

Skalární procesory, řetězené zpracování

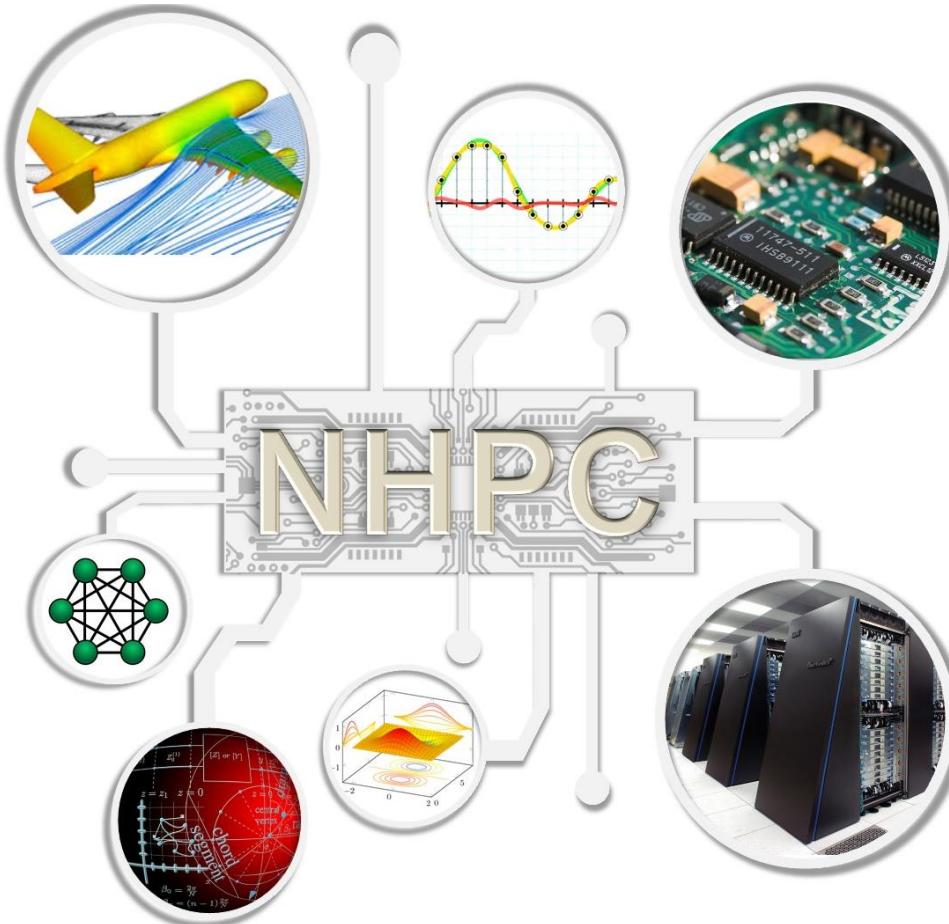
AVS – Architektury výpočetních systémů

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







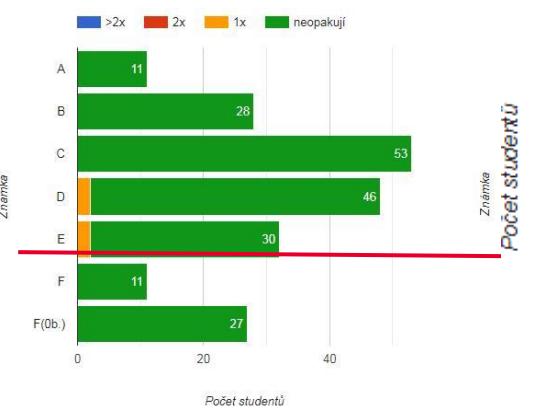
<https://www.facebook.com/skupina.stone>

- **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 + ...**



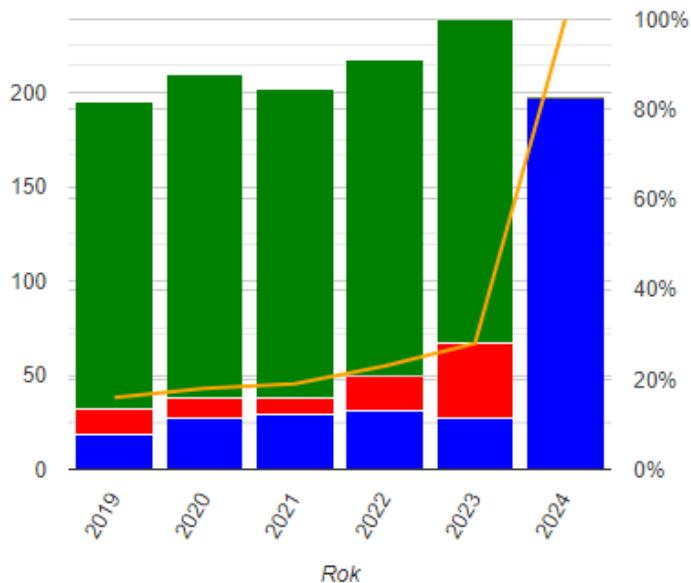
- Nejrychlejší a nejfektivnější způsob, jak se něco naučit.
- Nebojte se zeptat (nás výuka baví).
- Pokud máte pocit, že je předmět k ničemu nebo vás nebaví, dejte nám zpětnou vazbu.

2020 - Distančně, 14:00 - 16:00

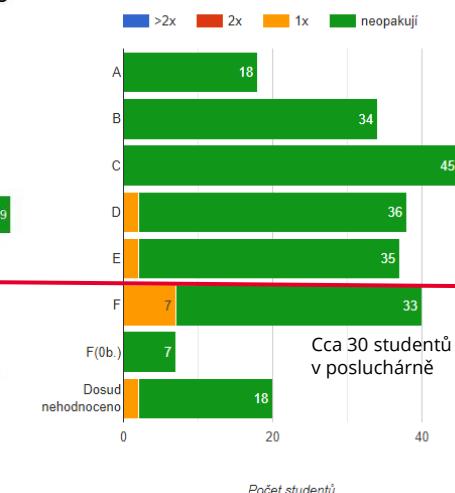


0b neabs. abs.

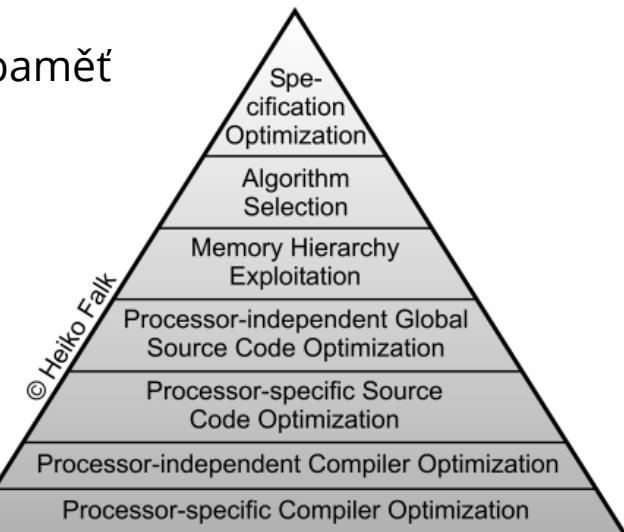
◀ 1/2 ▶



2023 - Prezenčně, 16:00-18:00



- **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 síť, 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 rychlosť 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.*

Rozdělení zátěže:

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

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

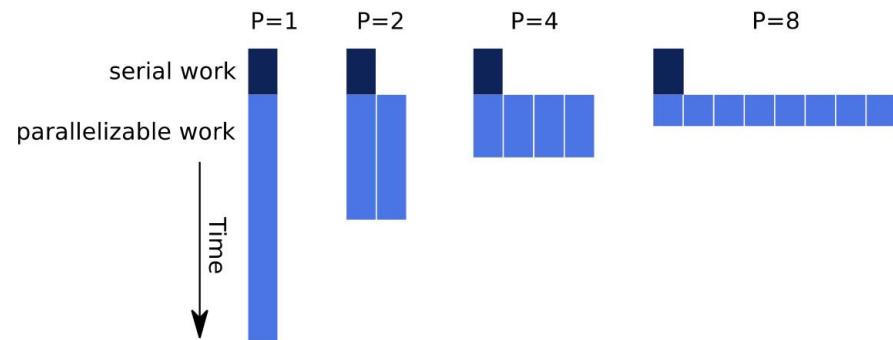
Zrychlení

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

Efektivita

$$\lim_{P \rightarrow \infty} E = 0$$

Amdahl



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>

I Gustafson's law: S počtem CPU roste zátěž

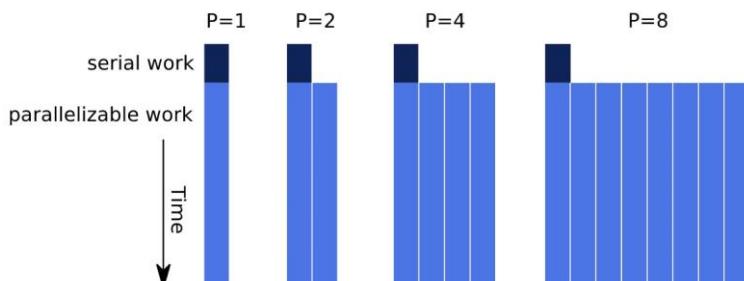
- S rostoucí velikostí úlohy (a tedy i práce W) často α relativně klesá a dostáváme dobré zrychlení.
- Při zachování stejného T_P pak roste T_S .
- Jestliže paralelní počítač pracoval na úloze **dobu T_P** , z toho dobu $\alpha_G T_P$ jen **jedno jádro** a po dobu $(1 - \alpha_G) T_P$ všech **P jader**,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

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

Gustafson

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

!



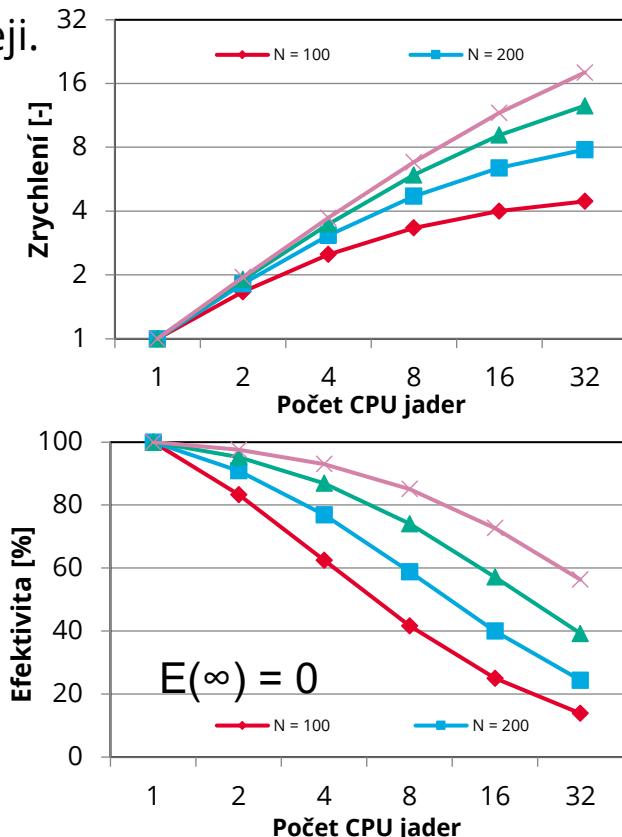
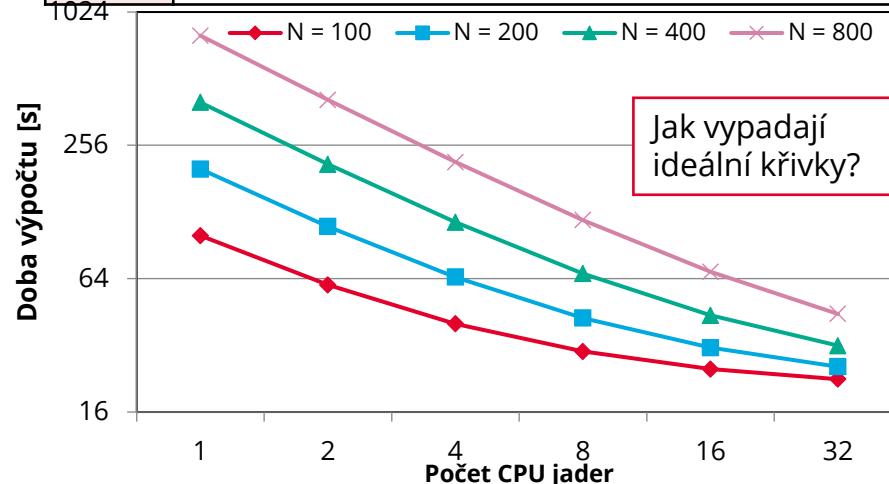
John L. Gustafson: Reevaluating Amdah's law (1998) <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.85.6348>

Silné škálování – Strong scaling

- **Silné škálování** – Konstantní celková práce (Amdahl)

- Snažíme se vykonat úlohu dané velikost N co nejrychleji.

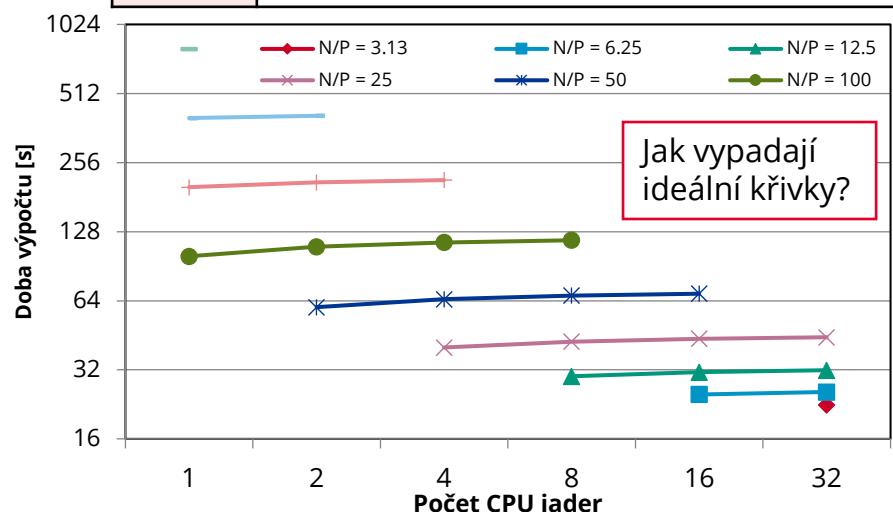
Doba výpočtu úlohy o velikosti N na P jádrech						
Úloha / P	1	2	4	8	16	32
$N = 100$	100	60	40	30	25	22.5
$N = 200$	200	110	65	42.5	31.25	25.625
$N = 400$	400	210	115	67.5	43.75	31.875
$N = 800$	800	410	215	117.5	68.75	44.375



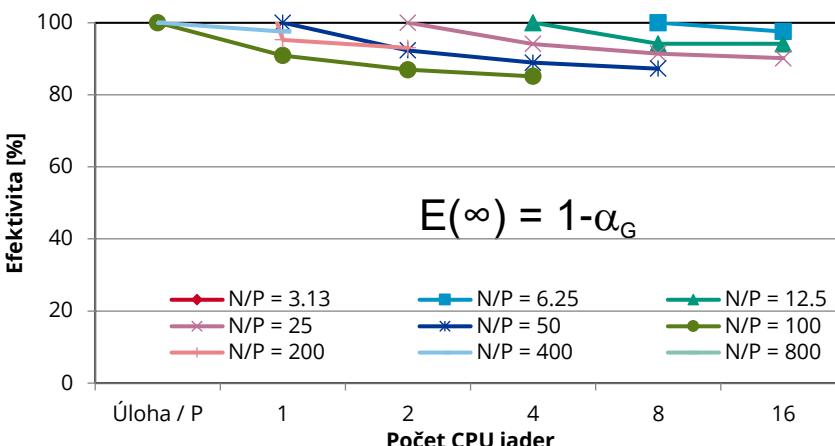
Slabé škálování – Weak scaling

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

Úloha / P	Doba výpočtu úlohy o velikosti N na P jádrech					
	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



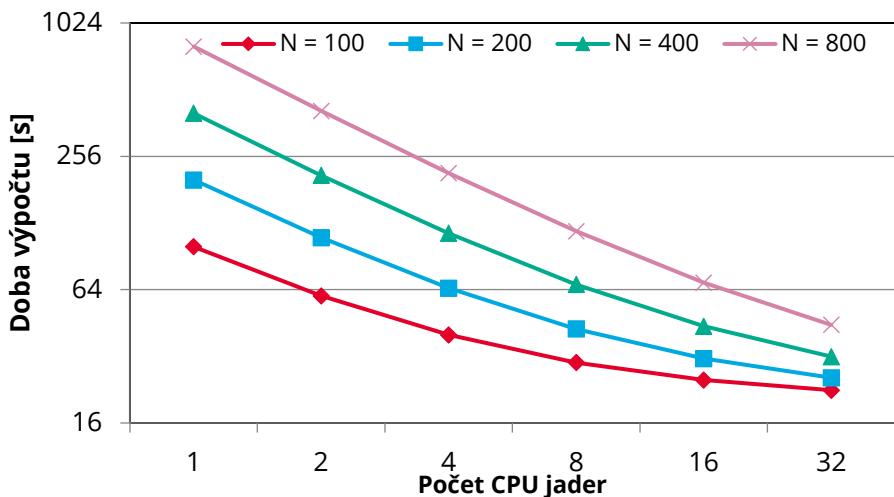
Úloha / P	Doba výpočtu úlohy velikosti N na P jádrech					
	1	2	4	8	16	32
N/P = 3.13						22.5
N/P = 6.25						25
N/P = 12.5						31.25
N/P = 25						31.87
N/P = 50					40	43.75
N/P = 100				65	115	144.37
N/P = 200			100	110	117.5	
N/P = 400		200	210	215		
N/P = 800	400	410				
N/P = 1000	800					



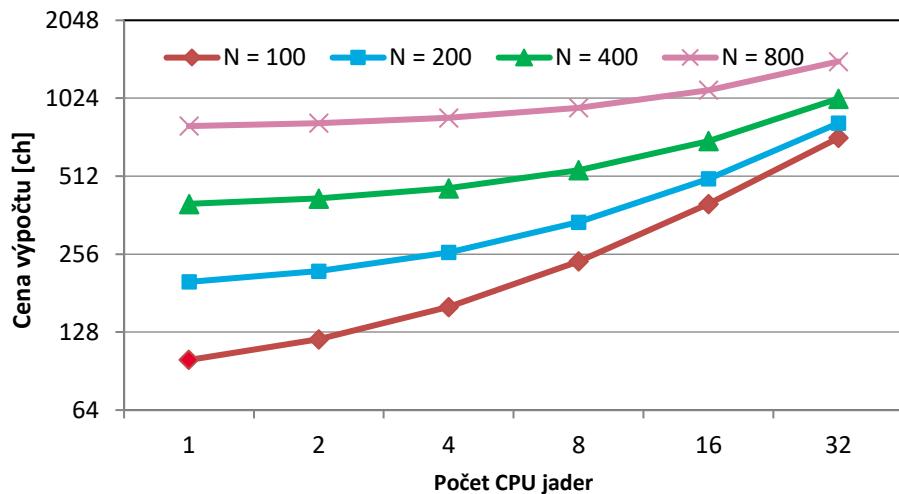
- Běžně účtovaná v jádrohodinách, nově ale i kWh

$$C = P * T_P$$

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



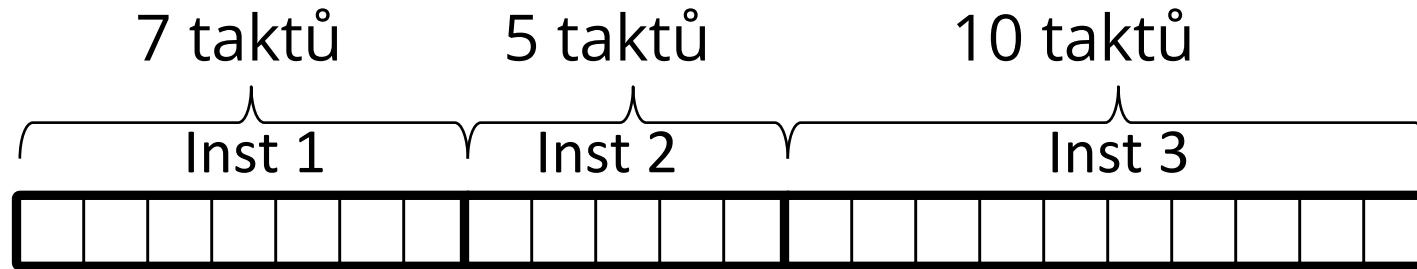
Architektura	Vydávání instrukcí	Provádění instrukcí
Subskalární von Neumann	Sekvenční 0 až 1	Sekvenční Několik taktů
Skalární (řetězené)	Sekvenční 0 až 1	Paralelní $CPI > 1$
Superskalární a VLIW	Paralelní 0 až m	Paralelní $IPC < m$
Vícejádrové s časovým MT	1 jádro	Z více vláken
Vícejádrové s časovým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

ILP ↑
↓ TLP

CPI – Clocks per instruction

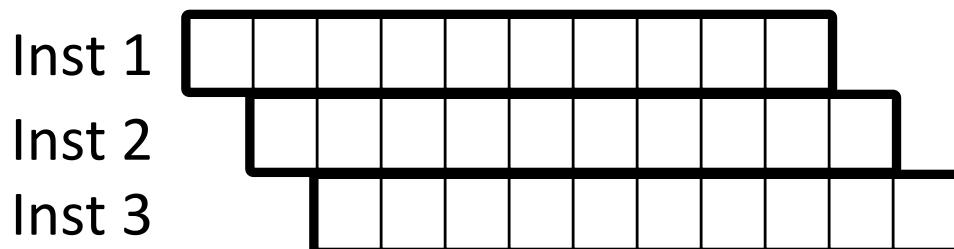
IPC – Instructions per clock

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

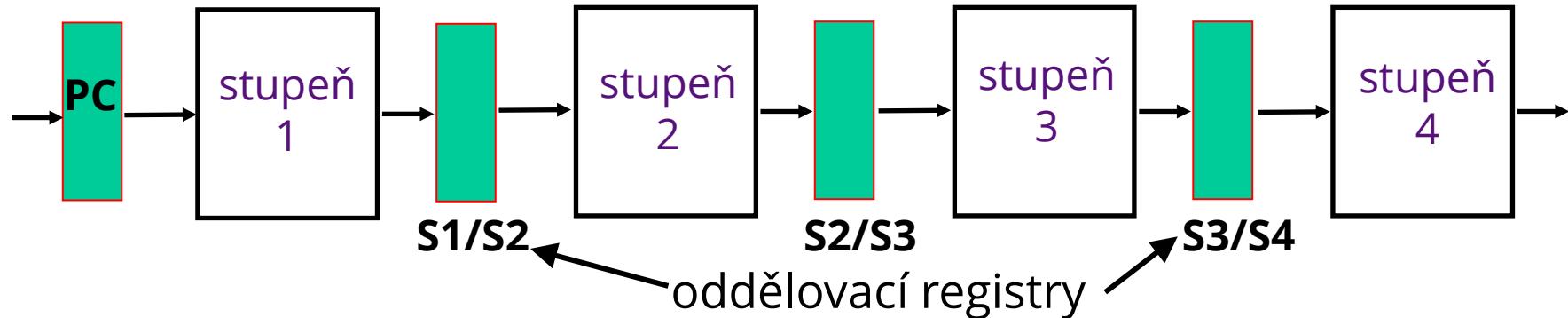


3 instrukce, 22 taktů, **CPI = 7,33**

- Řetězená (skalární) CPU



CPI = $(10 + 2) / 3 = 4$
ale v limitě $N \rightarrow \infty$
je ideálně CPI $\rightarrow 1$



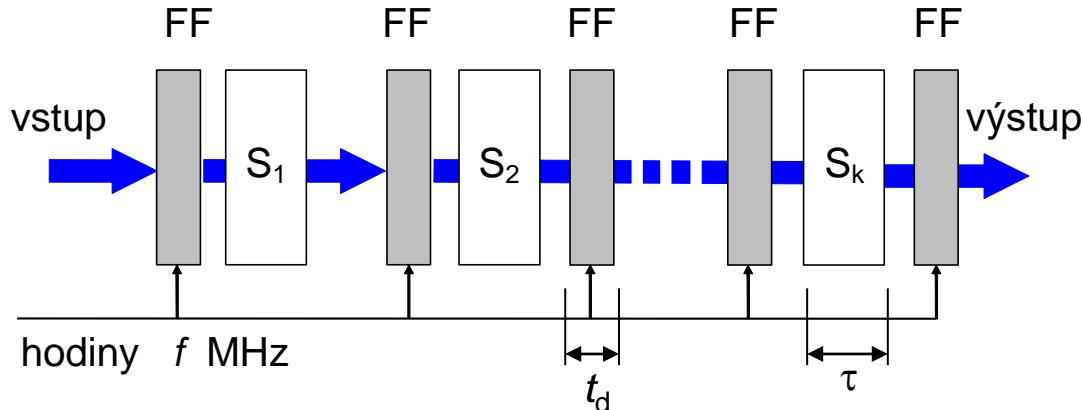
- Všechny objekty procházejí stejnými stupni (uniformní délka instrukcí).
- Žádná dvojice stupňů nesdílí HW prostředky, doba průchodu stupni je stejná.
- Načasování objektu vstupujícího do linky není ovlivněno objekty v jiných stupních.
- ***Tyto podmínky obecně platí pro průmyslové linky nebo VŠ výuku, 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:**

- **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 $\tau = t_1 / k$

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

1. Neřetězená CPU: průměrné trvání jedné instrukce

$$t_1 = 20 \text{ ns, výkonnost } R = 1 / t_1 = 50 \text{ MIPS}$$

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

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

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

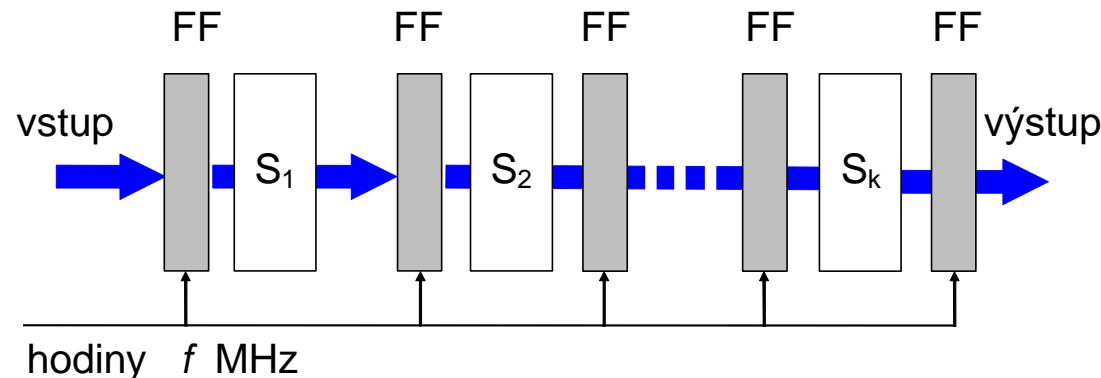
Pro počet instrukcí $N \rightarrow \infty$:

$$S_\infty = \frac{k \tau}{\tau + t_d} = \frac{20}{4 + 1} = 4$$

- 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 $CPI = 1 + q$.

$$\underset{N \rightarrow \infty}{S} = \frac{k\tau}{(\tau + t_d)(1+q)} \rightarrow \frac{k}{1+q} = \frac{k}{CPI}$$



I 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 $N \rightarrow \infty$ a $t_d \ll \tau$.

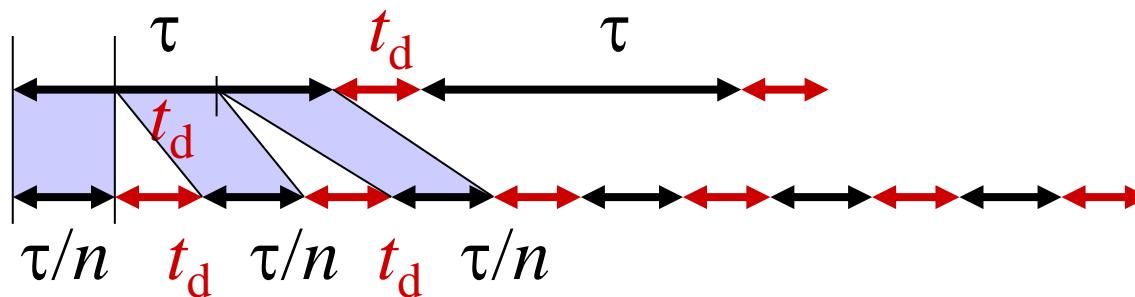
Řešení: CPI = 1 + pokuty vztažené na 1 instrukci

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

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

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

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



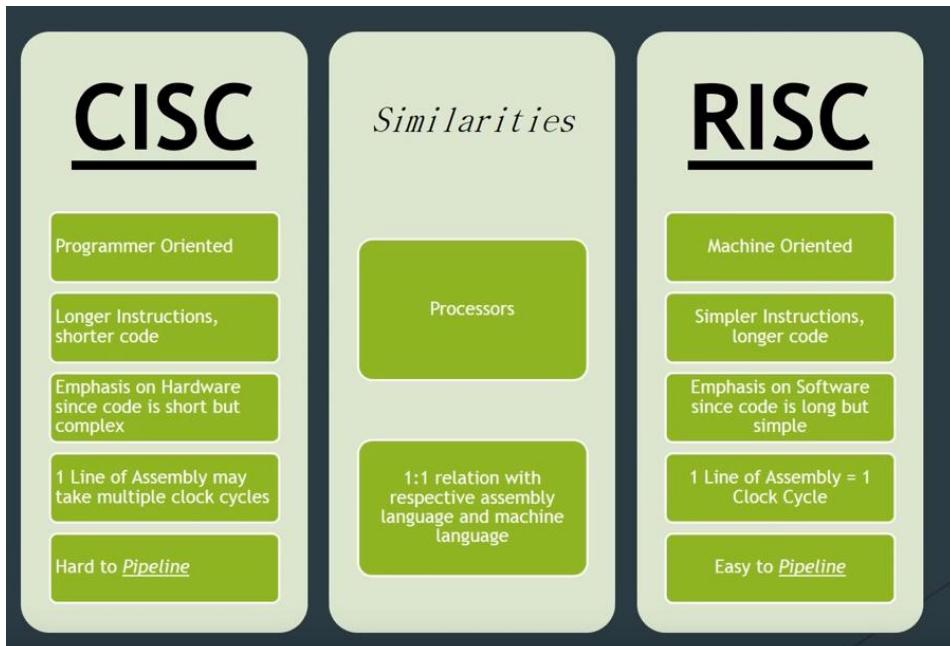
Např.

$$\tau = 2 \cdot t_d, n = 3: \quad S = 3 / (2 / 3 + 1) = 1,8$$

ŘETĚZENÝ PROCESOR RISC

ARCHITEKTURY DLX

- **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



6 5 5 5 5 6 bitů

[op | src1 | src2 | dst | shamt | funct] Register-type

[op | src | dst **address/immediate**] Imm-type

[op | **target address**] Jump-type



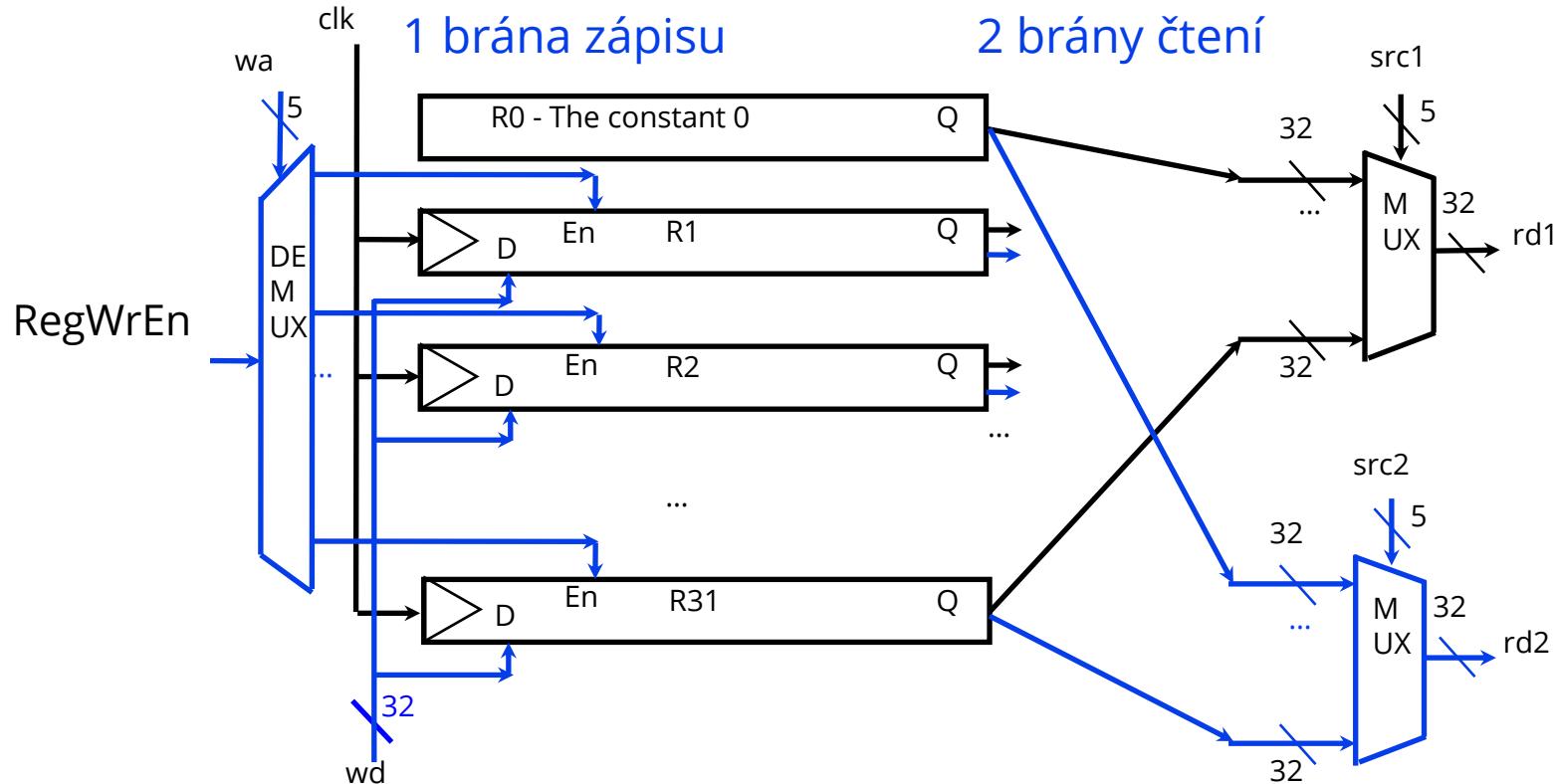
rozšíření znaménka (sign extension)

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

nepodmíněný 26 bitů

podmíněný 16 bitů

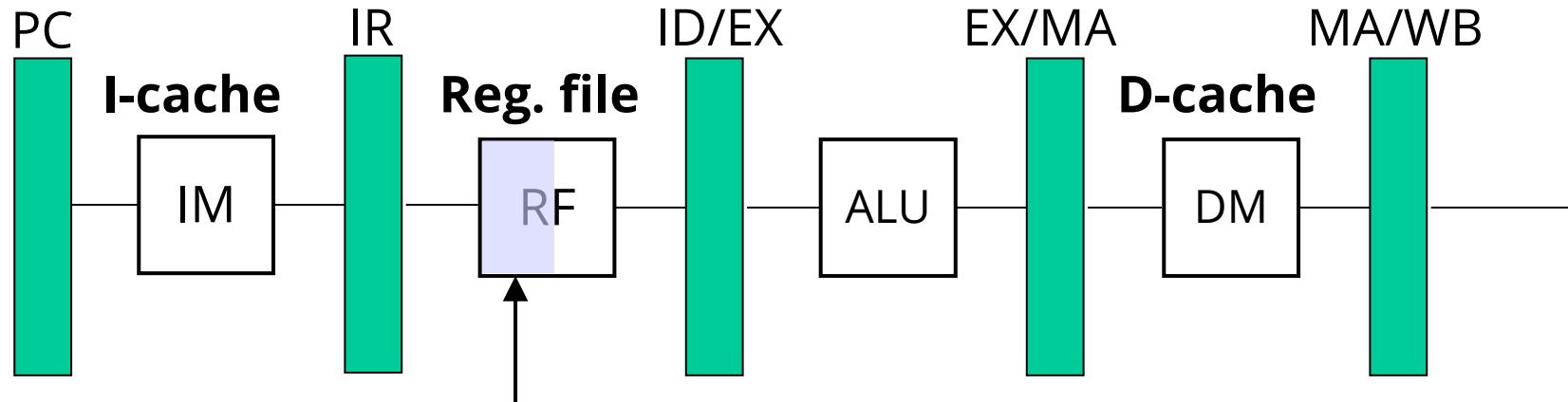
I Soubor registrů RF procesoru DLX



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

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

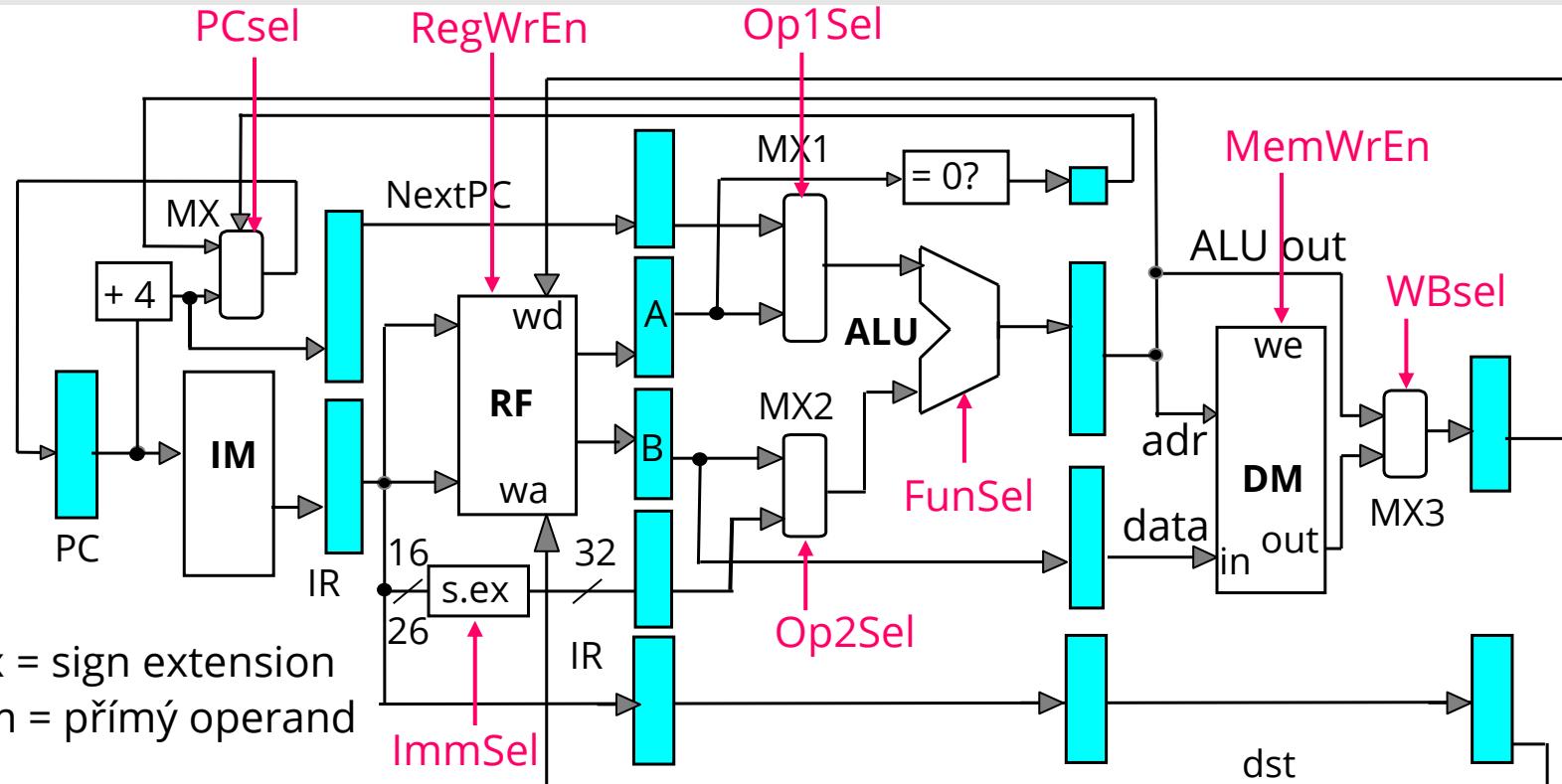
AG = Address Generation	generování adresy
E/C = Execute / Cache access	



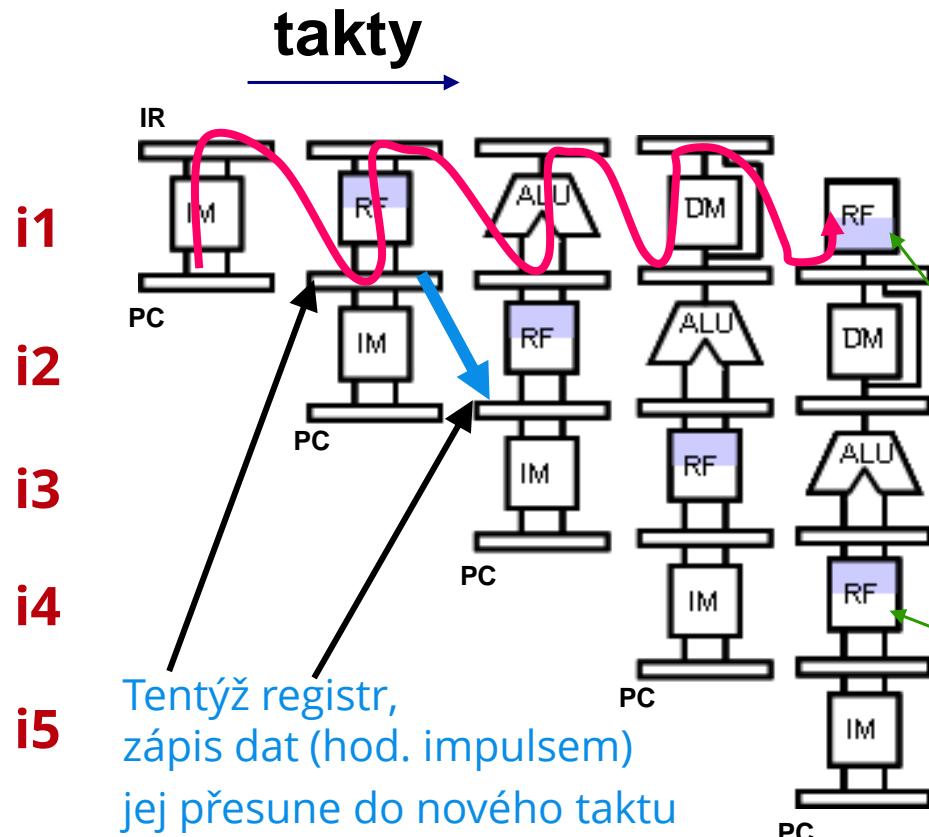
IFetch načtení instrukce	-	IDecode načtení operandů	-	EXecute výpočet (adresy)	-	Memory (Cache) Access	-	Write Back do reg.
--------------------------------	---	--------------------------------	---	--------------------------------	---	-----------------------------	---	--------------------------

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

I Klasická RISC Pipeline (MIPS/DLX)

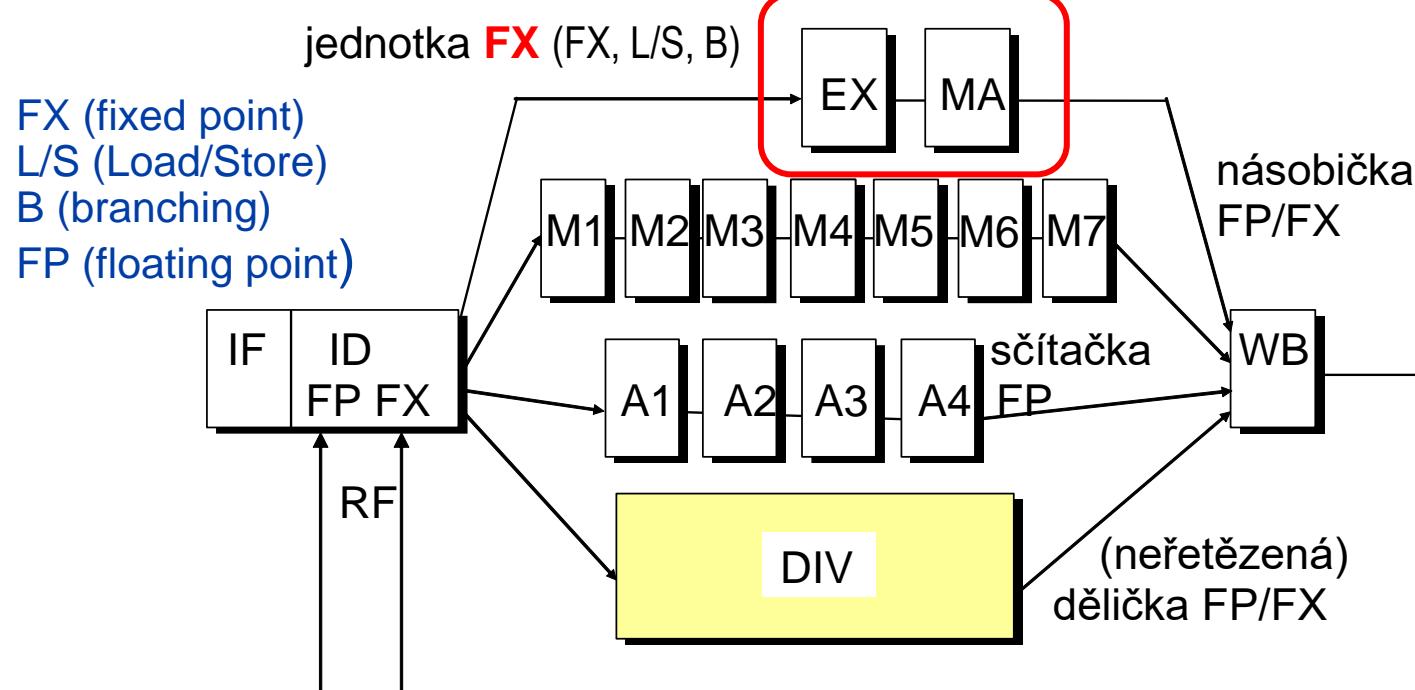


- Linkou putuje **op-kód, src1, src2** a **dst** dokud je třeba.
- Op-kód je dekódován ve stupni ID, **řídicí signály** putují linkou přes oddělovací registry do míst určení.
- <https://comparch.edu.cvut.cz/>



Pozor, HW linka směruje 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.!



- 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

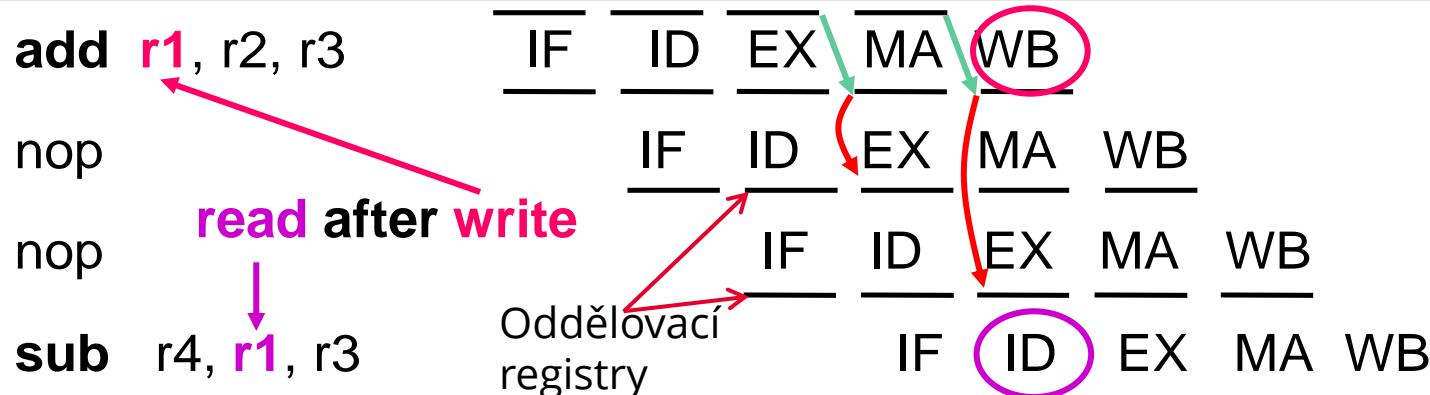
- Instrukce může záviset na něčem co produkuje dřívější instrukce
 - závislost se může týkat hodnoty dat
→ **datová závislost (konflikt)**
 - závislost se může týkat adresy příští instrukce
→ **řídicí závislost (větvení, výjimky)**
- Instrukce v řetězené lince může potřebovat prostředek, který právě používá jiná instrukce
→ **strukturní závislost (konflikt)**

DATOVÉ KONFLIKTY

- **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 komplátorem 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.

I | 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.

add r1, r2, r3



sub r4, r1, r3



and r6, r1, r7



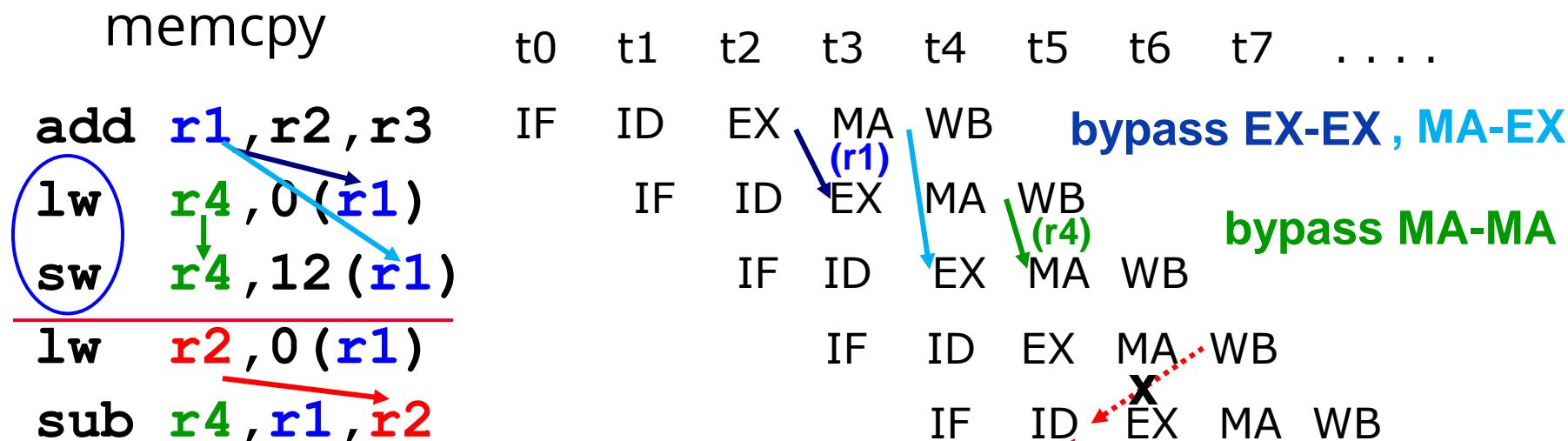
or r8, r1, r9



?
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.)



- **(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

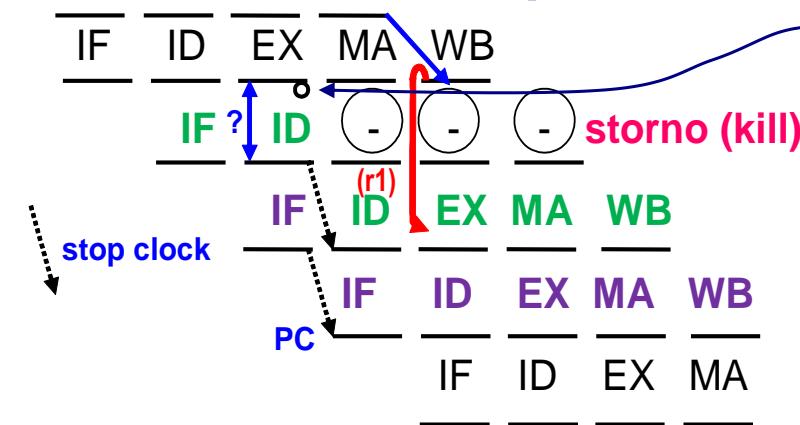
lw r1, 0(3)

sub r4, r1, r2

add/sub

add r5, r1, r3

or r4, r1, r2



„o“ = bubble, injekce
NOP nebo kill bitu
(mění instrukci na
NOP =)

= detekce konfliktu

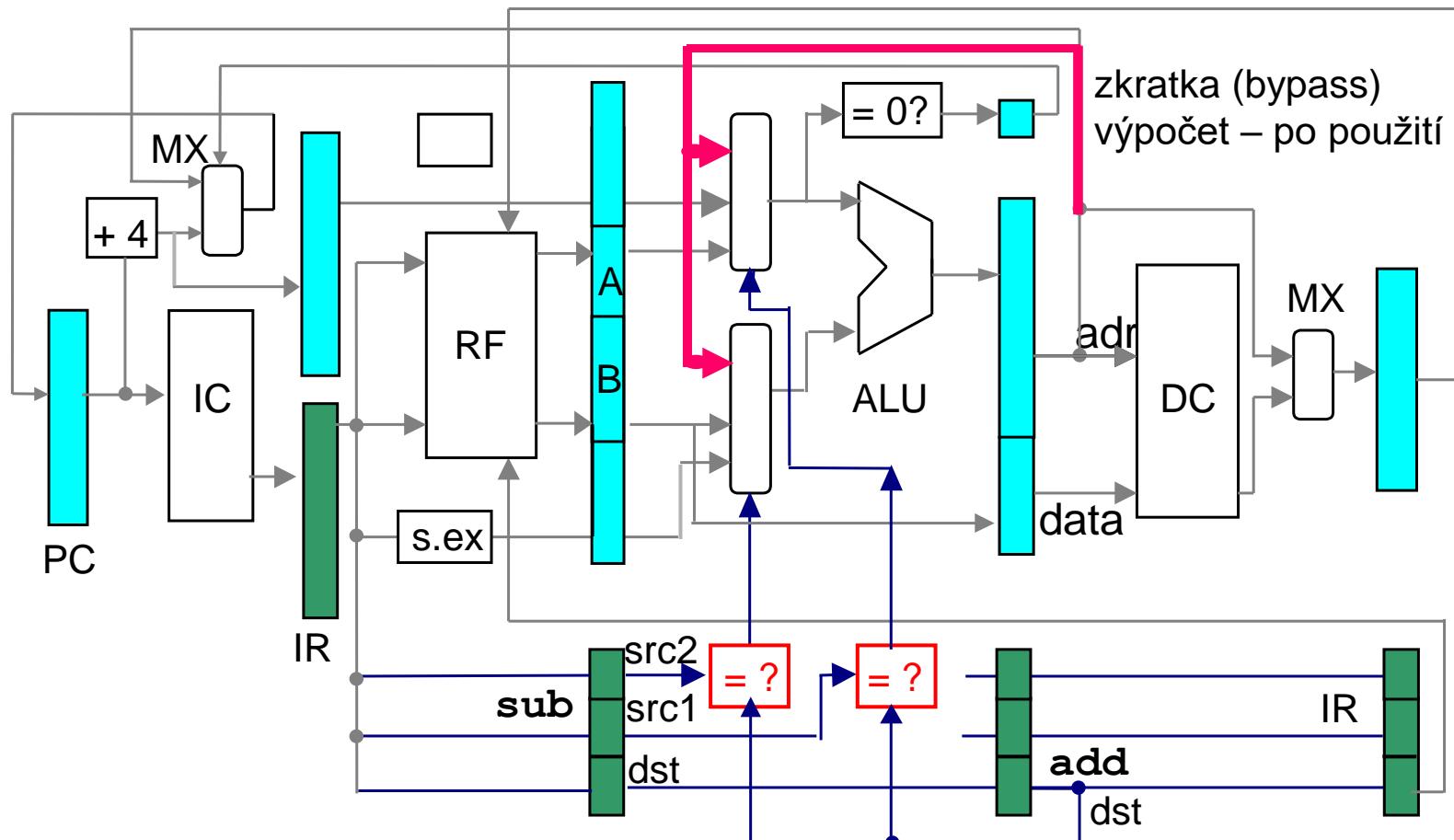
Původní verze sub je stornována, zpožděná sub pokračuje;
add (resp. cokoliv po sub) musí být též zpožděno.

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

stop clock
= stall

I HW podpora předávání dat EX → EX (RAW)



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

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

pořadí zápisů změněno



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

odstranění konfliktu přejmenováním
cílového registru

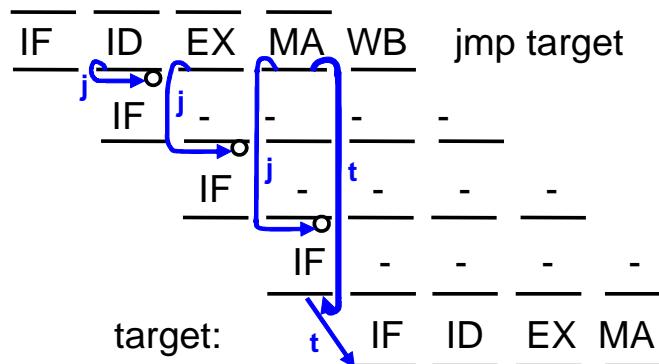
V klasické 5 stupňové lince FX konflikty WAR a WAW vzniknout nemohou.

ŘÍDÍCÍ KONFLIKTY

- 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 podmínsku.

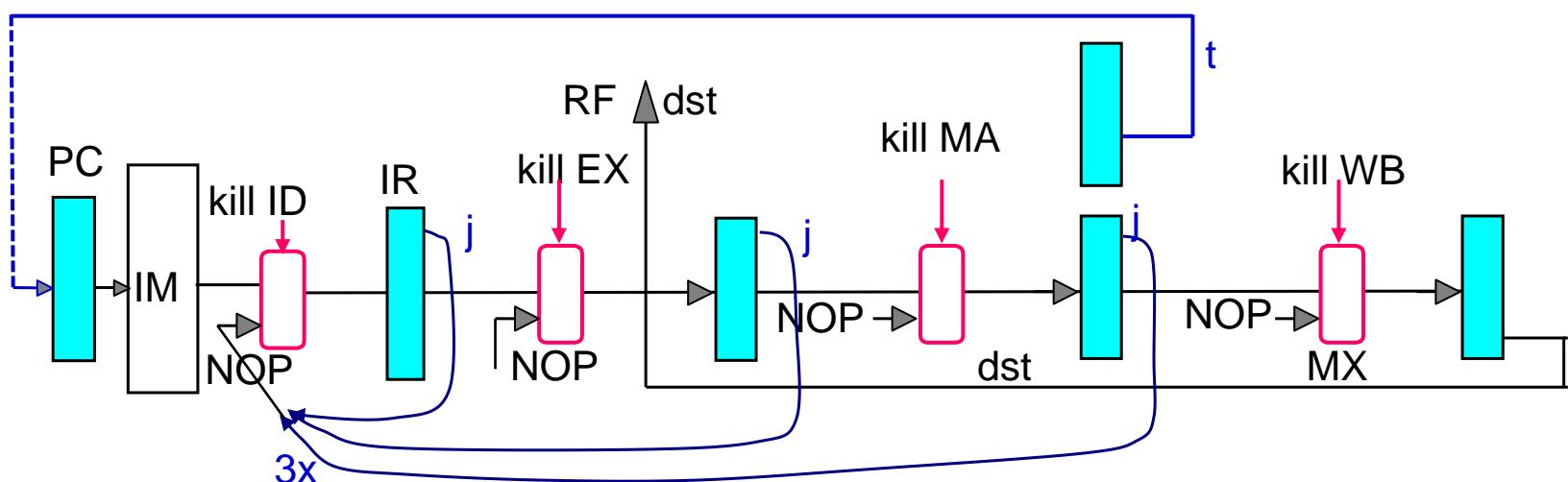
Výpočet cílové adresy (PC + rel. adresa) a podmínky je třeba co nejvíce urychlit!

I Nepodmíněný skok j relativně k PC

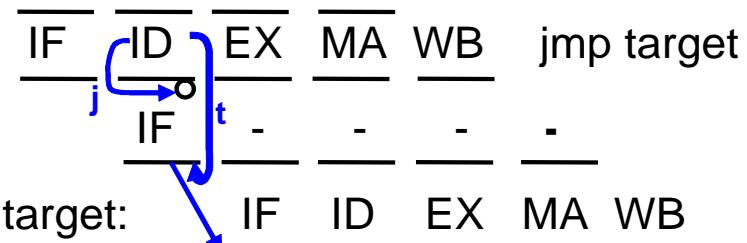
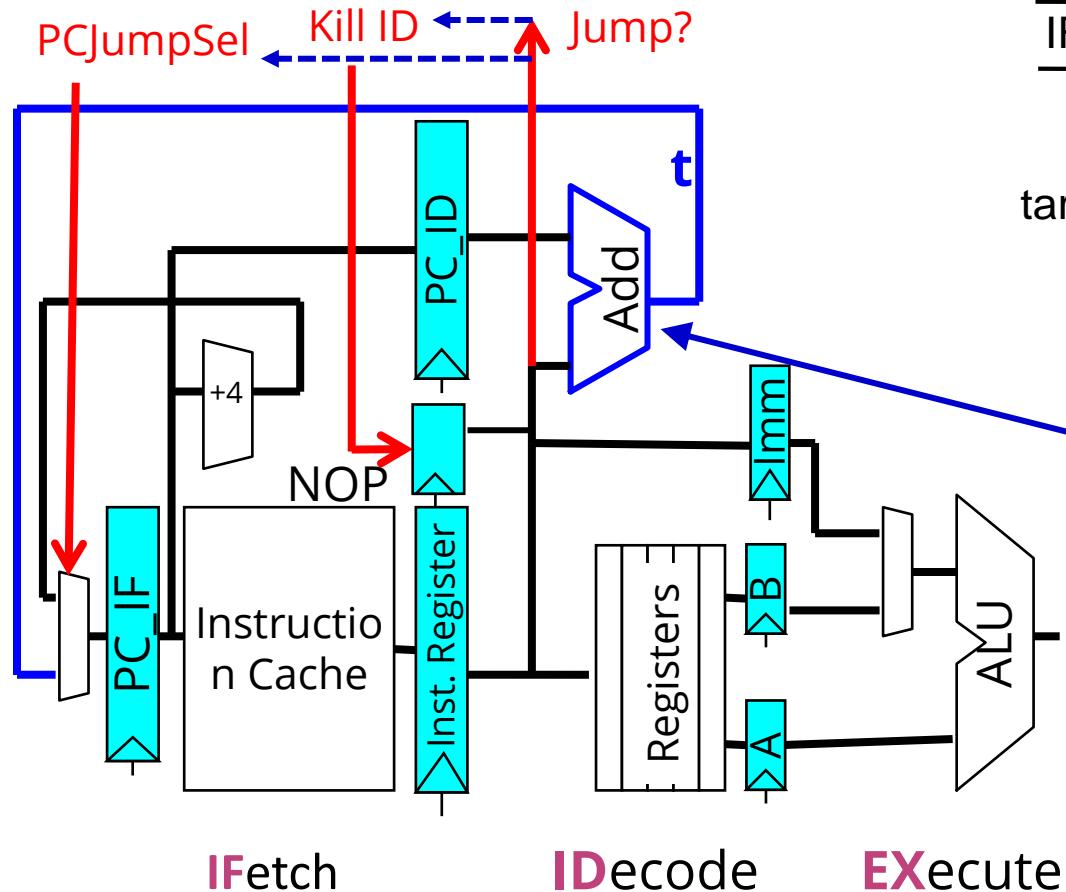


Adresa se spočte v EX, během MA se přepíše do PC → pokuta 3 takty.

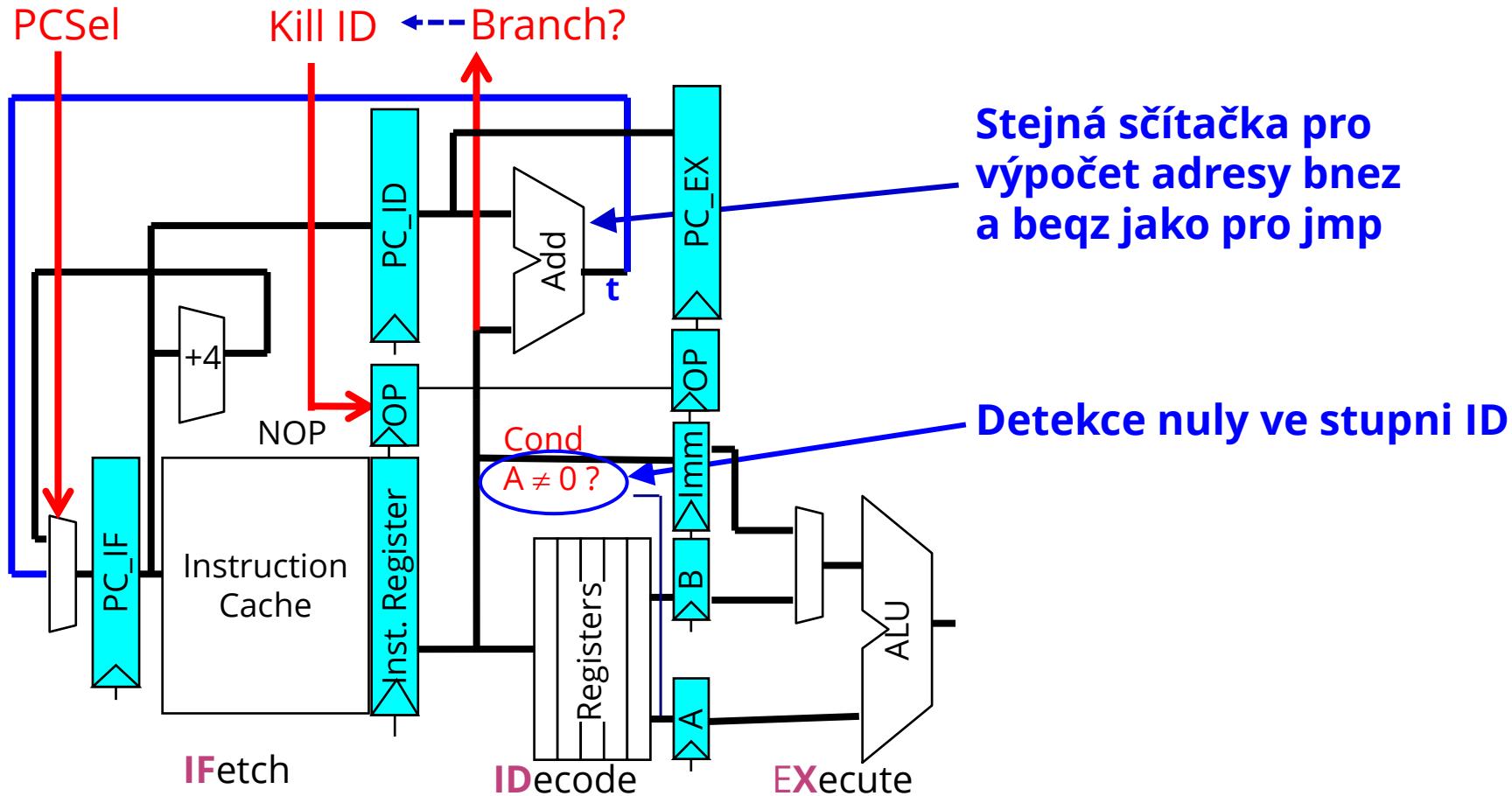
Storno již načtené nebo
rozpracované instrukce (**kill**):
injekce NOP do IR.



Redukce pokuty u nepodmíněného skoku

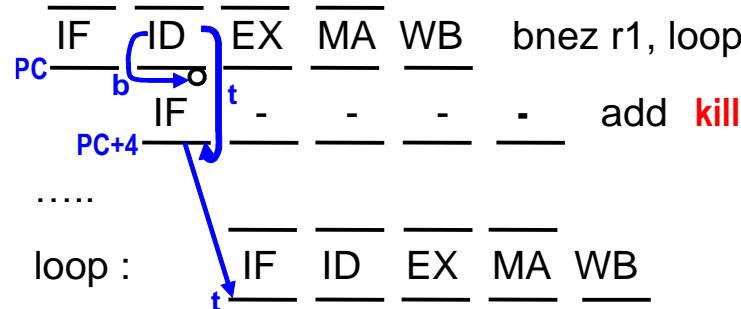


Nová sčítačka pro výpočet adresy t ve stupni ID: pokuta jen 1 takt



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

```
loop: ...
      bnez r1, loop
      add ...
```



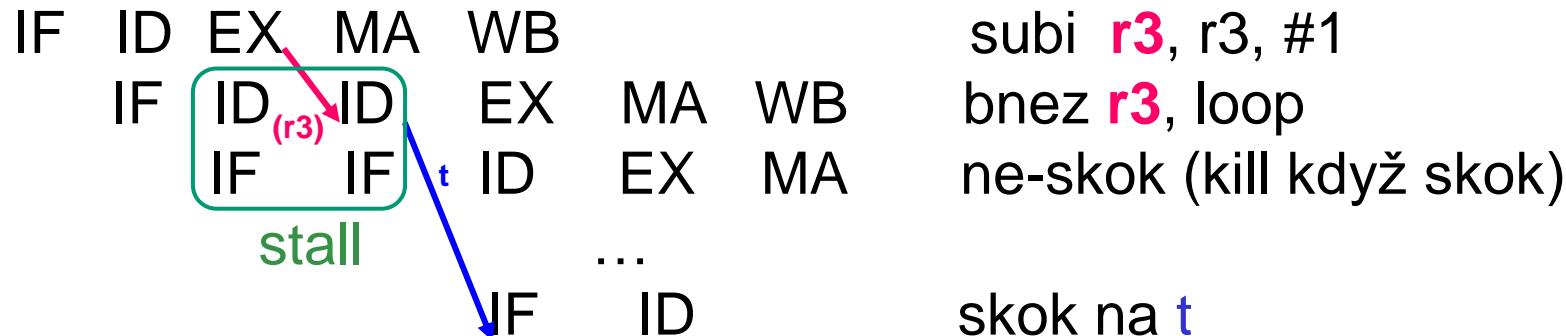
Fixní negativní predikce: jedeme dál s add. Pouze je-li test true, stornujeme add a aktivujeme bypass pro cílovou adresu *t* (skok na loop).

Pokuta pak bude:

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



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



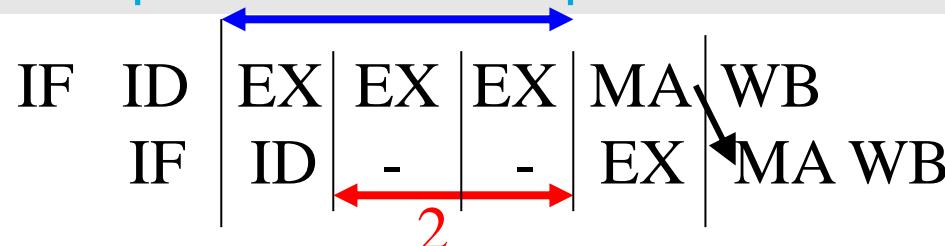
Pokuta je tedy

- 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).

- 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

fadd
sd

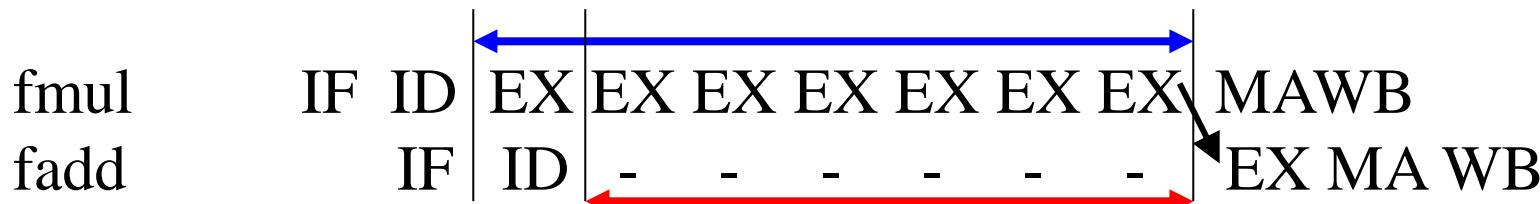


Latence fadd = 3

Mezioperační latence (fadd, sd) = 2

(producent, konzument)

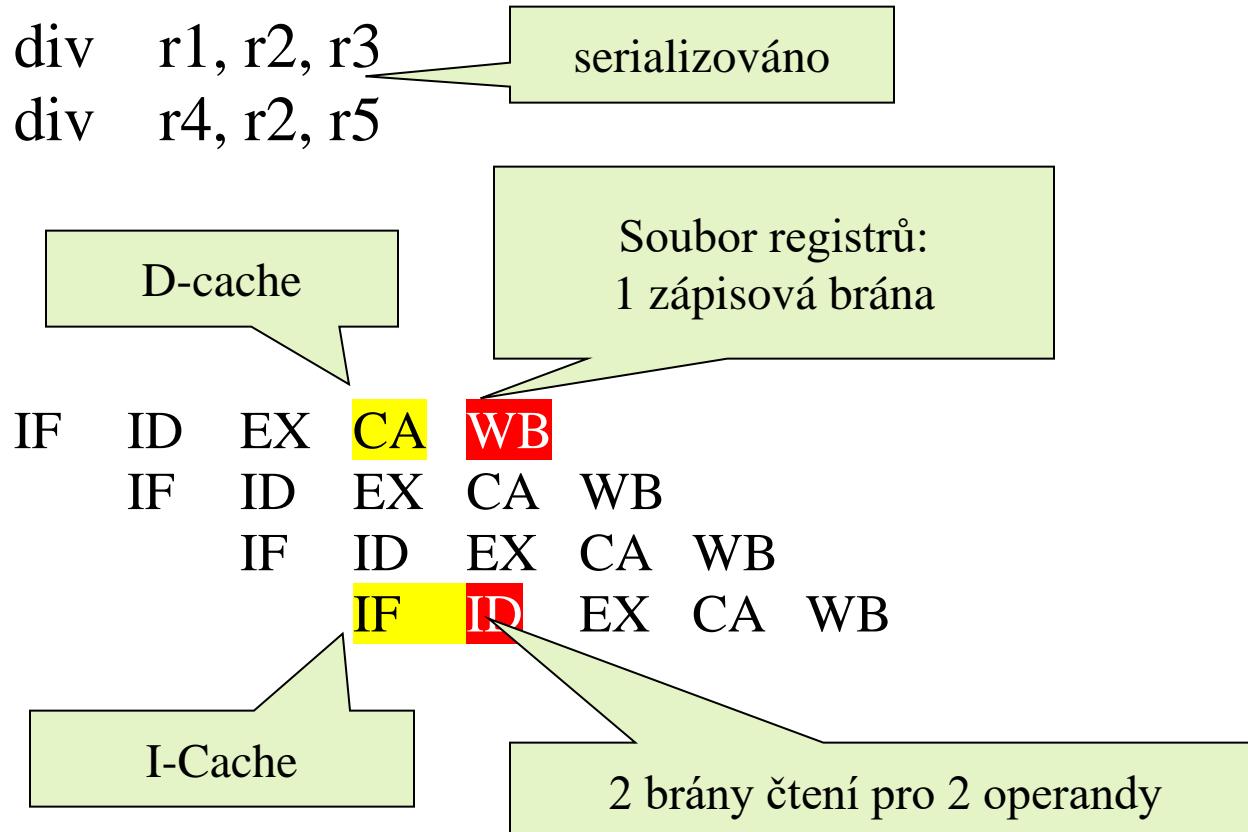
Latence fmul = 7



Mezioperační latence (fmul, fadd) = 6

STRUKTURNÍ KONFLIKTY

... když 2 instrukce potřebují stejný prostředek.



PŘÍKLADY

```
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 přeskladáním instrukcí (provádí kompilátor).

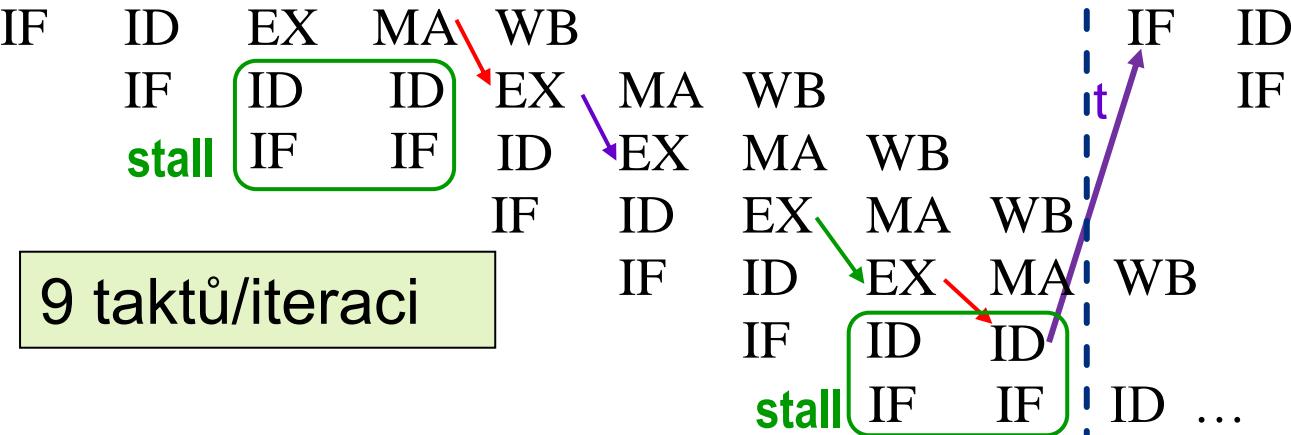
Nakreslete diagramy řetězového zpracování.

I Příklad 3: Smyčka s předáváním dat

loop:

```
lw r1, 0(r2)      IF ID EX MA WB  
addi r1, r1, #1   IF ID EX MA WB  
sw r1, 0(r2)      stall IF ID EX MA WB  
addi r2, r2, #4   IF ID EX MA WB  
sub r4, r3, r2    IF ID EX MA WB  
bnez r4, loop     stall IF ID EX MA WB
```

...



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!**

loop:

lw r1, 0(r2)	IF	ID	EX	MA	WB		IF		ID ... predikce O.K.
addi r2, r2, #4		IF	ID	EX	MA	WB		IF ... bez pokuty	
sub r4, r3, r2		IF	ID	EX	MA	WB			
addi r1, r1, #1		IF	ID	EX	MA	WB			
sw r1, -4(r2)		IF	ID	EX	MA	WB			
bnez r4, loop		IF	ID	EX	MA	WB			

pozitivní predikce (příště)

Diagram illustrating the predicted execution flow for the next iteration (positive prediction). A dashed vertical line separates the current iteration from the next. Arrows indicate the flow of control between stages: IF → ID → EX → MA → WB. The final stage of the current iteration (WB) has three arrows pointing to the corresponding stages (IF, ID, EX) of the next iteration, representing the predicted sequence of operations.

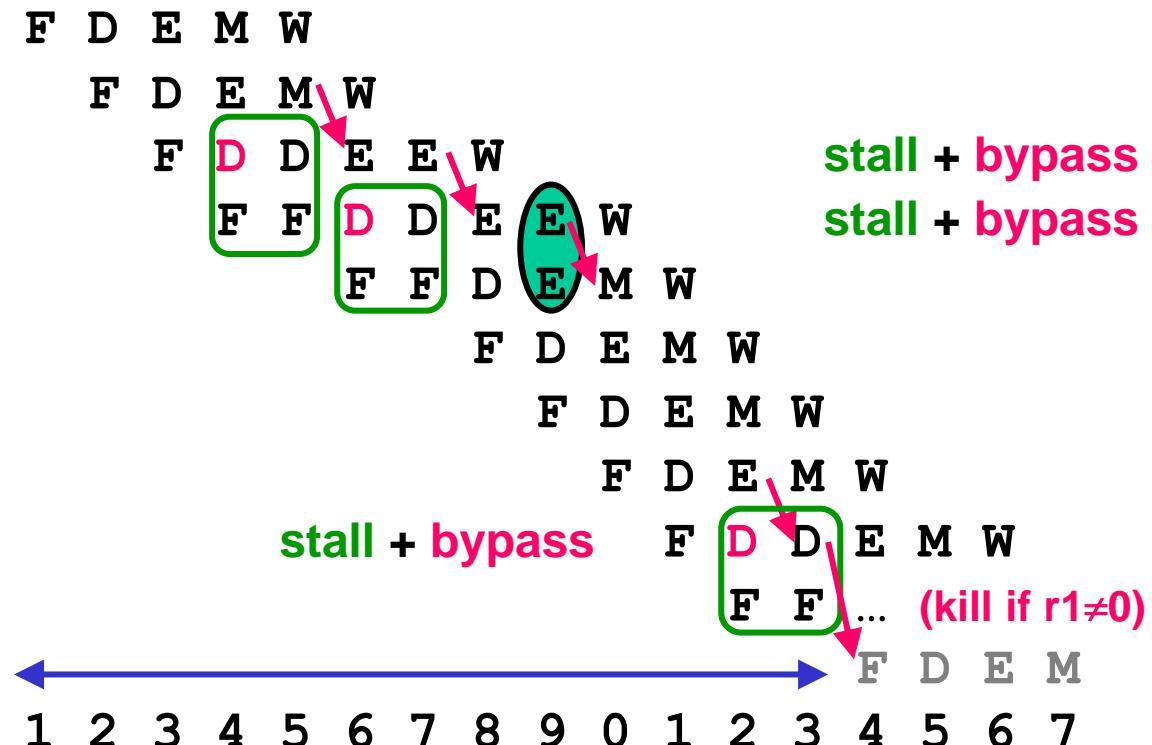
Po úspěšné **pozitivní predikci** ve stupni IF(bnez) **hned na cílovou adresu**: 6 taktů/iteraci.

- 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.

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
	...	



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

I Příklad 5: Inkrementace prvků vektoru ve FP

loop: ld f0, 0(r1) ; f0 = prvek vektoru

 nop

 fadd f4, f0, f2 ; přičti skalár z f2

 nop

 nop

 sd f4, 0(r1) ; ulož výsledek zpět

 subi r1, r1, #8 ; dekrementuj ukazovátko o 8 bajtů

 nop

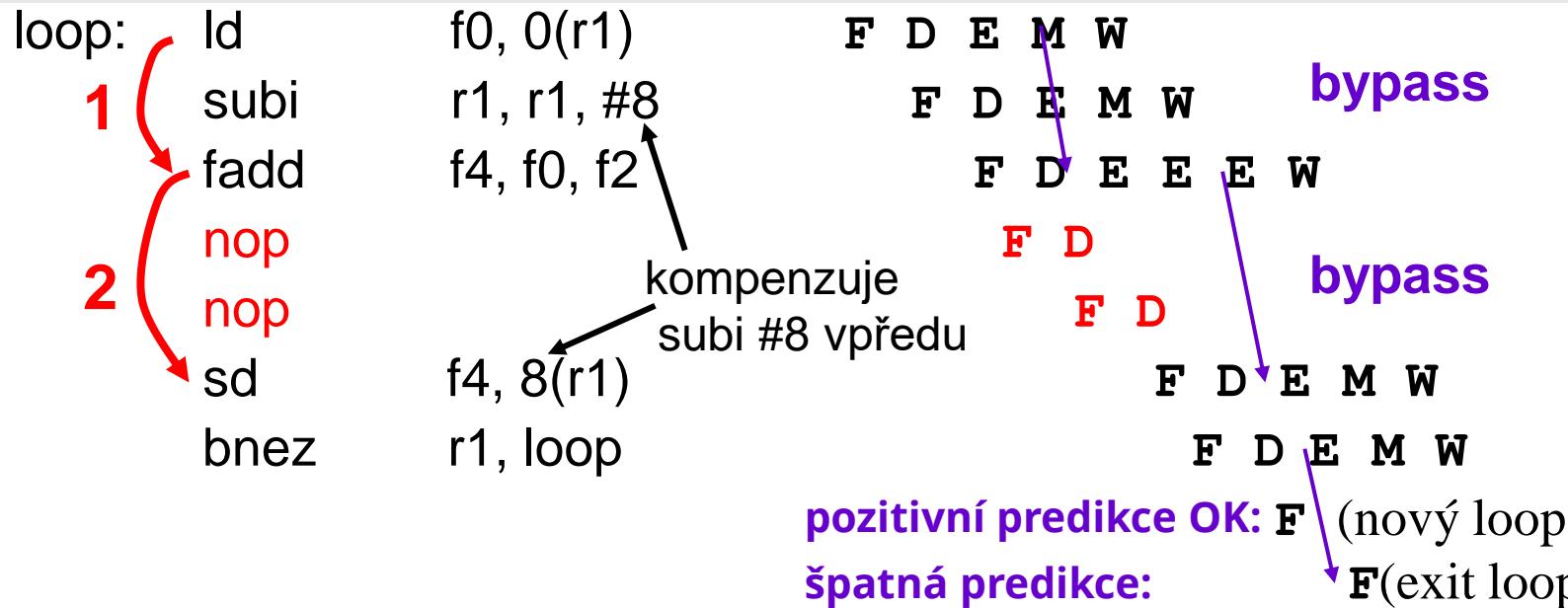
 bnez r1, loop ; skoč když r1 ≠ 0

**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

I 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:	Id	f0, 0(r1)	
	fadd	f4, f0, f2	mezera 1 takt mezera 2 takty
	sd	0(r1), f4	vynech subi & bnez
	Id	f0, -8(r1)	přejmenej f0 → f6, f4 → f8
	fadd	f4, f0, f2	vynech subi & bnez
	sd	-8(r1), f4	vynech subi & bnez
	Id	f0, -16(r1)	přejmenej f0 → f10, f4 → f12
	fadd	f4, f0, f2	vynech subi & bnez
	sd	-16(r1), f4	vynech subi & bnez
	Id	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: Id f0, 0(r1)
fadd f4, f0, f2
sd 0(r1), f4

Id f0, -8(r1)
fadd f4, f0, f2
sd -8(r1), f4

Id f0, -16(r1)
fadd f4, f0, f2
sd -16(r1), f4

Id f0, -24(r1)
fadd f4, f0, f2
sd -24(r1), f4

subi r1, r1, #32
bnez r1, loop

loop: Id f0, 0(r1)
Id f6, -8(r1)
Id f10, -16(r1)
Id 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



27 taktů včetně prázdných
6,8 taktu/element

14 taktů, žádný prázdný
3,5 taktu/element

Změna
pořadí
zpracování
instrukcí

4 iterace
prolnuty

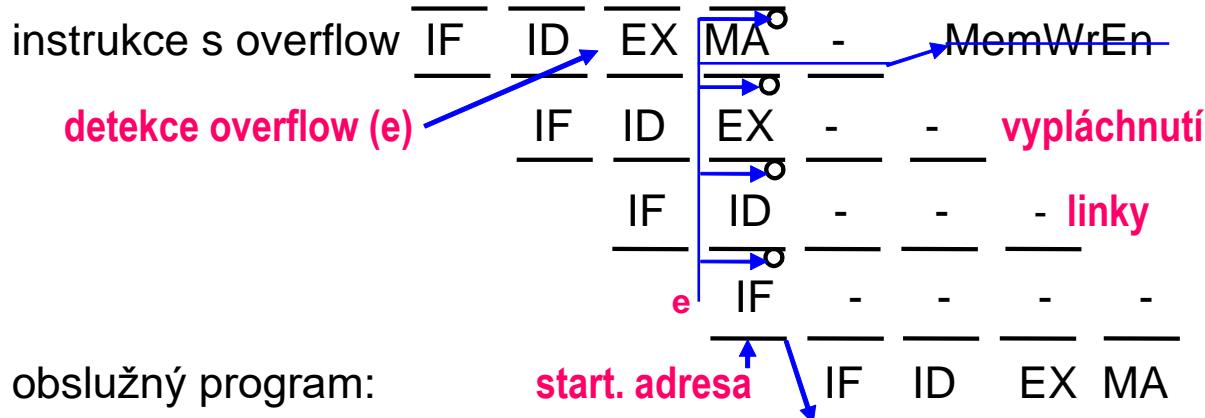
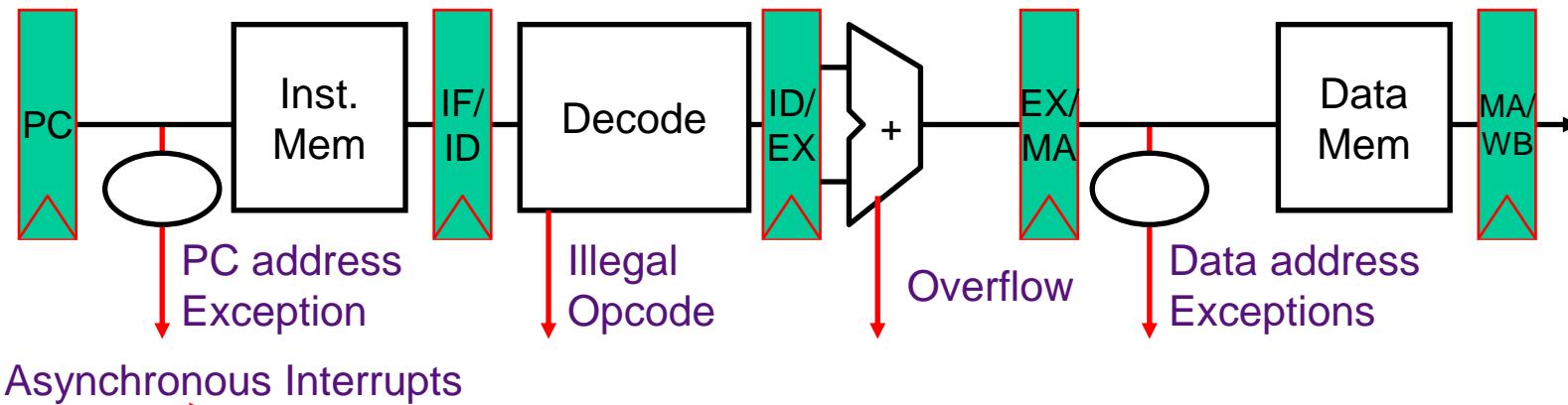
Odstraněny
všechny
RAW
konflikty

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)

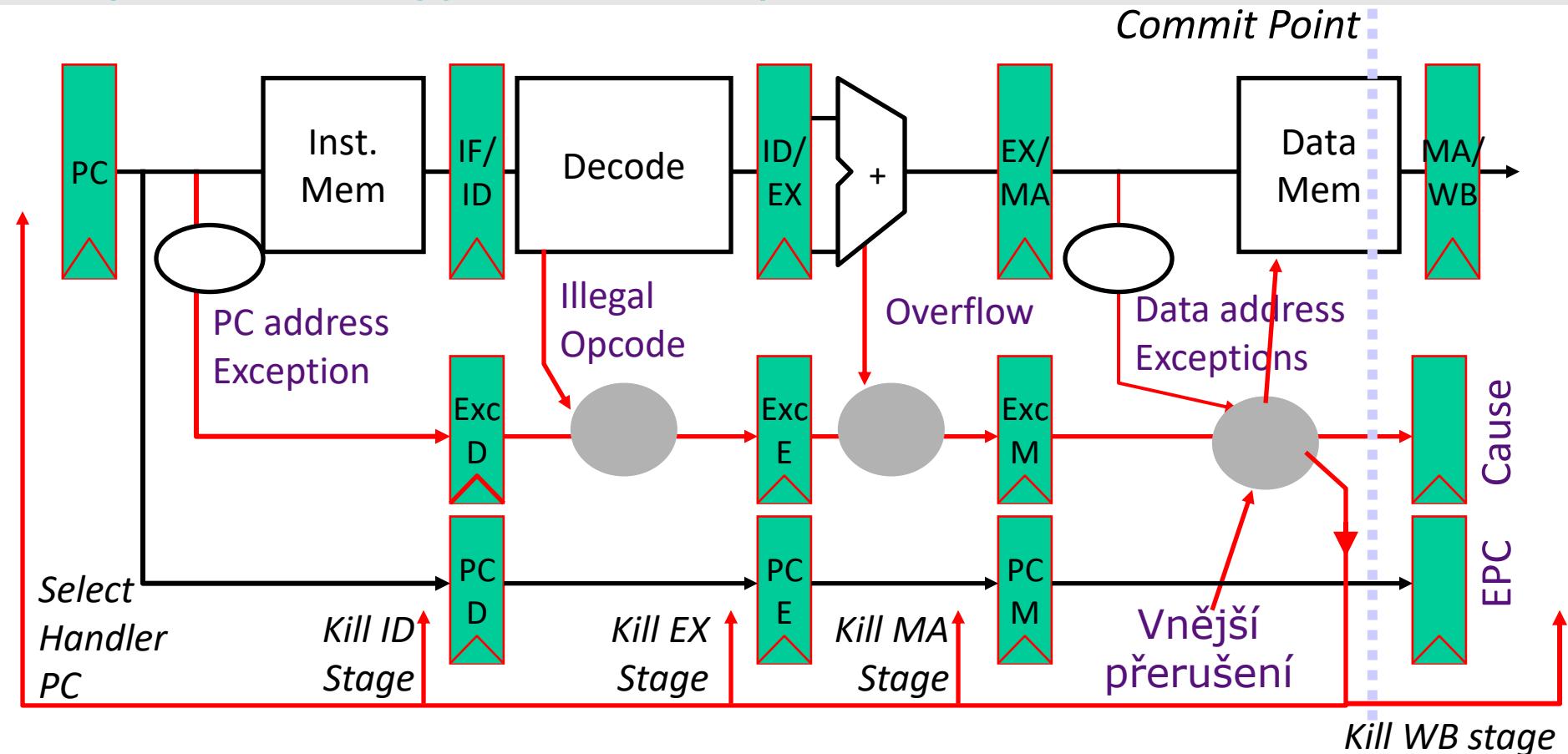
- Když se procesor rozhodne zpracovat přerušení
 - zastaví běžící program u instrukce I_i , a dokončí všechny instrukce až do I_{i-1}
 - uloží PC instrukce I_i do speciálního registru (EPC, Extra PC),
 - zablokuje další přerušení a předá řízení určenému programu obsluhy přerušení běžícímu v kernelu OS.
- 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.

I Zpracování výjimek (5 stupňová linka)



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

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, zahod' (kill) částečně provedené instrukce (NOP do oddělovacích registrů, do sekce op-kódu).
- Injektuj adresu obslužného programu do PC.

Pokračování příště

Dynamické plánování instrukcí

AVS – Architektury výpočetních systémů

Týden 2, 2024/2025

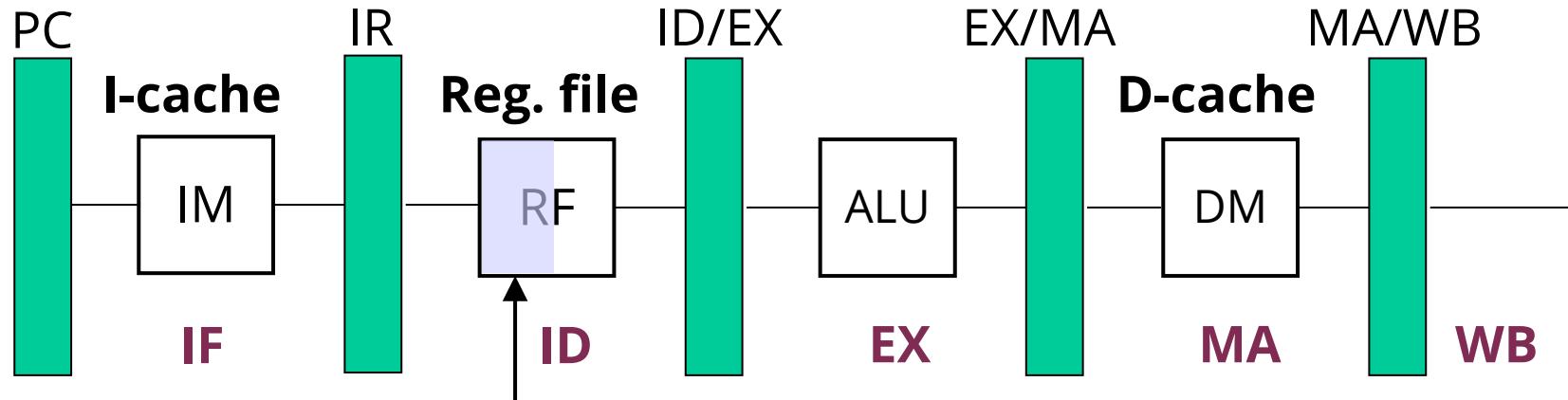
Jirka Jaroš

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



OPAKOVÁNÍ

Ref: [Henessy, str 147-162,](#)
[Appendix C](#)



Cena za řetězení

$$S_{\infty} = \frac{k\tau}{(\tau + t_d)}$$

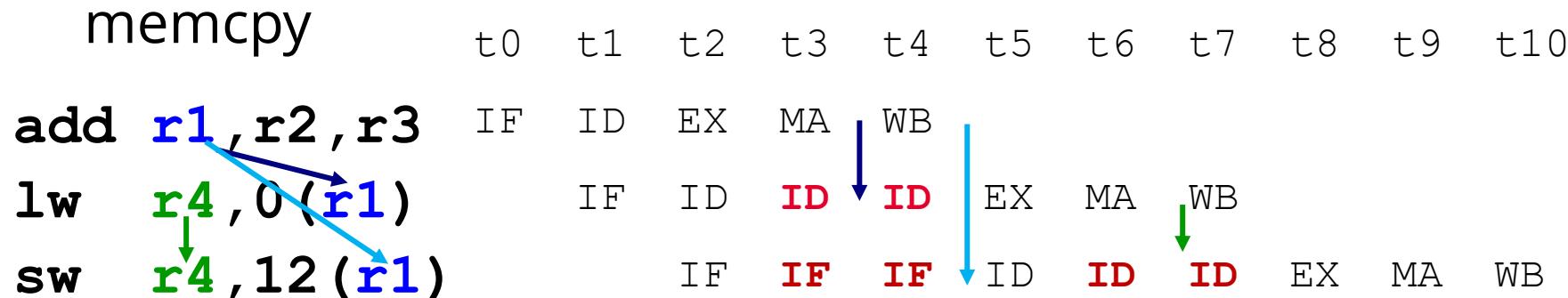
Průměrná cena za konflikty

$$S = \frac{k\tau}{(\tau + t_d)(1+q)} \rightarrow \frac{k}{1+q} = \frac{k}{CPI}$$

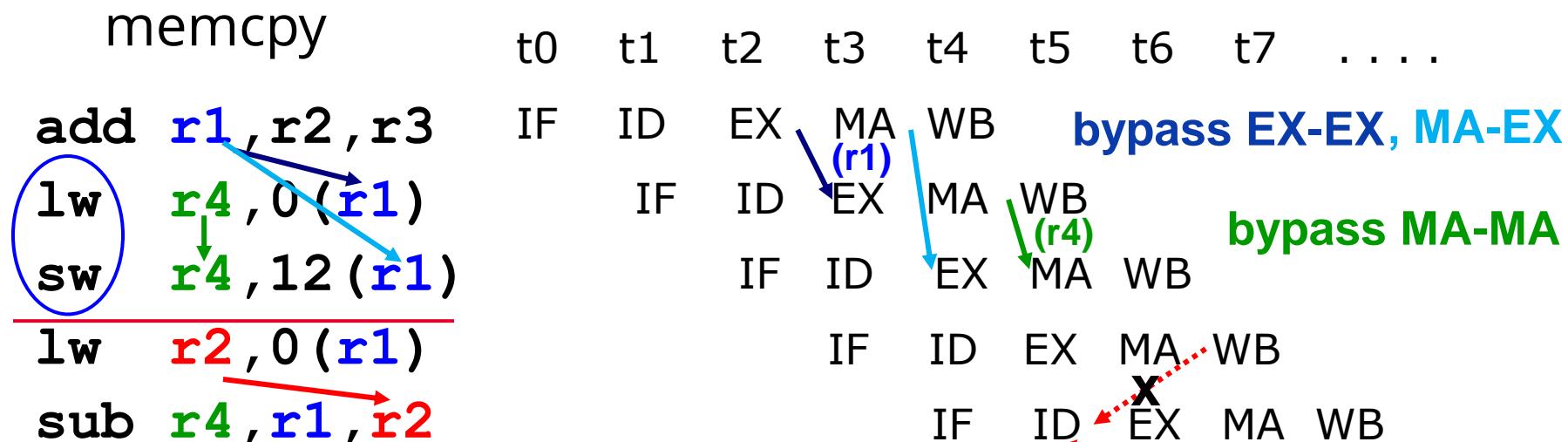
$\tau \gg t_d$

- **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 komplátorem 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.

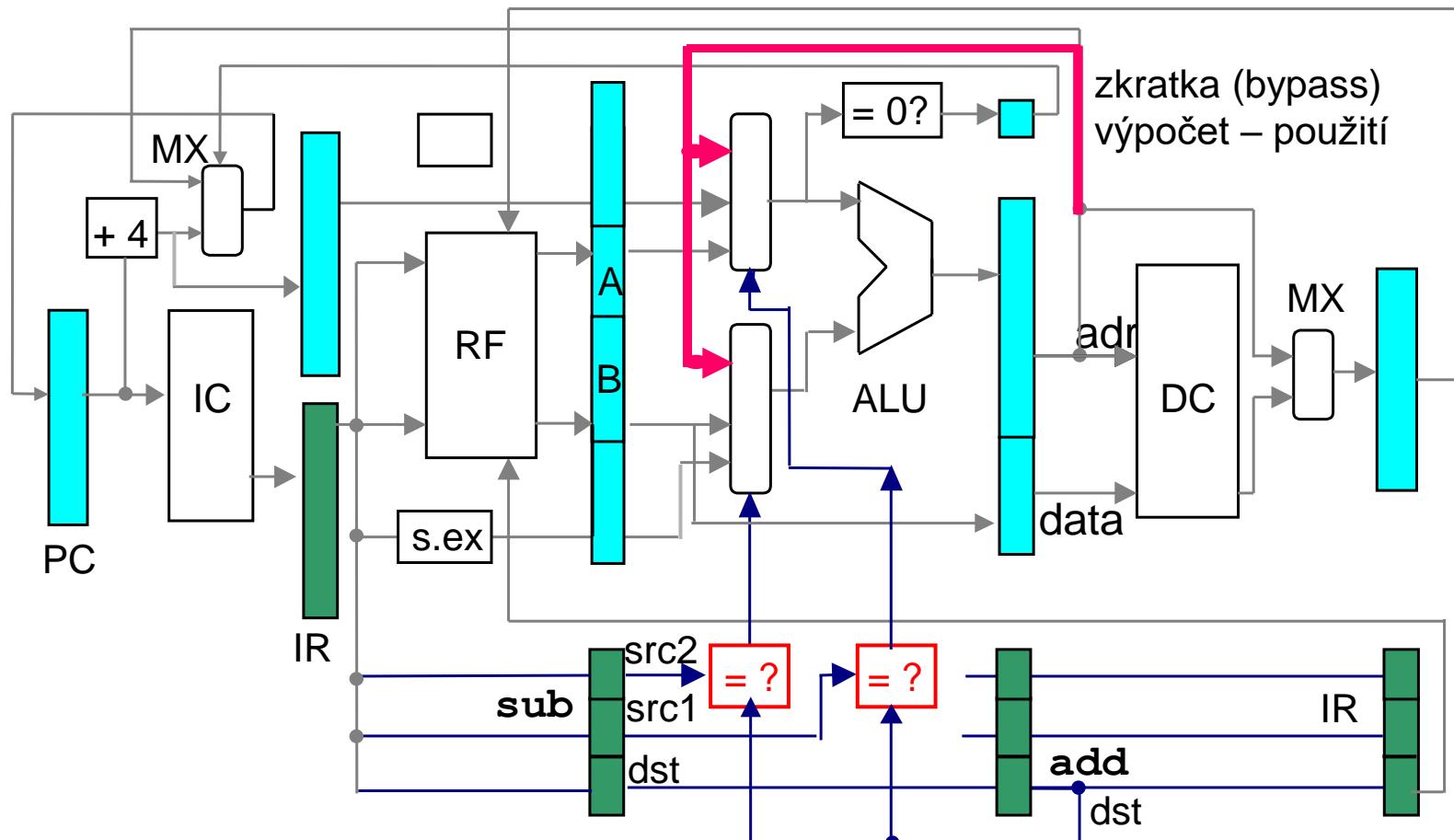


- r1 sice vznikne ve fázi EX, ale do registru se dostane až na začátku fáze WB
 - Následující instrukce čtou R1 ve fázi ID
 - **Pozor, nemohu mít dvě instrukce ve stejné fázi**
- r4 se sice načte ve fázi MA, ale do registru se dostane až na začátku fáze WB
 - Následující instrukce čte již ve fázi ID
 - Jelikož je zpoždění na předchozí instrukci, stráví instrukce SW (stall) ve fázi IF 3 taky a ve fázi ID také 3 takty



- **(r1) = výsledek add, který teprve bude zapsán do reg. r1**
- Kopírování dat lw – sw bez pokuty.
- **Předávat data do minulého taktu nelze !!
sub musí počkat další tak ve fázi EX!**

I HW podpora předávání dat EX → EX (RAW)



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

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

pořadí zápisů změněno



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

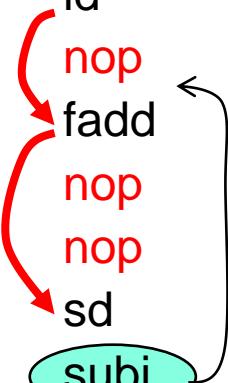
odstranění konfliktu přejmenováním
cílového registru

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

```

loop:    ld      f0, 0(r1)      ; f0 = prvek vektoru
          nop
          fadd   f4, f0, f2      ; přičti skalár z f2
          nop
          nop
          sd     f4, 0(r1)      ; ulož výsledek zpět
          subi   r1, r1, #8       ; dekrementuj ukazovátko o 8 bajtů
          nop
          bnez   r1, loop        ; skoč když r1 ≠ 0

```



**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

Ref: [Hennessy, str 147-162](#)

Příklad: Rozbalení smyčky minimalizující prostoje

loop: Id f0, 0(r1)
fadd f4, f0, f2
sd 0(r1), f4

Id f0, -8(r1)
fadd f4, f0, f2
sd -8(r1), f4

Id f0, -16(r1)
fadd f4, f0, f2
sd -16(r1), f4

Id f0, -24(r1)
fadd f4, f0, f2
sd -24(r1), f4

subi r1, r1, #32
bnez r1, loop

loop: Id f0, 0(r1)
Id f6, -8(r1)
Id f10, -16(r1)
Id 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



27 taktů včetně prázdných
6,8 taktu/element

14 taktů, žádný prázdný
3,5 taktu/element

Změna
pořadí
zpracování
instrukcí

4 iterace
prolnuty

Odstraněny
všechny
RAW
konflikty

SUPERSKALÁRNÍ PROCESORY

Ref: [Henessy, str 167-193,](#)
[233-247, Appendix C](#)

Jak zrychlit dobu výpočtu?

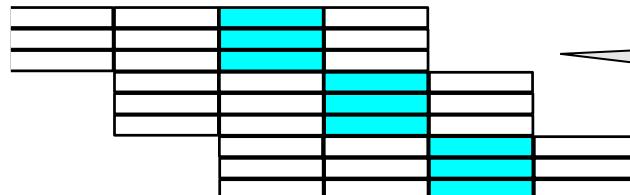
$$\text{doba výpočtu} = \mathbf{IC} * \mathbf{CPI} * \mathbf{T}$$

- **IC** = Instruction Count, počet provedených instrukcí – optimalizace kódu.
- Snížení doby taktu **T**
 - větší počet stupňů linky, až 30 (**hlubší linka**, superřetězení)
 - ale velké pokuty, vyšší příkon
 - dolní hranice: doba zpoždění hradel a příkon
- Snížení **CPI**, tj. zvýšení **IPC**:
 - více instrukcí (až *m*) v jednom stupni
 - *m*-cestný superskalární procesor (tedy širší linka)
 - složitější, nižší možný kmitočet
 - reálně dosažitelná hodnota IPC (kolik instrukcí průměrně končí v jednom taktu) je vždy značně nižší než *m*

Moderní mikroprocesory používají oba přístupy.

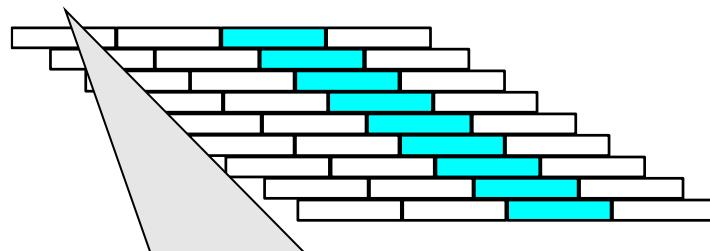
Zpoždění 1 stupně skalárního procesoru τ je rozděleno na n kratších taktů, počet cest navýšen na m nebo obojí.

Superskalární CPU, $m = 3$

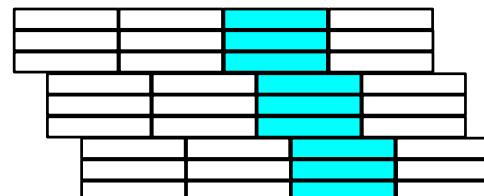


$$t_1 = (\tau + t_d) / m$$

Superřetězená CPU, $n = 3$



$$t_1 = [\tau / n + t_d]$$



Superskalární i superřetězená CPU, $m = n = 3$

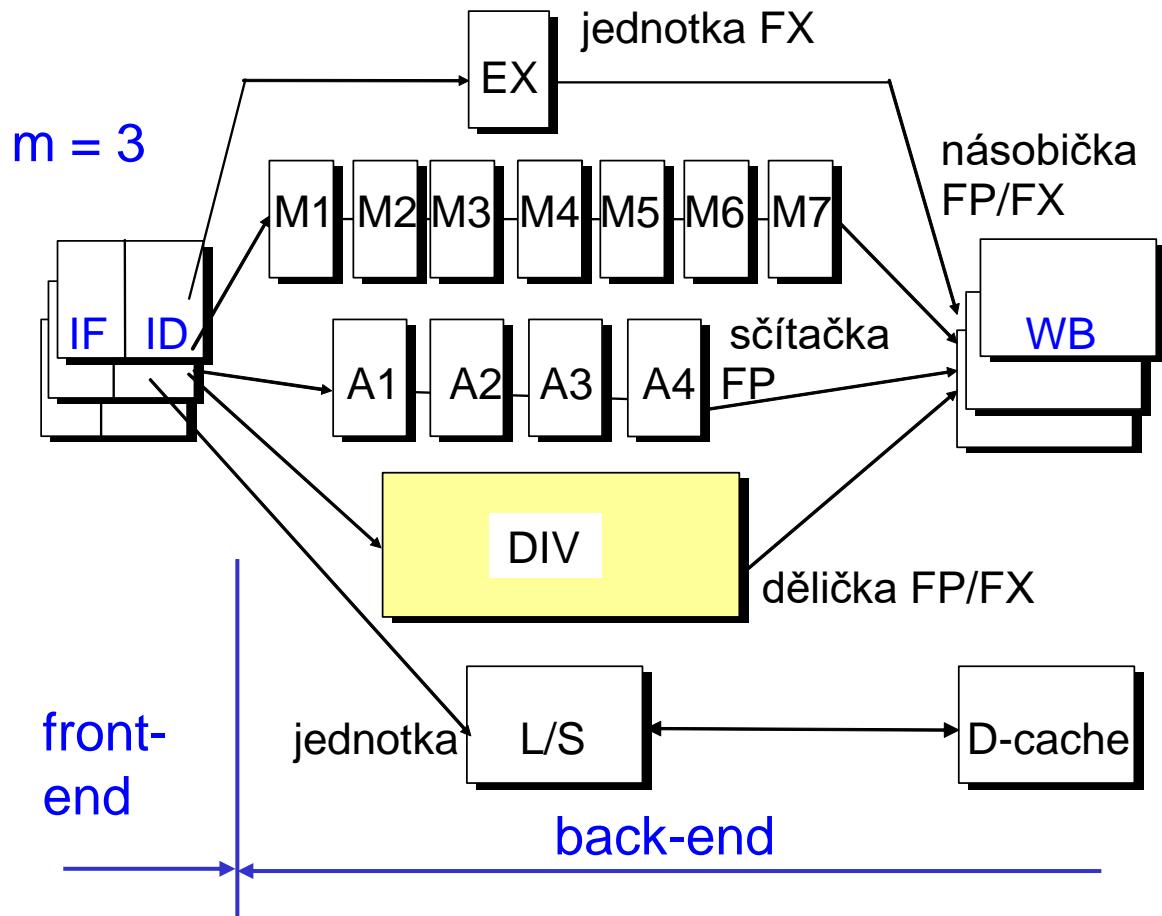
$$t_1 = [\tau / n + t_d] / m$$

- **Front-end** odpovídá stupňům **IF** a **ID**:
 - Načítá a dekóduje několik instrukcí najednou, počet se mění dynamicky.
 - *m*-cestný superskalár vydává až *m* instrukcí do funkčních jednotek v jednom taktu.
- **Back-end** odpovídá stupňům **EX**, **MA**, **WB**:
 - Provádí a ukládá výsledky několika instrukcí souběžně.
 - Některé stupně jsou rozděleny na pod stupně.
 - Počet (skupin) funkčních jednotek je minimálně *m*, ale počet propojovacích cest roste jako *m*².

Dnes *m* nejvýš 6 až 9, ale důležité je **IPC**: kolik instrukcí průměrně končí za 1 takt a to je o dost méně než *m*.

- **Dělení dle způsobu jakým instrukce opouštějí front-end:**
 - Podle pořadí v programu, po vyřešení konfliktů (in-order, INO), jednoduchý HW.
 - Mimo pořadí (out-of-order, OOO). Nepravé konflikty vyřešeny přejmenováním v HW, RAW řešeny čekáním rozpracovaných instrukcí. Zápis výsledků v původním pořadí zajištěn seřazovací pamětí (ROB).
- **Příklady:**
 - **INO**: první superskalární procesory (Pentium, DEC Alpha 21164), ale i nejnovější (IBM Power6, Intel Atom, ARM nebo Intel Xeon Phi).
 - **OOO**: Intel P6, Pentium4, Intel Core, ..., až Rocket Lake či Zen

I Příklad skalárního in-order procesoru



- Paralelní řetězené linky
- Jednotlivé stupně mohou mít různé zpoždění nebo propustnost instrukcí za takt
- Front end opouští až m instrukcí v jednom taktu
- Samostatná funkční jednotka L/S místo stupně MA

- **Paralelní řetězené linky** (INO i OOO)
 - Časový i prostorový parallelismus (paralelní načítání, dekódování, vydávání instrukcí do FJ, jejich paralelní provádění a dokončování).
- **Přejmenování registrů v HW** (OOO)
 - Odstraní konflikty WAR a WAW
- **Dynamické plánování instrukcí out-of-order** (OOO)
 - Po dekódování čekají instrukce na své operandy, které se tvoří. Jakmile jsou operandy připraveny, spustí se operace.
 - Instrukce, včetně přístupů do paměti, jsou zpracovávány v jiném pořadí oproti pořadí v programu (OOO).
- **Seřazovací paměť** (OOO)
 - Stupeň WB pomocí ní zajišťuje ukládání výsledků v pořadí určeném zdrojovým kódem.
- **Spekulativní zpracování instrukcí** (OOO)
 - Spekulace, že skok dopadne podle predikce nebo že dopředu načtená data se již nezmění.

	max. vydaných instr./takt
• Intel Pentium Pro, Pentium II, III (P6)	2/3
• Pentium 4 (NetBurst)	3/6
• Compaq Alpha EV7	6
• AMD Opteron	9
• IBM PowerPC 7400 (G4)	6
• HP PA 8700	4
• MIPS R14000	4
• Intel Core	4/5
• Intel Nehalem	5/6
• Intel Sandy Bridge – Cascade Lake	6/6
• AMD Zen	4/6FX, 4FP
• ARM Cortex A15	3
• Apple M1	8

dekódovaných
instrukcí x86
(CISC) za takt

6
9
6
4
4

vydaných
μoperací
(RISC) za
takt

http://www.agner.org/optimize/instruction_tables.pdf

DYNAMICKÉ PLÁNOVÁNÍ INSTRUKCÍ

Instrukce jsou vydávány do FJ a prováděny **mimo pořadí** v programu, pokud mezi nimi **nejsou konflikty** a **FJ** jsou **volné**.

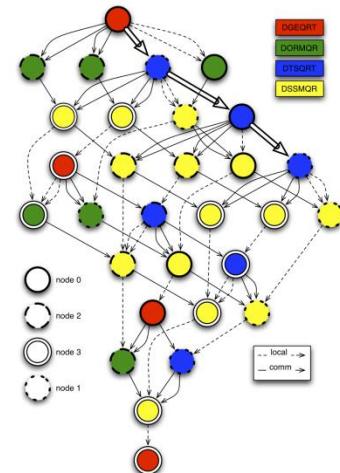
1. ScoreBoarding (Thorntonův algoritmus, 1964)

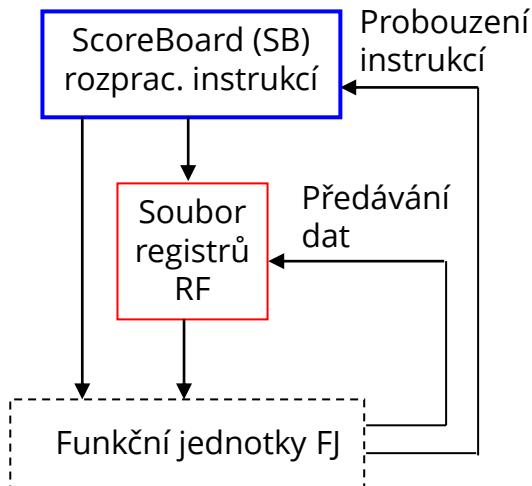
- Registruje všechny **konflikty** (RAW, WAW, WAR) v **tabulce rozpracovaných instrukcí** a udržuje jejich skóre (SB).
- SB vydá instrukce dál jen když nejsou v konfliktu s ostatními instrukcemi v SB. **Přejmenování registrů neprobíhá**.

2. Rezervační stanice (Tomasulův algoritmus, 1967)

- Konflikty WAW a WAR se řeší přejmenováním
- **Rezervační stanice RS** (bufery) umožňují odložit čekající instrukce a pracovat dopředu na dalších – tím řeší RAW.
- Rezervační stanice centrální (instruction window) nebo individuální u FJ či skupinové pro skupiny FJ.

http://users.utcluj.ro/~sebestyen/_Word_docs/Cursuri/SSC_course_5_Scoreboard_ex.pdf





Formát jedné položky **ScoreBoard (SB)**:

- **stav** instrukce (vydána do FJ, operandy načteny, hotová)
- funkční jednotka **FJ busy?**
- **operace FJ**
- **dst** (**adresa** cílového registru)
- **src1** (**adresa** zdrojového reg. 1)
- bit **V1** (operand 1 platný?)
- **src2** (**adresa** zdrojového reg. 2)
- bit **V2** (operand 2 platný?)

Formát **registrů v RF(dst)**:

Rezervační bit **V** | value

0 – neplatný (**rezervovaný**)

1 – platný (někdo, ale ještě může potřebovat)

Valid bit V1 a V2

0 – neplatný **nebo použitý**

1 – platný, ale ještě nepoužitý

- **Vydání nové instrukce – Fáze ID**
 - **Rezervace cílového registru v RF** (Serialize zápisů – WAW)
 - Registr **platný (V = 1)**, zapíše V = 0 a vloží položku do SB.
Zde neřeším, že hodnotu ještě někdo může potřebovat.
 - Registr již **neplatný (V = 0)** – nějaká instrukce do něj právě generuje výsledek – **čekáme** až dokončí předchozí instrukce.
 - **Vydání instrukce** (Eliminace RAW)
 - Jsou-li oba zdrojové operandy **platné (V1 a V2 = 1)**,
 - odešli op-kód instrukce do FJ
 - odešli adresy src1 a src2 do RF a odtud jejich hodnoty do FJ
 - **vynuluj příznaky V1 a V2 ve SB a změň stav instrukce (operandy načteny)**
 - Pokud není některý operand platný, **čekej**
- **Vykonání instrukce – Fáze EX**
 - Výsledek a adresa dst registru se objeví na výstupu FJ

- **Zápis výsledků do registru – Fáze WB** (Eliminace WAR)
 - Dokud se **shoduje dst** výsledku se **src1** nebo **src2** v nějaké instrukci v SB s bitem **V1** nebo **V2 = 1** („platný a ještě nepoužitý“)
 - musí **se zápis výsledku do registru RF(dst)** pozdržet i když je registr pro tento výsledek již rezervován ($V = 0$).
 - stávající obsah RF(dst) totiž ještě nepřečetla nějaká čekající instrukce.
 - Jakmile jsou relevantní byty **V1** a **V2** v SB **vynulovány**
 - zapíšeme **výsledek** a **V = 1** do **RF(dst)**, tedy data platná.
 - **prohledáme SB** a změníme bit **V1** nebo **V2** na 1 ve všech položkách SB, které čekaly na výsledek ($\text{src1} \mid \text{src2} = \text{dst}$)
- Jakmile je instrukce dokončena, její záznam je smazán z tabulky score.

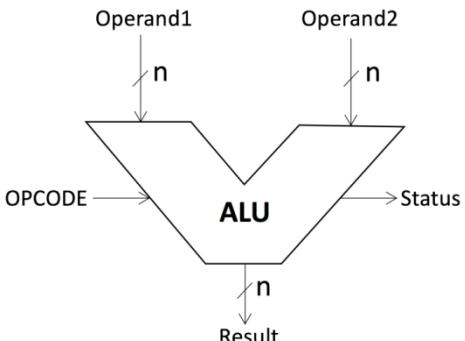
Závěr: konflikty **RAW**, **WAR** a **WAW** se u této varianty řeší čekáním.

ScoreBoarding – příklad

Registry

Name	Valid	Value
R1	0	?
R2	1	50
R3	1	30
R4	1	40
R5		
R6		

op1 **r1**, r2, r3
 op2 r2, **r1**, r4
 op3 r6, **r3**, **r1**
op4 **r1**, **r2**, **r3**
 op5 r7, r8, **r1**
 op6 **r1**, r5, r4



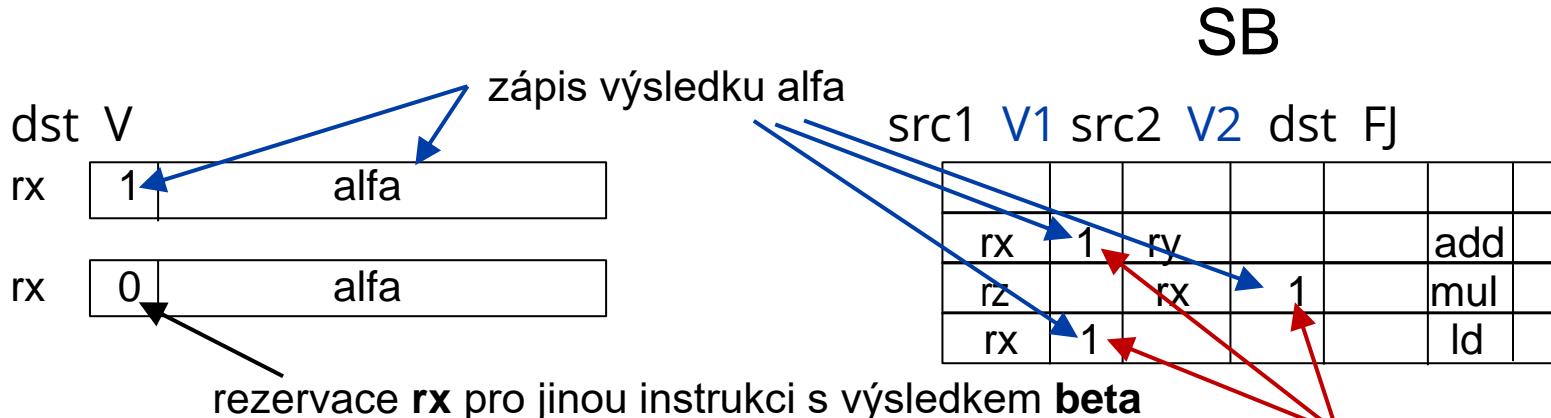
Score Board

Dst	S1	V1	S2	V2

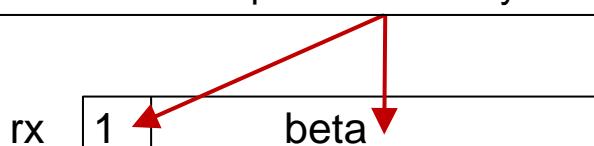
I Algoritmus ScoreBoarding (SB) graficky

ScoreBoard – bez přejmenování registrů: je třeba signalizovat

- okamžik čtení src operandů čekajícími instrukcemi opx
 $0 \rightarrow SB(opx).V1/V2$ pro WAR
- a zápisu cílového registru $1 \rightarrow RF(dst).V$ pro WAW.

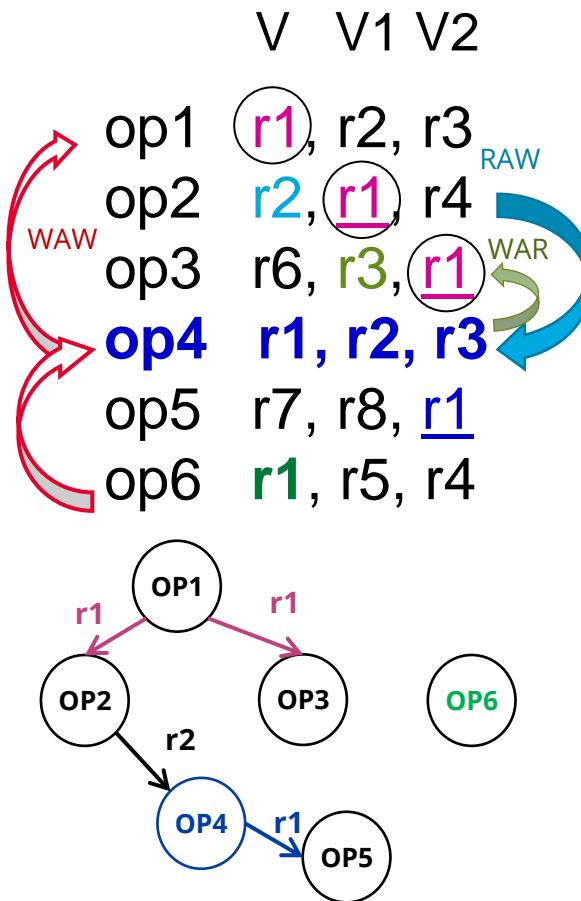


výsledek beta lze zapsat do rx až bylo alfa přečteno všemi čekajícími instrukcemi ($1 \rightarrow 0$ ve V1/V2)



nelze rezervovat pro gama, dokud je V = 0,
tj. dokud je rx rezervován pro beta

I Příklad 1: ScoreBoarding



Rezervace r1 pro op4:

- když $RF(r1).V = 1$, rezervuj r1 pro op4 zápisem $0 \rightarrow RF(r1).V$, tj. tvoří se nový obsah r1, ale starý r1 platí!

Rezervace r1 pro op6:

- když bit $RF(r1).V = 0$, čekej až bude tento bit 1 (eliminace WAW).

Čtení operandů r2, r3 a instrukce op4 do FJ: Když

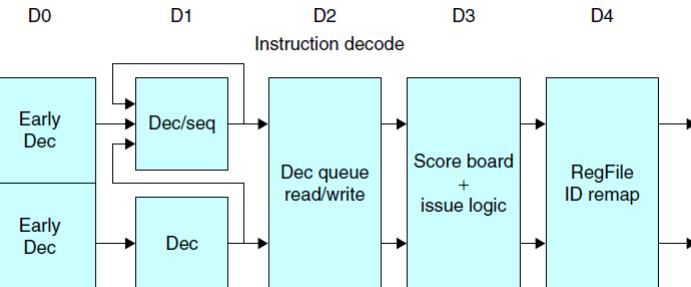
- $SB(op4).V1 = 1 \& SB(op4).V2=1$ a $FJ(op4) = \text{free}$, pošli $r2, r3 \rightarrow RF$, $[RF(r2).\text{value}, RF(r3).\text{value}, op4] \rightarrow FJ$, (elim. RAW),
- nastav $0 \rightarrow SB(op4).V1$ a $0 \rightarrow SB(op4).V2$ (tj. r2 a r3 přečteno).

Zápis výsledku op4 do r1:

- čekej, až op2 a op3 přečtou r1 vytvořený op1, tj. až bude $SB(op2).V1 = 0 \& SB(op3).V2 = 0$ (eliminace WAR).
- pak zapiš výsledek op4 do $RF(r1).\text{value}$ a nastav $1 \rightarrow RF(r1).V$ a také $1 \rightarrow SB(op5).V2$ (platný a ještě nepřečtený).

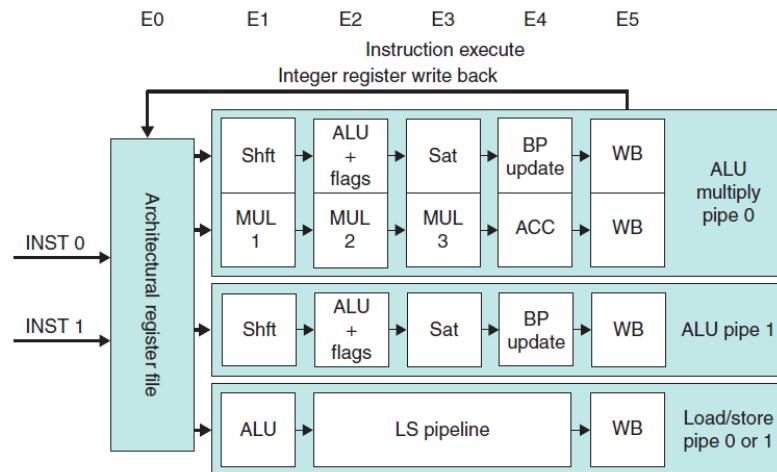
- Dekódovací a vydávací jednotka**

- Dekódují se až 2 instrukce za takt
- Pokud žádná z nich není skok, PC se posune o dvě instrukce dále
- ScoreBoard logika se stará o plánování instrukcí



- Exekuční jednotka**

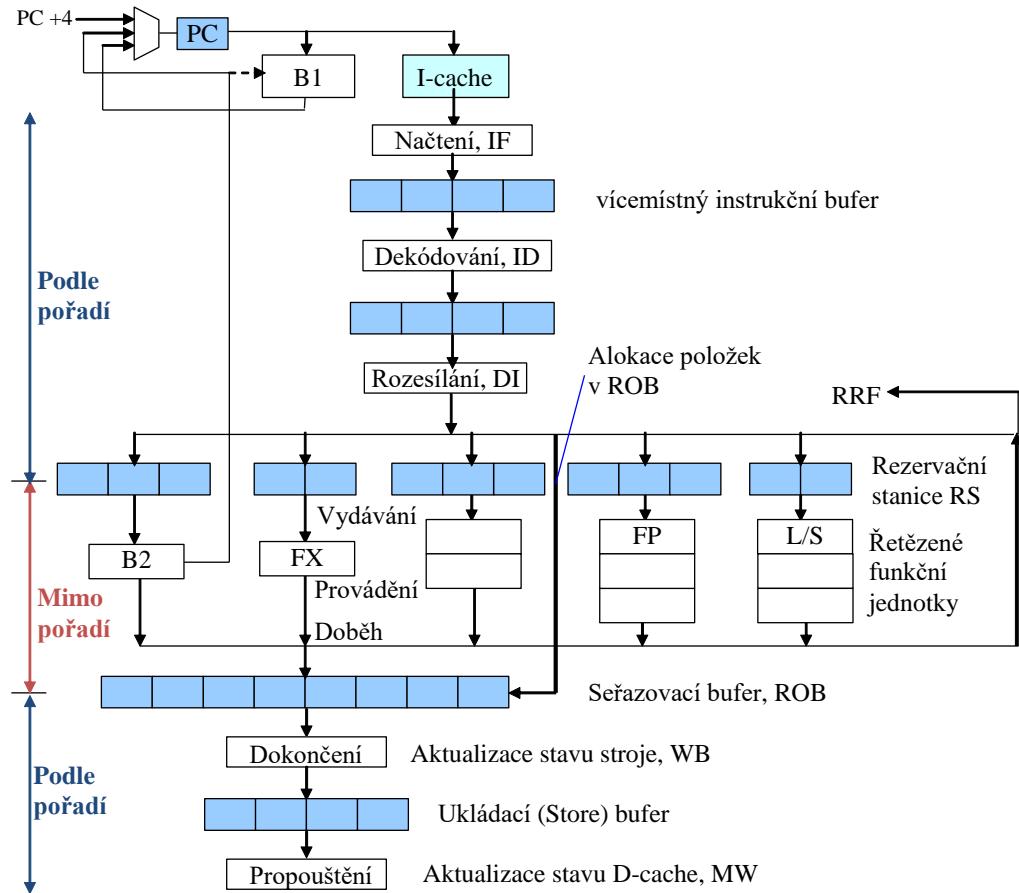
- Pokud nedokáže pomocí forwarding budou se závislé instrukce vykonávat sériově
- Forwarding je implementován mezi všemi linkami
- Pokud dojde k závislosti jsou buď obě nebo ta druhá pozastaveny



TOMASULŮV ALGORITMUS REZERVAČNÍ STANICE

PC	programový čítač, instrukční pointer
FX	integer funkční jednotka, integer instrukce
FP	jednotka/instrukce s plovoucí desetinnou čárkou
L/S	jednotka/instrukce přístupu do paměti load/store
B	jednotka zpracování skoků, má 2 části: B1 a B2
RF	soubor registrů (ARF architekturních v instrukčním souboru, RRF přejmenovaných – rename reg. file)
RS	rezervační stanice
IF	načtení skupiny instrukcí, predikce skoků
ID	dekódování instrukcí a přejmenování registrů
DI	rozeslání instrukcí (Dispatch) do RS a ROB
EX	vydání instrukcí do FJ a jejich provedení
WB	aktualizace ARF podle pořadí v ROB
MW	aktualizace stavu D-cache (Memory Write)

Generická OOO superskalární architektura



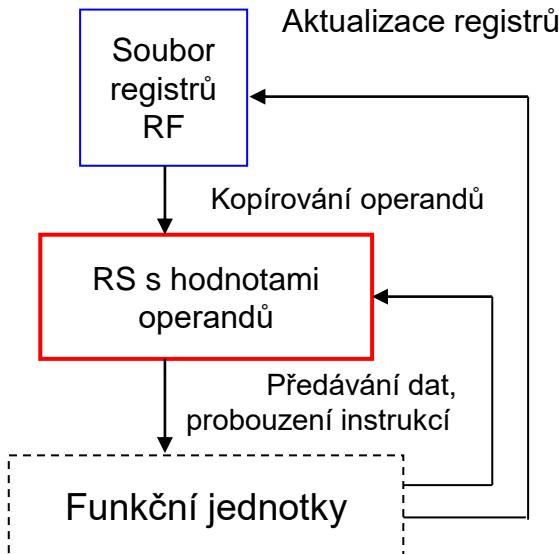
B, Branch unit,
jednotka pro zpracování skoků
B1 – spekulativní, B2 – reálné

RRF – registry pro
přejmenování (cca 256)

RS – zde individuální rezervační
stanice

Předávání výsledků do RS a RF
přes **CDB** (společná datová
sběrnice)

- Intel **P6**: centrální RS s 5 brána (5 skupinami FJ), **20** položek; obsahuje μops s kopími operandů.
- Intel **Pentium 4** (NetBurst): 5 skupinových RS, každá s frontou 8–12 položek, obsluhují 5 skupin FJ: (MEM, ALU0, ALU1, FP/move, FP/MMX/SSE)
- Intel **Core i7** (Nehalem): centrální RS, **36** položek
- Intel **Sandy Bridge**: centrální RS, **54** položek, 6 bran
- Intel **Haswell**: centrální RS, **60** položek, 8 bran
- Intel **Cascade Lake** : centrální RS, **97** položek, 8 bran
- **Opteron** (AMD, 64 bitová CPU kompatibilní s IA-32): ALU + AGU: 3 krát skupinová RS s 8 položkami, FP operace: skupinová RS s 36 položkami. Celkem **60** položek v RS
- AMD **Ryzen**: Skupinová RS, **6 x 14** položek, 6 bran

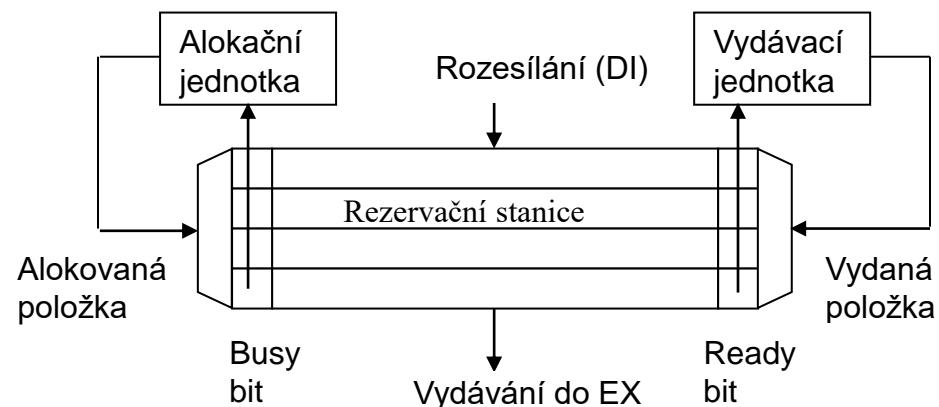


Formát registrů v RF(dst):

RF(dst) **V | tag | hodnota**

Formát instrukcí v RS(i):

- busy bit // Je položka v tabulce volná
- operace
- src1 (hodnota, tag1, valid bit V1)
- src2 (hodnota, tag2, valid bit V2)
- dst (adresa, tag)
- ready bit // Je instrukce připravena



Cílový operand

- Dynamicky vytvoříme **příznak** (přejmenujeme)
 - Zapíšeme do RS (**tag**)
 - Zapíšeme do RF (**tag**), lze přepsat i předchozí příznak
- Zatím neplatný obsah RF(dst) se označí zápisem $V = 0$.
- Jen naposled přejmenovaný dst reg bude mít kopie v RS i RF.
- Dříve přejmenované instance téhož registru existují jen v RS.

Zdrojové operandy

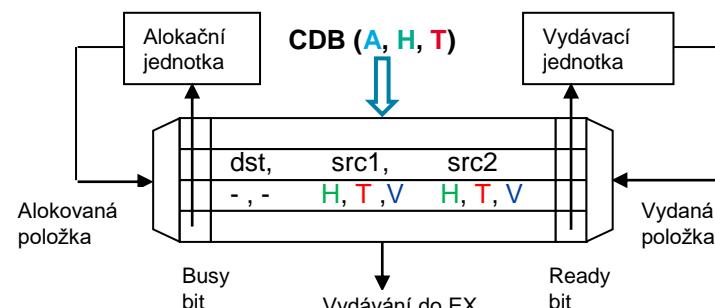
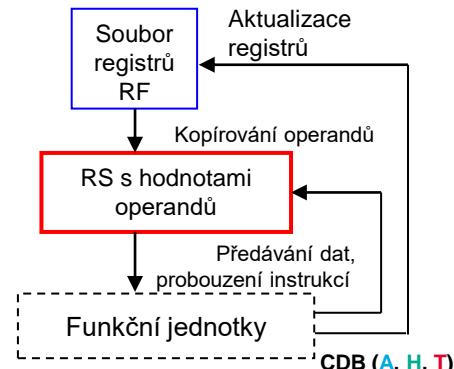
- platné** hodnoty ($V = 1$) zdrojových operandů z RF včetně tagu se načtou do RS současně s instrukcí; $V1/V2 = 1$.
- zatím ještě **neplatné** operandy ($V = 0$): načte se pouze tag s $V1/V2 = 0$.

Formát položky RS:

- busy bit
- operace
- zdrojový operand1
(hodnota, tag1, valid bit V1)
- zdrojový operand2
(hodnota, tag2, valid bit V2)
- cílový operand
(dst reg., **tag** – příznak)
- ready bit

RF(dst): **V | tag | hodnota**

- Pokud src operandy instrukce v RS ready**
 $(V1 \& V2 = 1)$ & FJ volná
 - Odešle se instrukce z RS (včetně dat, dst addr, tag) do FJ
 - Jinak čekej až budou ready (eliminace RAW)
- FJ vytvoří výsledek**
 - Rozhlásí hodnotu výsledku na CDB (pokud je volný) spolu s adresou dst i tagem dst registru
- RS monitoruje CDB**
 - Instrukce čekající v RS monitorují CDB, porovnávají tagy operandů s tagem na CDB
 - V případě shody zachytí hodnotu z CDB do příslušného políčka RS, případně do store buferu (instrukce store) (eliminace WAR, WAW);
- RF monitoruje CDB**
 - Při shodě tagů na CDB i v reg. RF(dst) se hodnota zapíše paralelně i do reg. RF(dst) a nastaví se v něm $V = 1$.
 - Při neshodě tagů (tj. RF(dst) byl již znova přejmenován) se hodnota v RF(dst) vůbec neobjeví, je zachycena pouze v polích RS, které tak slouží jako tagem přejmenované registry.



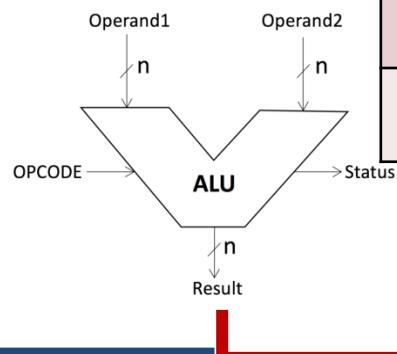
Registry

Name	Valid	Tag	Value
R1			
R2			
R3			
R4			
R5			
R6			



Rezervační stanice

DST	Tag	V	Tag	Val	V	Tag	Val



CDB



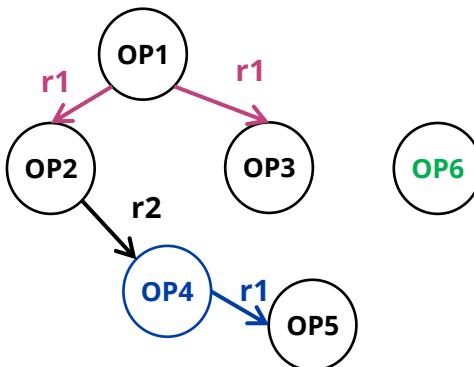
I Příklad 2: Tomasulo – algoritmus

Rezervační stanice – přejmenování registrů příznakem (tag) odstraní nepravé konflikty

i1	r1, r2, r3
i2	r2, <u>r1</u> , r4
i3	r6, r3, <u>r1</u>
i4	r1, r2, r3
i5	r7, r8, <u>r1</u>
i6	r1, r5, r4

tag1 → RS(i1), tag1 → r1
tag2 → RS(i2), tag2 → r2
tag3
tag4 → RS(i4), tag4 přepíše tag1 v r1
tag5...
tag6 → RS(i6), tag6 přepíše tag4 v r1

i1	t1 , r2, r3
i2	t2, <u>t1</u> , r4
i3	t3, r3, <u>t1</u>
i4	t4, t2, r3
i5	t5, r8, <u>t4</u>
i6	t6 , r5, r4



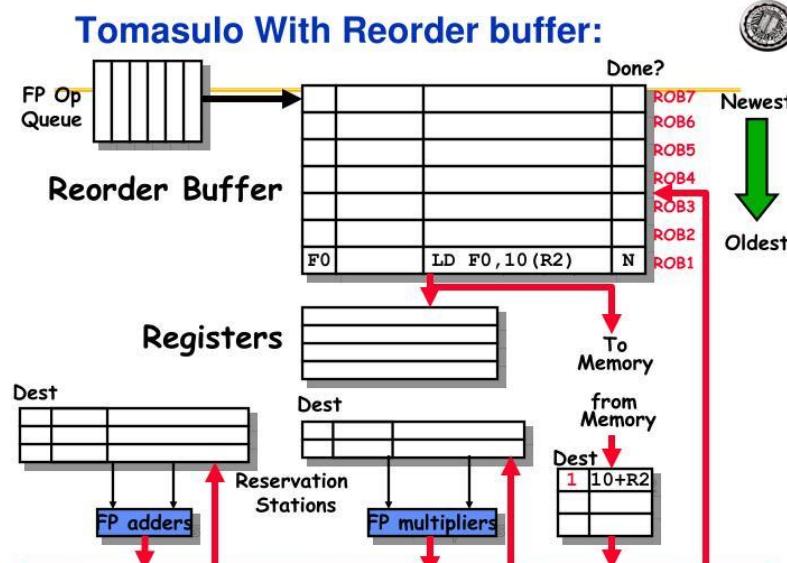
- Do RS(i) se načítá tag cílového reg. a operandy nebo tagy zdrojových reg.
- Instance registru r1 existují v src políčkách v RS označených tagem1, tagem4 a tagem6,
- V RF existuje jen instance označená tagem6 (poslední přejmenování).

- Instrukční závislosti **RAW** se detekují a řeší čekáním v RS
 - u skalárních CPU to bylo čekání/storno instrukce po zjištění problému během fáze ID,
 - podobně u superskalárních INO CPU).
- **Instrukce způsobující konflikty WAR a WAW:**
 - registry viditelné programátorovi (architekturní, ARF) jsou mapovány (**přejmenovány příznakem**) na položky RS (15–60 celkem).
 - Konflikty WAR a WAW je ale lépe řešit explicitním přejmenováním (viz příští přednáška).
- U superskalárních procesorů INO (in-order) přejmenování registrů neprobíhá.

SEŘAZOVACÍ PAMĚТЬ REORDER (RETIRE) BUFFER

Seřazovací paměť (Reorder Buffer, ROB)

- ROB je kruhová vyrovnávací paměť **rozpracovaných instrukcí obsahující**
 - informace o stavu instrukce
 - předběžné/spekulativní výsledky.
- Instrukce jsou vloženy do ROB při vydání do RS
- Všechny rozpracované instrukce jsou zde udržovány ve frontě FIFO v programovém pořadí.
- **Propouštění instrukcí pouze z čela ROB**, po propuštění všech předchozích instrukcí.
- **Zápis do arch registrů/paměti při OPUŠTĚNÍ ROB**
- ROB může být použit i pro přejmenování.
- **Současné procesory:** Intel Cascade Lake: 224 položek, AMD ZEN2 : 224 položek Retire Queue.



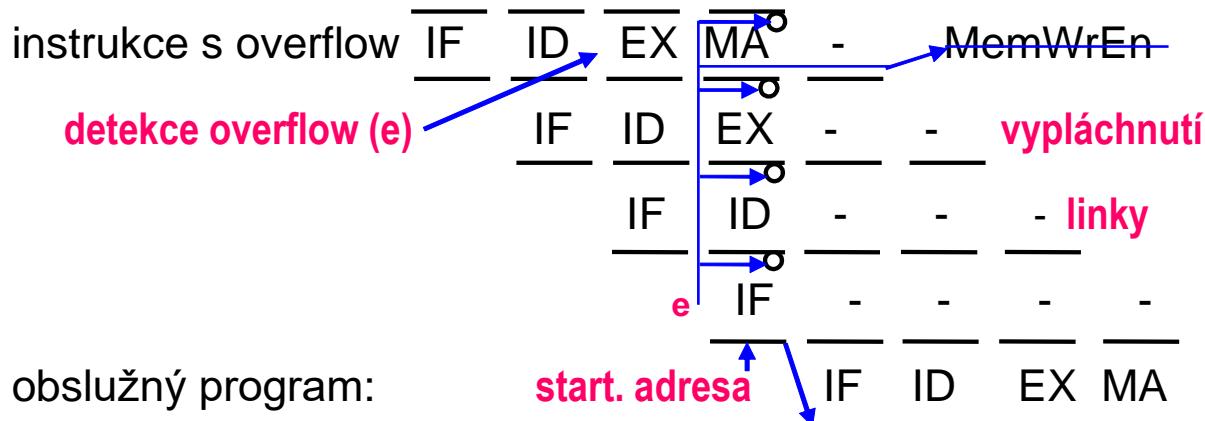
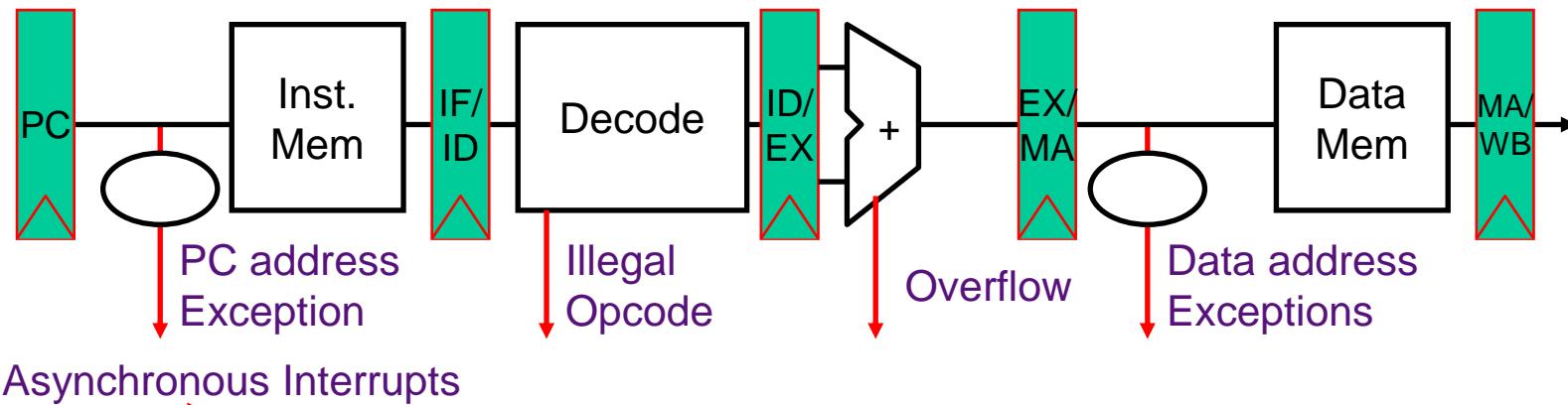
- **Typ instrukce** (aritmetická, L/S, branch,...)
- **Destinace** – adresa arch. registru pro uložení výsledku.
- **Flag** – stav instrukce
 - binární flag ukazuje jestli instrukce doběhla ve FJ (done),
 - vícebitový flag může udávat i jiný stav (dispatched v RS, exec, ready)
- **Hodnota** spočtená instrukcí, zatím ale **nezávazná**
 - tato pole představují sadu registrů RenameRF (jedna možnost přejmenování).
 - závazná hodnota je v arch. registru.
- Při vydávání instrukcí je pro každou instrukci podle jejího pořadí rezervována první volná položka na konci aktivního úseku ROB.
- ROB poskytuje historii více přejmenování téhož registru: nejstarší verze registru již může být v ARF, předchozí uvnitř fronty v ROB a nejmladší na jejím konci.

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)

- 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.

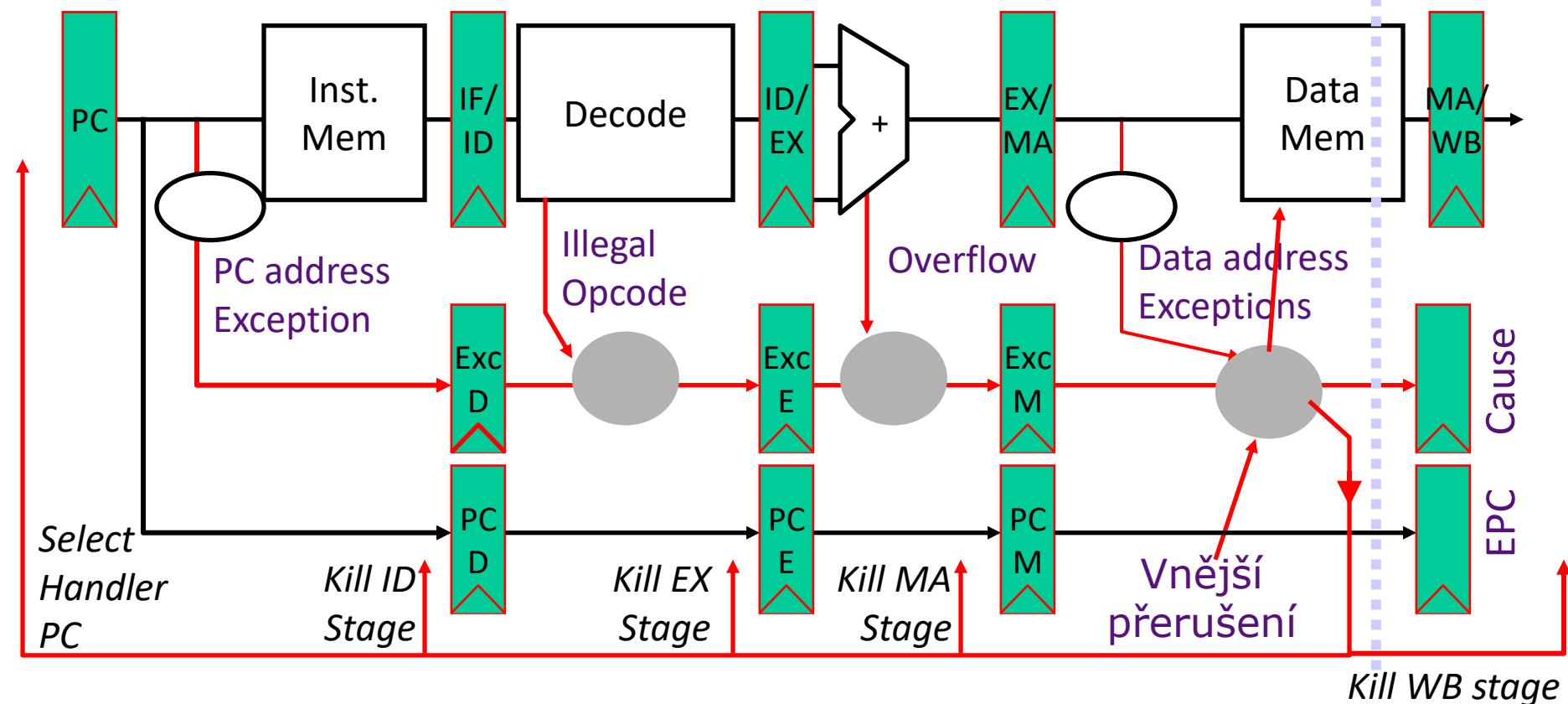
I Zpracování výjimek ve skalární 5 stupňové lince



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

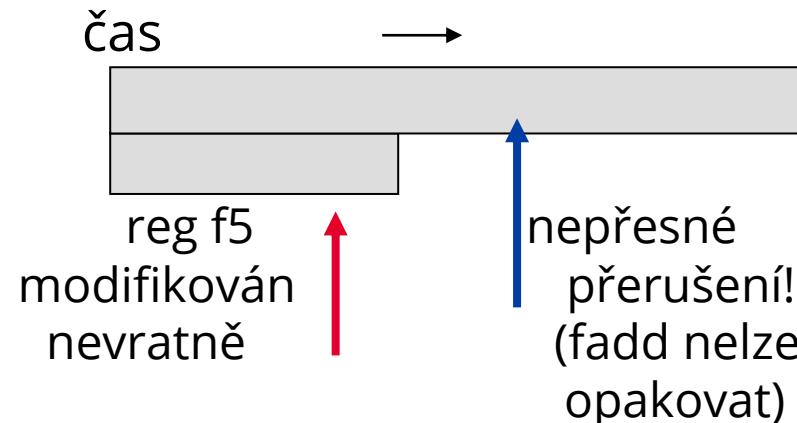
I Zpracování výjimek ve skalární 5 stupňové lince

Commit Point



Paralelně prováděné operace mohou skončit v jiném pořadí než v programu:

i-1		
i	fdiv	f2, f1, f0
i+1	fadd	f5, f4, f5
i+2		



Odstanění: Výsledek operace je třeba uchovat jako prozatímní v dočasně paměti a přepsat do cílového architekturního registru (f5) až po skončení všech předchozích instrukcí – **Propouštění z čela ROB.**

- Ošetření přerušení v ROB
 - Instrukce jsou propouštěny v programovém pořadí, přerušení je poznačeno u instrukce v ROB **speciálním bitem**.
 - CPU vyřizuje přerušení jen od instrukce na čele fronty v ROB.
 - Následná instrukce ještě nezapsala prozatímní výsledek z ROB do ARF a dá se opakovat po ošetření přerušení, tedy jde o **přesné přerušení**.
- Některé procesory dovolují práci ve vybraném režimu:
 - Pomalejší zpracování s přesnými přerušeními (ladění).
 - Rychlé zpracování s nepřesnými přerušeními.

FP násobení 7 taktů, FP sčítání 4 takty, L/S 2 takty.

- Fáze IF, ID nejsou v diagramu níže zobrazeny. Začíná se rozesíláním instrukcí do RS (fáze DI).
- Výsledek poslední fáze EX na CDB podržen, poslední takt EX zvýrazněn.

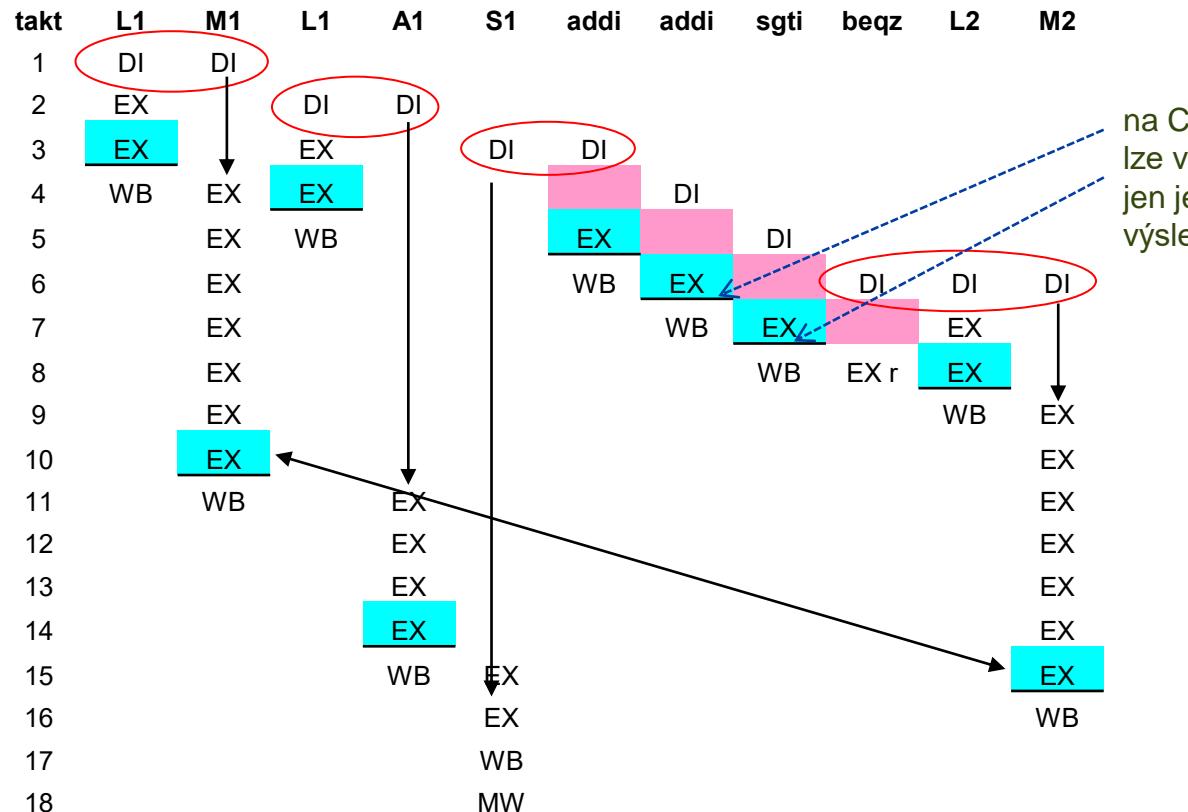
• **V diagramu je vidět (ověrte sami):**

- rozesílání instrukcí do RS po skupinách 1 až 3,
- čekání instrukcí v RS,
- provádění instrukcí OOO (dokonce L2 předbíhá S1),
- výsledek na CDB (na konci EX),
- čekání instrukcí na volný CDB (strukturní konflikt),
- překrytí první a druhé iterace (tj. dynamické rozbalování smyčky),
- rozestup prvních mezivýsledků (i výsledků) 5 taktů.

$$y[i] = \text{alpha} * x[i] + y[i]$$

loop:	ld	f2, 0(r1)
	fmul	f4, f2, f0
	ld	f6, 0(r2)
	fadd	f6, f4, f6
	sd	0(r2), f6
	addi	r1, r1, #8
	addi	r2, r2, #8
	sgti	r3, r1, done
	beqz	r3, loop

Příklad 3: Řešení



$$y[i] = \text{alpha} * x[i] + y[i]$$

loop:	ld	f2, 0(r1)
	fmul	f4, f2, f0
	ld	f6, 0(r2)
	fadd	f6, f4, f6
	sd	0(r2), f6
	addi	r1, r1, #8
	addi	r2, r2, #8
	sgti	r3, r1, done
	beqz	r3, loop

na CDB
lze vložit
jen jeden
výsledek

3 cestný procesor

- FP násobení 7 taktů,
- FP sčítání 4 takty,
- L/S 2 takty.

OPTIMALIZACE TOKU DAT PŘES REGISTRY

- **Přejmenování registrů** v jednoduché formě již v rezervačních stanicích, ale
 - počet položek RS omezen
 - počet přejmenování registrů za 1 takt omezen.
- Při zpracování 4 a více instrukcí/takt muže být potřeba přejmenovat několik registrů v jednom taktu, případně jeden a tentýž registr vícekrát v jednom taktu:

mul r3, r0, **r2** ... 4 instrukce postupující současně

add **r2**, r1, r4

sw **r2**, 0(r5) **r2, r2 ; r2, r2** ... WAR

sub **r2**, r3, r1 **r2, r2** ... WAW

- Stačí přejmenovat 2x cílový registr r2 v 1 taktu:
r2 → r6, r2 → r7.

- **Přejmenování pomocí registrů v ROB:**
 - **1 i více výsledků** dokončených instrukcí z čela ROB se přesunuje v každém taktu najednou do ARF (souboru architekturních, tj. viditelných registrů).
 - Intel P6, Pentium M: 40 položek, Nehalem 128 položek
- **Přejmenování v HW** pomocí sady (128 a více) registrů pro přejmenování (**Rename Register File RRF**) s více branami
 - RRF obsahuje potvrzené i spekulativní výsledky.
 - Soubor registrů pro přejmenování RRF může být:
 - oddělen od architekturních (ARF)
 - monolitický, integrován s ARF (Intel Pentium 4, Sandy Bridge)

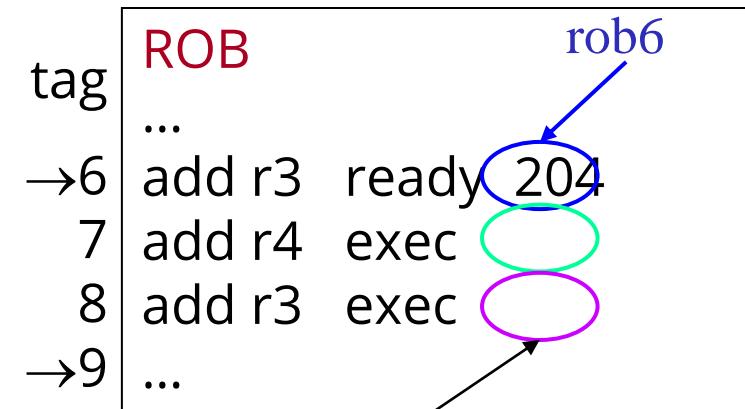
- Pro novou instrukci se **vytvoří položka na konci ROB** {**typ instrukce, dst reg., flag, hodnota vypočtená instrukcí**}.
- Do RAT (Register Alias Table) se zapíše (podle pořadí v programu poslední) přejmenování
arch.reg. → tag = adresa ROB.
- Tabulka přejmenovaných registrů RAT udává pro každý arch. registr adresu ROB nebo adresu registru v ARF.
- **Ready operand** se hledá nejdříve v ARF, pak v ROB(tag).
 - ARF obsahuje již potvrzené výsledky.
 - ROB udržuje nepotvrzené a spekulativní výsledky.
- **Po provedení instrukce** je
 - výsledek a flag = ready zapsán do ROB
 - Přepis (WB) do ARF se provede až bude instrukce na čele ROB!

I Přejmenování registrů pomocí kruhového ROB

Před : add r3, r3, 4
 add r4, r7, r3
 add r3, r2, r7

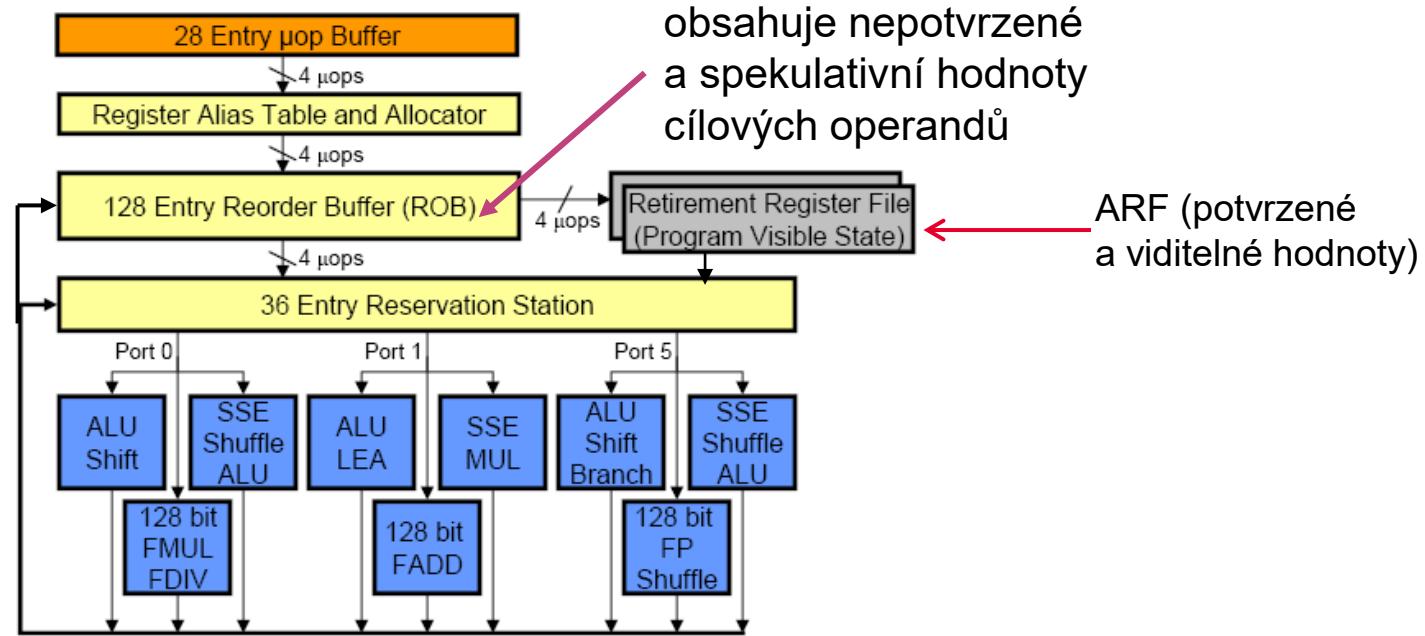
Po: add rob6, r3, 4
 add rob7, r7, rob6
 add rob8, r2, r7

RAT	poslední přejmenování
...	
r2	r2
r3	rob8
r4	rob7
...	tag



Je-li jeden dst reg. přejmenován 2x a 2 výsledky by se měly zapsat v jednom taktu do téhož registru ARF, zapíše se tam jen poslední výsledek v program. pořadí.

Intel Nehalem: RAT může přejmenovat až 4 µops v každém taktu a každé přidělí cílový registr v ROB. Přepis až 4 výsledků hotových instrukcí z čela ROB do ARF v programovém pořadí.



- **Každý fyzický registr** z Physical Register File (PRF) může být použit jako architekturní (viditelný) nebo **přejmenovaný**.
- **Při dekódování se**
 - načte **tag** z FIFO volných tagů
 - přejmenuje se dst registr ARF(x) → PRF(tag)
 - do mapovací tabulky **RAT** (Register Alias Table) se uloží nové mapování x → tag, **unready**. (tag = adresa registru v PRF)
 - Register je unready, když registr čeká na zápis výsledku.
 - Ke stejnemu ARF registru může být přiřazeno několik tagů, PRF(tag) je ale určen jednoznačně.
 - Do RAT se dá jen **poslední přejmenování**.
- **Po provedení instrukce**
 - cílový operand s tagem rozhlášen do PRF, RS a do RAT.
 - V RS se operand načte do políček operandů se shodným tagem.
 - V RAT se u tagu zapíše **ready**.

free list: r37, r38, r39

Před : add r3, r3, 4
 add r4, r7, r3
 add r3, r2, r7

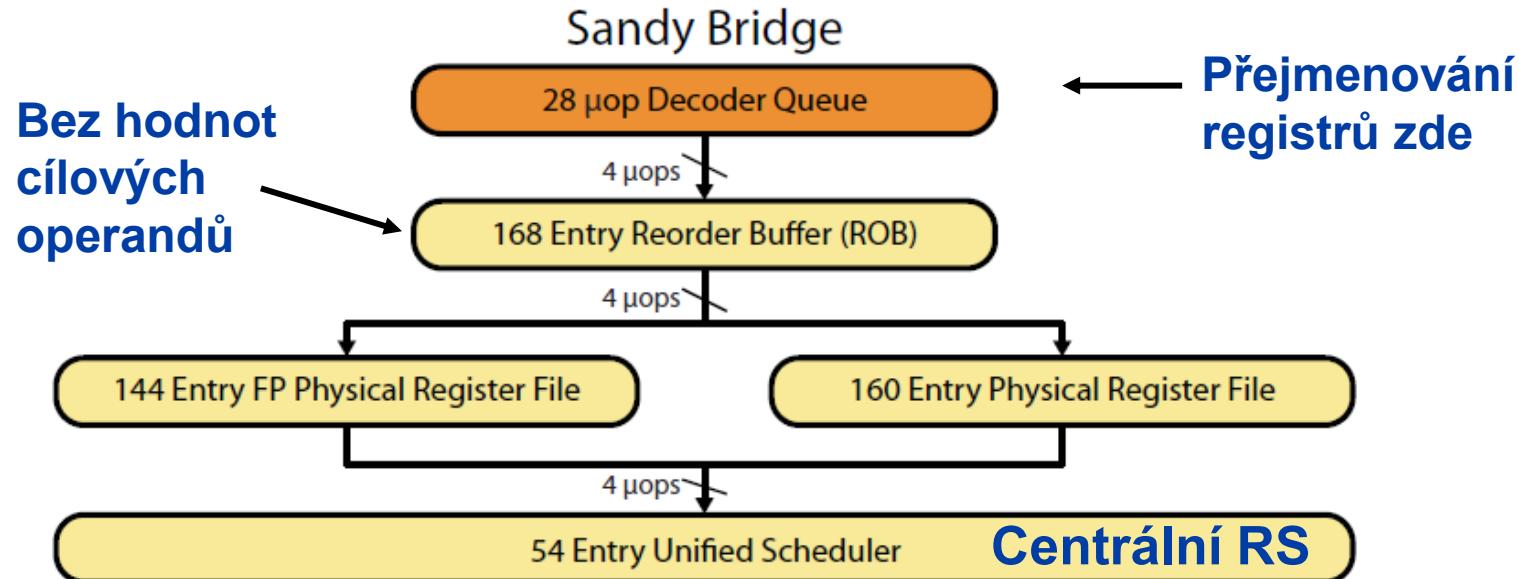
Po: add r37, r3, 4
 add r38, r7, r37
 add r39, r2, r7

přejmenován podruhé




Při **propouštění instrukce** se

- mapování dst registru uloží do další tzv. **propouštěcí RAT**, která definuje **okamžitý soubor ARF**.
- Do propouštěcí RAT se uloží jen poslední mapování v programovém pořadí ($r3 \rightarrow r39$).
- Ostatní volné tagy → FIFO ($r37 \rightarrow \text{free list}$).
- **U každého skoku** je pořízen snímek propouštěcí RAT kvůli zotavení ze špatné predikce. Při špatné predikci nebo přepnutí kontextu je třeba se vrátit k poslednímu platnému mapování v propouštěcí RAT.



- ROB **neobsahuje hodnoty** operandů. Odpadá tak kopírování výsledků z ROB do ARF při dokončování (propouštění) instrukcí. ☺
- **Write-back** aktualizuje jen stav (flag) instrukcí v ROB.

Pokračování příště

Organizace paměti cache, L/S jednotky

AVS – Architektury výpočetních systémů

Týden 3, 2024/2025

Jirka Jaroš

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



OPAKOVÁNÍ

Instrukce jsou vydávány do FJ a prováděny **mimo pořadí** v programu, pokud mezi nimi **nejsou konflikty** a **FJ** jsou **volné**.

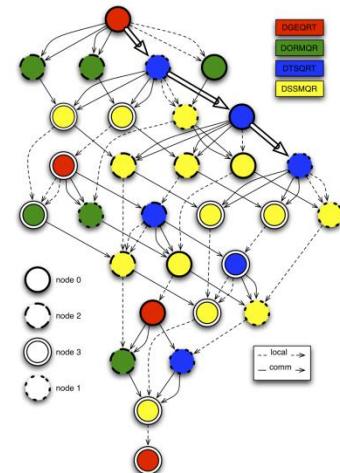
1. ScoreBoarding (Thorntonův algoritmus, 1964)

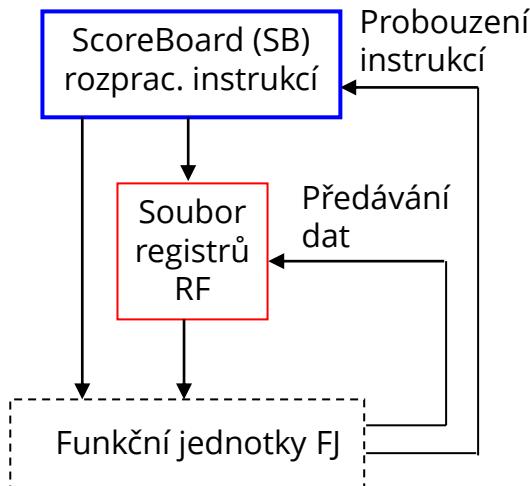
- Registruje všechny **konflikty** (RAW, WAW, WAR) v **tabulce rozpracovaných instrukcí** a udržuje jejich skóre (SB).
- SB vydá instrukce dál jen když nejsou v konfliktu s ostatními instrukcemi v SB. **Přejmenování registrů neprobíhá**.

2. Rezervační stanice (Tomasulův algoritmus, 1967)

- Konflikty WAW a WAR se řeší přejmenováním
- **Rezervační stanice RS** (bufery) umožňují odložit čekající instrukce a pracovat dopředu na dalších – tím řeší RAW.
- Rezervační stanice centrální (instruction window) nebo individuální u FJ či skupinové pro skupiny FJ.

http://users.utcluj.ro/~sebestyen/_Word_docs/Cursuri/SSC_course_5_Scoreboard_ex.pdf





Formát jedné položky **ScoreBoard (SB)**:

- **stav** instrukce (vydána do FJ, operandy načteny, hotová)
- funkční jednotka **FJ busy?**
- **operace FJ**
- **dst** (**adresa** cílového registru)
- **src1** (**adresa** zdrojového reg. 1)
- bit **V1** (operand 1 platný?)
- **src2** (**adresa** zdrojového reg. 2)
- bit **V2** (operand 2 platný?)

Formát **registrů v RF(dst)**:

Rezervační bit **V** | value

0 – neplatný (**rezervovaný**)

1 – platný (někdo, ale ještě může potřebovat)

Valid bit V1 a V2

0 – neplatný **nebo použitý**

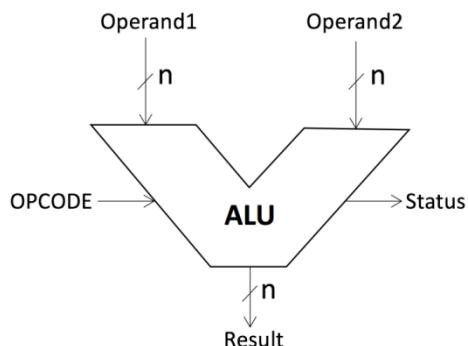
1 – platný, ale ještě nepoužitý

ScoreBoarding – příklad

Registry

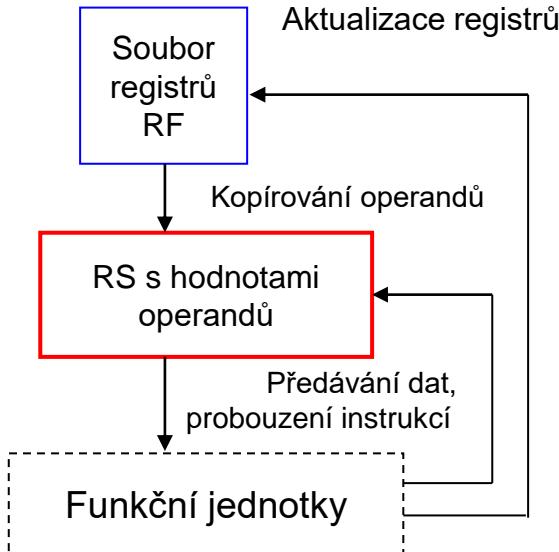
Name	Valid	Value
R1	0	100
R2	0	200
R3	1	300
R4		
R5		
R6		

op1 r1, r2, r3
op2 r2, r1, r4
op3 r6, r3, r1
op4 r1, r2, r3
op5 r7, r8, r1
op6 r1, r5, r4



ScoreBoard

Dst	S1	V1	S2	V2

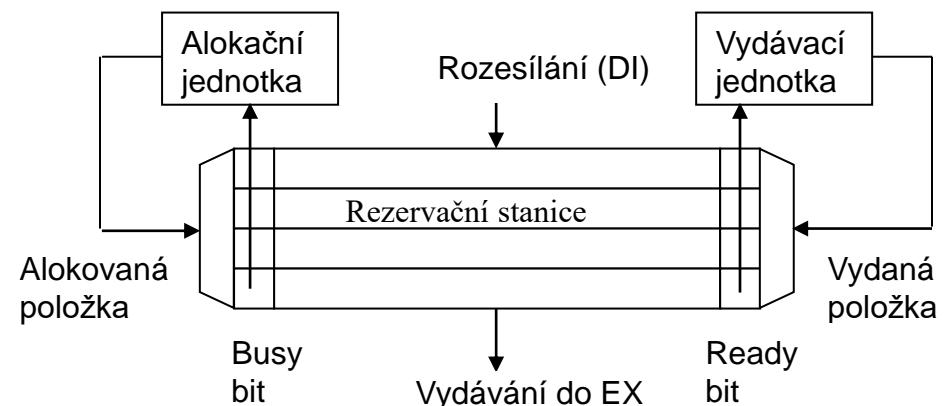


Formát registrů v RF(dst):

RF(dst) **V | tag | hodnota**

Formát instrukcí v RS(i):

- busy bit
- operace
- src1 (**hodnota**, tag1, valid bit V1)
- src2 (**hodnota**, tag2, valid bit V2)
- dst (**adresa**, tag)
- ready bit



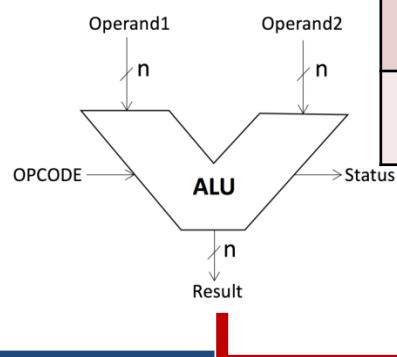
Registry

Name	Valid	Tag	Value
R1	0	T1	100
R2	0	T2	200
R3	1	T3	300
R4	1	T4	400
R5			
R6			



Rezervační stanice

DST	Tag	V	Tag	Val	V	Tag	Val

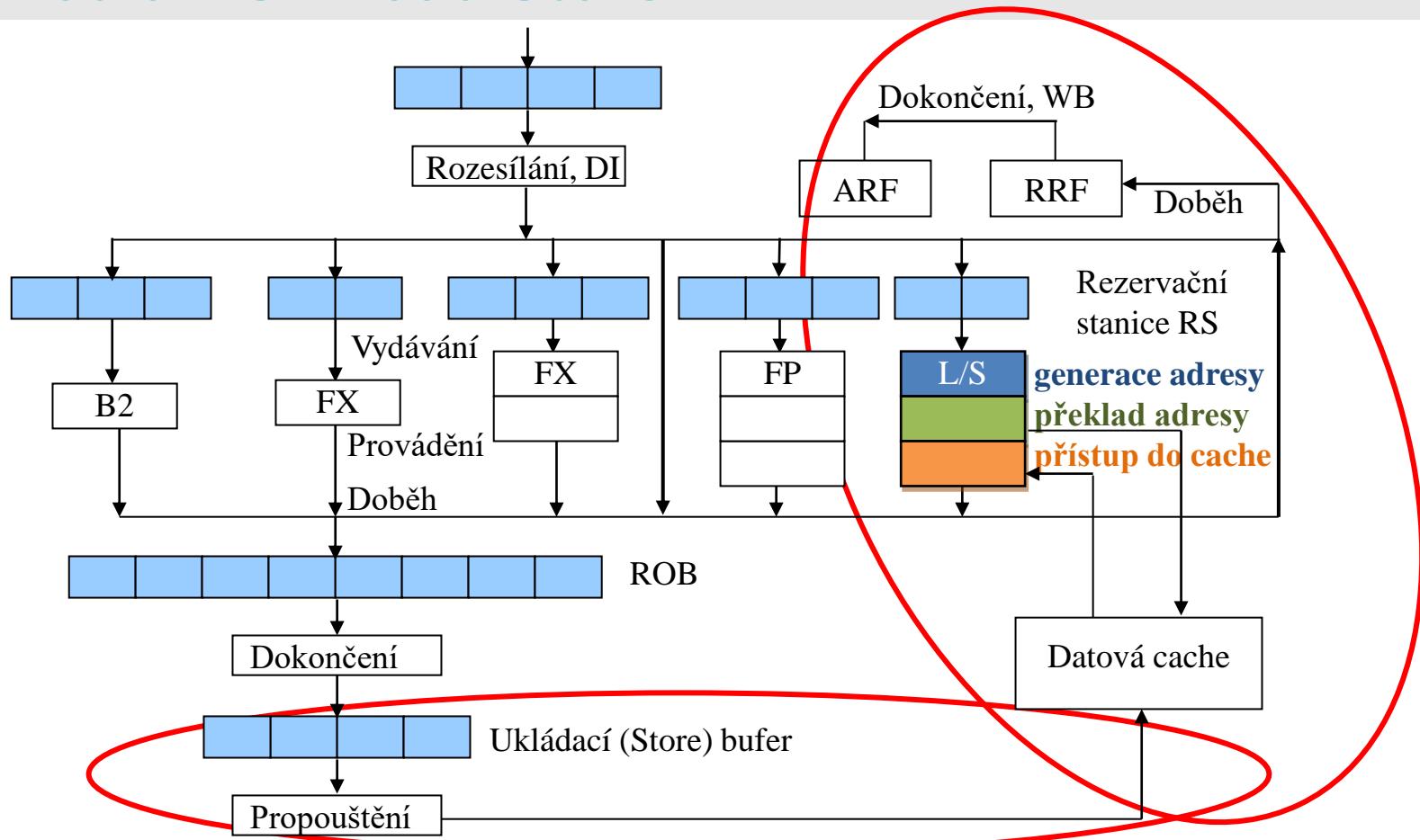


CDB

LOAD/STORE ČÁST PROCESORU

- V průměru každá 3. instrukce je L/S
 - Několik % z nich generuje výpadek v D-cache
 - Vyřízení výpadku může podle úrovně cache trvat mnoho taktů CPU
- Přistupovat do paměti v programové sekvenci s čekáním na vyřízení výpadků je neúnosné – brání rozbalování smyček.
- Výjimky je třeba zpracovávat podle původního pořadí L/S instrukcí v programu.
- Zisk v IPC překrytím paměťových operací je 35–100%.

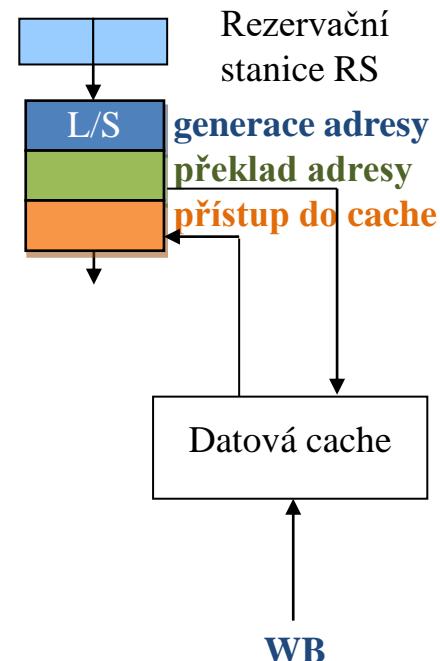
I Jednotka L/S – Load/Store



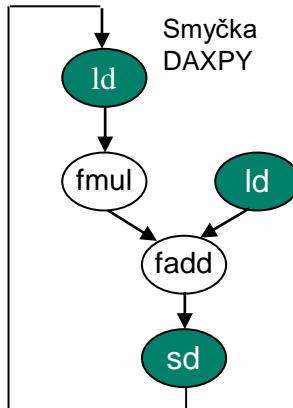
Bez výpadku:

1. generace adresy (sčítačka)
2. překlad adresy (DTLB)
3. čtení z cache (zápis není vidět)

- Překlad adresy v TLB a přístup do **fyzicky adresované** (P/P) **cache** probíhají většinou sériově, což znamená součet zpoždění (dva oddělené stupně v lince L/S).
- U **cache s virtuálním indexem** (V/V a V/P) je možný **souběžný přístup** do TLB i do cache, který dovoluje překrytí obou zpoždění.

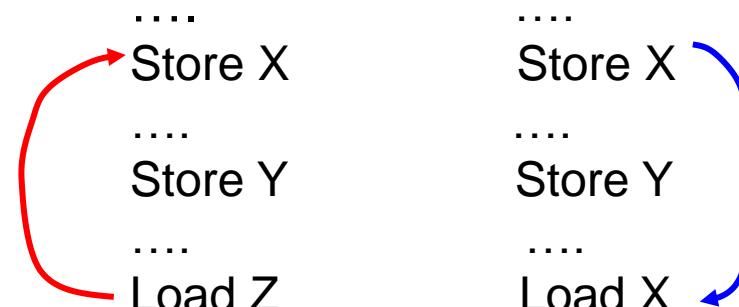


- Mezi instrukcemi **Load a Store se stejnou adresou** existují datové závislosti, podobně jako u registrů.
 - **RAW, WAR a WAW.**
 - Není je možné odhalit dříve než jsou spočteny adresy paměťových operandů.
 - Tyto závislosti musí být respektovány, aby se zachovala sémantika programu.
- **Instrukce Load a Store je možné**
 - Vykonávat v programovém pořadí (pomalé)
 - Instrukce Load a Store lze za jistých okolností provádět OOO.
 - **RPR** Read can Pass Read
 - **RPW** Read can Pass Write
 - **WPR** Write can Pass Read
 - **WPW** Write can Pass Write



- OOO procesor v podstatě provádí HW rozbalení smyčky.
- Nový load může předběhnout aktuální store, **iterace smyčky se mohou částečně překrýt!**
- RPW je hlavním zdrojem lepší výkonnosti, načítání bývá totiž na začátku těla smyček se závislými instrukcemi 2 způsoby:

Load z adresy **Z**
předběhne Store
s **jinou** adresou **X**,
které ještě nezačalo
(*bypassing*)



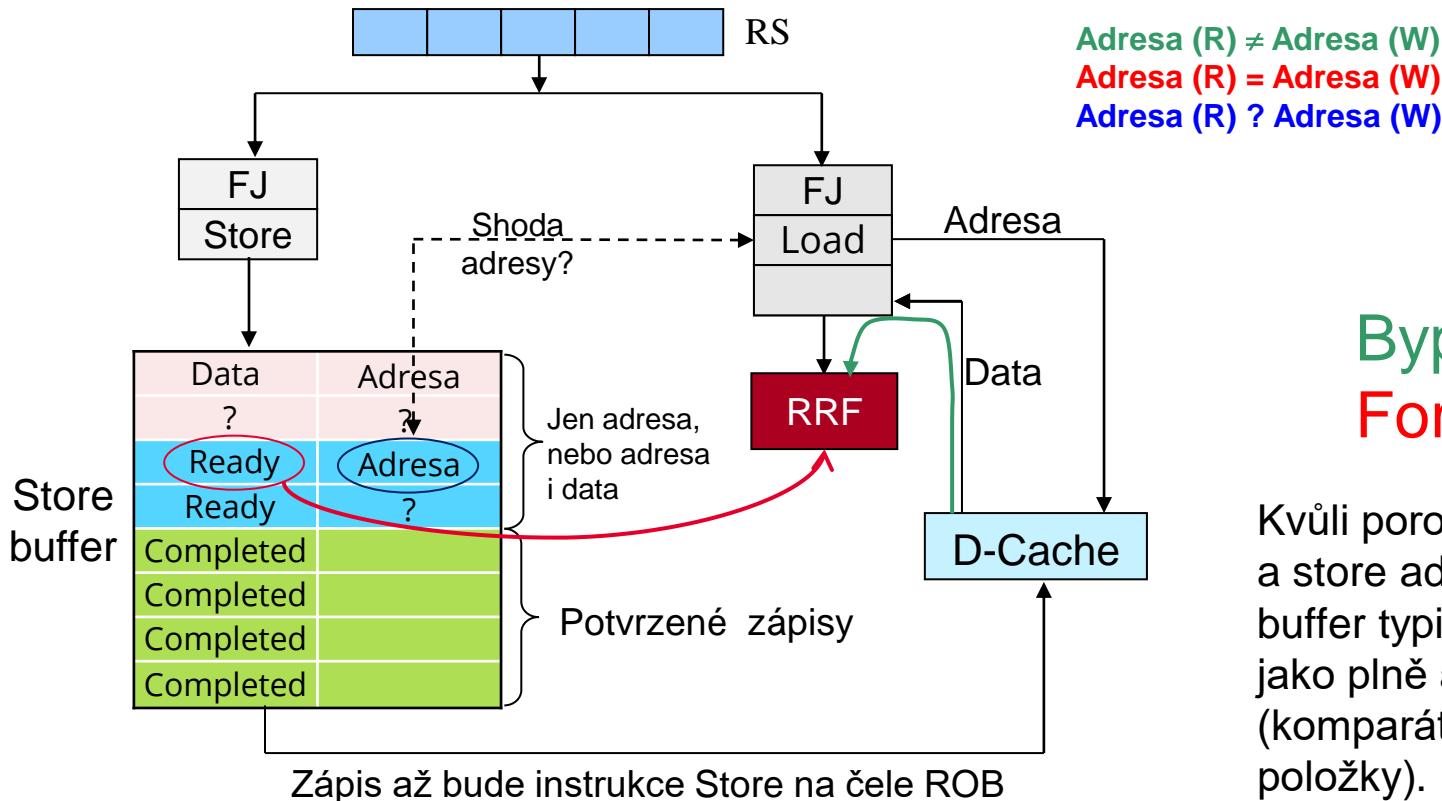
Load načte data z **ještě nedokončené instrukce**
Store se stejnou
adresou **X** (*load from store forwarding*,
předávání).

Technika RPW vyžaduje **dynamické rozlišování adres**:
adresa (R) ≠ adresa (W) ?

- Ano: neshoda adres, (není konflikt RAW);
R (load) nečeká na W (store) a předběhne jej
načte data z D-cache do dst reg (samozřejmě v RRF nebo ROB).
- Ne: shoda adres, R (load) načte data do dst reg (v RRF nebo ROB).
z čekajícího nedokončeného zápisu (store buferu)
- Neví: čekej (nebo spekuluj, že se adresy liší).

RPW může zvýšit výkonnost o 11–19% (bypassing) a o 1–4% (forwarding).

- Položka je ve store bufferu alokována v době dekódování (DI). Pokud je store buffer plný, musí se čekat.
- Adresy instrukcí Store jsou ve bufferu v programovém pořadí.
- Nové adresy Load se kontrolují s čekajícími adresami Store na shodu / neshodu.
- **Stavové byty** v každé položce indikují:
 - Available – položka je volná, k dispozici
 - Adr only – adresa je již v bufferu, data ještě ne
 - Ready – adresa i data jsou již v bufferu
 - Completed – potvrzené zápisy, čekají až instrukce Store bude na čele ROB
- Když je **instrukce Store na čele ROB i store buferu**, dojde k propuštění Store, tj. k zápisu do D-cache. Položka přejde do stavu Available.
- V terminologii x86 nazýván **Store Queue**



Bypassing Forwarding

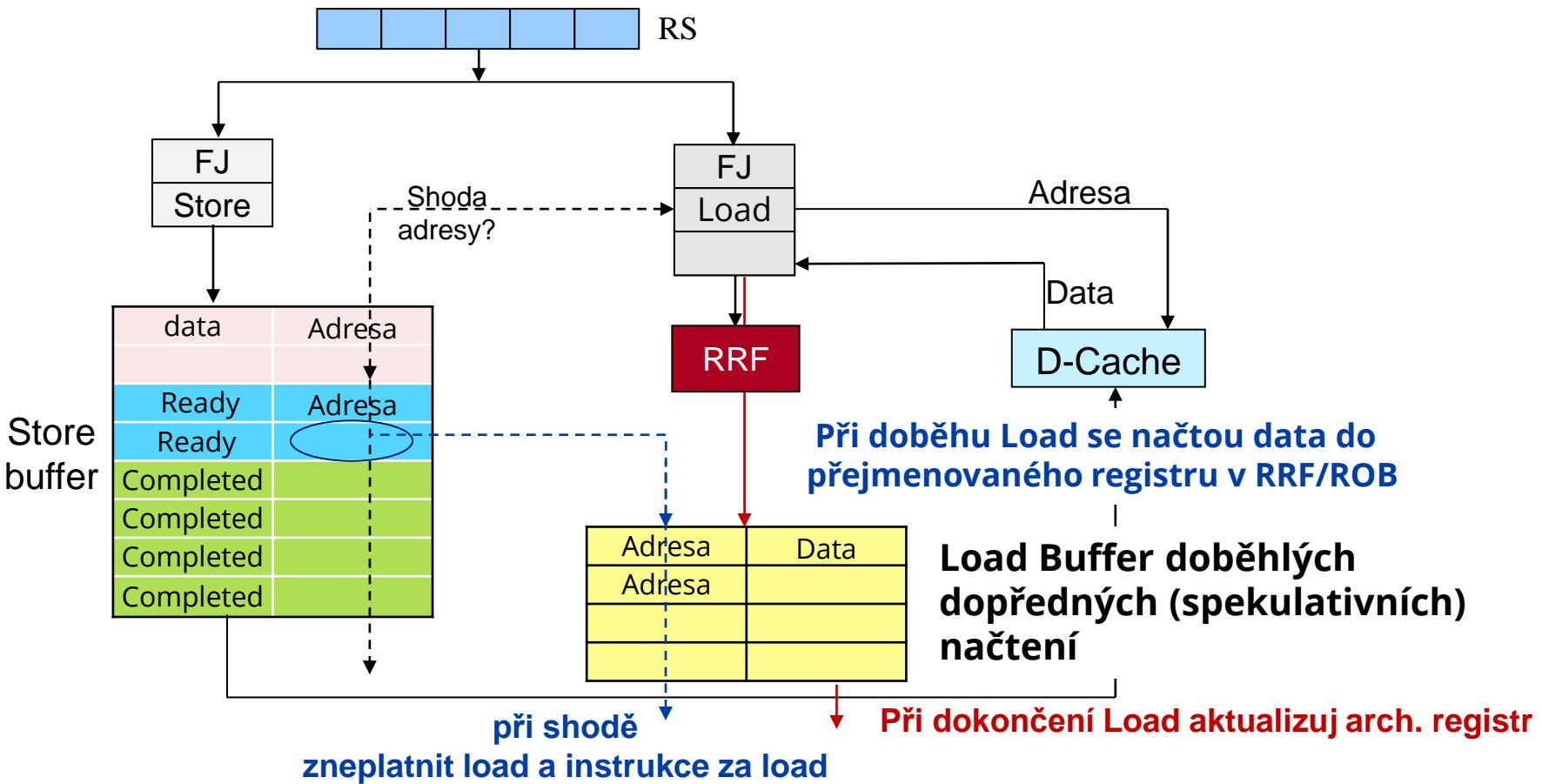
Kvůli porovnávání load a store adres je kruhový store buffer typicky organizován jako plně asociativní fronta (komparátor u každé položky).

Pokud některé adresy Store nejsou známy, je třeba spekulovat, že se budou lišit od adresy Load.

- **Pro ověření spekulace:**

- jsou adresy **doběhlých** spekulativních **načítání** uloženy do nového L-bufferu.
- Každá potvrzená instrukce **Store** pak musí **ověřit**, že nemá adresu shodnou s nějakou položkou L-bufferu, tj. s nějakým spekulativním čtením, které Store předběhlo.
 - **Při shodě** je potřeba postiženou instrukci **Load a další za ní** následující instrukce **zrušit a opakovat**.
 - Pokud k **žádné shodě nedošlo** až do dokončení Store, je proveden přepis Load dat z RRF do ARF.
- V terminologii x86 nazýván **Load Queue**

Out-of-order Load/Store jednotka



- **Sekvenční konzistence**, která zachovává pořadí přístupů do paměti, **není v současnosti zajímavá**. Je totiž překážkou modernímu hardware a optimalizujícímu kompilátorům.
- Předbíhání RPW je jen jedna možnost uvolnění pořadí přístupů do paměti, i když pro výkonnost nejvýznamnější.
- Existuje však řada ještě volnějších modelů se zcela volným pořadím čtení a zápisů.
- Moderní procesory vykazují **relaxovanou** (uvolněnou) **paměťovou konzistenci**, kdy se čtení a zápisy mohou vzájemně předbíhat, pokud to nemění správnost programu.

- **Relaxovaná paměťová konzistence**
 - Lepší výkonnost a jednodušší HW implementace
 - Je ponecháno na programátorovi, aby identifikoval a označil spec. instrukcemi (např. paměťovými bariérami) ty instrukce L/S, které musí být uspořádány.
 - Všechny ostatní instrukce se mohou provádět mimo pořadí.
- Na vyšší úrovni **musí programátor použít synchronizační příkazy** pro vymezení oblastí předepsaného pořadí L/S.
 - Direktivou **flush** v OpenMP
 - proměnnými **volatile** aj.
- **Nevýhodou relaxovaných modelů je břímě navíc pro programátora, možnost záludných chyb.**

```
data = gendata (...)  
ready = 1;
```



```
while (!ready) ;  
usedata (data)
```

- Speciální instrukce **paměťových zábran fence** F (ohrádka, zábrana) zabraňují přeskládání L/S tam, kde je to nežádoucí.
- Všechny instrukce L/S v programu před fence musí dokončit a teprve po fence mohou začít další, takže v sekvenci WFR nemůže nastat RPW.
 - Plná zábrana (viz výše)
 - Částečné zábrany (se týkají jen čtení nebo jen zápisu)
 - Jednosměrné zábrany: dvojice instrukcí **acquire** (brání přesunům L/S nahoru) a **release** (brání přesunům L/S dolů).
- Deklaraci proměnné „**volatile**“ interpretuje kompilátor jako **zápis s release a čtení s acquire**.

```
data = gendata(...)  
ready = 1;
```



```
while (!ready) ;  
usedata(data)
```

- Sekvenci po sobě jdoucích zápisů **na čele fronty** do téhož bloku D-cache lze provést **současně**.
- Důležitější je optimalizovat čtení než zápis, protože mohou bránit postupu výpočtu. Čtení se vyskytuje 2x častěji než zápis a četnost výpadků je zhruba stejná.
- Při výpadku čtení v běžné (**blokující**) cache L1 se **zastaví linka L** a další instrukce se nevydávají až do vyřízení výpadku.
- Jednotka L/S dovolující jen jedno čtení nebo zápis v jednom taktu je značně omezující u shluků L instrukcí. Proto se používá více jednotek L/S, **neblokující** cache, **dvoubránová** nebo levnější **prokládaná paměť cache**.
- **Výhody:**
 - **Další** přístupy po výpadku čtení, které nepotřebují data z výpadku, nejsou blokovány
 - Dovolují zpracování několika souběžných výpadků a zásahů.
 - Mnoho L1C a většina L2C jsou **neblokující** D-cache.
 - Výpadky při zápisu jsou ošetřeny zápisovými bufery.

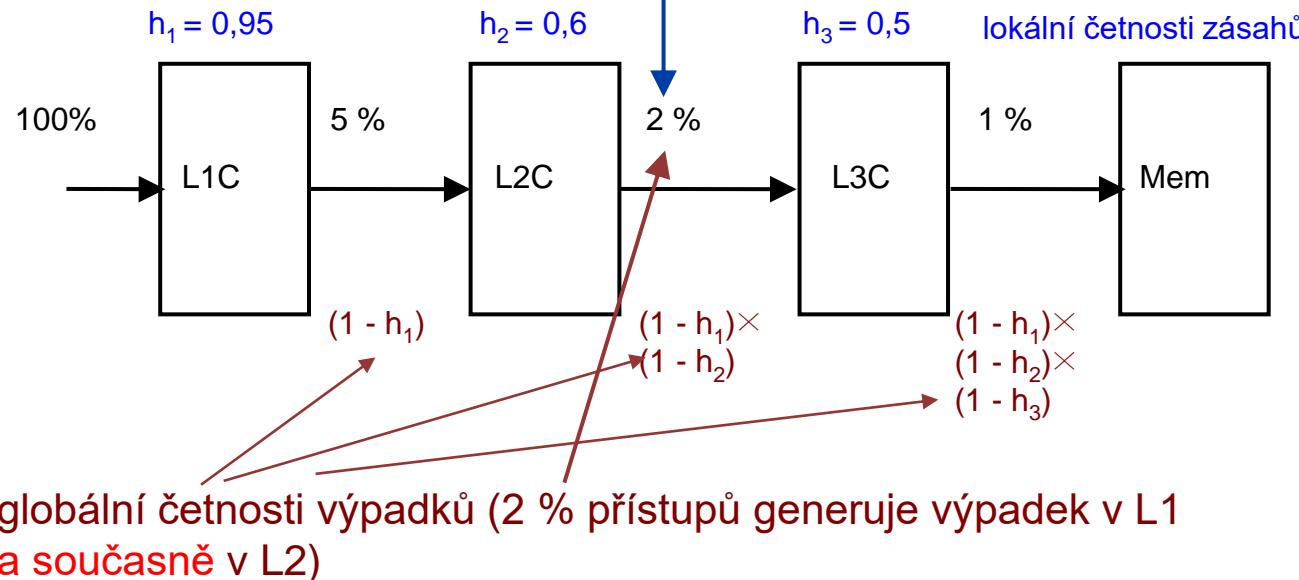
PAMĚTI CACHE (RYCHLÉ VYROVNÁVACÍ PAMĚTI)

- **Doposud jsme uvažovali:**
 - jen cache L1,
 - reálnou paměť,
 - fyzickou adresu PA,
 - četnost zásahů (hit rate) $h = 100\%$.
- **Nyní budeme uvažovat:**
 - L1C, L2C, L3C = cache úrovně L1, L2, L3,
 - virtuální paměť, virtuální adresa VA, $h_{1,2,3} < 100\%$.
- **Názvosloví:**
 - Sjednocená/rozdělená cache pro data a instrukce
 - Cache privátní/sdílené
 - Strategie zápisu:
 - write through nebo write back (při zásahu)
 - write around nebo write allocate (při výpadku)

- **2 úlohy pamětí cache:**
 - Snížení objemu komunikace s hlavní pamětí on 2 až 3 řády.
 - Snížení průměrné doby přístupu o 1 až 2 řády.
- Jelikož malé paměti jsou rychlejší a dražší, velké paměti pomalejší a levnější, je optimální paměťový systém heterogenní, s několika úrovněmi pamětí cache
- **Parametry paměti cache:**
 - (lokální) četnost zásahů h_i : počet zásahů v cache úrovni i dělený počtem přístupů do této cache
 - (globální) četnost výpadků v úrovni i : $(1 - h_1) \times (1 - h_2) \times \dots (1 - h_i)$ znamená četnost výpadků generovaných současně v úrovni 1, 2... i
 - latence zásahu t_i : doba od vyslání adresy do návratu dat z cache i .
 - latence výpadku (pokud procesor při výpadku blokuje) na úrovni i : doba přenosu bloku z cache úrovně $i + 1$ do úrovně i a načtení slova z cache i .
- **U procesorů OOO nelze latenci výpadku přímo měřit.**

I Lokální a globální četnost výpadků a zásahů

98 % přístupů se trefí v L1 nebo v L2
 (= globální četnost zásahů v L1 + L2)

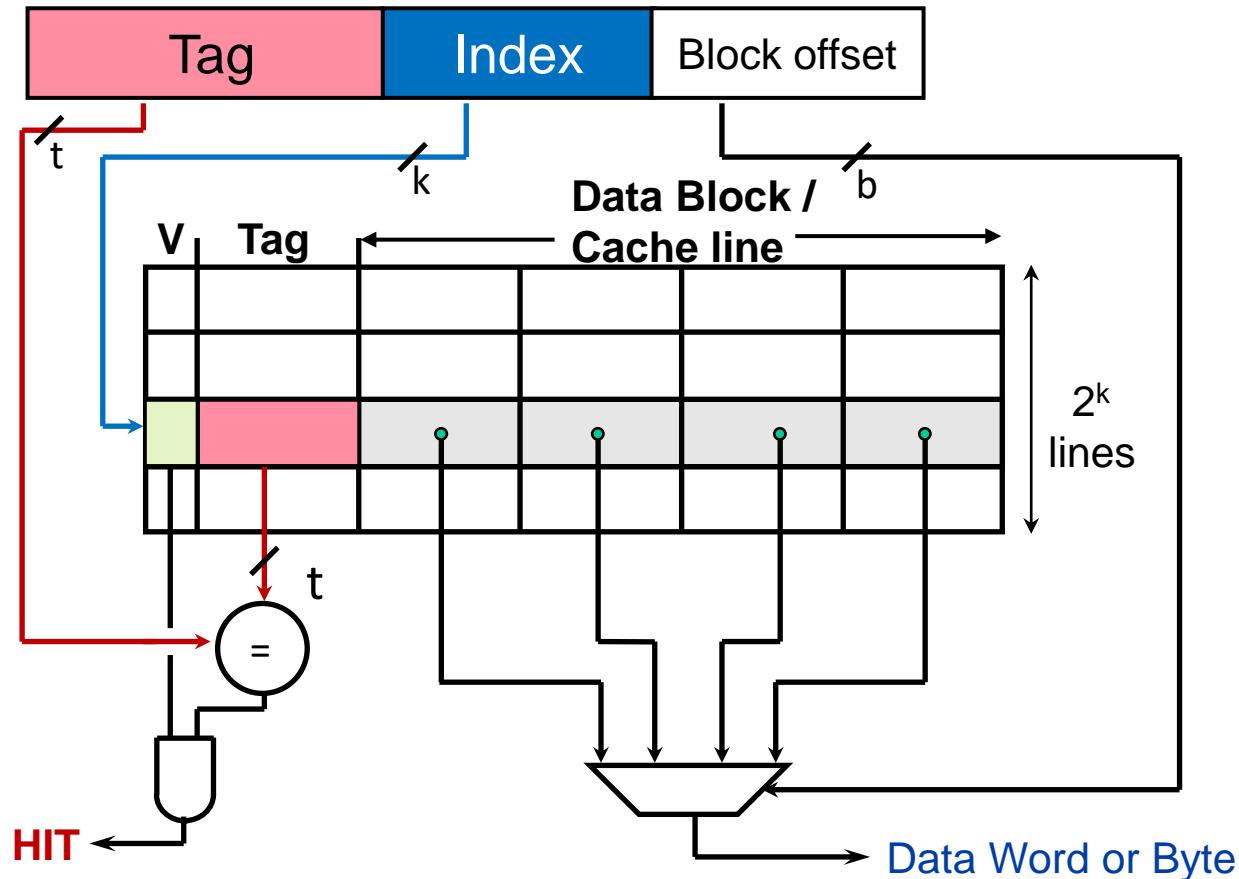


Uvedené parametry se týkají konkrétního programu nebo skupiny programů.

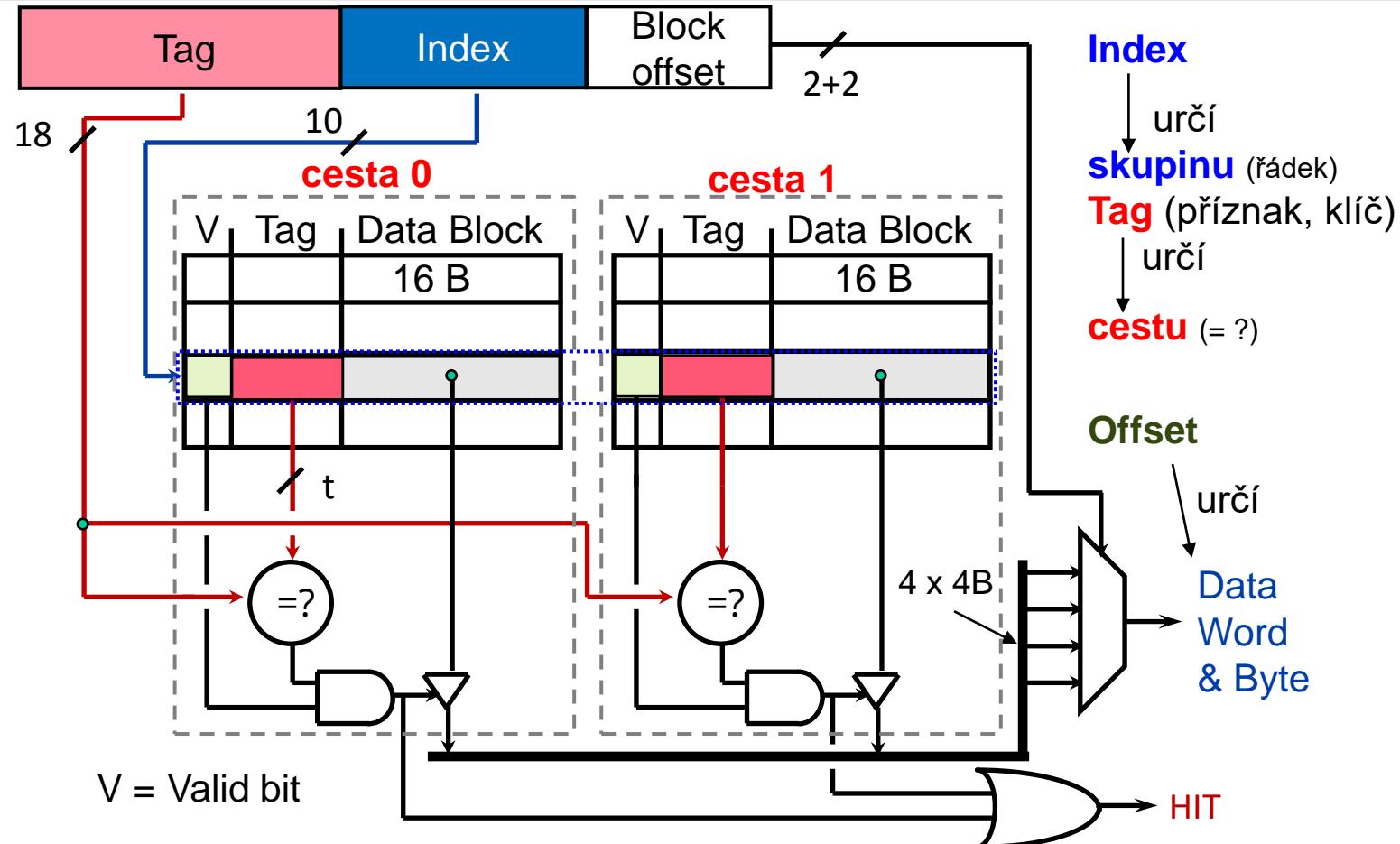
- Výběr dat po blocích na základě
 - obsahu (příznaku bloku, tag)
 - nebo i adresy (index).
- Asociativita **$m = \text{počet cest (ways)}$, komparátorů pro tag.**
Přístup do bloku i :
 - shodný tag existuje: hit,
 - neexistuje: miss.
- **Zjednodušení HW:**
 - počet bloků C celé paměti
 - lze rozdělit do S skupin (sets)
 - a pro výběr skupiny použít adresu (index).
- Podle toho máme paměti cache
 - **Přímo mapované (PM):** $m = 1, S = C$ (skupina = 1 blok)
 - $\text{index} = i \bmod S$ vybírá 1 blok cache
 - shoda tagů se hledá jen v nalezeném bloku.
 - **Plně asociativní (PA):** $C = m, S = 1$ skupina, jen tag
 - hledá se shodný tag v celé paměti (tj. skupině).
 - **Skupinově asociativní (SA):** index + tag
 - index určuje skupinu
 - $\text{index} = i \bmod S$ (má $\log_2 S$ bitů)
 - shodný tag se hledá v indexované skupině.

I Přímo mapovaná (PM) cache

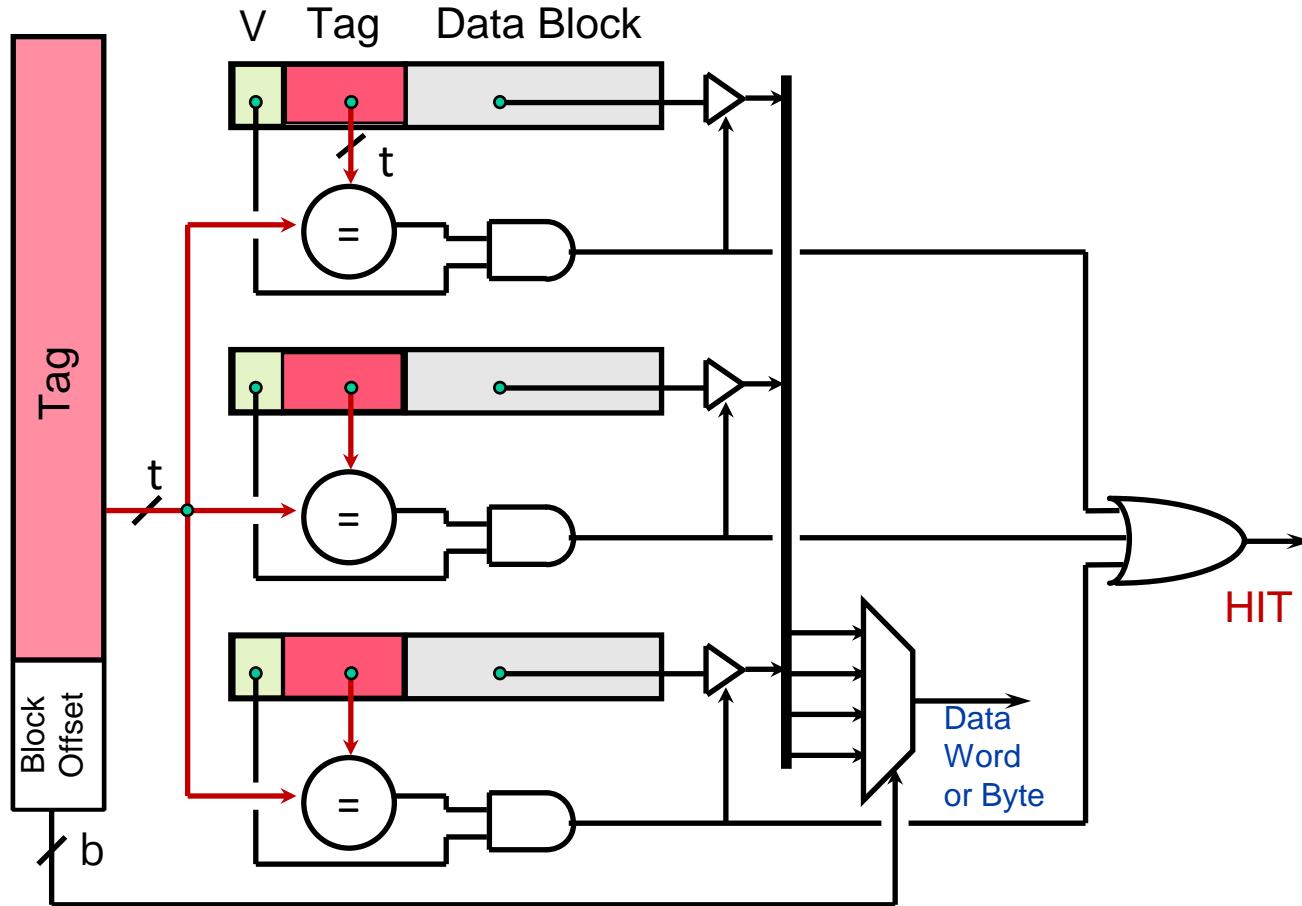
Adresa:



| Skupinově asociativní (SA) cache 32 kB

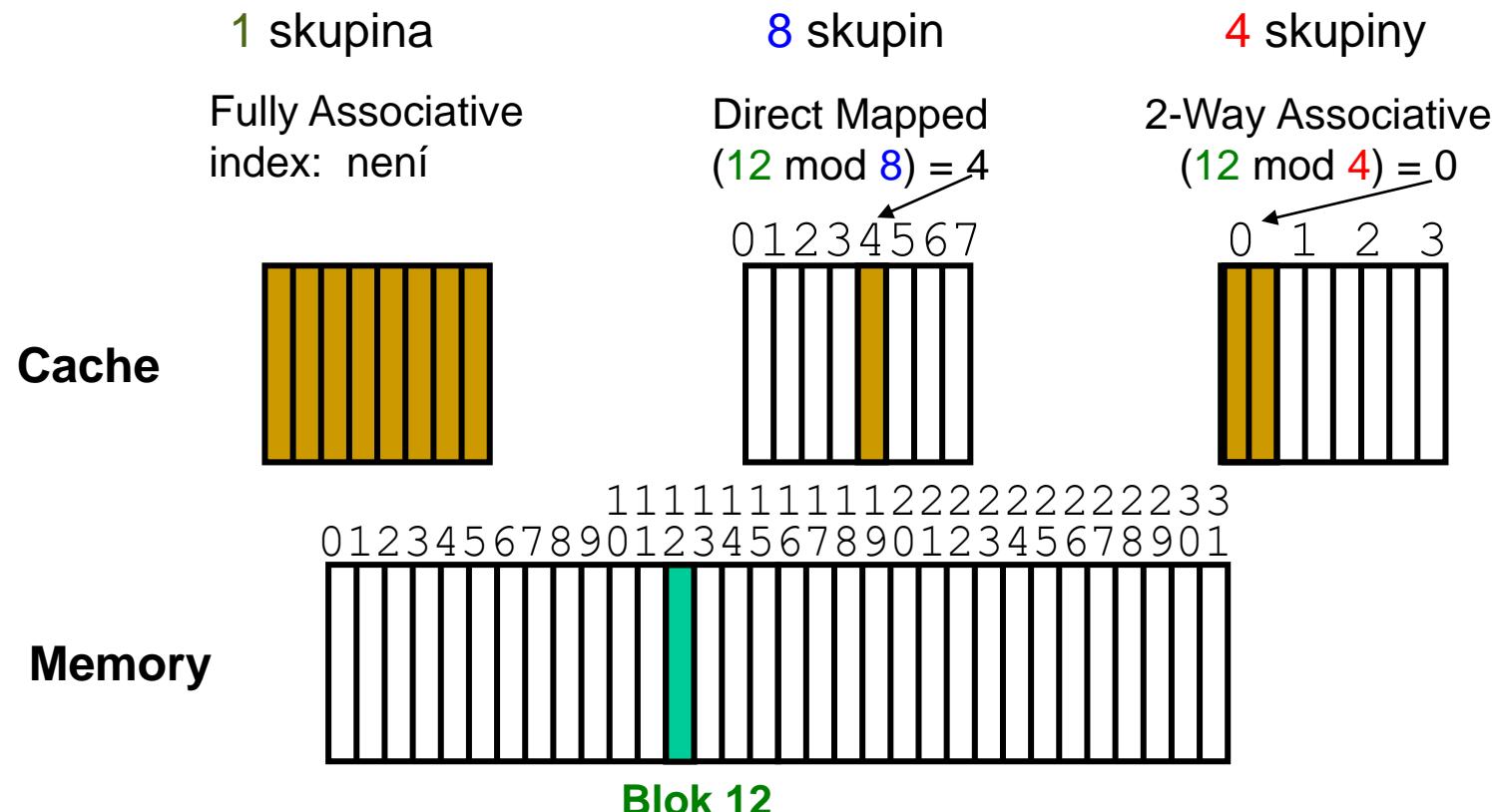


I Plně asociativní (PA) cache



I Příklad mapování bloků paměti do bloků cache

Umístění bloku $i = 12$ do cache o velikosti 8 bloků (kam mohu vložit):



- **Výpadek při zápisu slova do cache** (write allocate):
 - Najde se volný blok a provede se zápis.
 - Není-li v cache volné místo, musí se některý blok z cache přemístit do paměti (**block replacement**).
- **Strategie výběru bloku pro přemístění:**
 - LRU (blok nejdéle nepoužitý, zaznamenává se počet přístupů),
 - FIFO (nejstarší blok),
 - nebo náhodný blok.
- **Pokud je vyměněný blok později potřeba**
 - nastane výpadek kvůli **kolizi** (jen jedna cesta u PM cache nebo málo cest u SA cache)
 - **kapacitě** (PA cache).
- **Výpadky vznikají na začátku**, když je cache prázdná (**povinné** výpadky)
- **Výpadky udržováním koherence** v multiprocesorovém systému (**koherenční**).

- U vícejádrových procesorů jsou **cache kromě poslední úrovně privátní** a **LLC** (Last Level Cache) **sdílená**.
- **Inkluzivní paměti cache (Intel do Skylake-X)**
 - **Vlastnost inkluze** mezi úrovněmi cache (obsah L2C) je nadmnožina obsahu L1C) zajišťuje, že L2C odfiltruje zbytečné požadavky na L1C:
 - **invalidace** bloků, které nejsou v L2C, nepostupují do L1C
 - je-li blok v L2C, **inclusion bity** u něj ukazují, zda je i v některé L1C.
 - 1 : je v dané L1C, musí se zneplatnit,
 - 0 : žádná akce.
 - **Udržování inkluze:**
 - Při výpadku v L1C musí být blok načten do všech úrovní L_i
 - Když je blok z úrovně L_i přemístěn do paměti, musí být odstraněn ze všech úrovní pod L_i

- **Exkluzivní paměti cache (AMD):**

- Je-li blok v úrovni L_i , pak není v žádné jiné úrovni
- Při výpadku v L1C se tam **přesune** (nekopíruje) blok z některé vyšší úrovně
- Když je blok z úrovni L_i **přepsán** (zničen) blokem z paměti, je před tím přesunut na úroveň $L_i + 1$
- V systému L1C-L2C je pak L2C velká cache „obětí“.

- **Cache obětí (Victim Cache, VC)**

- Přidá se malý buffer pro umístění (LRU) bloků vyhozených z L1C
- Tato PA **cache obětí** již jen se 4–16 položkami může efektivně eliminovat mnoho výpadků PM cache.
- Používá se mezi úrovněmi L1 a L2. Dojde-li v L1C k výpadku, tak
 - při zásahu ve VC se nalezený blok vymění s LRU blokem v PM cache
 - při výpadku ve VC se v L1C udělá místo pro nový blok. Oběť (LRU blok v L1C) se zapíše do VC. Je-li VC plná, pak do další úrovně. Nový blok je pak zaslán do L1C.

PODPORA VIRTUÁLNÍ PAMĚTI NA PROCESORU

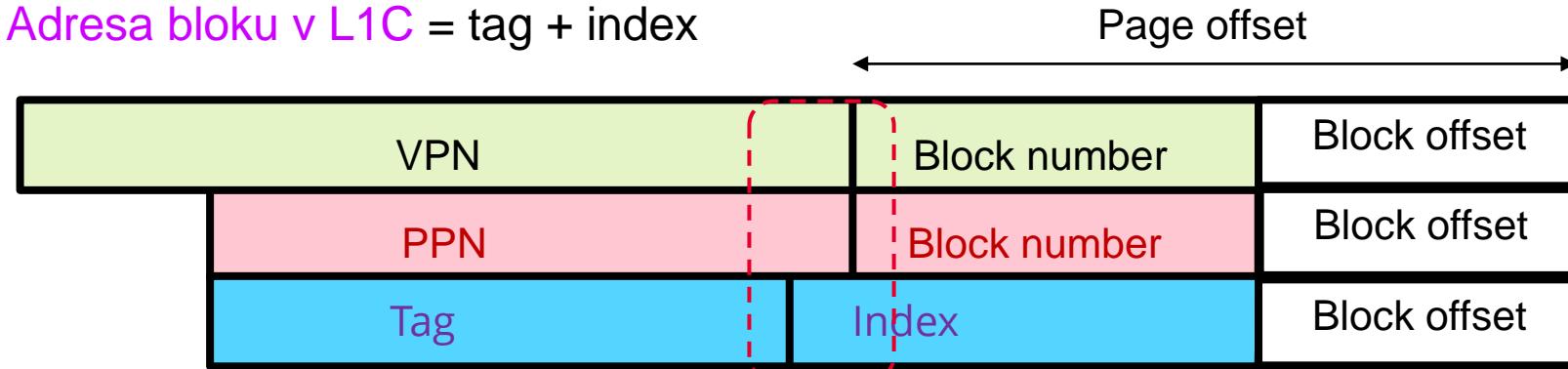
VA = Virtual Address = virtual page number **VPN** + Pg. offset

PA = Physical Address = phys. page number **PPN** + Pg. offset

Pg.offset = Block number (číslo bloku na stránce) + **block offset** (které slovo, byte)

Adresa bloku v paměti = PPN + Block number

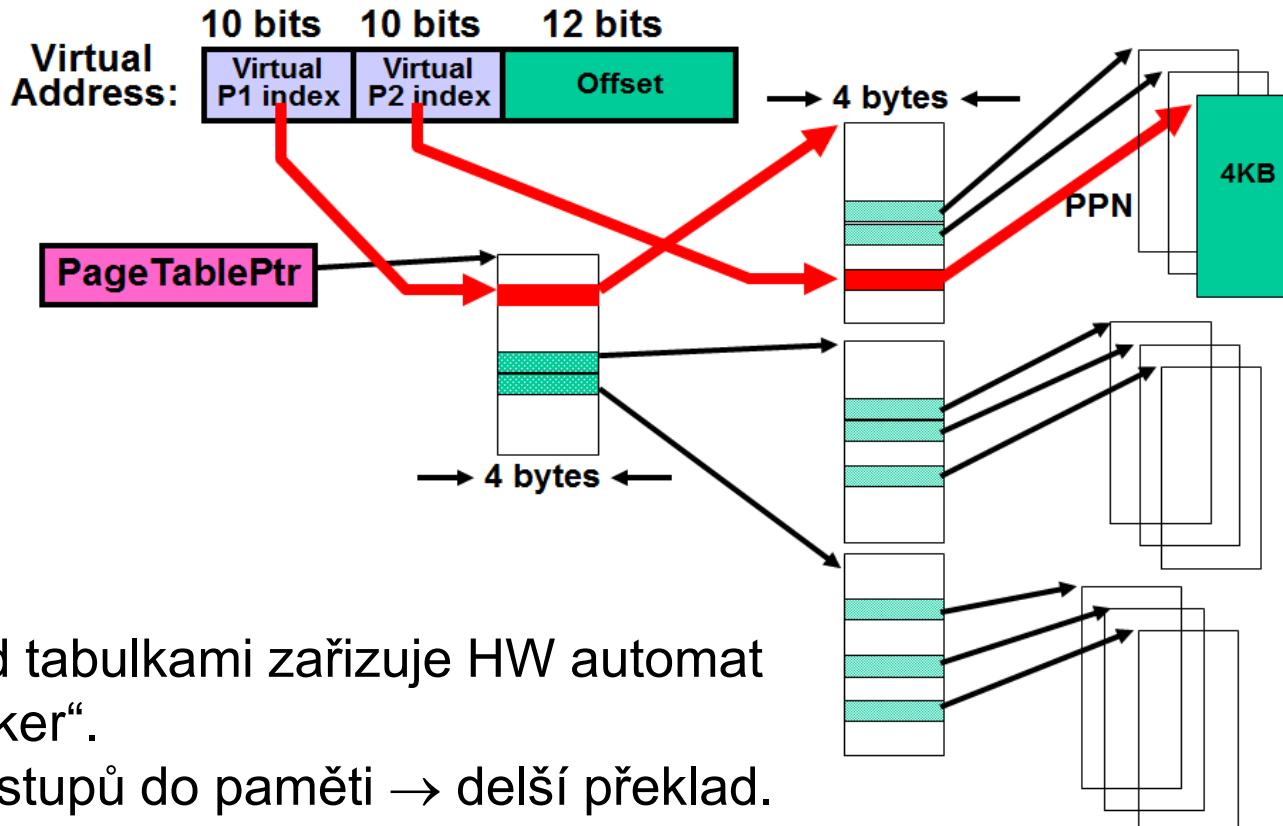
Adresa bloku v L1C = tag + index



Velikost stránky: 4, 8 kB, ale i 64 kB, 2 MB, 4 MB

Cache index: 16 kB, 32 kB, 512 kB, 16 MB

- Položka PT (PTE, Page Table Entry) mapuje číslo virtuální stránky na číslo fyzické stránky, VPN → PPN; Vyhledání je snadné!
- **PT je v paměti.**
- Pro 32 bitové adresy, stránky **4 KB** a PTEs 4 byte:
 - $2^{32}/2^{12} = 2^{20}$ PTEs, tj. **tabulka stránek 4 MB** na 1 proces
 - až $2^{32} = 4$ GB dat v celém virtuálním prostoru na 1 proces
- **Větší stránky?**
 - Vnitřní fragmentace (celá stránka se neužije) ☹
 - Větší pokuta při výpadku stránky (delší čas čtení z disku) ☹
 - Méně překladu při zpracování velkých dat (matice) ☺
- **A co teprve 64 bit virtuální adresový prostor???**
 - Dokonce při velikosti stránek **1MB** bychom potřebovali $2^{64}/2^{20} = 2^{44}$ PTEs 8 byte (2⁷ TB!)
- **Naštěstí je obsazení virtuálního adresového prostoru řídké**



Průchod tabulkami zařizuje HW automat „PT walker“.

Více přístupů do paměti → delší překlad.

I Translation Lookaside Buffer (TLB)

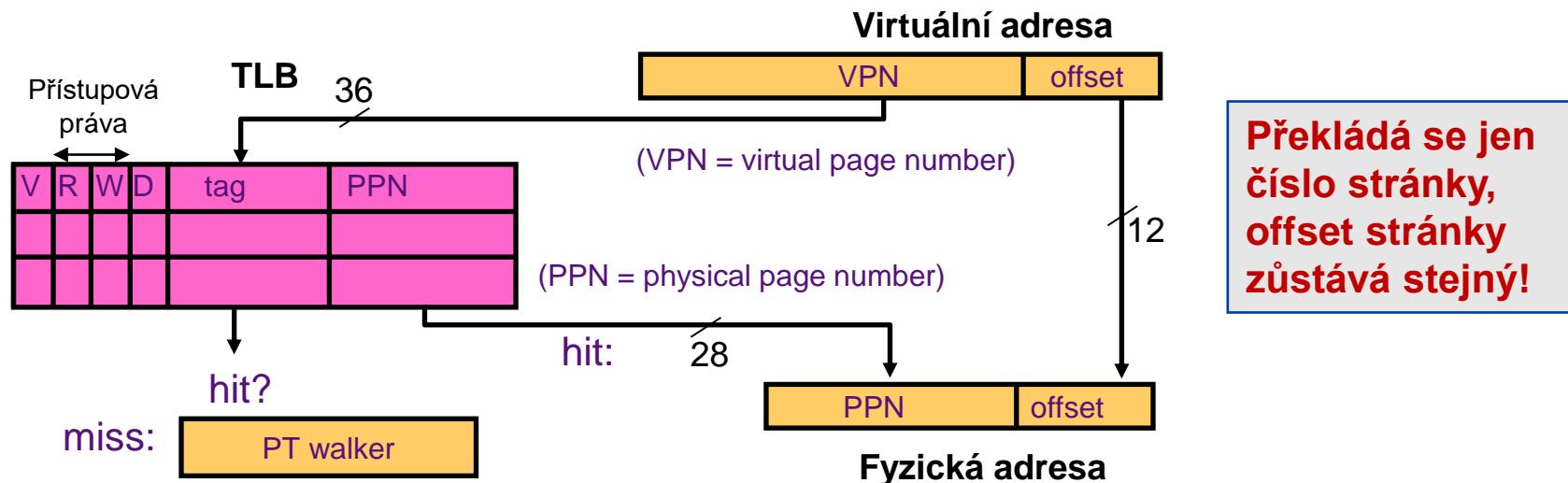
Překlad adresy je velmi drahý!

V hierarchické tabulce stránek stojí každý překlad několik přístupů do paměti.

Řešení: *Překladová tabulka TLB na čipu* obsahuje sadu

aktuálně používaných dvojic {VPN, PPN}

- TLB hit \Rightarrow překlad za 1 takt,
- TLB miss \Rightarrow průchod PT k doplnění TLB



- **Velmi malá** (32–128 položek) a tak velmi rychlá cache.
- **Plně asociativní.** Někdy jsou větší TLBs (256–512 položek) skupinově asociativní, 4–8 cestné
- Může být rozdělena na instrukční a datovou část.
- **Strategie výměny:** LRU, pseudo LRU, random nebo FIFO
- **Správa TLB:** HW nebo SW.
- Větší systémy někdy mívají více-úrovňové TLB (L1 a L2). TLB L2 bývá sjednocená.

Intel Haswell:

- **I-TLB**
 - 4 kB stránky
 - 128 položek, 4-way asociativita, dynamicky rozdělena mezi 2 vlákna.
 - 2/4 MB stránky
 - 8 položek, plně asociativní. Každé vlákno má svoji tabulkou.
- **D-TLB**
 - 4 kB stránky
 - 64 položek, 4-way asociativita, fixně rozdělena mezi vlákna.
 - 2/4 MB stránky
 - 32 položek, 4-way asociativita.
 - 1 GB stránky
 - 4 položky, 4-way asociativita.
- **STLB**
 - 4 kB + 2 MB stránky
 - 1024 položek, 8-way asociativita.
 - Sdílená pro data i instrukce na úrovni L2.

I Paměti cache v systému virtuální paměti

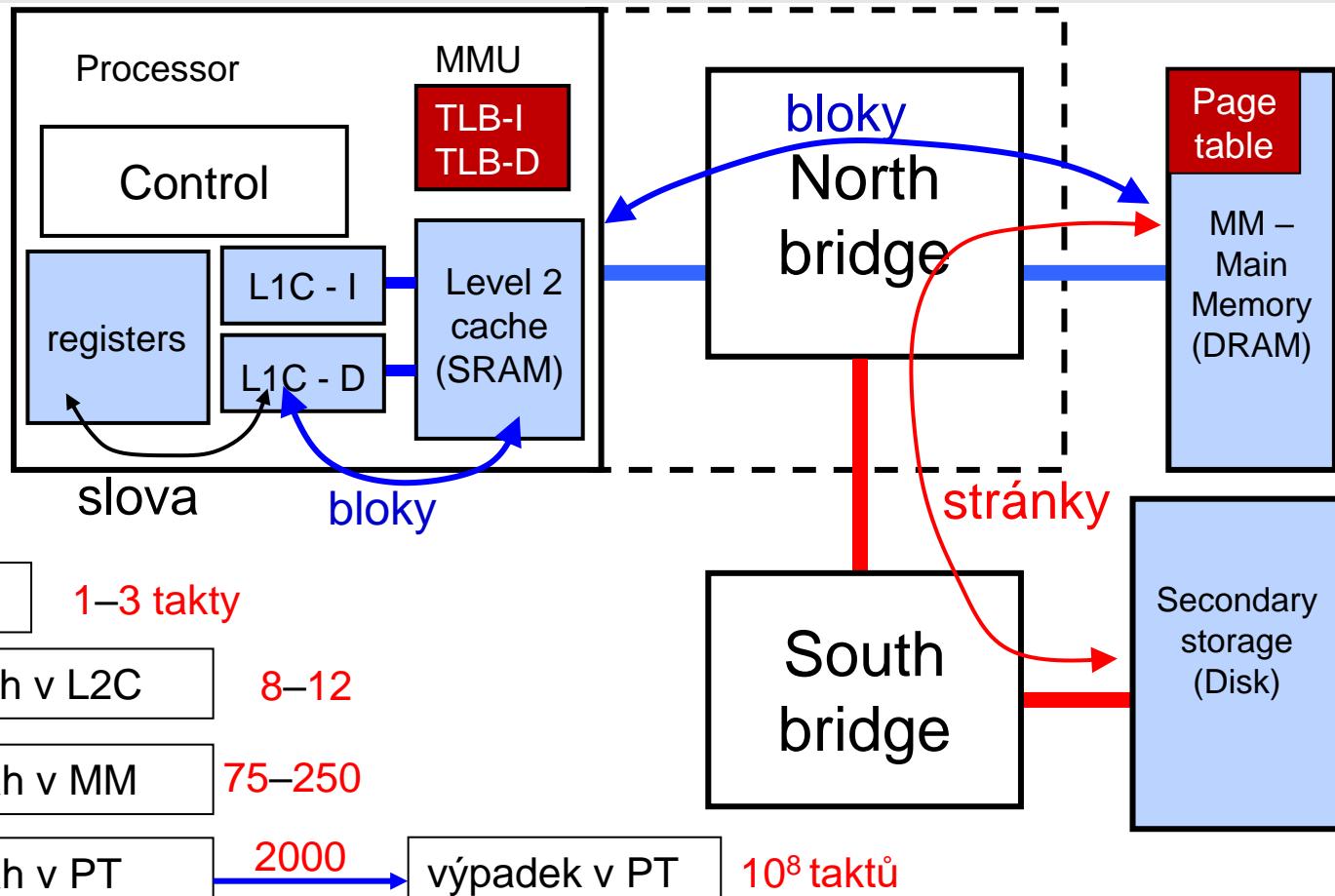
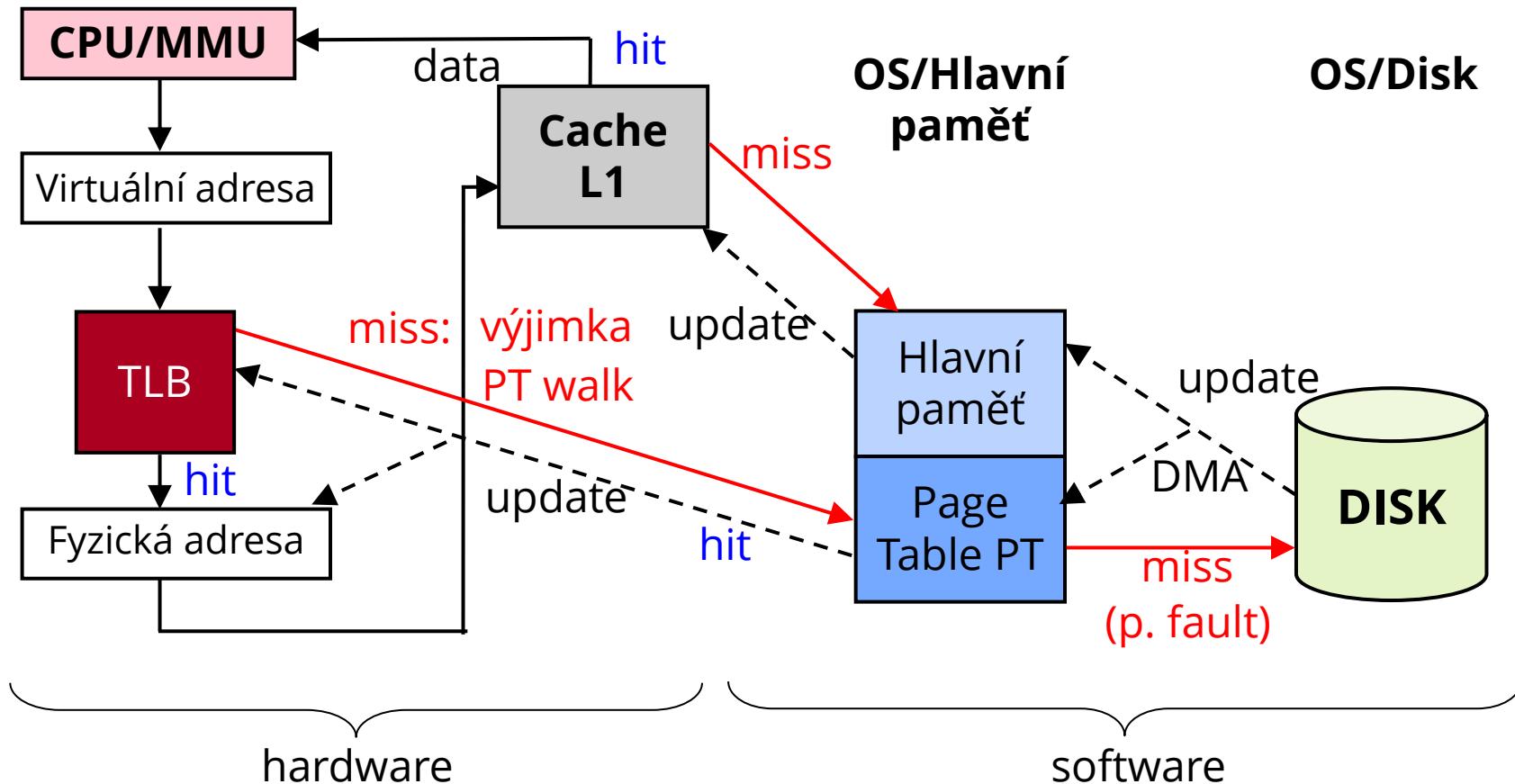
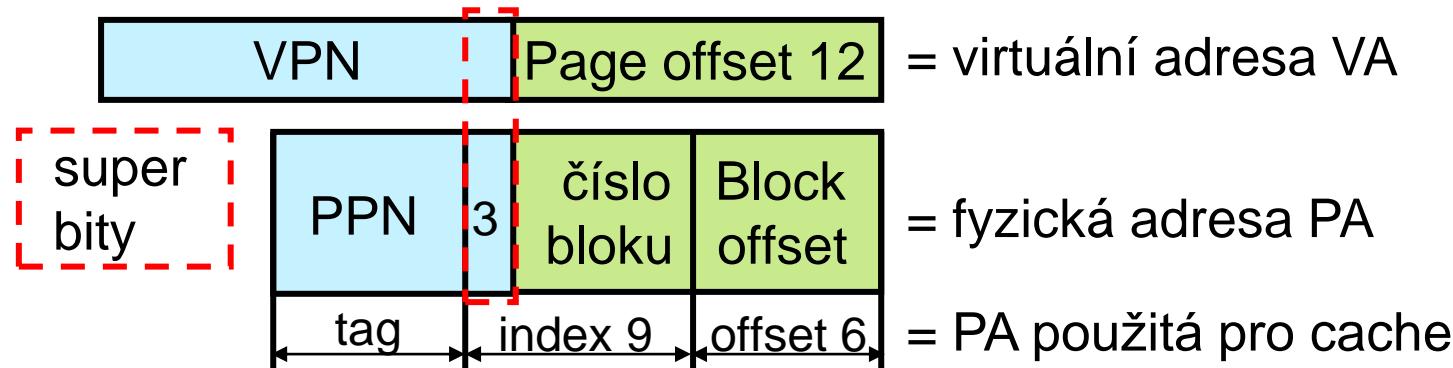


Schéma přístupu do paměti s L1C

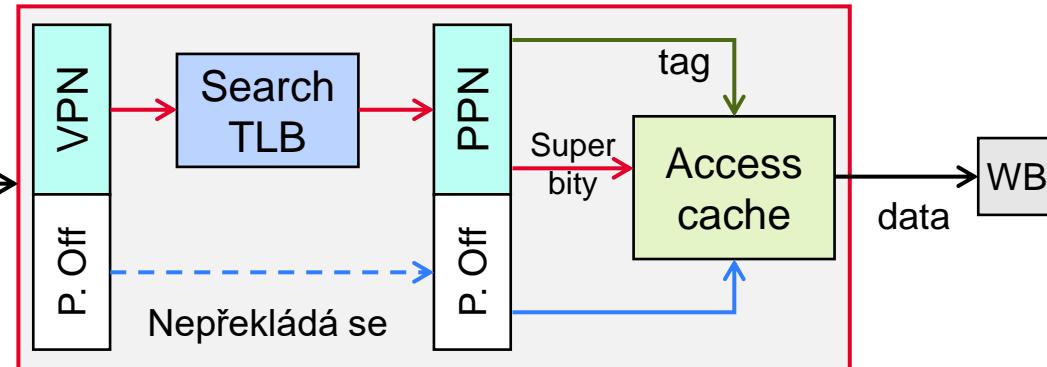
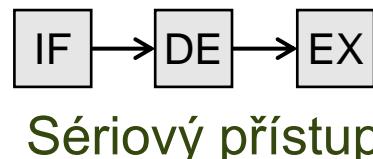


- Spolupráce CPU s L1C vyžaduje co nejrychlejší přístup do cache.
- Jak dospět od VA k adrese bloku (**index, tag**) v L1C?
 - VA = virtuální adresa je k dispozici ihned
 - PA = fyzická adresa je k dispozici až po **překladu** VA.
 - Která adresa by se mohla použít pro přístup?
- **Tři možnosti:**
 - P/P cache: fyzický index, fyzický tag
 - V/V cache: virtuální index, virtuální tag
 - V/P cache: virtuální index, fyzický tag
virtuální index se dá použít hned, paralelně s překladem VA.
 - P/V cache: fyzický index, virtuální tag - nepoužívá se, index by se musel získat překladem a čas by se neuspořil.

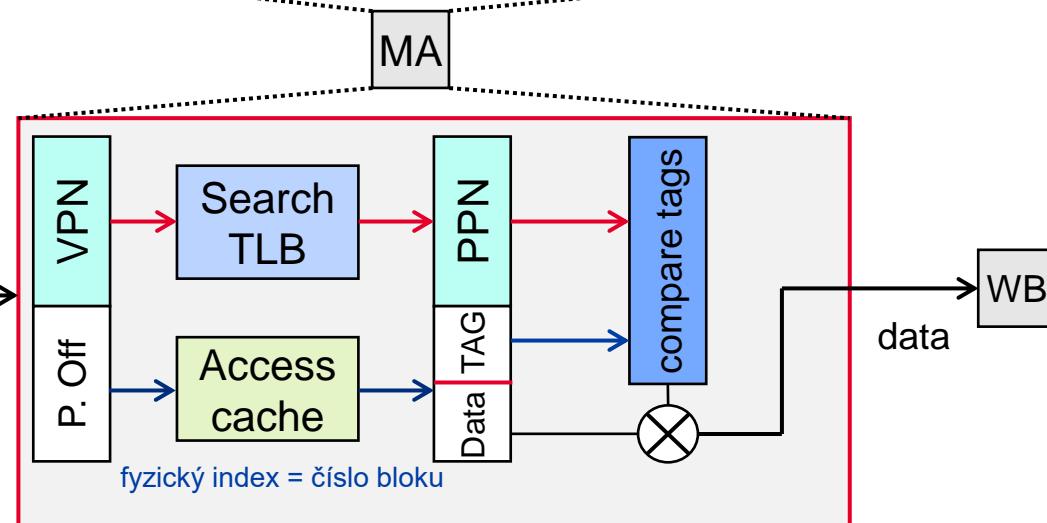
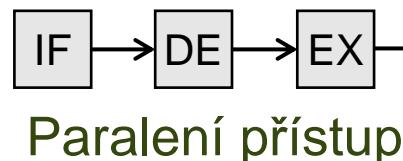


- **Sériový přístup** napřed TLB, pak cache L1 je **pomalý**.
- **Paralelní přístup** je možný
 1. Když se **index** kryje s **číslem bloku**: nemusí se překládat, indexuje bloky na 1 stránce dané PPN.
Např. pro stránku 4 kB, velikost bloku 64 B je na stránce $4\text{ kB}/64\text{ B} = 64$ bloků a index má 6 bitů. Kapacita cache je zde ale omezena na m (cest) stránek. Např. 8 cestná cache má kapacitu $8 \times 4\text{ kB} = 32\text{ kB}$ (Intel Haswell)
 2. Když se **bity VPN** a **PPN** použité v **indexu** (**superbity**) shodují (stránky mají stejnou „barvu“). To může zařídit např. OS/SW. Pak se také index získá bez překladu.

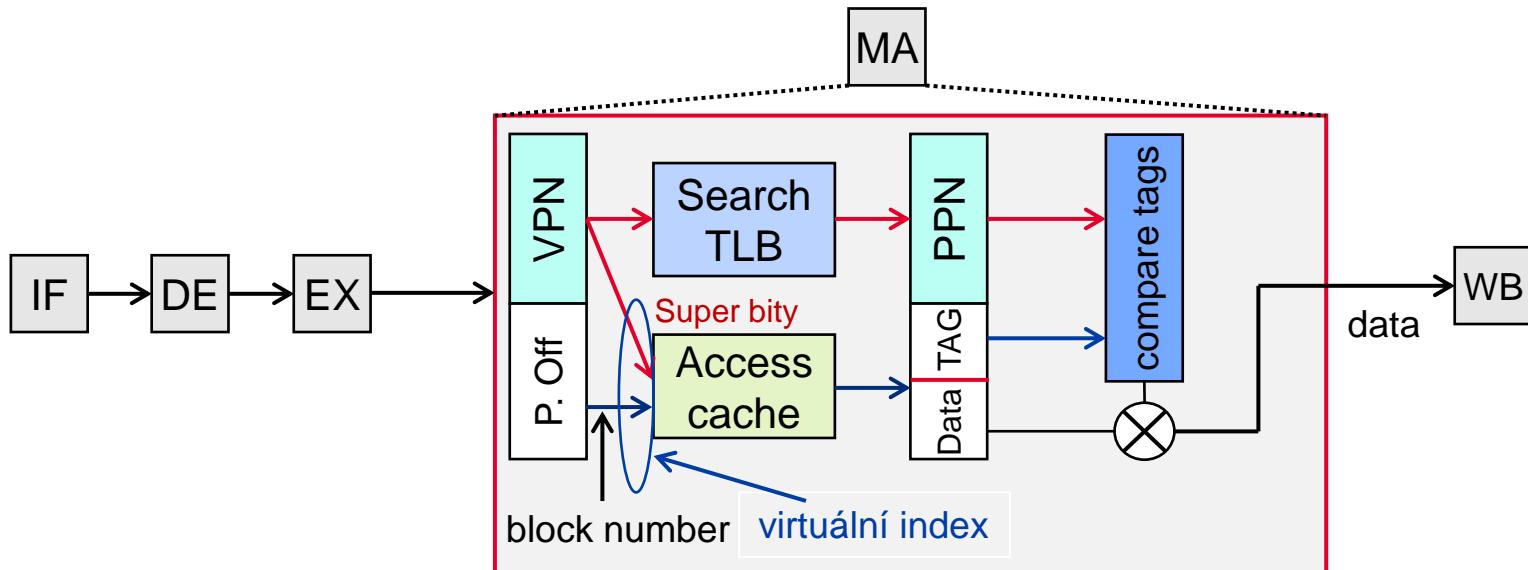
VPN Virtual Page Number
 PPN Physical Page Number
 P.Off Page offset



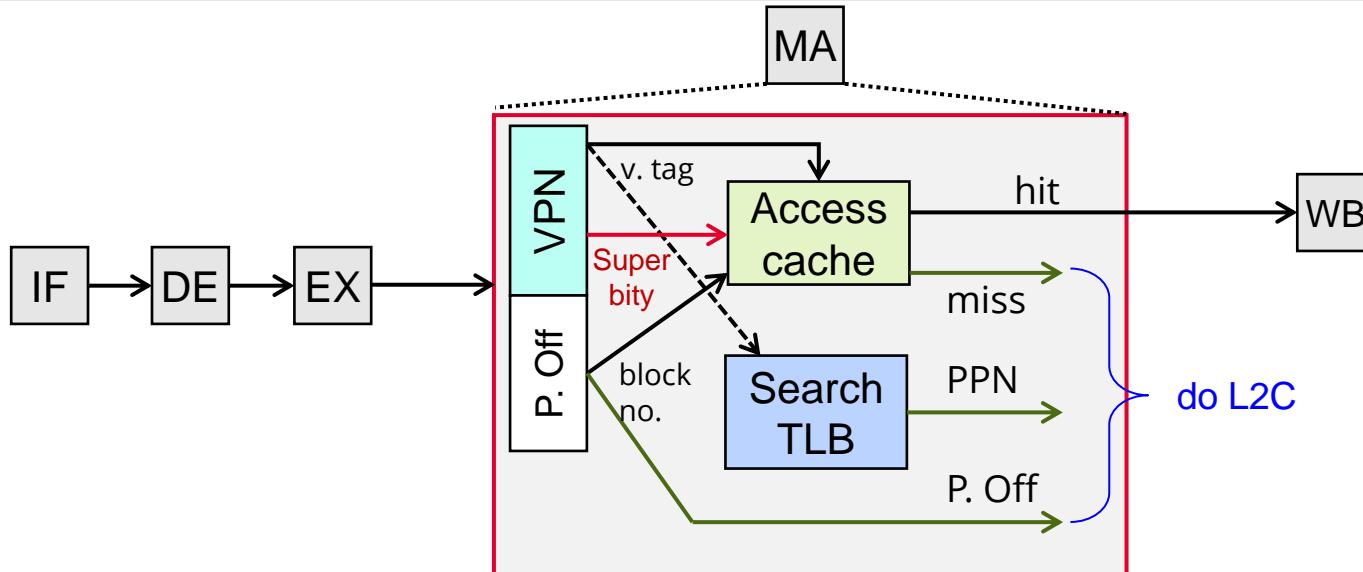
fyz. index = číslo bloku
bez překladu pokud se
index kryje číslem bloku



- **Indexování L1C** může začít hned, spolu s překladem v TLB.
- Následuje **porovnání tagu** z L1C s tagem získaným z TLB.



- **Problém Synonym**
 - 2 nebo více VA v TLB se přeloží na stejný fyzický index



Virtuální index = block number + **virtuální superbity**

- **Při zásahu** v L1C se TLB neuplatní, **překlad není třeba**.
- **Výstup TLB** je použit jen **při výpadku v L1C**, tedy zřídka. Nalezené PPN se použije pro hledání v L2C.

Další komplikace:

- Jelikož TLB se většinou obchází, je třeba režijní bity stránky překopírovat z TLB do L1C.
- Kromě **synonym** je třeba ještě řešit **homonymy**: stejná VA se mapuje do rozdílných PA (přepnutí kontextu).

Pokračování příště

Predikce skoků, přednačítání instrukcí a dat AVS – Architektury výpočetních systémů Týden 4, 2023/2024

Jirka Jaroš

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



OPAKOVÁNÍ

Instrukce jsou vydávány do FJ a prováděny **mimo pořadí** v programu, pokud mezi nimi **nejsou konflikty** a **FJ** jsou **volné**.

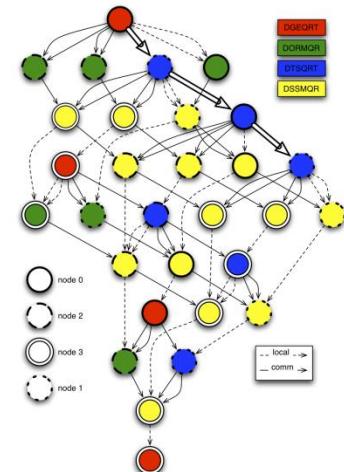
1. ScoreBoarding (Thorntonův algoritmus, 1964)

- Registruje všechny **konflikty** (RAW, WAW, WAR) v **tabulce rozpracovaných instrukcí** a udržuje jejich skóre (SB).
- SB vydá instrukce dál jen když nejsou v konfliktu s ostatními instrukcemi v SB. **Přejmenování registrů neprobíhá**.

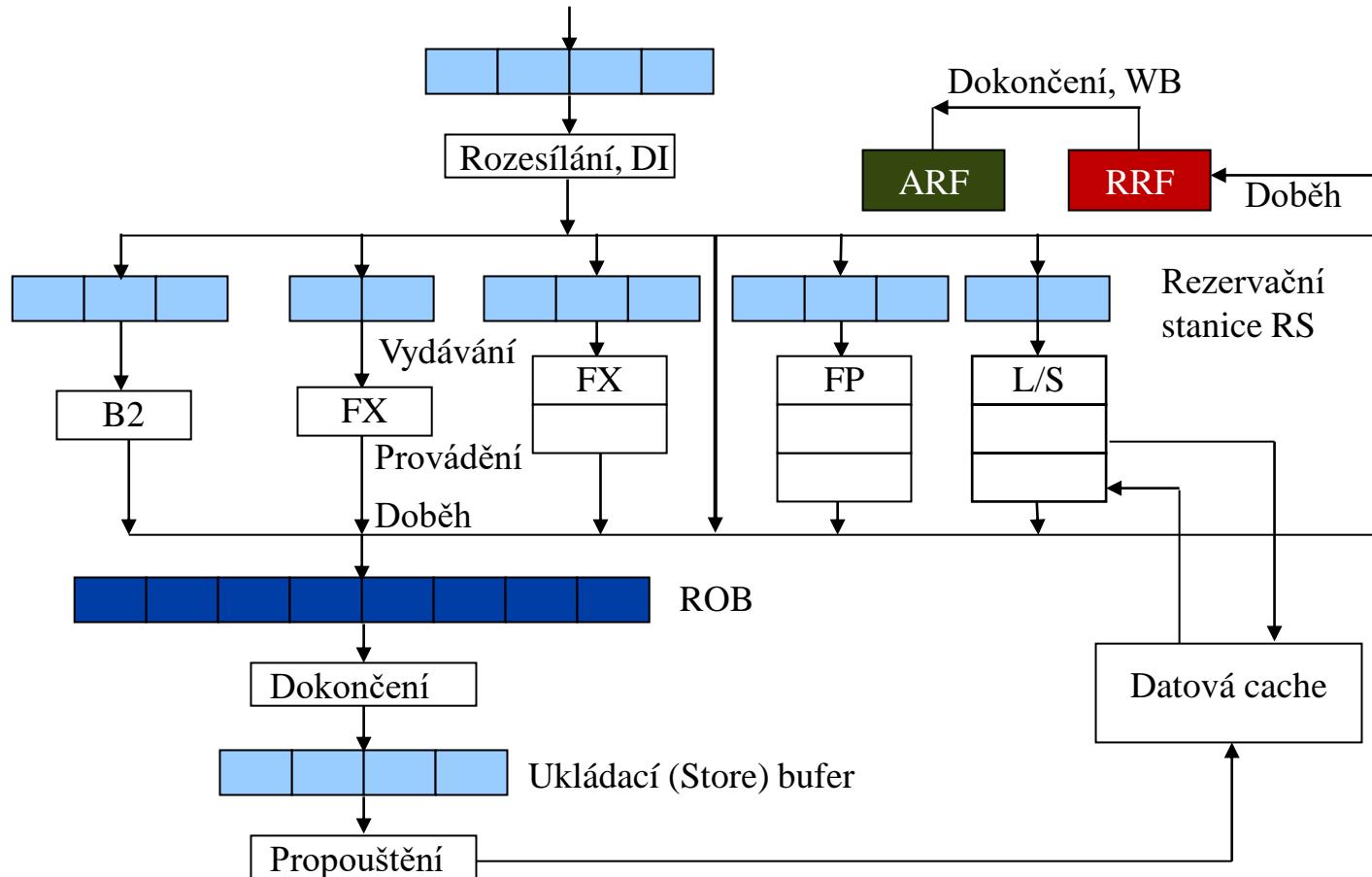
2. Rezervační stanice (Tomasulův algoritmus, 1967)

- Konflikty WAW a WAR se řeší přejmenováním
- **Rezervační stanice RS** (bufery) umožňují odložit čekající instrukce a pracovat dopředu na dalších – tím řeší RAW.
- Rezervační stanice centrální (instruction window) nebo individuální u FJ či skupinové pro skupiny FJ.

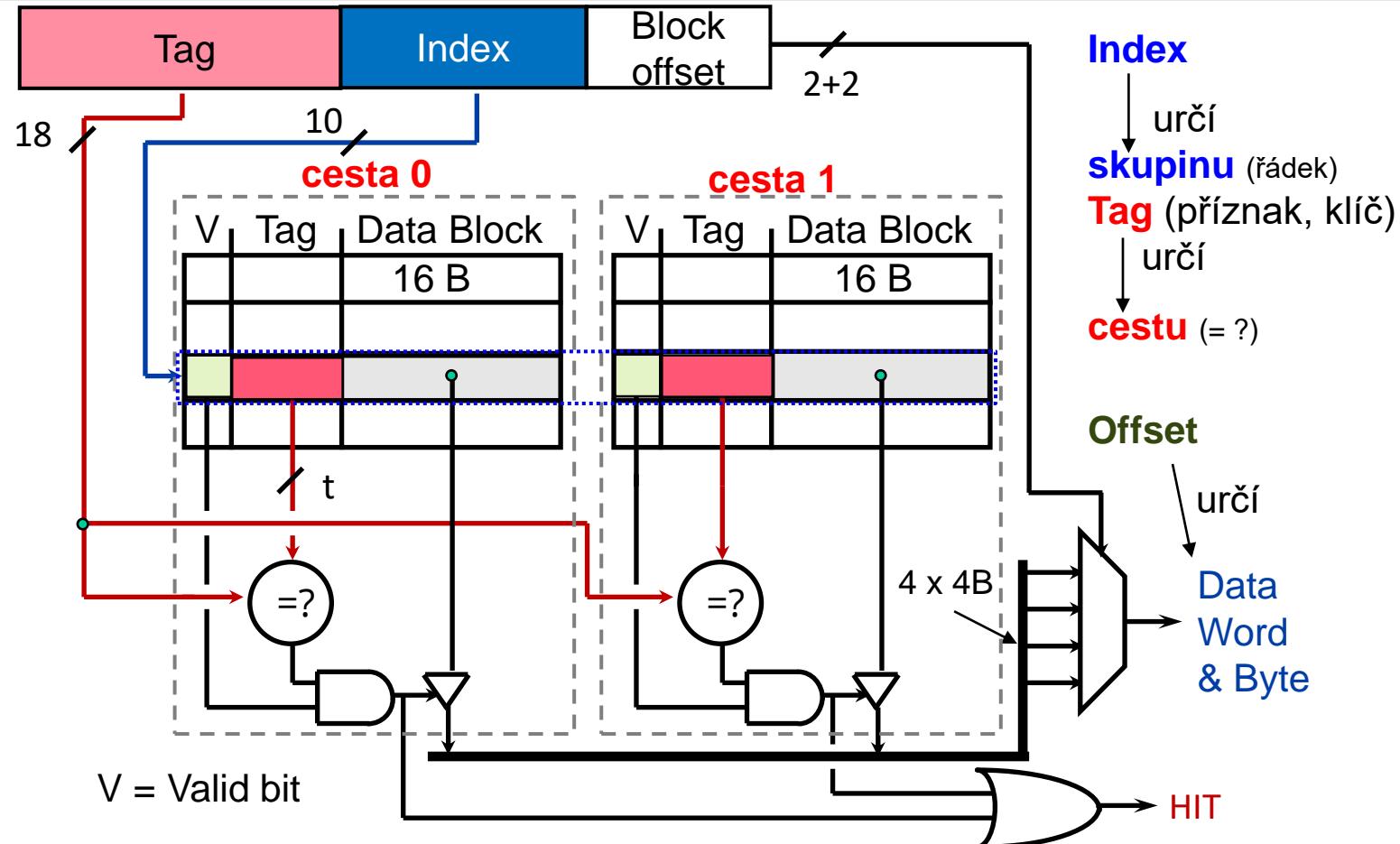
http://users.utcluj.ro/~sebestyen/_Word_docs/Cursuri/SSC_course_5_Scoreboard_ex.pdf



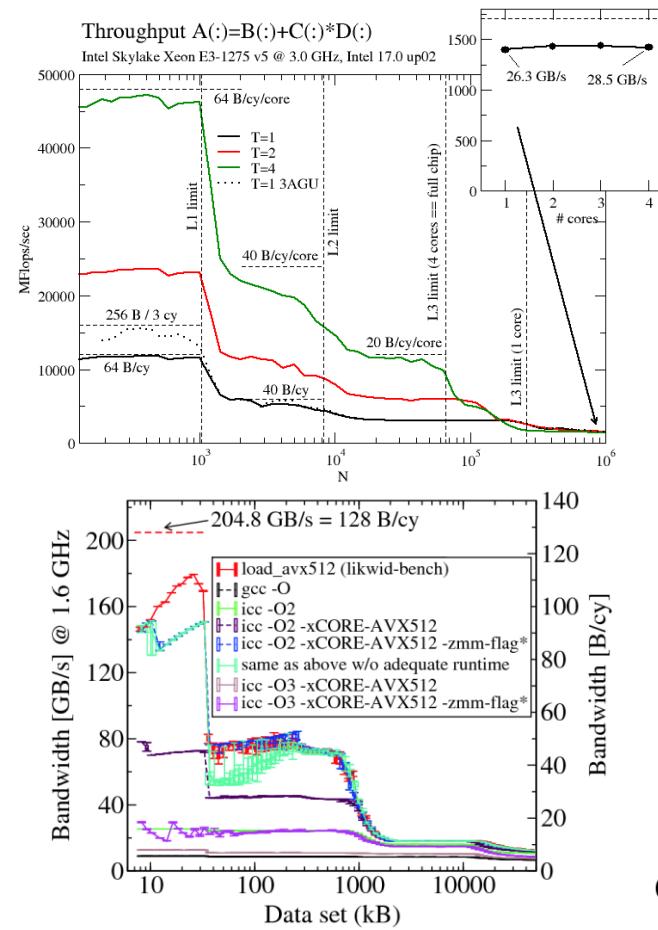
I Superskalární procesor – Back end



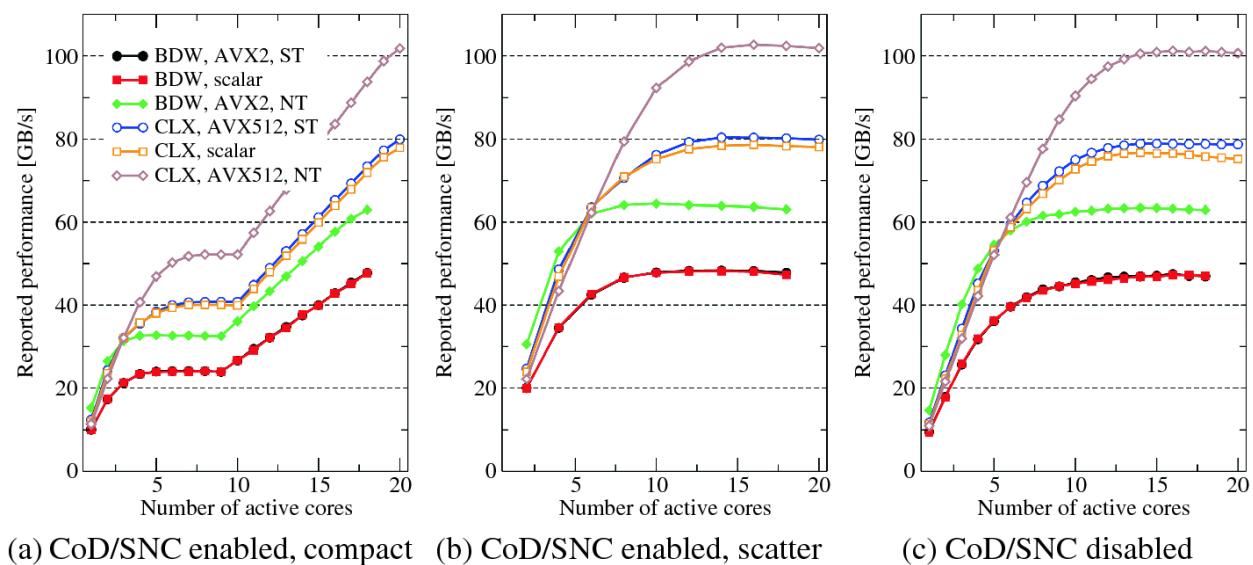
| Skupinově asociativní (SA) cache



Propustnosti a latence cache



- Stream benchmark testující propustnost paměti a cache
- Vlevo – jedno jádro
- Vpravo – více jader
- https://link.springer.com/chapter/10.1007/978-3-030-50743-5_21



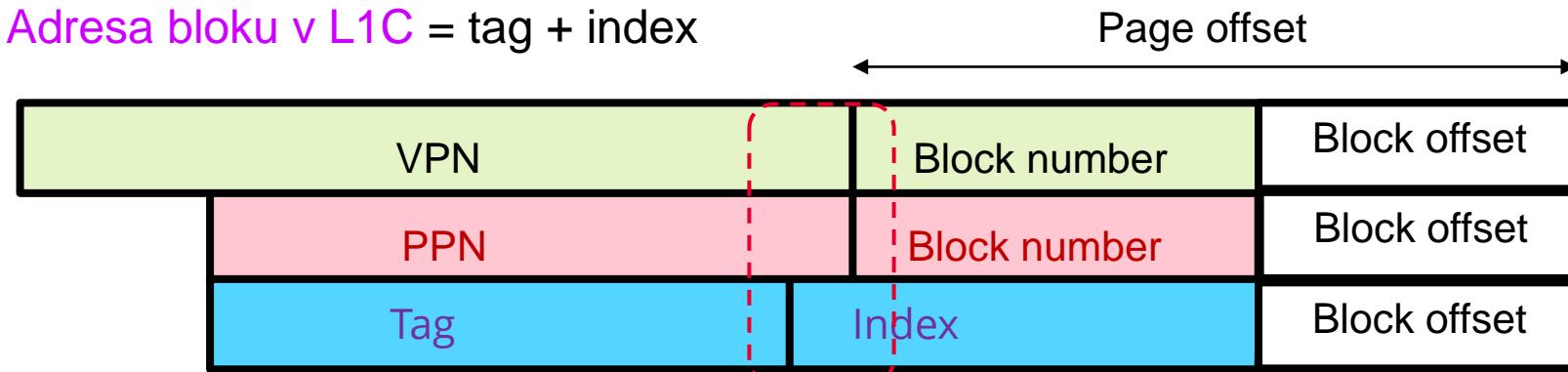
VA = Virtual Address = virtual page number **VPN** + Pg. offset

PA = Physical Address = phys. page number **PPN** + Pg. offset

Pg.offset = Block number (číslo bloku na stránce) + **block offset** (které slovo, byte)

Adresa bloku v paměti = PPN + Block number

Adresa bloku v L1C = tag + index



Velikost stránky: 4, 8 kB, ale i 64 kB, 2 MB, 4 MB

Cache index: 16 kB, 32 kB, 512 kB, 16 MB

- Spolupráce CPU s L1C vyžaduje co nejrychlejší přístup do cache.
- Jak dospět od VA k adrese bloku (**index, tag**) v L1C?
 - VA = virtuální adresa je k dispozici ihned
 - PA = fyzická adresa je k dispozici až po **překladu** VA
 - Která adresa by se mohla použít pro přístup?
- **Tři možnosti:**
 - P/P cache: fyzický index, fyzický tag
 - V/V cache: virtuální index, virtuální tag
 - V/P cache: virtuální index, fyzický tag
virtuální index se dá použít hned, paralelně s překladem VA.
 - P/V cache: fyzický index, virtuální tag - nepoužívá se, index by se musel získat překladem a čas by se neuspořil.

I Translation Lookaside Buffer (TLB)

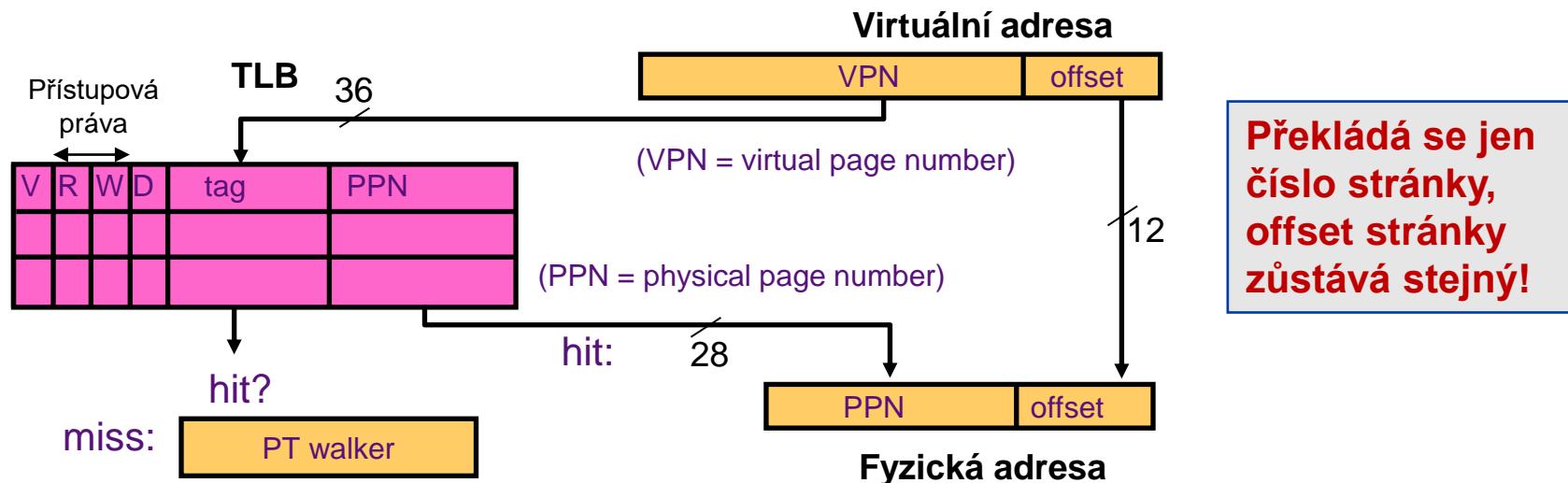
Překlad adresy je velmi drahý!

V hierarchické tabulce stránek stojí každý překlad několik přístupů do paměti.

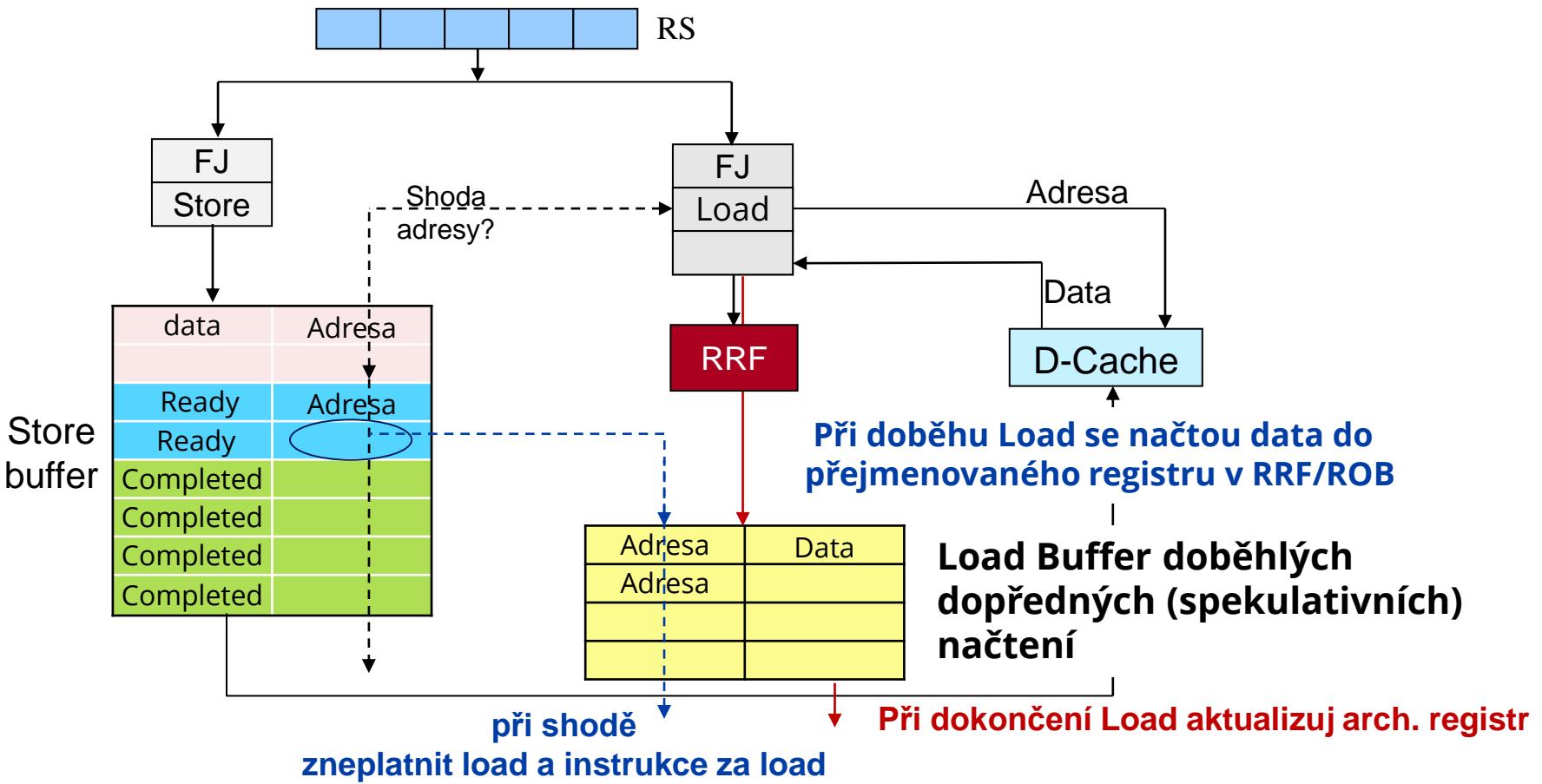
Řešení: *Překladová tabulka TLB na čipu* obsahuje sadu

aktuálně používaných dvojic {VPN, PPN}

- TLB hit \Rightarrow překlad za 1 takt,
- TLB miss \Rightarrow průchod PT k doplnění TLB



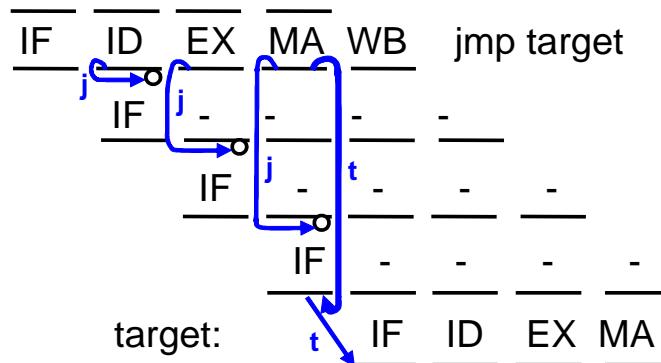
Out-of-order Load/Store jednotka



PREDIKCE SKOKŮ

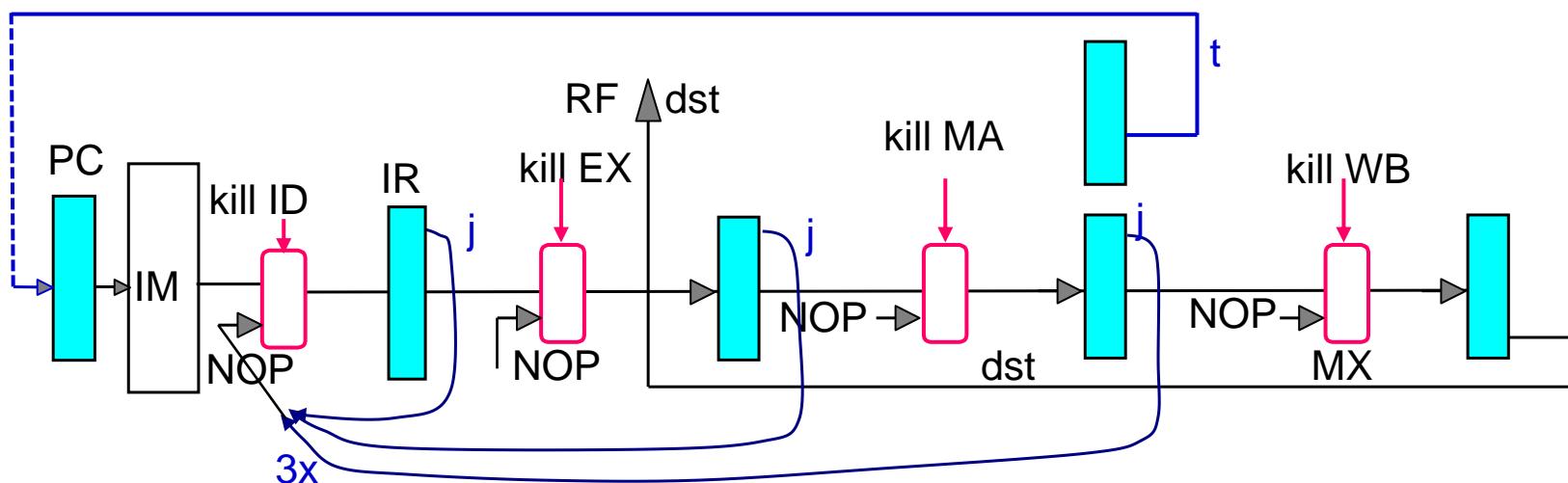
- V průměru je každá 6. – 9. instrukce skoková
 - nepodmíněný skok - jump **j**, jump register **jr**
- do podprogramu: jump and link, **jal**
 - podmíněný skok
 - **bnez/beqz** r1, target (test 1 registru ve stupni ID)
 - **bne/beq** r1, r2, loop (test 2 registrů ve stupni EX)
- Jaká data skok potřebuje: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

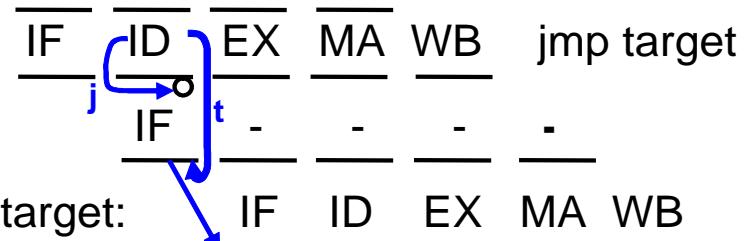
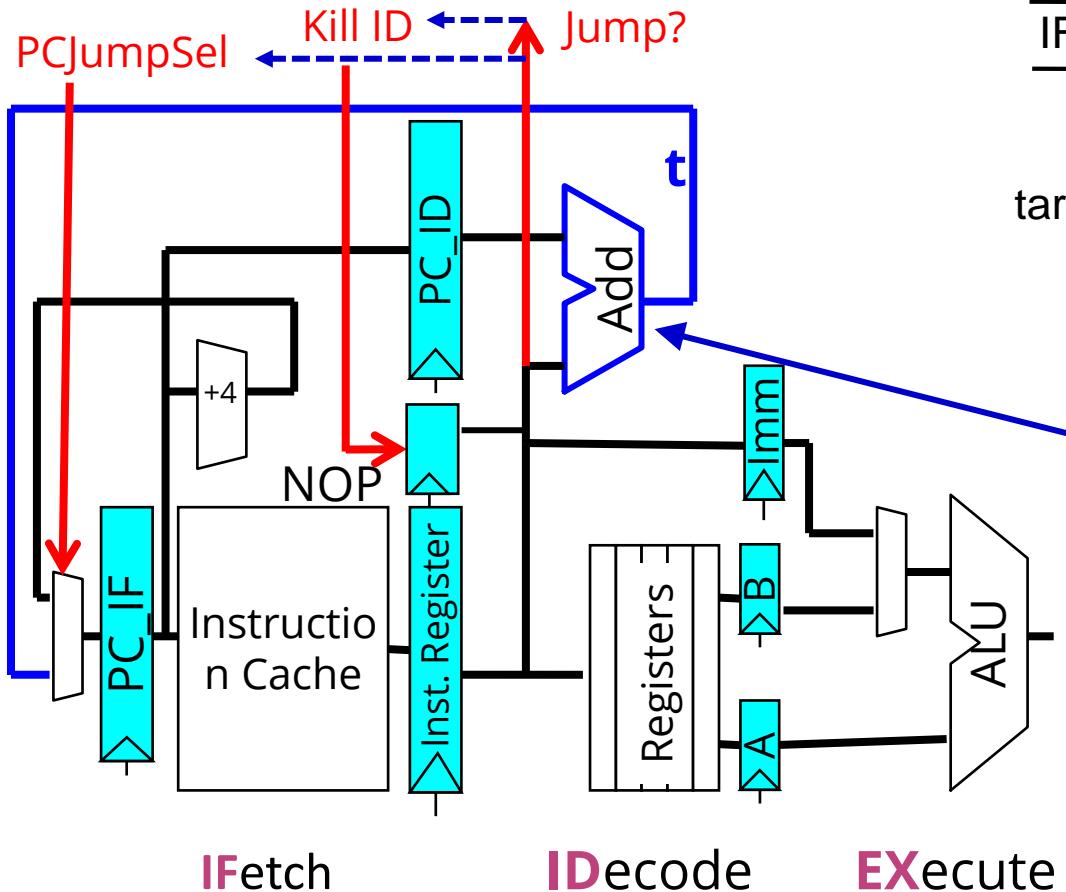
Výpočet cílové adresy (PC + rel. adresa) a podmínky je třeba co nejvíce urychlit!



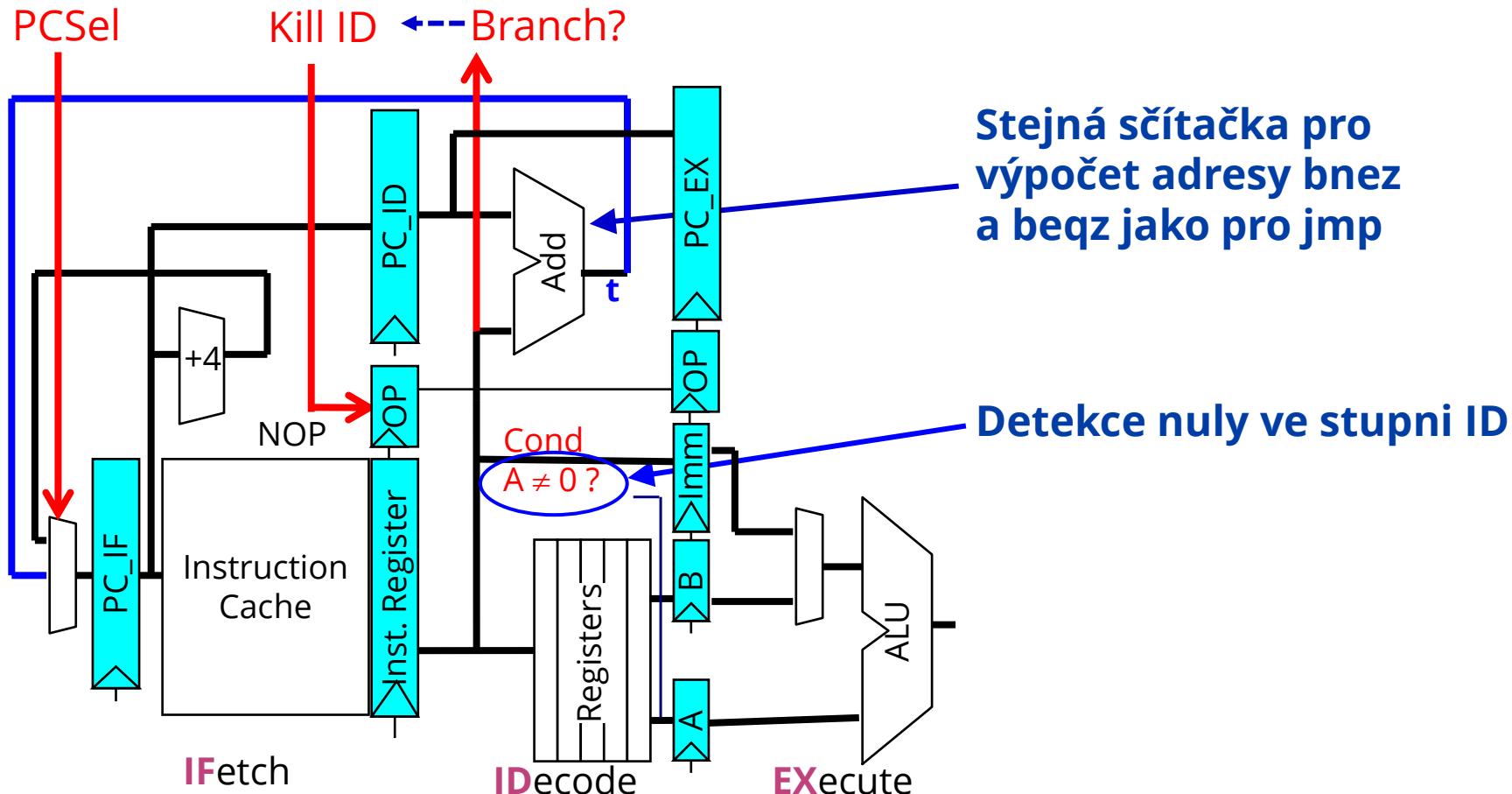
Adresa se spočte v EX, během MA se přepíše do PC → pokuta 3 takty.

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



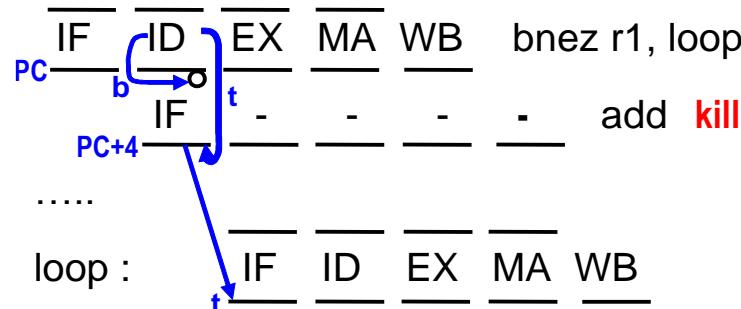


Nová sčítáčka pro výpočet adresy t ve stupni ID: pokuta jen 1 takt



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).

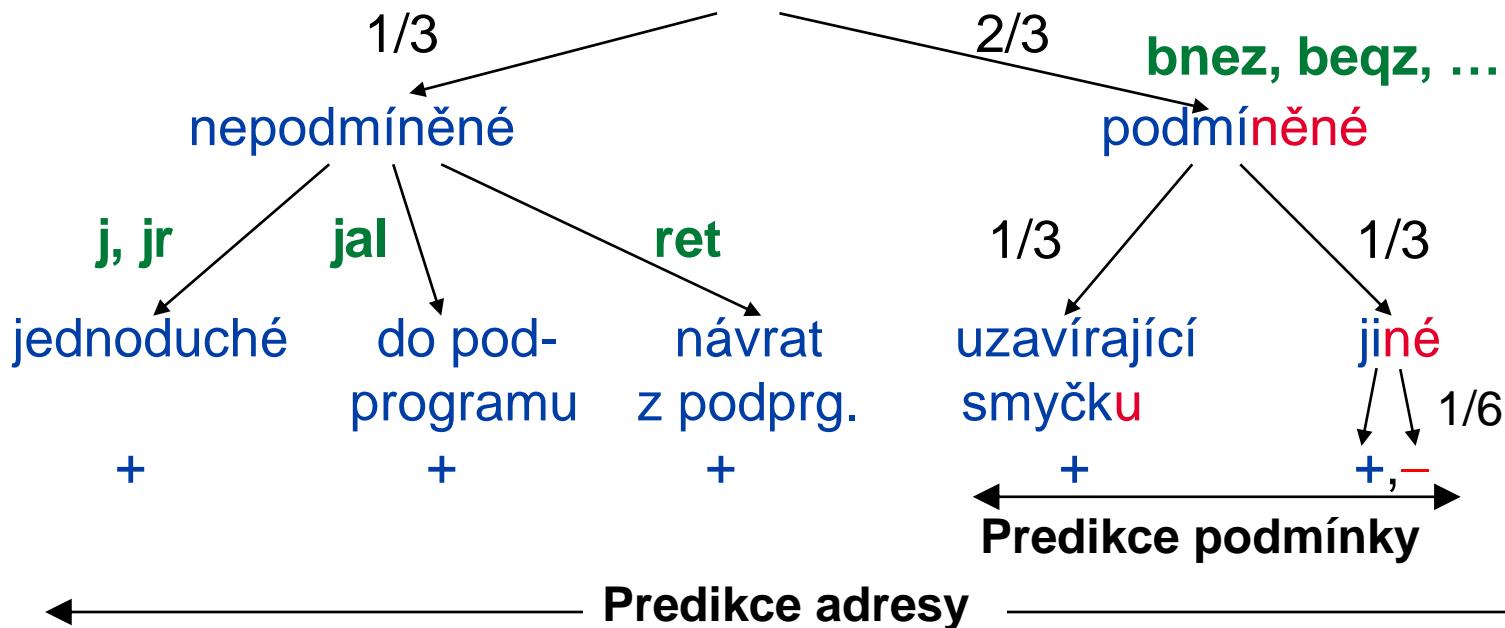
Pokuta pak bude:

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



Statistika:

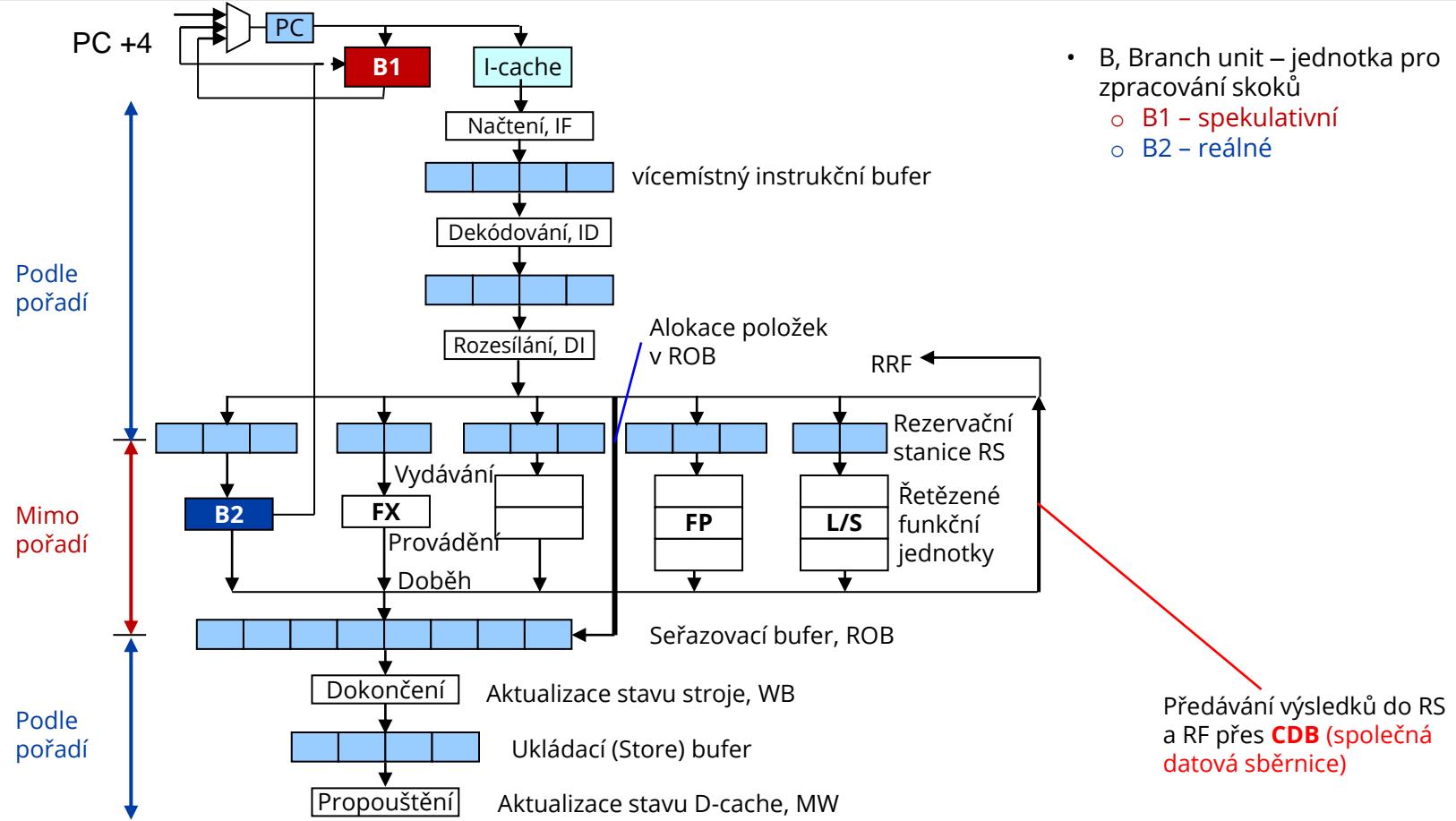
Skoky 20 % (univ.), 5–10 % (HPC programy)



Predikce podmínky:

- statická (podle testu $\neq 0$, >0 , ≥ 0 , směru skoku komplátorem – predict bit)
- dynamická (za běhu programu)

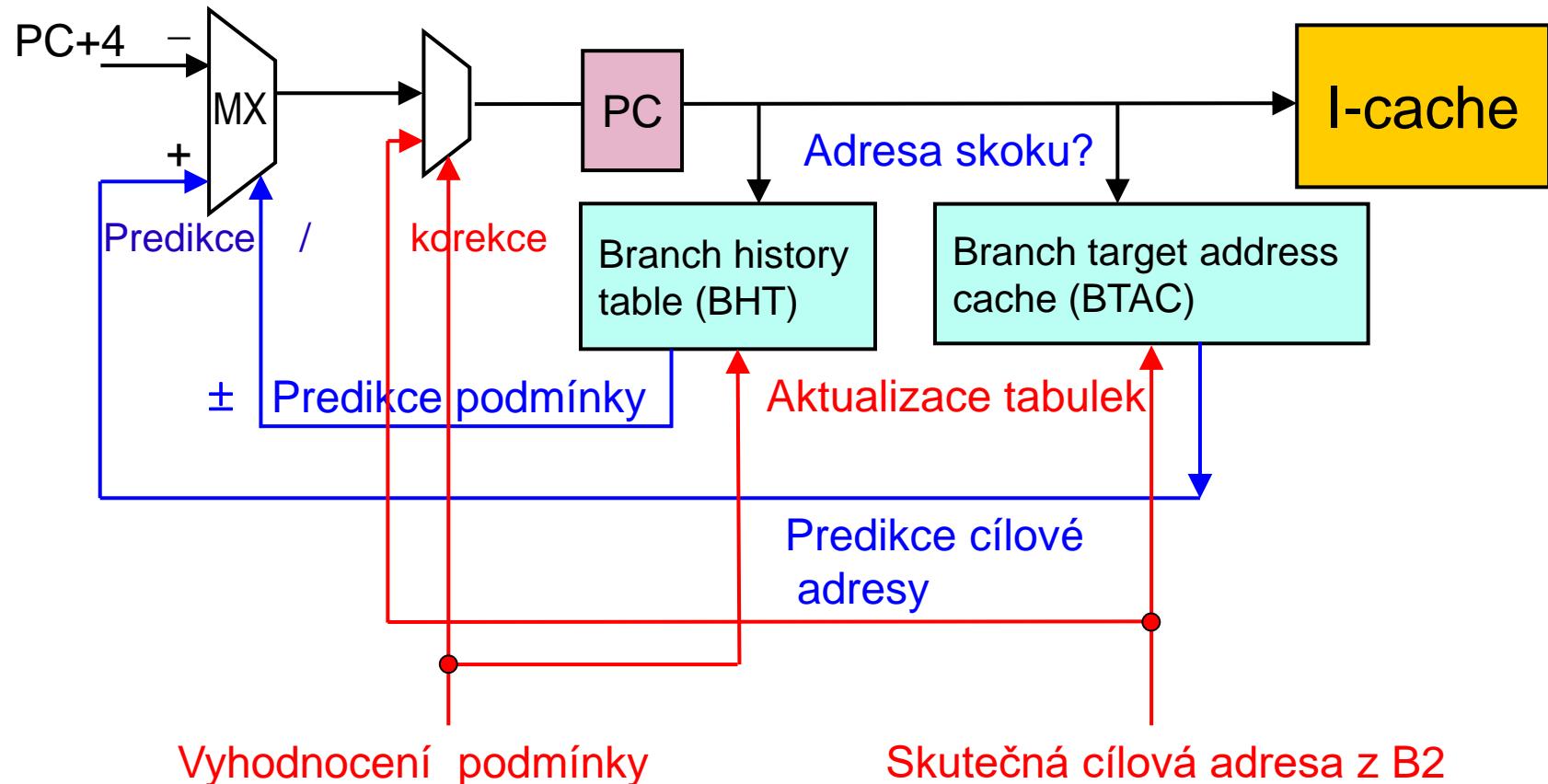
Superskalární procesor



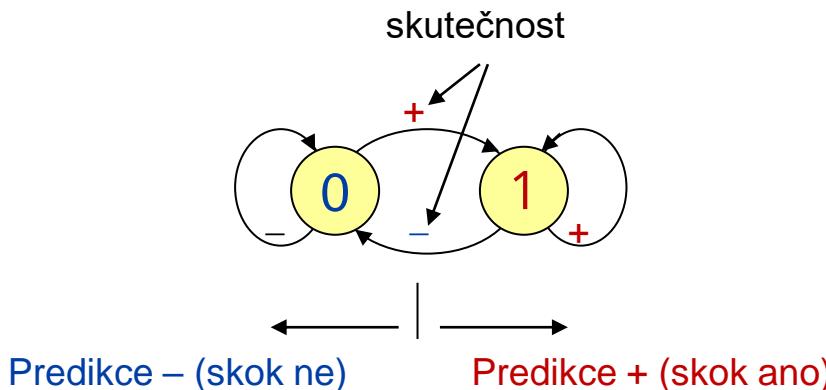
Pro provedení skoku potřebujeme předpovědět dvě věci:

- **Predikce podmínky skoku** – jen u podmíněných skoků
 - 1 bitový prediktor
 - 2 bitový prediktor spolupracující s **BHT** Branch History Table
 - 2 bitový prediktor spolupracující s **PHT** Pattern History Table
 - Korelační prediktory
- **Predikce cílové adresy** (cílové instrukce) u všech skoků
 - **BTB** Branch Target Buffer
 - **BTAC** Branch Target Address Cache
 - **BTIC** Branch Target Instruction Cache
 - **RSB** Return Stack Buffer (prediktor návratových adres)

Predikce podmínky a cílové adresy skoku (B1)



1 skok = 1 bit v BHT,
Branch History Table



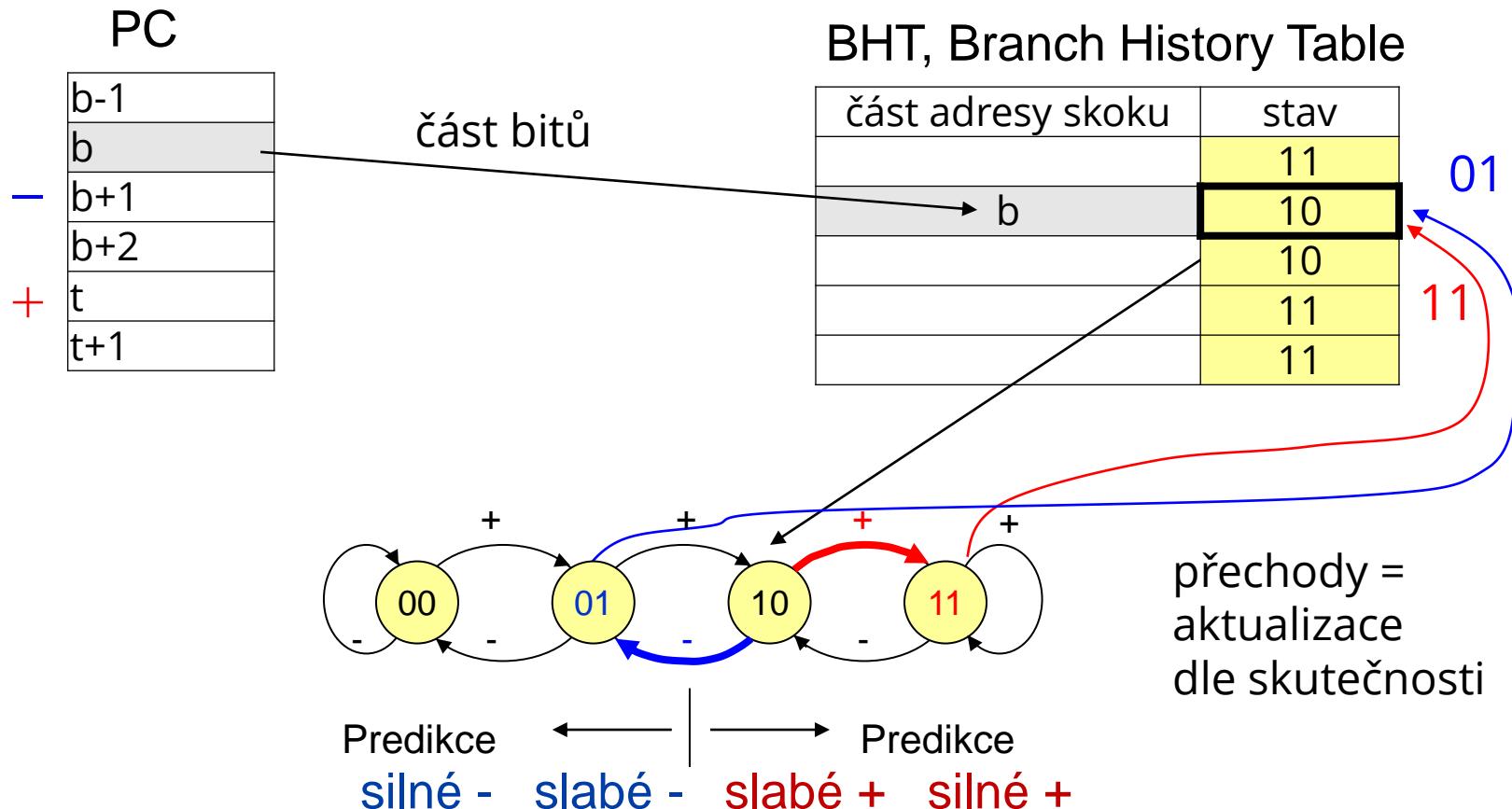
1 bitová predikce skoků:

- Když se skočilo, očekávám že se zase skočí (stav 1)
- Když se neskočilo, očekávám že se zase neskočí (stav 0).

Implementace:

- Pro každý skok je třeba uložit stav (1 bit) v tabulce BHT.
- **Kvůli rozměrům** je tabulka indexovaná jen vybranými bity PC nebo nějakou hashovací funkcí nad PC.

I 2 bitový prediktor podmínyky skoku



```

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        do_work(i, j);
    
```

Vnější smyčka, jiný skok

Skutečnost:

+++++− ... + ... +++++− ... + ... +++++− ... + ... +++++− ...

- **1 bitová predikce:**

−+++++− −+++++− −+++++− −+++++− −+++++−

2 chyby na 1 průchod vnitřní smyčky

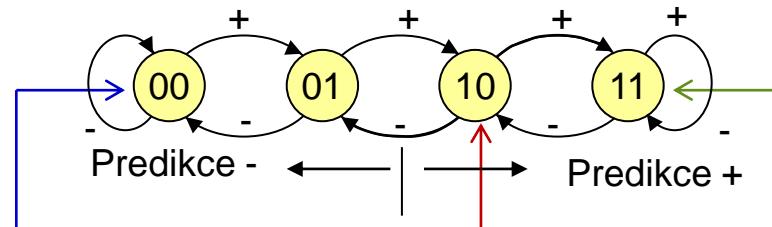
- **2 bitová predikce** (má jistou setrvačnost):

− − ++++− ++++++− ++++++− ++++++− ++++++−

1 chyba na 1 průchod vnitřní smyčky (zanedbáme-li 2 počáteční chyby
– význam počátečního nastavení prediktoru!)

Skutečnost:

— + — + — + — + — + + — ...



2 bitová predikce, počáteční stav silné –, levá smyčka

— — — — — — — — ... 50% úspěšnost

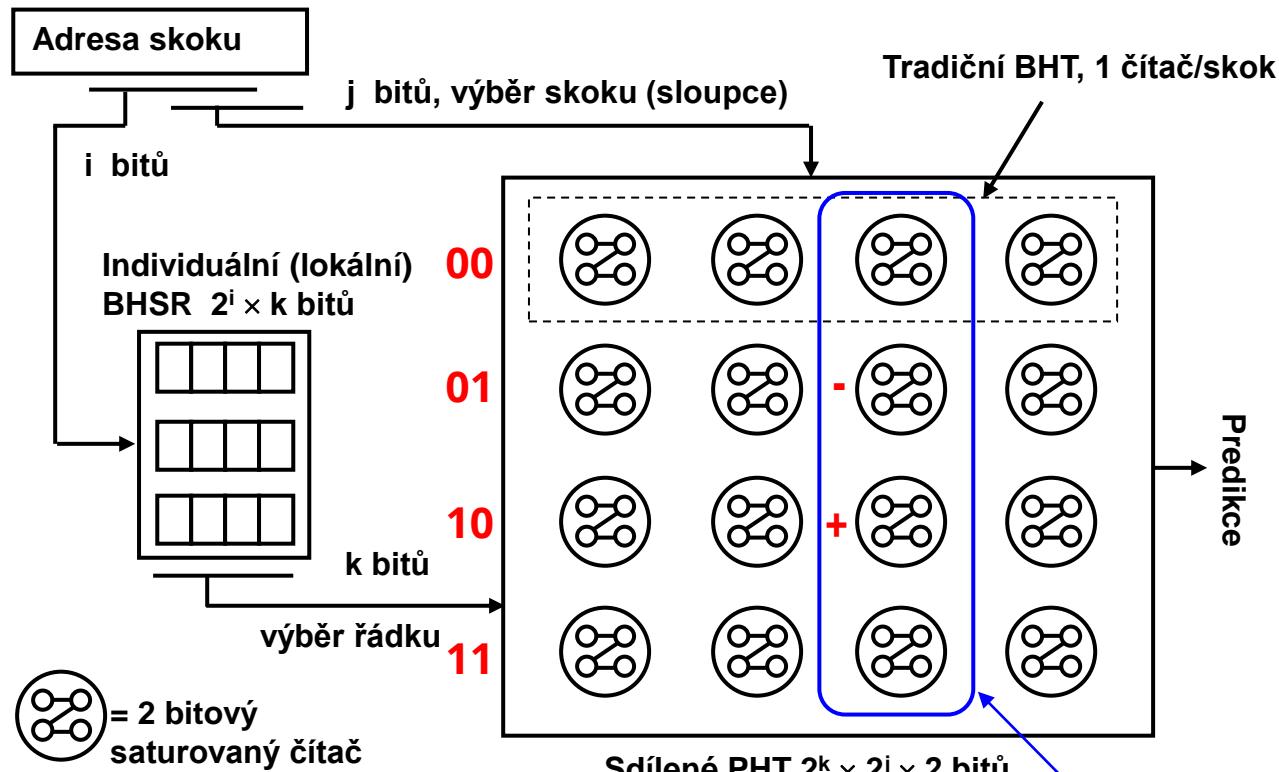
2 bitová predikce, počáteční stav silné +, pravá smyčka

+++ + + + + + + + + + ... 50% úspěšnost

2 bitová predikce, počáteční stav slabé +, střední smyčka

+ + + + + + + + + + ... nulová úspěšnost

Řešení (Yeh & Patt) adaptivní prediktor – podle posledních k výsledků každého skoku vybrat jeden z 2^k prediktorů a ten použít. ($k = 2$: prediktor **10** by předpovídal +, prediktor **01** by předpovídal -, další 2 prediktory **00** a **11** by se neuplatnily).



čítače odpovídající 2 bitovým vzorkům historie jednoho skoku

- Používá 2^k tradičních tabulek BHT. Tato 2D struktura čítačů se nazývá **Pattern History Table, PHT**.
- Pro daný **podmíněný skok** (identifikovaný obecně i bity PC) a podle k předchozích výsledků tohoto skoku, ukládaných do k -bitového posuvného registru historie skoků (**BHSR – Branch History Shift Register**) se vybere jeden BHT (řádek PHT); každý skok má svůj lokální BHSR.
- Část adresy skoku (j bitů) představuje index do vybrané BHT (sloupec PHT) a určí příslušný 2 bitový čítač.
- Nový obsah BHSR se získá vložením výsledku (1 bitu) příslušného skoku. PHT se aktualizuje jen u podmíněných skoků.

I Příklad: Adaptivní prediktor $k = 2$ bity historie

Skutečnost:

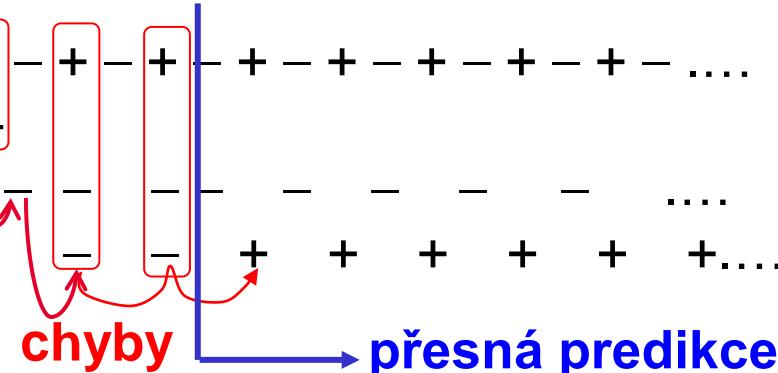
BHSR --: - + - + - + - + - + - + - + -

Čítač 00: - -

Čítač 01: - -

Čítač 10: - -

Čítač 11: - -



Skutečnost:

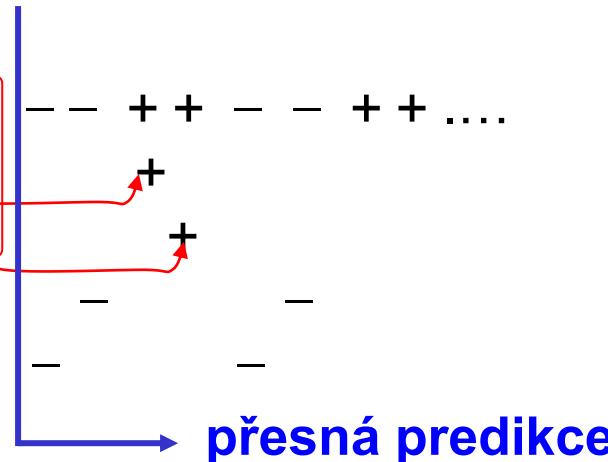
BHSR --: - - + + - - + + - - + + - - + +

Čítač 00: - -

Čítač 01: - -

Čítač 10: **chyby =
adaptace**

Čítač 11: - -



Umí se naučit přesně predikovat každou opakující se sekvenci
(adaptuje se na určitý vzorek)

- délky $\leq k + 1$ nejvýš po **třech** pokusech,
- délky $k + 1$ až 2^k , pokud všechny pod-sekvence délky k jsou různé.
- nepravidelné sekvence mají více špatných predikcí než u prediktoru bez BHSR.

Příklad: $k = 4$

000011 umí, protože 0000, 0001, 0011, 0110, 1100, 1000 jsou různé,

00001 umí, protože 0000, 0001, 0010, 0100, 1000 jsou různé,

000001 neumí, 0000, 0000, 0001 ... nejsou různé

- **Problémy adaptivních prediktorů:**

- Velikost prediktorů roste exponenciálně s počtem bitů historie.
- Neberou v úvahu možnou korelaci mezi provedením uvažovaného skoku a skoků na cestě k němu (např. vnořené smyčky).

- **Korelační prediktory**

- Nepoužívají lokální BHSR, ale jeden globální **GBHSR** pro všechny skoky.
- Tím zohledňují dynamický kontext posledně provedených skoků programu (a nikoliv výsledky jen jednoho skoku v čase).
- Při $k = 8$ až 16 je jejich přesnost lepší než 95 %.
- **Nevýhoda:** PHT je příliš velké a využití položek nerovnoměrné.

- **Statická** – Založena pouze a jen na instrukci (bit v OP code)
- **Dynamická** – Založena na historii vykonání (x-bitový)
- **Lokální** – Historie udržována jen pro danou instrukci (adaptivní)
- **Globální** – Historie udržována přes sekvenci skoků (korelační)

```
// generates random numbers from uniform distribution [1, 10]
for (size_t i = 0; i < data.size(); i++){
    data[i] = dist(rng);
}

Type sum = 0;
for (int r = 0; r < REPETITIONS; r++)
    for (int i = 0; i < SIZE; i++)
        if (data[i] < 6)
        {
            sum += data[i];
        }
```

- **Perf**

- \$ perf stat -e branch-misses ./filter
212 565 554 branch-misses
- \$ perf list # list all available counters

- **Intel Vtune**

- \$ ml VTune
- \$ ampxle-cl -collect uarch-exploration -- ./filter
- Branch Mispredict: 33.7% of Pipeline Slots

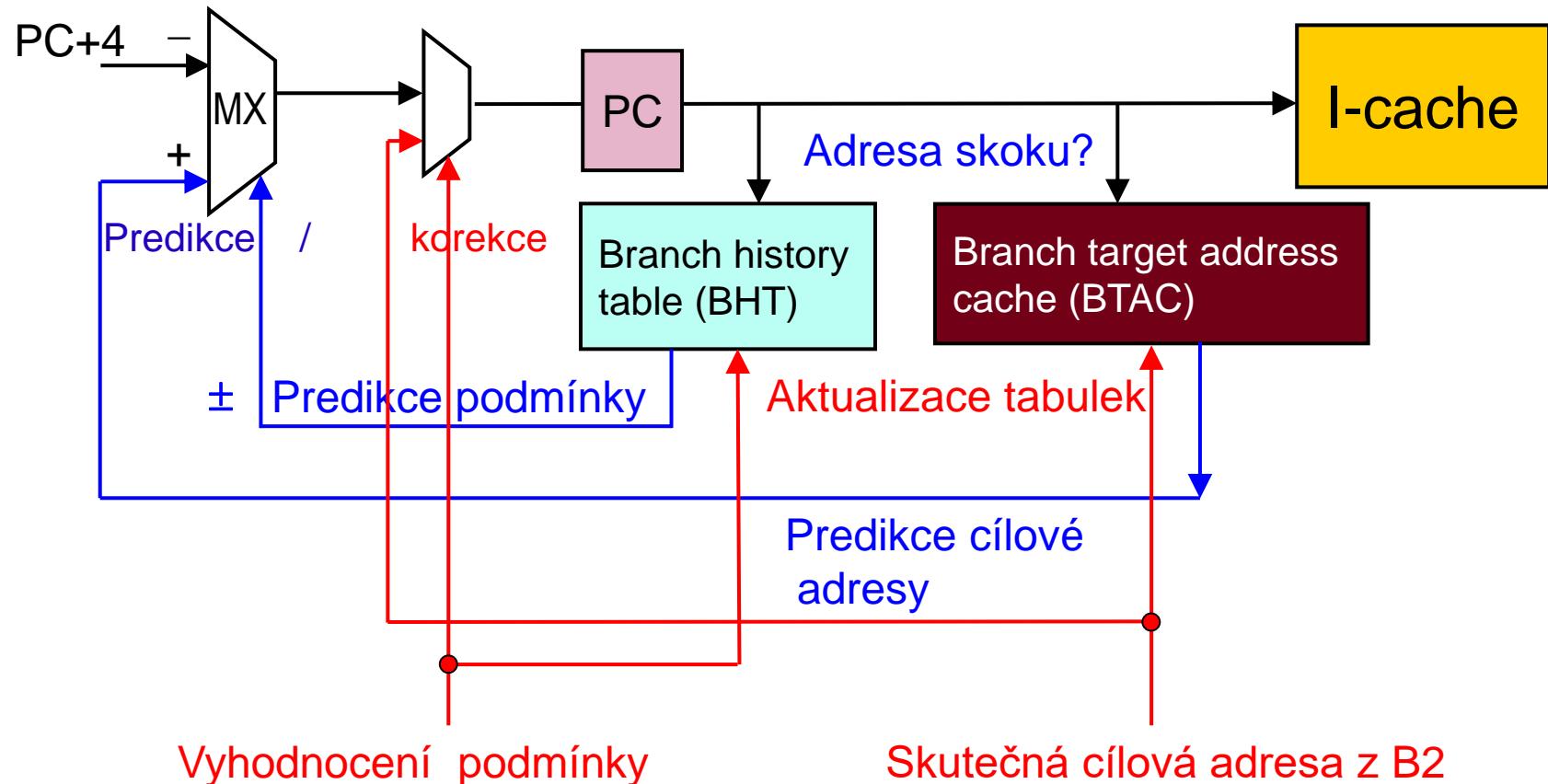
- **Programmatically PAPI**

- Více predikovatelná data
 - Seřadit pole
 - 212M branch-misses -> 7M
- Odstranění nebo nahrazení skoků
 - `sum += data[i] * (a < 6);`
 - `sum += (a < 6) ? data[i] : 0;`
- Compiler hints
 - `if (__builtin_expect(a < 6, 1)`
- PGO (profile-guided optimization)
- **Co když neznáme cílové adresy?**
 - Virtuální funkce, ukazatele na funkce, návraty z volání rutin

- Hned ve fázi IF, jakmile se rozpozná že PC ukazuje na skokovou instrukci (**b** nebo **j**), dá se **část adresy skoku** použít jako tag a index do malé cache, ve které jsou uloženy **předchozí cílové adresy skoků** (**t**, target = cíl).
- Tato malá cache se označuje jako
 - **BTB** (Branch Target Buffer) nebo
 - **BTAC** (Branch Target Address Cache)
- **Nepřímé skoky (jr)**: skáče se na různá t , v BTB může být
 - jen poslední t , takže dost špatných predikcí nebo
 - všechna t (Core 2) – oddělený BTB využívající GBHSR
- **Návraty z funkcí (ret)** se řeší jinak, viz dál (RSB)
- **Návraty z konce smyček** – oddělený BTB a loop counter

Velikost BTB:
512 až 8192 položek

Predikce podmínky a cílové adresy skoku (B1)



Pokud jsou v BTAC pouze **cílové adresy t** uskutečněných skoků, je s predikcí adresy automaticky spojena 1 bitová predikce podmínky. (skok se udělal: položka v BTAC existuje → příště skoč; skok se neudělal: položka se vyřadí → příště neskoč).

Častěji se k BTAC připojuje BHT se 2 bity.

- Když BHT říká „**skákat se bude**“ a **t existuje** v BTAC, **skočí se**
- Podobně „**skákat se nebude**“ a **t neexistuje**..., **neskočí se**
- Když BHT říká „**skákat se nebude**“ a **t existuje** v BTAC, **neskočí se**, pokračuje se na $b + 4$ (**predikce BHT je silnější**);
- Když BHT říká „**skákat se bude**“ a **t neexistuje** v BTAC:
nevíme, že jde o skok (alias v tabulce)
musí se čekat, až se spočítá efektivní adresa skoku a podmínka.

Aktualizace BHT a následně BTAC podle skutečnosti.

Predikce **podmíněných** skoků má 2 kroky:

- spekulaci (jednotka B1) v době IF:
 - $b \rightarrow BTB$, $b + 4 \rightarrow PC$ (predikce -)
 - $b \rightarrow BTB$, $t = BTB(b) \rightarrow PC$ (predikce +)
- její ověření: adresa a podmínka platí? (jednotka B2).

Než se skok ověří, provádějí se další instrukce včetně skoků s predikcí.

- Tyto **spekulativní instrukce** (jejichž provedení nebo storno závisí na správnosti predikce) je třeba identifikovat **příznaky**.
- Při **správné predikci** je ze spekulativních instrukcí příznak odstraněn.
- Při **špatné predikci** je špatná větev ukončena a příznaky se použijí k odstranění těchto instrukcí z buferů ROB i RS.

- 85 % nepřímých skoků jsou **nepodmíněné skoky *ret*** z konce volané funkce zpět do hlavního programu.
Pokud je adresa návratu na **zá sobníku v paměti D-cache**, instrukce *ret* provádí její načtení a **WB** do PC.
- **Běžná predikce pomocí BTB by byla nepřesná** (funkce se volá z více míst → více návratových adres, takže při návratu jinam než na předešlou adresu → BTB miss)
- **Řešení:** ukládáním návratových adres do **malého zásobníku přímo v CPU** (Return Stack Buffer, RSB nebo Return Address Stack, RAS)
 - Při zjištění v ID, že instrukce je návrat, načte se adresa návratu z RSB, hned v dalším taktu je již v PC a pokračuje hlavní program.
 - Tím se současně redukuje zpoždění při načtení adresy návratu z D-cache.

| | Bez RSB | | | | | |
|------------------|--------------|----|----|----|----------|----|
| ret | IF | ID | EX | MA | WB | |
| itarget | | | | | IF | IF |
| pokuta = 4 takty | | | | | WB do PC | |
| | S pomocí RSB | | | | | |
| ret | IF | ID | EX | MA | WB | |
| itarget | | IF | IF | | | |
| pokuta = 1 takt | | | | | | |

Za předpokladu, že instrukcí návratu je v programu 5 % a neuvažujeme-li pokuty u dalších instrukcí (CPI = 1), je zrychlení **skalární CPU** dosažené pomocí RSB $\sim 14\%$:

$$S = \frac{CPI + 5\% * 4 \text{ takty}}{CPI + 5\% * 1} = \frac{1 + 0,2}{1,05} = 1,14$$

- Skok na konci smyčky se chová tak, že z n průchodů
 - jde $n-1$ jedním směrem (na začátek)
 - 1 průchod na další instrukci.
- Skoky na konci (vnořených) smyček je možné predikovat samostatně pomocí počítadla iterací – **loop counter (LC)** a **oddělené cache cílových adres skoků (BTB)** jen pro smyčky s přídavnou informací o skoku (n).
- Při prvním průchodu LC napočítá do n , při dalších průchodech se LC porovnává s n a predikuje se exit pro n -tý běh.
- Např. 6 bitový čítač LC umí přesně predikovat skoky až do 64 iterací smyčky.

Instrukce za podmíněným skokem jsou vydávány, dynamicky plánovány i prováděny, jako kdyby predikce byla správná.

Spekuluje se i s více skoky za sebou, i se 2 skoky v 1 taktu. Vyžaduje to:

- dynamickou predikci skoků
- místo pro spekulativní výsledky (hodí se ROB! nebo RRF),
ROB: [typ instrukce, dst reg., stav instrukce, hodnota]
- označení operandů a instrukcí „spekulativní/potvrzený“,
- storno špatně spekulovaného úseku a repete,
- ignorování výjimek dokud není jasné, že k nim dojde.

- **Spekulativní zpracování instrukcí po predikci skoku:**

- Spekulativní instrukce v ROB jsou označeny (1 bit) a nemohou být propuštěny, dokud se nerozhodnou příslušné predikované skoky.
- CPU zpracuje výjimku pouze u potvrzené instrukce na čele ROB.

- **Spekulativně lze také načítat data dopředu:**

- Spekuluji, že se předem načtená data do použití již nezmění.
- Jsou použity instrukce nevyvolávající výjimky, např. **sld** (spekulativní load) a existence výjimek se testuje odděleně později.

PŘEDNAČÍTÁNÍ INSTRUKCÍ

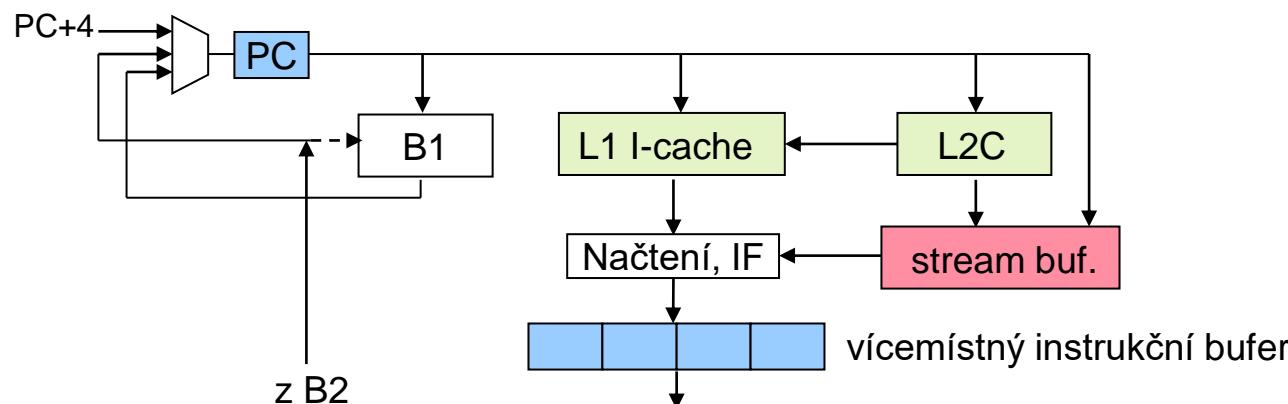
Fakta:

- OOO procesory načítají více instrukcí najednou.
- Instrukce jsou v I-cache nebo v L2C (nebo výš).
 - V bloku I-cache 64 B je např. sekvence 8 instrukcí po 8 B.
 - Přes hranice bloků je třeba načítat bez dalšího zpoždění (i při výpadku).
- Fragmentace bloků I-cache
 - Načítání sekvenčního sledu instrukcí končí skokem.
 - Z výchozího a cílového bloku I-cache se tak při skoku použije jen fragment, při špatné predikci dojde k výpadku.
- Instrukce načtené v 1 taktu mohou obsahovat více skoků, jejichž predikce ovlivní další načítání.

Závěr:

- K vyloučení výpadků v L1C je třeba načítat instrukce z L2C do L1C dopředu před jejich použitím – přednačítat.

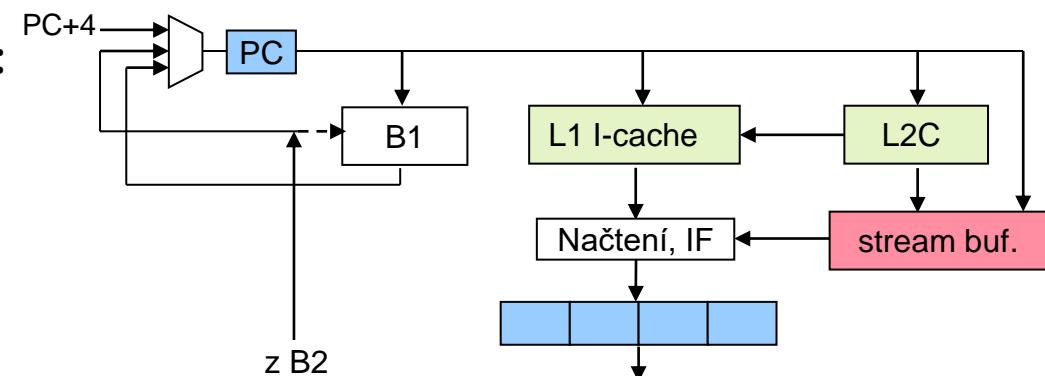
- Autonomní jednotka, která zásobuje instrukcemi další stupně linky
- **Tato jednotka integruje více funkcí:**
 1. predikci skoků
 2. přednačítání instrukcí
 3. přístupy do více bloků cache, jejich spojování
 4. dočasné uložení instrukcí v buferu



- **Typicky procesor při výpadku v I-cache načítá z L2C 2 bloky:**
 - žádaný blok do I-cache
 - s předstihem následující blok na adresu získané inkrementací PC nebo predikcí do **instrukčního streamového buferu**.

- **Je-li žádaný blok ve stream buferu:**

- žádost do cache je zrušena
- blok je načten ze stream buferu
- je vydána žádost na **další přednačtení** (jen při 1. přístupu do bloku).



- **Načítání do vícemístného instrukčního buferu má ještě další účel:**

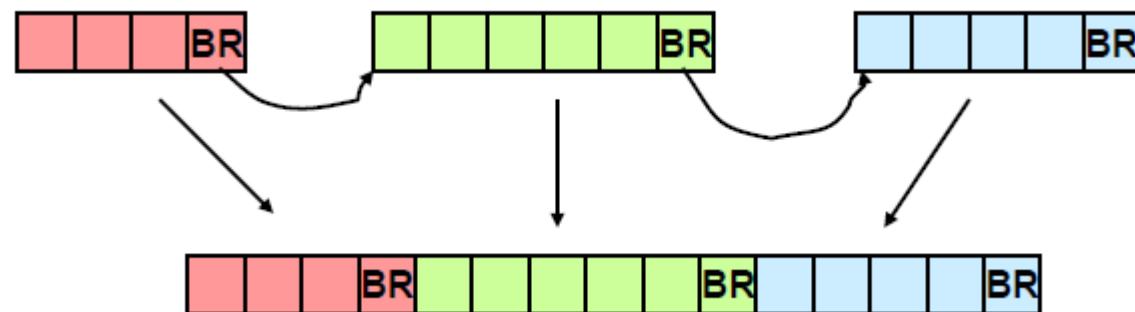
- jednotka načítání může **pospojovat** a **zkombinovat** instrukce z bloků cache do front (buferů) a vytížit tak lépe dekodéry (načítání přes hranice bloků, fragmenty bloků vlivem skoků).

Problémy s načítáním instrukcí:

- Spojování instrukcí z různých bloků zvyšuje latenci.
- Při načítání skupin více než 4 instrukcí za takt je často ve skupině více skoků. Je třeba je predikovat v 1 taktu.

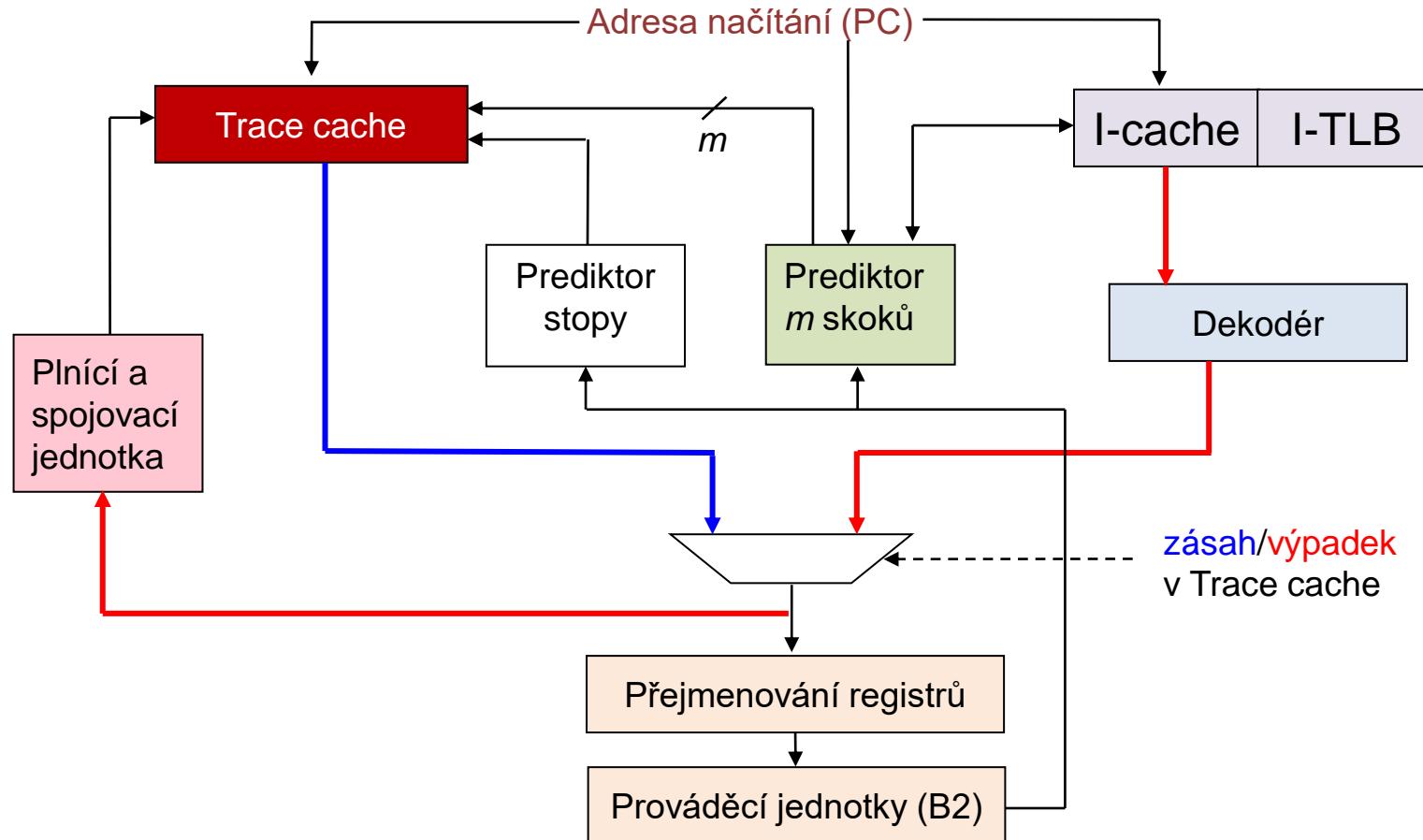
Řešení:

- Statické uspořádání instrukcí programu (v I-cache) nahradit **dynamickým pořadím** (v doplňkové Trace cache). Tj. sbalit několik nenavazujících základních bloků do jednoho souvislého bloku (stopy) v **Trace cache**.



- **Základní bloky instrukcí (zakončené skokem) se načítají**
 - na začátku programu z I-cache pomocí **jednotky načítání**, přičemž se pomocí **plnící jednotky** spojují a zapisují i do TC
 - při zásahu v TC se načítá stopa z TC
 - při výpadku v TC postup jako na začátku.
- Přístup do TC probíhá paralelně s přístupy do I-cache a do BTB s použitím aktuální adresy načítání.
- **Predikce několika skoků + adresa načítání** se porovnává s reálnou sekvencí uloženou v TC.
- Podle výsledku porovnání máme zásah nebo výpadek v TC.

Načítání instrukcí s využitím Trace cache



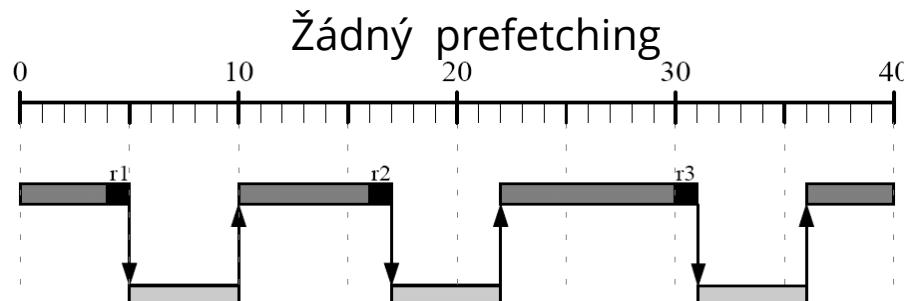
- Jedno načtení stopy dodá hned několik základních bloků.
- **Délka stopy v TC je omezena**
 - jistým max. počtem instrukcí (např. $n = 16$)
 - počtem skoků predikovatelných v 1 taktu m (např. $m = 3$)
 - nepřímým skokem, návratem nebo výjimkou.
- Blok paměti **trace cache** tak uchovává segment dynamické instrukční stopy přes několik provedených větvení.
- Na rozdíl od I-cache, uchovává **trace cache logicky navazující instrukce ve fyzicky navazujících paměťových místech**.

- **Zásah v trace cache nastává když**
 1. adresa načítání se shoduje s tagem stopy
 2. a současně predikce skoků se shodují s flagy skoků
(tj. s tím jak již byly skoky reálně ne/provedeny)
- **Při zásahu v TC** se celá stopa instrukcí nemusí dekódovat, cache instrukcí se neúčastní.
- **Při výpadku v TC**
 - se načítání provádí z I-cache do plnící jednotky, jeden základní segment za druhým.
 - Plnění skončí po n instrukcích nebo když se detekoval m -tý skok, nepřímý skok či návrat.
 - Jsou generovány postupové adresy stopy, flagy a masky skoků a obsah plnící jednotky je přepsán do TC.

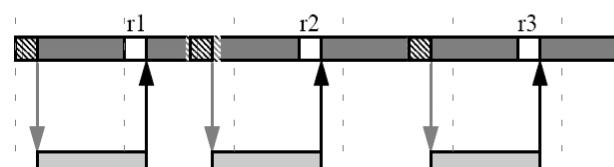
PŘEDNAČÍTÁNÍ DAT

- Podobně jako u instrukcí, než aby se čekalo na vyřízení výpadku (tj. načtení dat z paměti do cache), přednačítání provede načtení dopředu, čímž se přístup do paměti překryje s výpočtem.
- Může být implementováno v **HW nebo v SW**
- **Otázky:**
 - **kdy** je přednačtení iniciováno
 - **kam** jsou data přednačtena
 - do které úrovně cache, prefetch buferu nebo i registru
 - **jaká** je velikost dat u přednačítání:
 - bloky z paměti do D-cache
 - bloky s rozestupem (HW), slova spekulativně (SW)

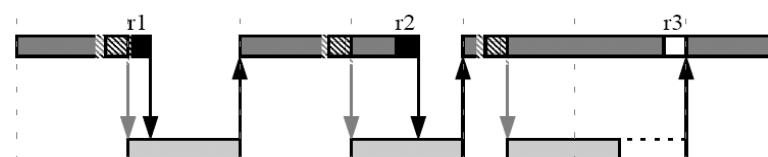
- Aby bylo **užitečné**, musí být **just-in-time**.
 - je-li příliš brzy, může být předem načtený blok ještě před použitím vyměněn nebo modifikován jinou CPU; (tj. **neužitečné** přednačítání)
 - je-li příliš pozdě, musí se na data nějakou, i když kratší dobu, čekat.
- **Špatné načasování** přednačítání má **vliv jen na výkonnost, ale ne na správnost**
 - přednačtená data mohou v cache nahradit data, která ještě procesor užívá → výpadek navíc
 - snaha využít všechna slova z bloku.



Perfektní prefetching



Degradovaný prefetching



- Mikroprocesory mají pro **přednačítání strojové instrukce**
 - mohou být vloženy komplátorem nebo programátorem při optimalizaci
- Instrukce přednačítání **nemůže vyvolat výjimku** (přednačtení nepokračuje).
- Přednačítání **může předbíhat** regulérní rozpracované operace v cache.
- **Přednačítání** má také svou **režii** (přídavné instrukce, použití jistých registrů pro adresy, zabírá část propustnosti paměti), která znamená vyšší provoz mezi CPU a pamětí.
- U přednačítání se testuje, zda daný blok v cache už je nebo ne.
Provede se **jen když tam není**.
- **Nejčastěji** se používá u smyček počítajících **s velkými poli** (vědecké výpočty s predikovatelnými vzory přístupů do polí).
- Existuje též přednačítání ukazatelů (průchod stromy, seznamy).
- SW přednačítá data nikoliv slepě, ale jen ta data, která budou velmi pravděpodobně potřeba.
- Vyžaduje neblokující cache.
- Může se aplikovat mezi libovolnými úrovněmi paměťové hierarchie. My si ukážeme jen do L1C.

Bez přednačítání

```
for (i = 0; i < N; i++)
{
    sum += a[i] * b[i];
}
```

- **Předpoklad:**
 - v bloku cache jsou 4 prvky vektorů
 - kód generuje 2 výpadky na 4 iterace

Naivní přednačítání

```
for (i = 0; i < N; i++)
{
    fetch(&a[i+1]);
    fetch(&b[i+1]);
    sum += a[i] * b[i];
}
```

- V bloku cache jsou 4 prvky a[0], a[1], a[2], a[3]
- přednačítání každého zvlášť je proto zbytečné

```

for (i = 0; i < N; i+=4) {
    fetch(&a[i+4]);
    fetch(&b[i+4]);
    sum += a[i] * b[i];
    sum += a[i+1] * b[i+1];
    sum += a[i+2] * b[i+2];
    sum += a[i+3] * b[i+3];
}

```

- při první iteraci výpadky
- zbytečné přednačítání při poslední iteraci
 $i = N - 4$
- rozbalení tolikrát, kolik je slov v bloku (4).

| | | |
|---|---|--|
| $i = 0$
fetch(&a[4]);
fetch(&b[4]);
sum += a[0]*b[0];
sum += a[1]*b[1];
sum += a[2]*b[2];
sum += a[3]*b[3]; | $i = 4$
fetch(&a[8]);
fetch(&b[8]);
sum += a[4]*b[4];
sum += a[5]*b[5];
sum += a[6]*b[6];
sum += a[7]*b[7]; | \dots
$i = N-4$
fetch(&a[N]);
fetch(&b[N]);
sum += a[N-4]*b[N-4];
sum += a[N-3]*b[N-3];
sum += a[N-2]*b[N-2];
sum += a[N-1]*b[N-1]; |
|---|---|--|

```
fetch (&sum);  
fetch (&a[0]);  
fetch (&b[0]);  
  
for (i = 0; i < N-4; i+=4)  
{  
    fetch (&a[i+4]);  
    fetch (&b[i+4]);  
    sum += a[i]*b[i];  
    sum += a[i+1]*b[i+1];  
    sum += a[i+2]*b[i+2];  
    sum += a[i+3]*b[i+3];  
}  
  
for (i = N-4; i < N; i++)  
    sum = sum + a[i]*b[i];
```

- Instrukce prefetch nemají registrový operand
- Blok obsahující slovo s uvedenou adresou je načten z té úrovně, kde právě je, do D-L1C, případně do cache zadанé úrovně
- **Implicitní předpoklad, že 1 iterace (4 x výpočet sum) stačí překrýt latenci paměti**
- Poslední 4 iterace pracují s daty přednačtenými již dříve

- Programátor ani kompilátor nemusí zasahovat.
 - Není nutno modifikovat programy, žádné instrukce navíc.
 - Může využít informací z běhu programu ke zefektivnění přednačítání.
 - Generuje ale více zbytečných přednačtení než SW způsob (znečisťuje cache).
-
- **Varianty:**
 - sekvenční přednačítání (s rozestupem 1)
 - přednačítání s libovolným rozestupem
 - **Neumí nepravidelné vzory přístupů k datům**

- Po načtení bloku i se přednačte sousední blok $i+1$ (OBL, One-Block Lookahead).
Zvládne smyčky s $i++$, ale už né $i--$.
- Jestliže blok již v cache je, přednačtení není iniciováno.
- **Existují 2 implementace**

1. Přednačtení při výpadku:

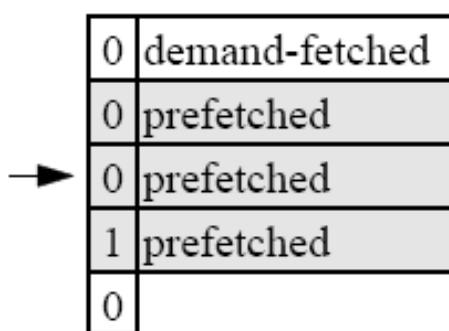
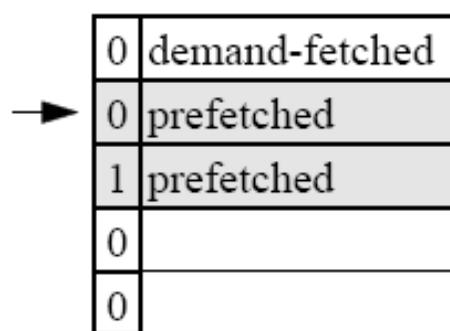
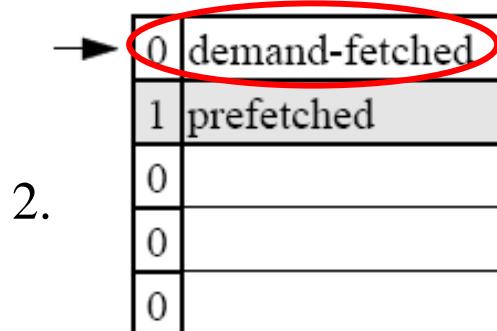
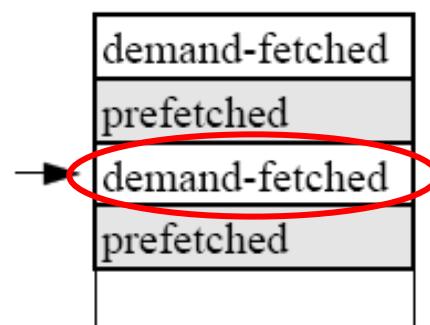
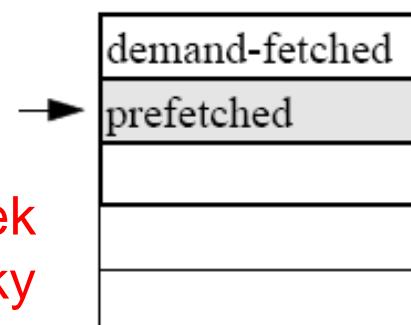
- Při výpadku v bloku i se načte blok i a přednačte $i+1$.
- Jak se to liší od 2x většího bloku cache? – Ten by se musel přesunout nebo zneplatnit celý, větší možnost falešného sdílení.

2. Přednačtení s příznakem (jak se mě dotkneš, přednačti další):

- při výpadku v bloku i se načte blok i (s příznakem 0) a přednačte se blok $i+1$ (s příznakem 1)
- při zásahu v bloku $i+1$:
 - s příznakem 1 (první zásah): změň příznak na 0 a přednačti blok $i+2$ (s příznakem 1)
 - s příznakem 0 (druhý a další zásah): žádná akce

I | Porovnání obou přednačítání OBL

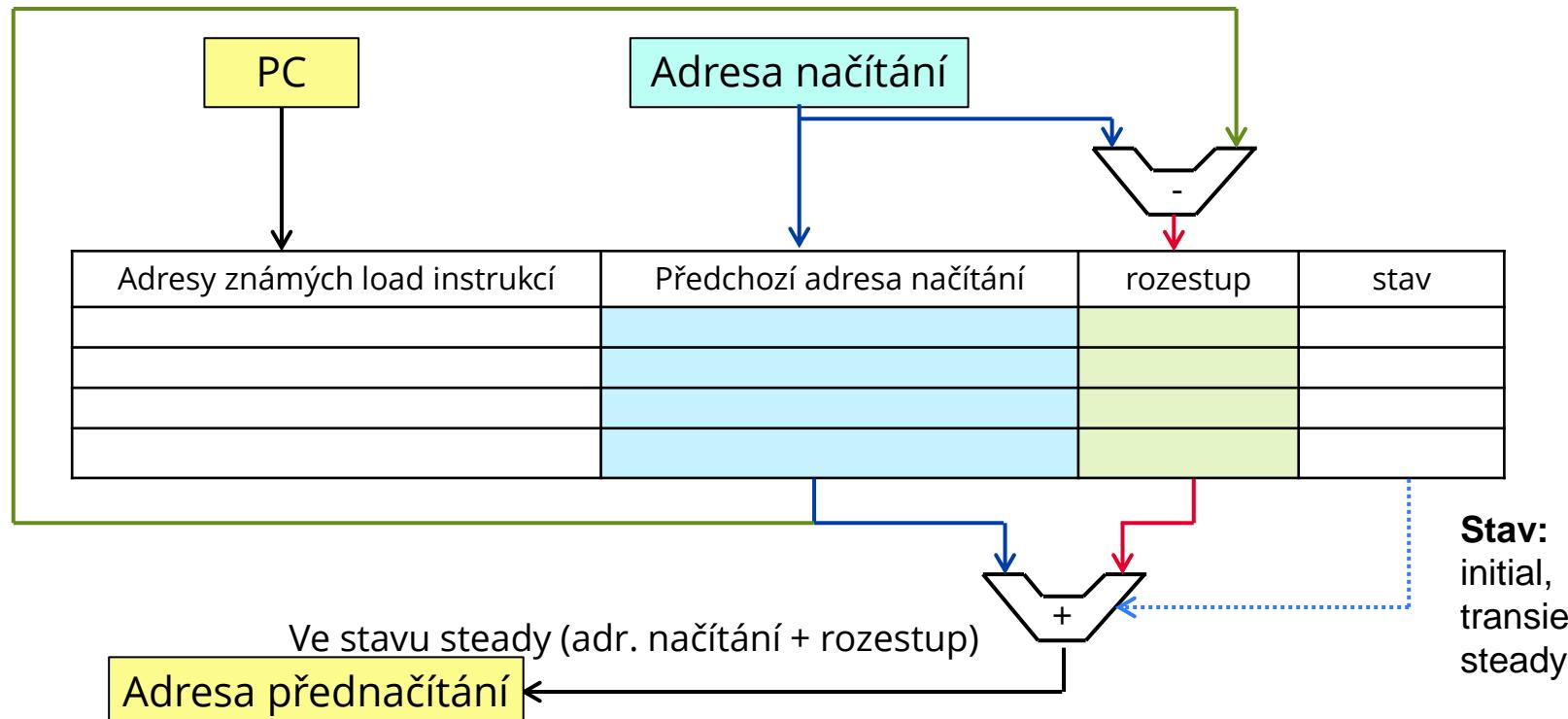
Přístupy do po sobě jdoucích bloků (vyžádaných, demand-fetched nebo přednačtených, prefetched)



1 výpadek na celou posloupnost bloků

- Je to analogie přednačítání instrukcí do instrukčního buferu.
- Fixní počet 4–8 bloků (např. mediální data) se sekvenčně načítá předem do **FIFO stream buferu** a ne do L1 D cache, aby se vyloučilo její znečistění, jelikož se data použijí jen 1x.
- Hledá se současně v cache i na čele stream buferu.
 - Když je nalezen blok v buferu, přesune se do cache, pointer na čelo se posune a nový blok se přednačte na konec buferu.
 - Při výpadku v cache a když blok není nalezen ani na čele buferu (konec sekvence) se bufer začne plnit daty od nové adresy výpadku.
- Výhoda stream buferu se ukáže jen když přístup k přednačítaným blokům je přísně sekvenční.

- Load History Table (LHT) = malá cache již provedených instrukcí L
- Rozestup se získá ze dvou L na téže adrese.



Pokračování příště

Datový paralelismus SIMD

AVS – Architektury výpočetních systémů

Týden 5, 2024/2025

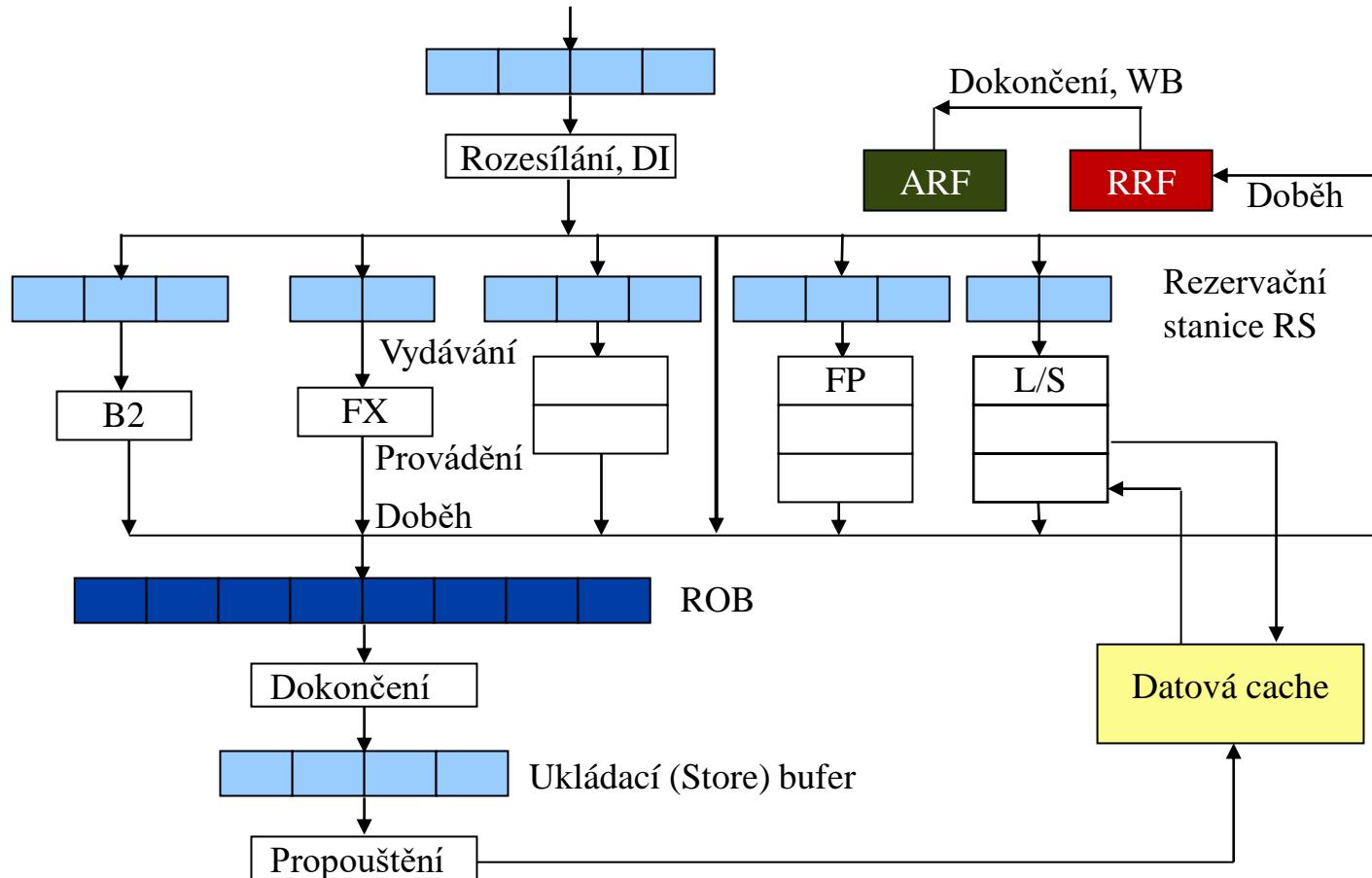
Jirka Jaroš

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

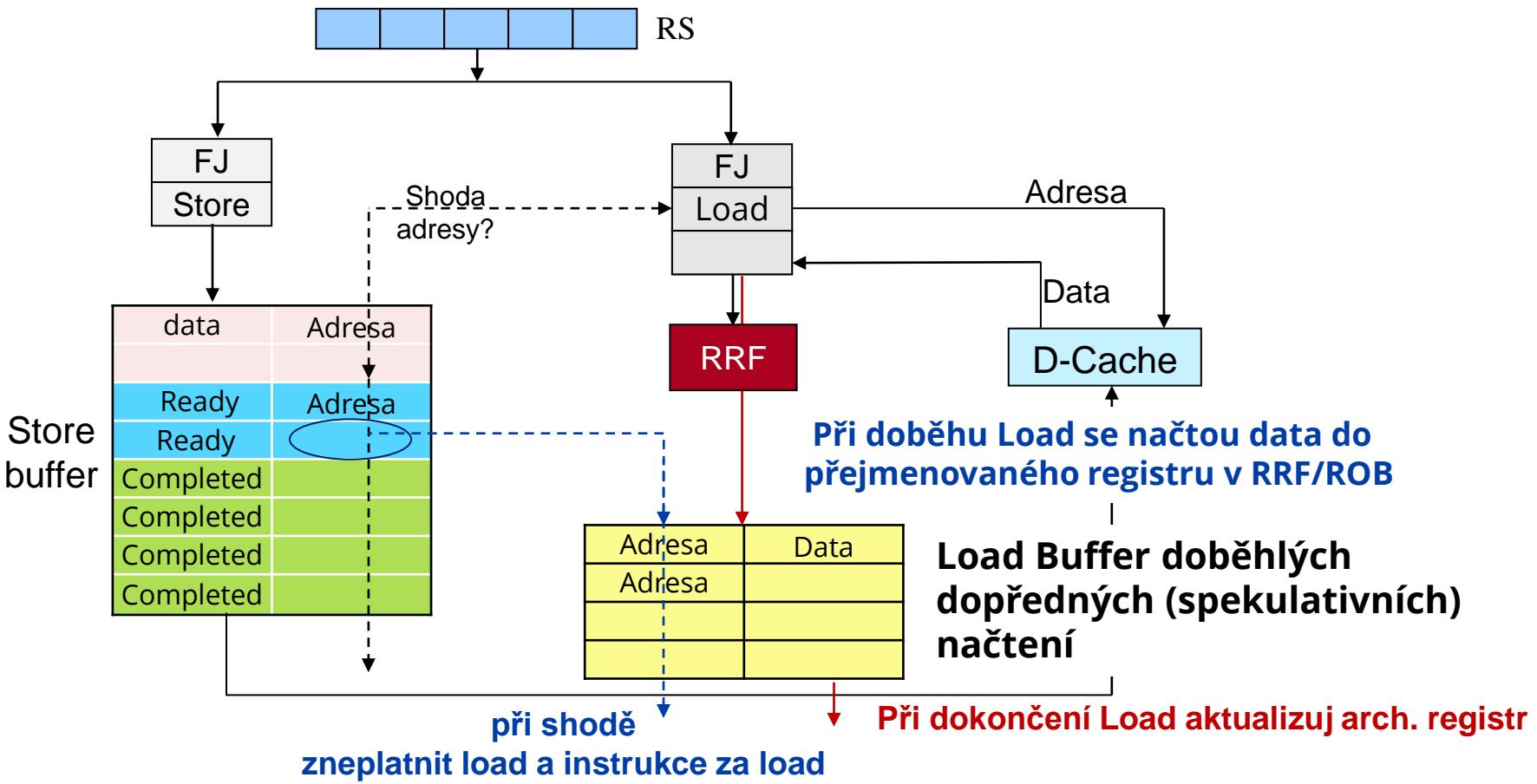


OPAKOVÁNÍ

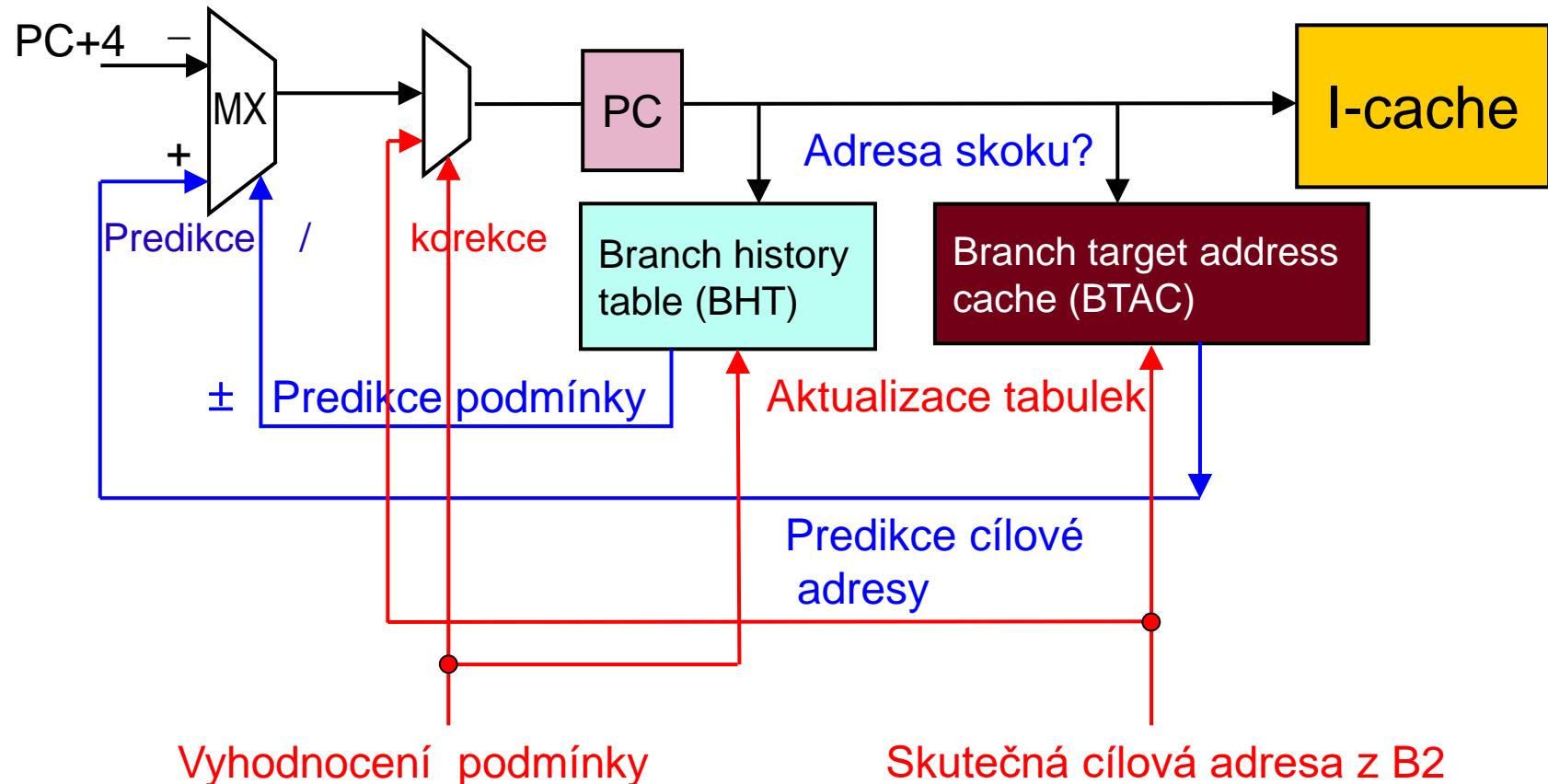
I Superskalární procesor – Backend



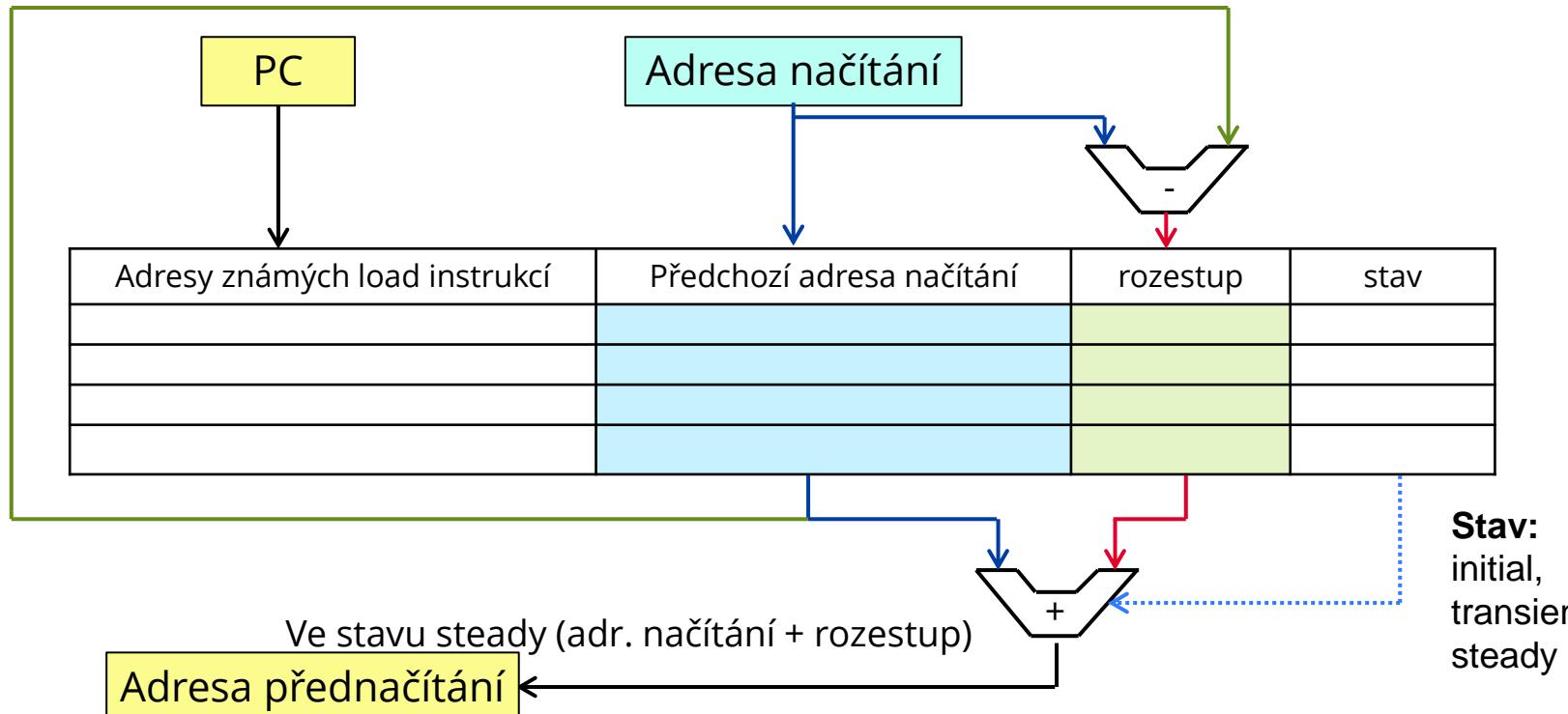
Out-of-order Load/Store jednotka



Predikce podmínky a cílové adresy skoku (B1)



- Load History Table (LHT) = malá cache již provedených instrukcí L
- Rozestup se získá ze dvou L na téže adrese.



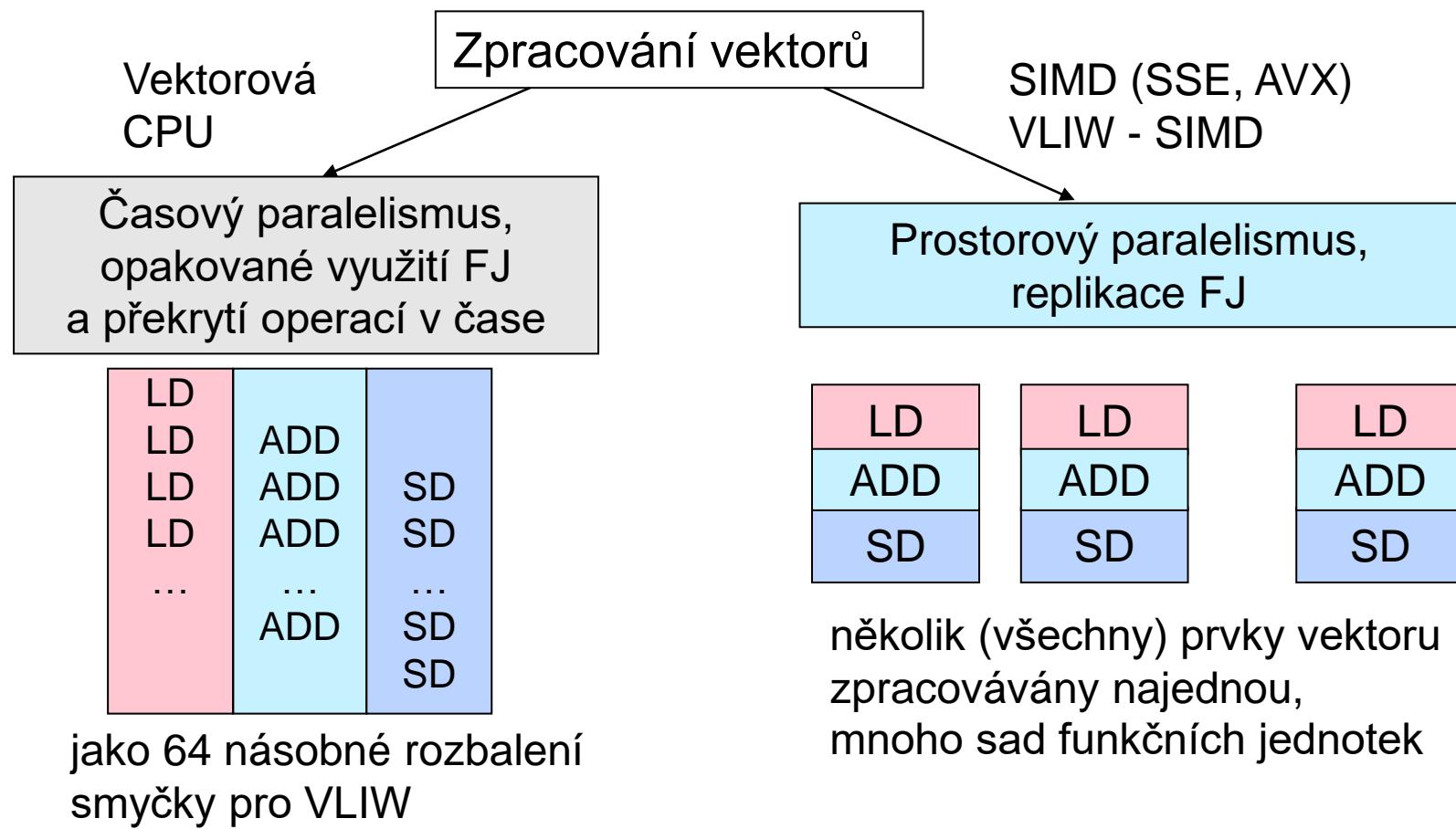
DATOVÝ PARALELISMUS

1. **Funkční paralelismus** využívá obecně možnosti paralelního provádění funkcí v jemné (ILP)

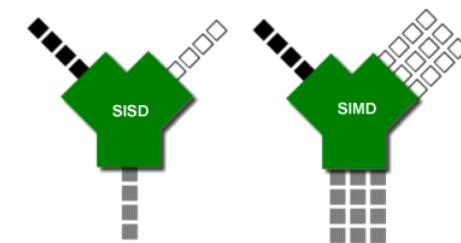
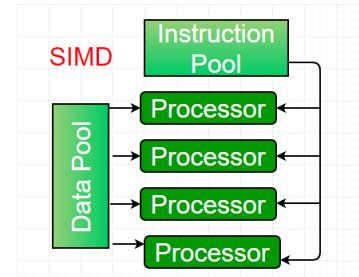
- Superskalární procesory – ILP řízen pomocí hardware (OOO, pipeline)
- Procesory s dlouhým instrukčním slovem VLIW – ILP řízen software (kompilátor) nebo hrubé granularitě (vlákna – TLP, procesy).

2. **Datový paralelismus** využívá možnosti provádět stejné operace s mnoha nezávislými datovými položkami

- **v čase** řetězeným zpracováním (vektorové procesory)
- **v prostoru** na replikovaných funkčních jednotkách (multiprocesory SIMD, SIMD Within A Register = SWAR)
- **v obou dimenzích** (paralelní vektorové linky nebo technologie Single Instruction Multiple Threads = SIMT)



- SIMD – Single Instruction Multiple Data
 - Soubor procesorů řízený centrální jednotkou pracuje **synchronně po instrukcích** - všechny procesory dělají totéž
 - Některé procesory mohou stát (NOP)
- Výpočet jednotlivých prvků probíhá nezávisle na ostatních
- **Silná stránka:**
 - Zpracování polí, matic, trenzorů, seznamů
- **Slabá stránka:**
 - Podmíněné sekce a příkazy switch.
N alternativ se provádí v podmnožinách procesorů sekvenčně.
- Při mapování algoritmů na tyto architektury je nezbytné změnit způsob uvažování.

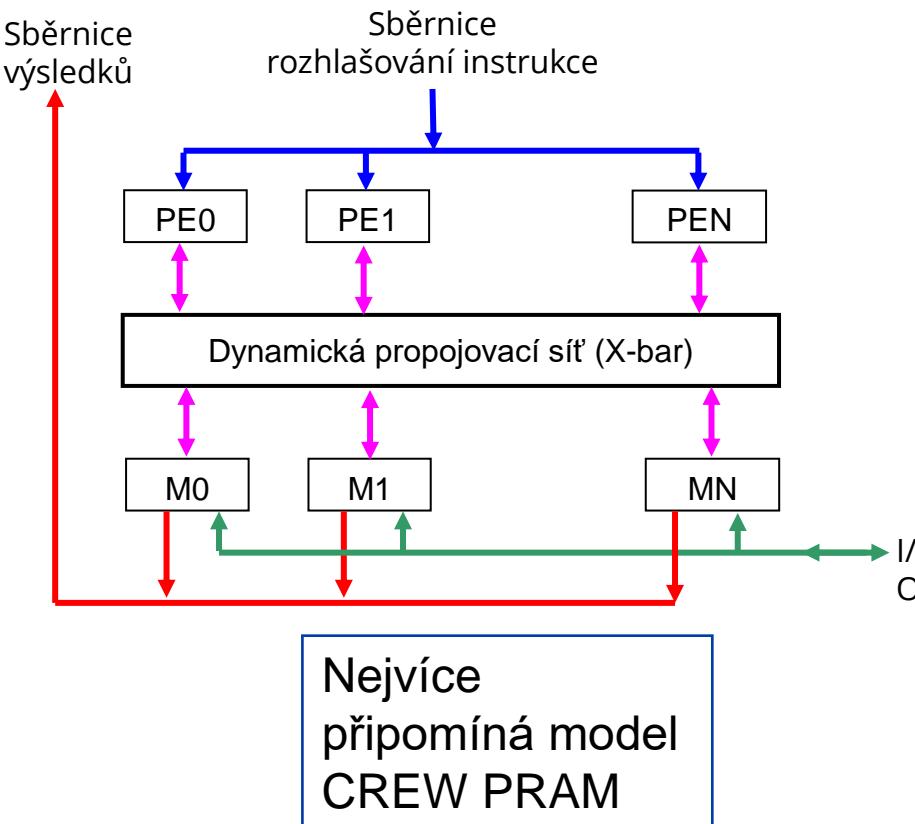


- SIMD architektury obsahují **velký počet výpočetních jednotek** (Processing Units PE) a řídící procesor.
- Počet PE jednotek na CPU a GPU v rozsahu 4–32.
- SIMD architektury **vyžadují méně HW prostředků** než MIMD (pouze jedna řídící jednotka).
- **Základní problém:** Vhodný poměr mezi složitostí PE jednotky a jejich počtem.
- **Častý kompromis:** Velký počet základních jednotek podporujících operace (ADD, SUB, CMP, MUL, ...) doplněných o několik málo specializovaných jednotek realizujících speciální operace (DIV, SIN, LOG, SQRT, ...)
- Nejsou považovány za univerzální výpočetní systémy; pro některé testy špičková výkonnost, pro jiné jen nízká.
- **Použití jako HW akcelerátory a koprosesory (GPU, AVX)**



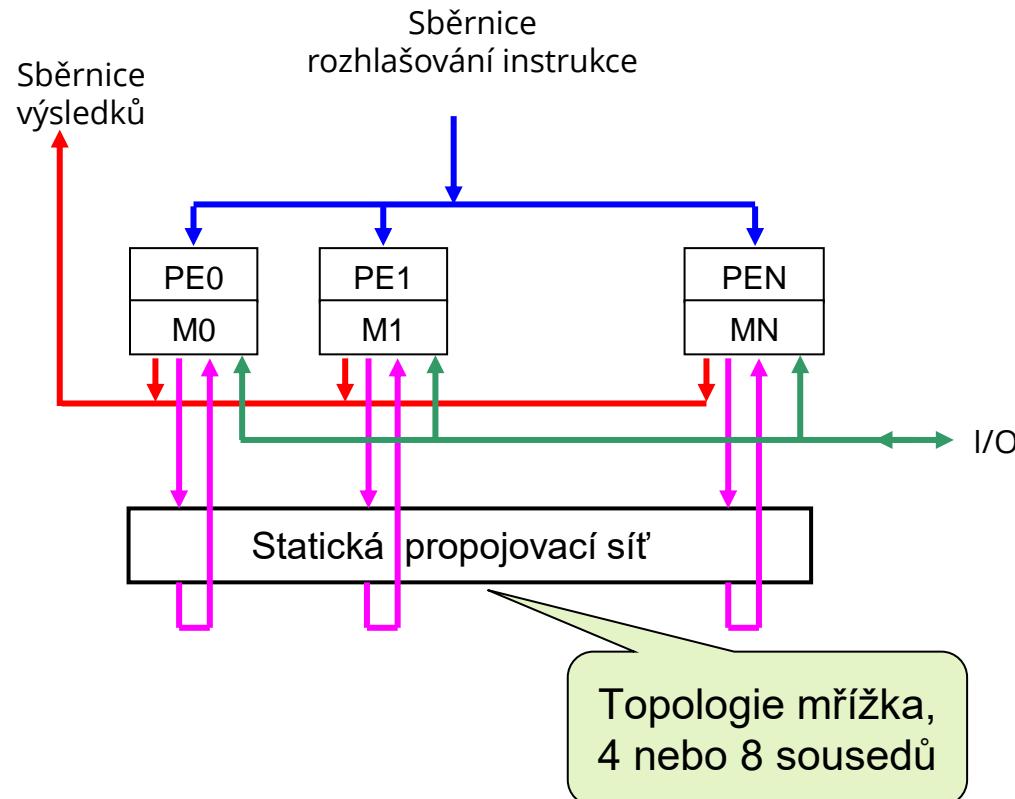
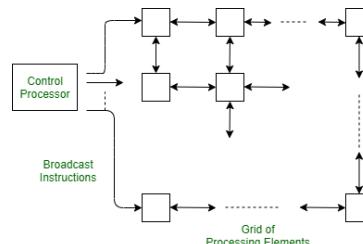
- SIMD se sdílenou pamětí

- Paměťové moduly sdíleny všemi PE elementy
- Čtení i zápis dat probíhá skrze dynamickou propojovací síť (např. křížový přepínač)
- Jednotlivé PE elementy komunikují pouze přes sdílenou paměť
- Současné architektury AVX jednotek v procesorech a SM procesorů v GPU se nejvíce podobají tomuto modelu

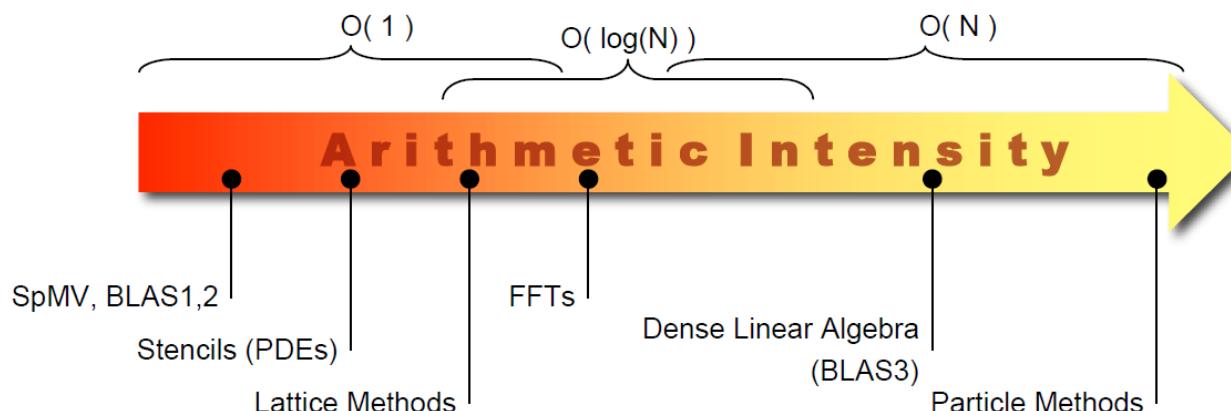


- SIMD s distribuovanou pamětí**

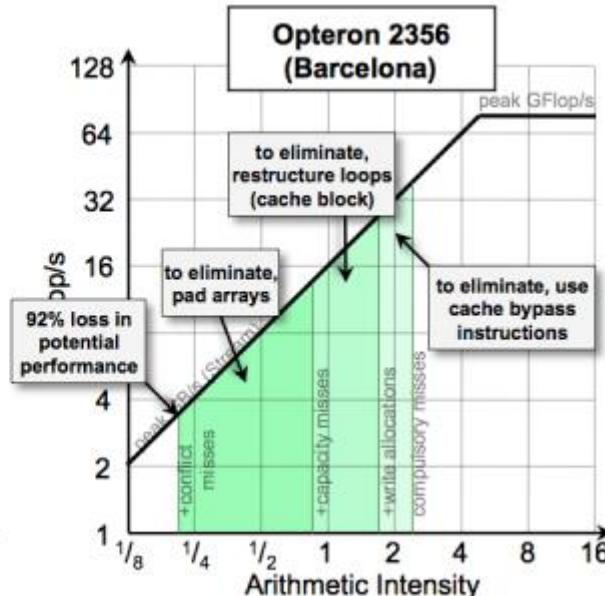
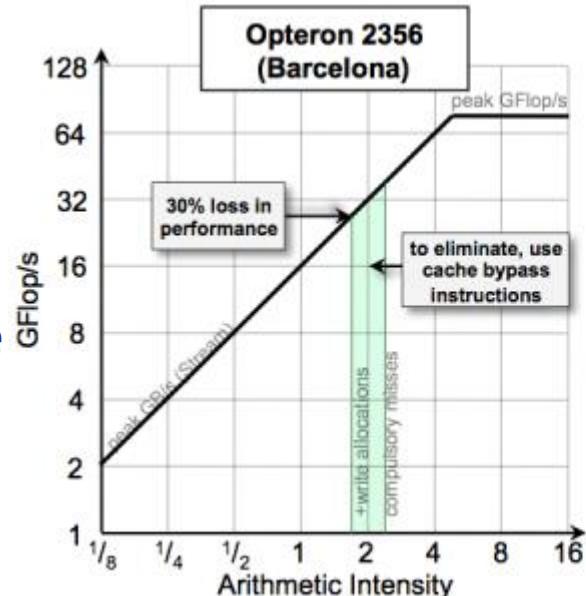
- Každý PE má svůj vlastní paměťový modul – většinu výpočtu provádí nad lokálními daty.
- Komunikace resp. výměna dat s ostatními PE elementy probíhá skrze statickou propojovací síť (např. mřížka, torus, popř. složitější topologie).
- Použito např. v IBM Cell procesory (Sony Playstation 3)



- Výkonnost každého algoritmu je shora omezena buď propustností (FP) ALU (GFLOP/s) nebo paměti (GB/s).
- I L1 je stěží schopna zásobovat žravé AVX jednotky a zabránit vkládání NOP.
- Pokud nedosáhneme rozumné aritmetické intenzity (FLOP:B), nemá smysl se do vektorizace vůbec pouštět.
- SIMD-friendly algoritmus tedy musí maximálně vyžívat paměť cache (cache blocking), efektivně přednačítat data z RAM, minimalizovat výpadky v TLB, ...

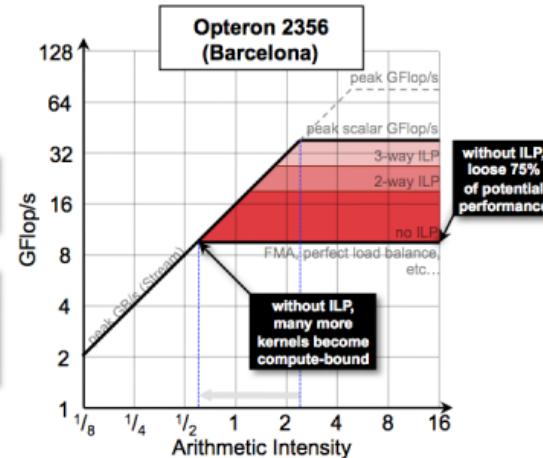
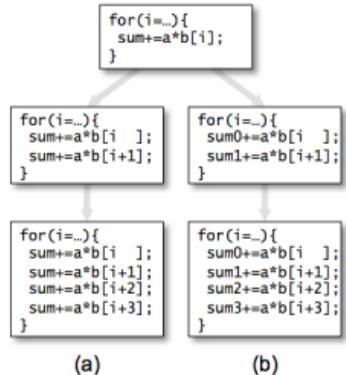


- **Roofline popisuje výkon algoritmu na základě**
 - Aritmetické intenzity
 - Propustnosti paměti
 - Výpočetního výkonu
- **Výpadky v paměti cache mají velký vliv na aritmetickou intenzitu**

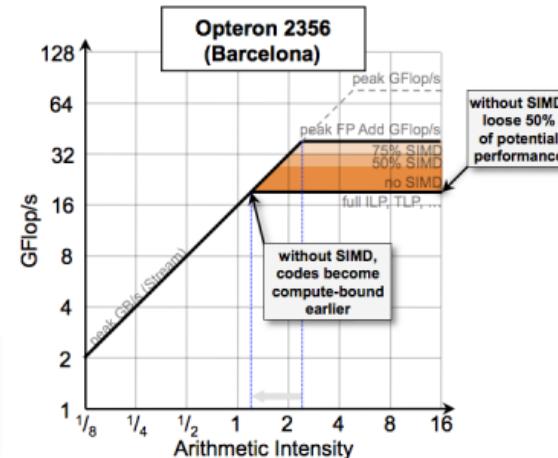
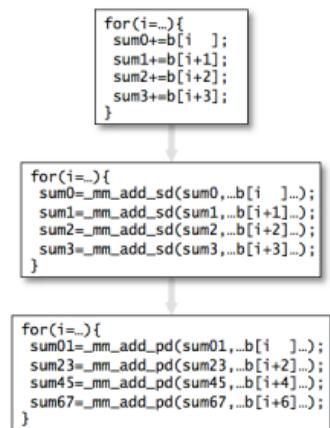


<https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/>

ILP



SIMD



Vectorize & Thread or Performance Dies

Threaded + Vectorized can be Much Faster Together than Either Alone



<https://www.intel.com/content/www/us/en/developer/videos/vectorize-or-performance-dies-tune-for-the-latest-avx-simd.html>

SIMD UVNITŘ REGISTRŮ SWAR

I Historie vektorových instrukcí v procesorech x86

| 1999 | 2000 | 2004 | 2006 | 2007 | 2008 | 2009 | 2011 | 2012\2013 | 2012 |
|--|--|---|---|---|---|---|--|--|--|
| SSE | SSE2 | SSE3 | SSSE3 | SSE4.1 | SSE4.2 | AES-NI | AVX | AVX2 | MIC |
| 70 instr
Single-Precision
Vectors
Streaming
operations | 144 instr
Double-precision
Vectors
8/16/32
64/128-bit
vector
integer | 13 instr
Complex Data | 32 instr
Decode | 47 instr
Video
Graphics
building
blocks
Advanced
vector instr | 8 instr
String/XML
processing
POP-Count
CRC | 7 instr
Encryption
and
Decryption
Key
Generation | ~100 new
instr.
~300
legacy sse
instr
updated
256-bit
vector
3 and 4-
operand
Instructions | Int. AVX
expands to
256 bit
Improved
bit manip.
fma
Vector
shifts
Gather | 512-bit
vector |
| | | | | | | | | | |
| Intel® Xeon®
processor
64-bit | Intel® Xeon®
processor
5100
series | Intel® Xeon®
processor
5500
series | Intel® Xeon®
processor
5600
series | Intel® Xeon®
processor
code-named
Sandy
Bridge EP | Intel® Xeon®
processor
code-named
Ivy Bridge
EP | Intel® Xeon®
processor
code-named
Haswell
EP | Intel® Xeon®
Processor
codenamed
Skylake
EP | Intel® Xeon
Phi™
coprocessor
Knights
Corner | Intel® Xeon
Phi™
processor
& coprocessor
Knights
Landing ¹ |
| Core(s) | 1 | 2 | 4 | 6 | 8 | 12 | 18 | 28 | 61 |
| Threads | 2 | 2 | 8 | 12 | 16 | 24 | 36 | 56 | 244 |
| SIMD Width | 128 | 128 | 128 | 128 | 256 | 256 | 256 | 512 | 512 |

AVX-512 instrukce jsou rozdělenu do několika skupin

- **Podpora na CPU i MIC (Xeon Phi)**

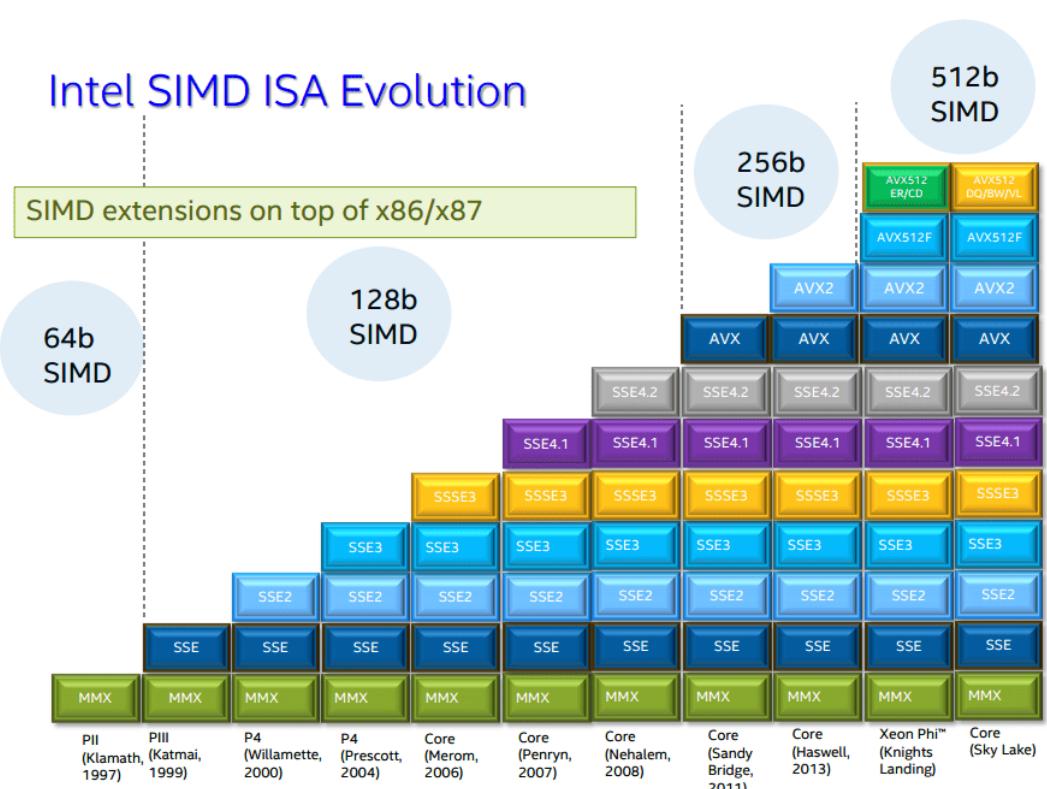
- Foundation Instructions (AVX512-F)
- Conflict Detection Instructions (AVX512-CD)

- **Podpora pouze na MIC**

- Exponential and Reciprocal Instruction (AVX512-ER)
- Prefetch Instructions (AVX512-PF)

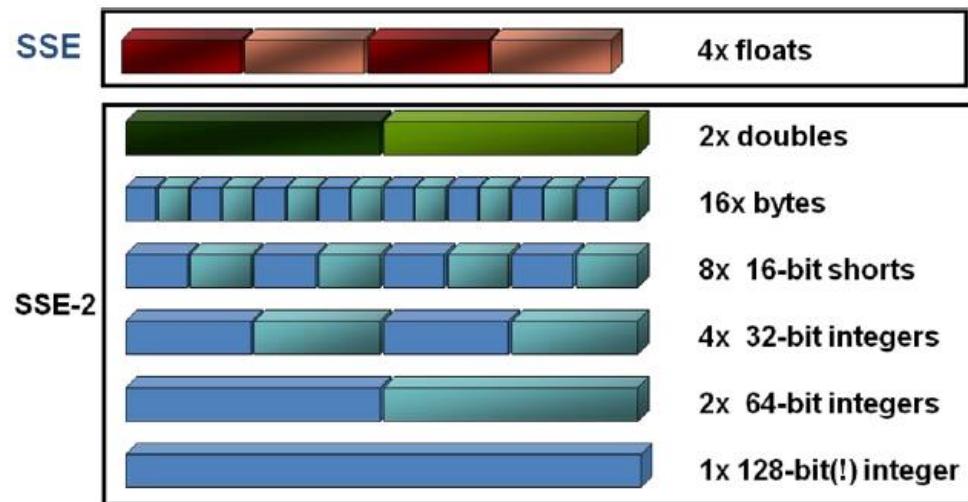
- **Podpora pouze na CPU**

- Byte (char/int8) and Word (short/int16) Instructions (AVX512-BW)
- Double-word (int32/int) and Quad-word (int64/long) Instructions (AVX512-DQ)
- Vector Length Extensions (AVX512-VL)



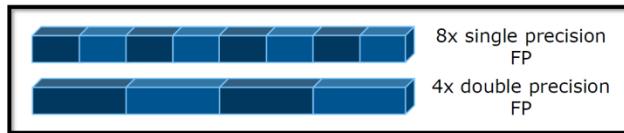
I Registry SSE

- Registry SSE již nejsou mapovány na registry FPU jako registry MMX.
- SSE obsahuje $8 \times 128b$ registrů (XMM0–XMM7)
- Jelikož tyto registry nejsou v původní sadě x86, OS nemá o jejich existenci tušení => nelze uložit jejich stav.
- Proto Intel modifikoval chráněný režim procesoru a vytvořil tzv. rozšířený mód procesoru (Enhanced mode), kde jsou registry SSE již viditelné pro OS.
- Pokud je OS schopen pracovat s registry SSE, musí přepnout procesor do rozšířeného módu.

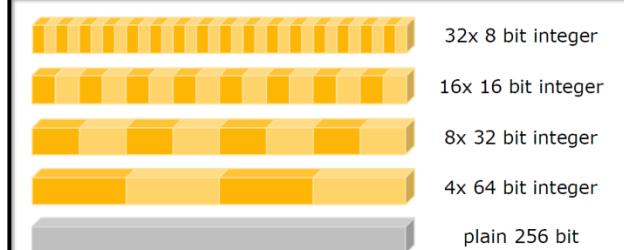


AVX512 state

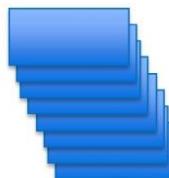
Intel® AVX



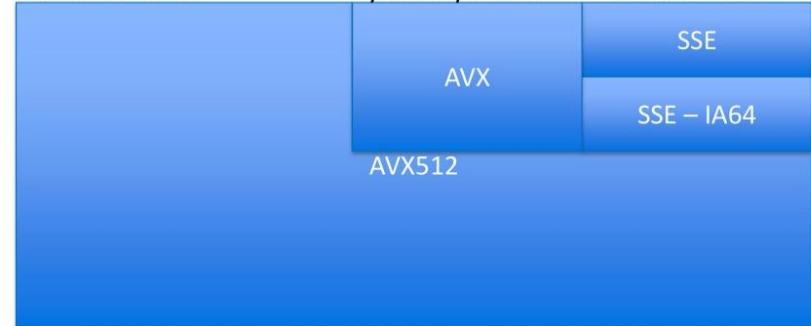
Intel® AVX2



Kmask k0..k7



zmm0..zmm31



ymm0..ymm15

AVX

xmm0..xmm7

SSE

SSE – IA64

High amounts of compute need large amounts of state to compensate for memory BW
AVX512 has 8x state compared to SSE (commensurate with its 8x flops level)

Instrukční sady AVX a AVX-512

Sandy Bridge

256b AVX1
16 SP / 8 DP
Flops/Cycle

Haswell

256b AVX2
32 SP / 16 DP
Flops/Cycle (FMA)

Future (in planning,
subject to change)

512b AVX512
Server: 64SP / 32 DP
Client: 32 SP / 16 DP
Flops/Cycle (FMA)

AVX

256-bit basic FP
16 registers
NDS (and AVX128)
Improved blend
MASKMOV
Implicit unaligned

AVX2

Float16 (IVB 2012)
256-bit FP FMA
256-bit integer
PERMD
Gather

AVX512

512-bit FP/Integer
32 registers
8 mask registers
Embedded rounding
Embedded broadcast
Scalar/SSE/AVX “promotions”
Native media additions
HPC additions
Transcendental support
Gather/Scatter

SNB-2011

HSW-2013

Future Processor (Knight
Landing & Skylake Xeon)

Intel Desktop

| Intel® AVX | Intel® AVX2 |
|--------------------|---|
| 128/256-bit FP | Float16 |
| 16 registers | 128/256-bit FP FMA |
| NDS (and AVX128) | 256-bit int |
| Improved blend | PERMD |
| MASKMOV | Gather |
| Implicit unaligned | Flag-based enumeration
Intel® Xeon P-core only |

Intel Server

AMD Zen 4

| Intel® AVX-512 |
|-----------------------------|
| 128/256/512-bit FP/Int |
| 32 vector registers |
| 8 mask registers |
| 512-bit embedded rounding |
| Embedded broadcast |
| Scalar/SSE/AVX “promotions” |
| Native media additions |
| HPC additions |
| Transcendental support |
| Gather/Scatter |
| Flag-based enumeration |
| Intel® Xeon P-core only |

Intel® AVX10.1 (pre-enabling)

| Optional 512-bit FP/Int |
|----------------------------------|
| 128/256-bit FP/Int |
| 32 vector registers |
| 8 mask registers |
| 512-bit embedded rounding |
| Embedded broadcast |
| Scalar/SSE/AVX “promotions” |
| Native media additions |
| HPC additions |
| Transcendental support |
| Gather/Scatter |
| Version-based enumeration |
| Intel® Xeon P-core only |

Intel® AVX10.2

| New data movement, transforms and type instructions |
|---|
| Optional 512-bit FP/Int |
| 128/256-bit FP/Int |
| 32 vector registers |
| 8 mask registers |
| 256/512-bit embedded rounding |
| Embedded broadcast |
| Scalar/SSE/AVX “promotions” |
| Native media additions |
| HPC additions |
| Transcendental support |
| Gather/Scatter |
| Version-based enumeration |
| Supported on P-cores, E-cores |

Figure 1-2. Intel® ISA Families and Features

- VADDPS ZMM0 {k1}, ZMM3, [mem]

 - Mask bits used to:

- Suppress individual elements read from memory*
 - hence not signaling any memory fault
- Avoid actual independent operations within an instruction happening*
 - and hence not signaling any FP fault
- Avoid the individual destination elements being updated,*
 - or alternatively, force them to zero (zeroing)

```
for (I in vector length)
{
    if (no_masking or mask[I]) {
        dest[I] = OP(src2, src3)
    } else {
        if (zeroing_masking)
            dest[I] = 0
        else
            // dest[I] is preserved
    }
}
```

Some instructions do no suppress memory exceptions as mask aligned to "out"

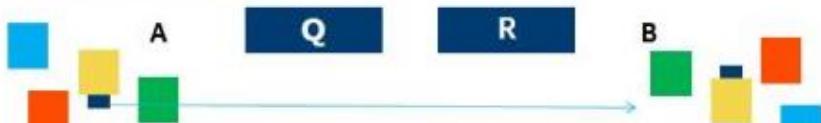
AVX512 Masking

Gather & Scatter

D/Q/SP/DP element types
D/Q Indices
Instruction can partially execute
k-reg Mask used as completion mask

VMOVDQU64 zmm1, Q[rsi]
VMOVDQU64 zmm2, R[rsi]
VGATHERQQ zmm0 {k2}, [rax+zmm1*8]
VSCATTERQQ [rax+zmm2*8] {k3}, zmm0

```
for(j=0, i=0; i<N; i++)
{
    B[R[i]] = A[Q[i]];
}
```



G/S implementation attempts to 'max out' DCU BW
Performance gains come from vectorizing REST of algorithm
Algorithm shown could get some gain (24 load dispatches → 10 per 8 elements)

- Intel® Advanced Vector Extensions 2 (Intel® AVX2)

- Includes

- 256-bit Integer vectors
 - FMA: Fused Multiply-Add
 - Full-width element permutes
 - Gather

- Benefits

- High performance computing
 - Audio & Video
 - Games

- New Integer Instructions

- Indexing and hashing
 - Cryptography
 - Endian conversion – MOVBE

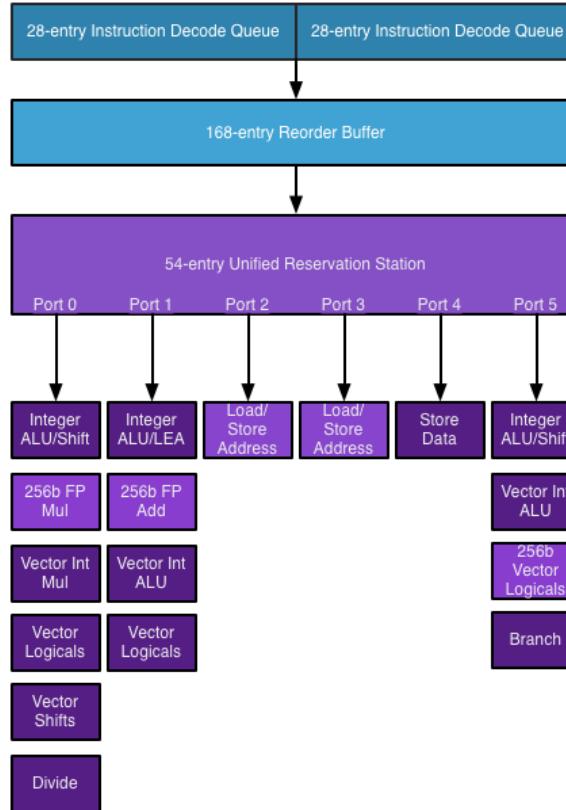
| | Instruction Set | SP FLOPs per cycle | DP FLOPs per cycle |
|--------------|-----------------|--------------------|--------------------|
| Nehalem | SSE (128-bits) | 8 | 4 |
| Sandy Bridge | AVX (256-bits) | 16 | 8 |
| Haswell | AVX2 & FMA | 32 | 16 |

| Group | Instructions |
|--|--|
| Bit Field Pack/Extract | BZHI, SHLX, SHRX, SARX, BEXTR |
| Variable Bit Length Stream Decode | LZCNT, TZCNT, BLSR, BLSMSK, BLSI, ANDN |
| Bit Gather/Scatter | PDEP, PEXT |
| Arbitrary Precision Arithmetic & Hashing | MULX, RORX |

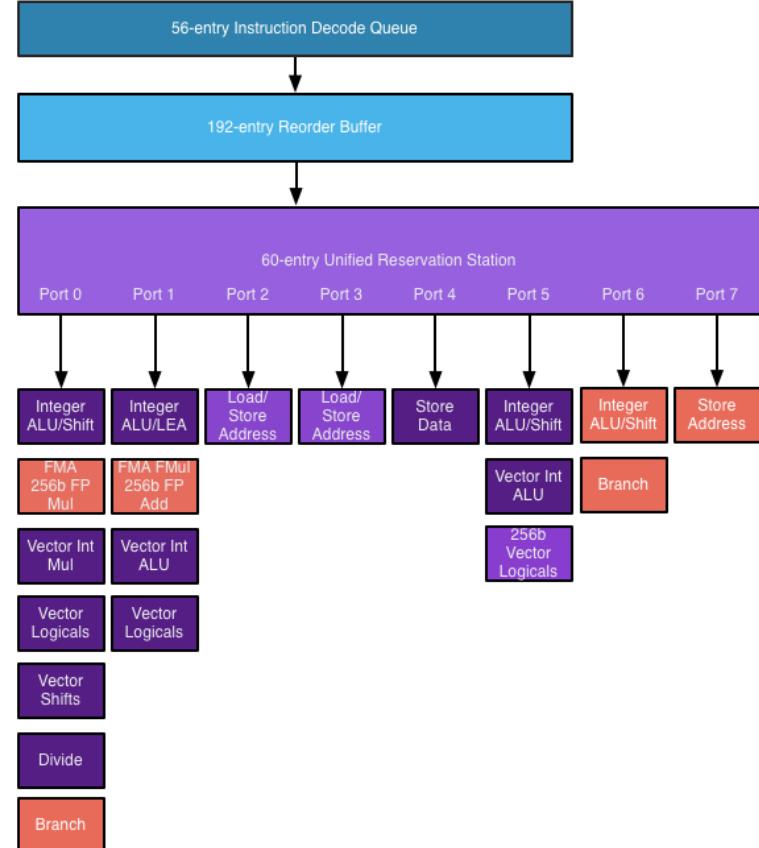
- Full Instruction Specification Available at: <http://software.intel.com/en-us/avx/>

Intel Sandy Bridge a Haswell backend

Intel Sandy Bridge Execution Engine

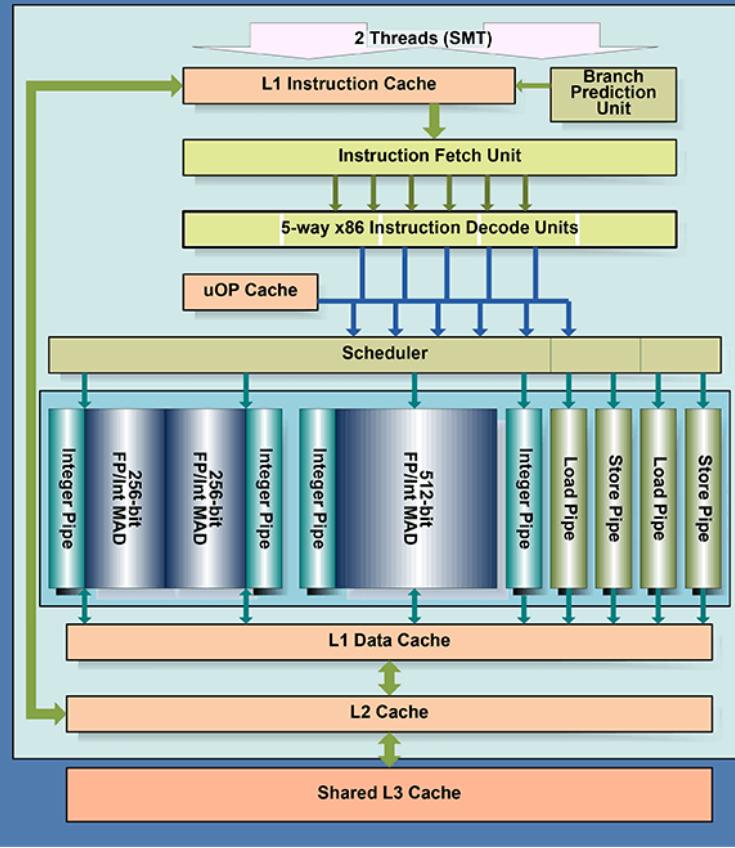


Intel Haswell Execution Engine

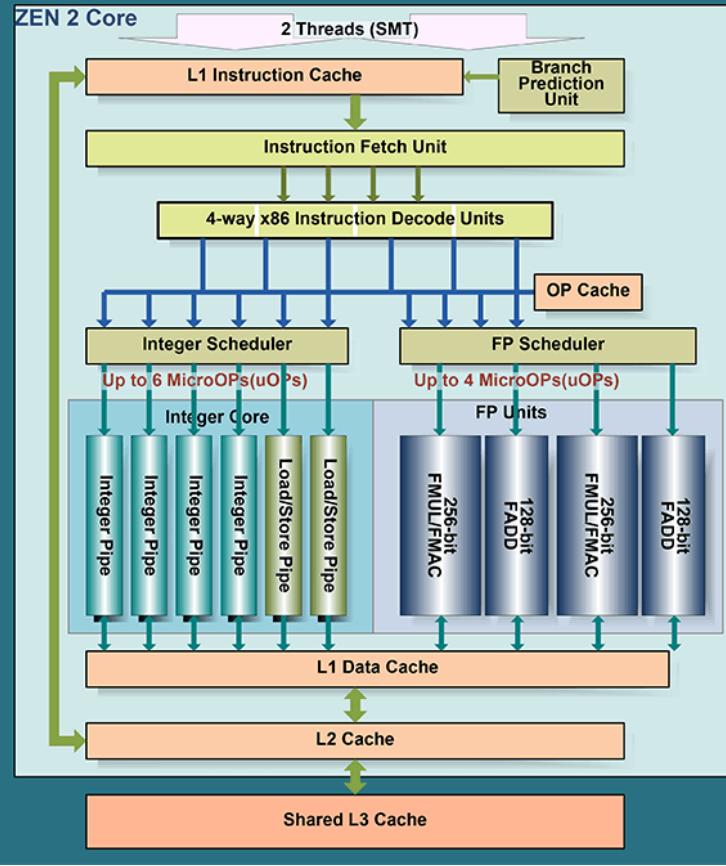


Intel vs AMD backend

Intel Sunny Cove Architecture



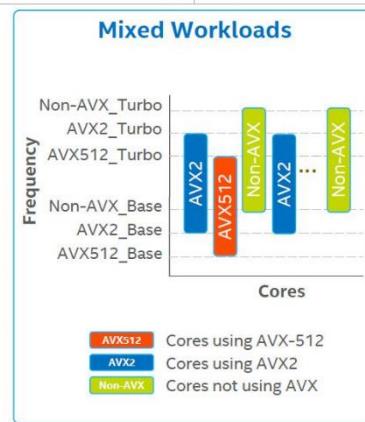
AMD ZEN 2 Architecture (estimated)



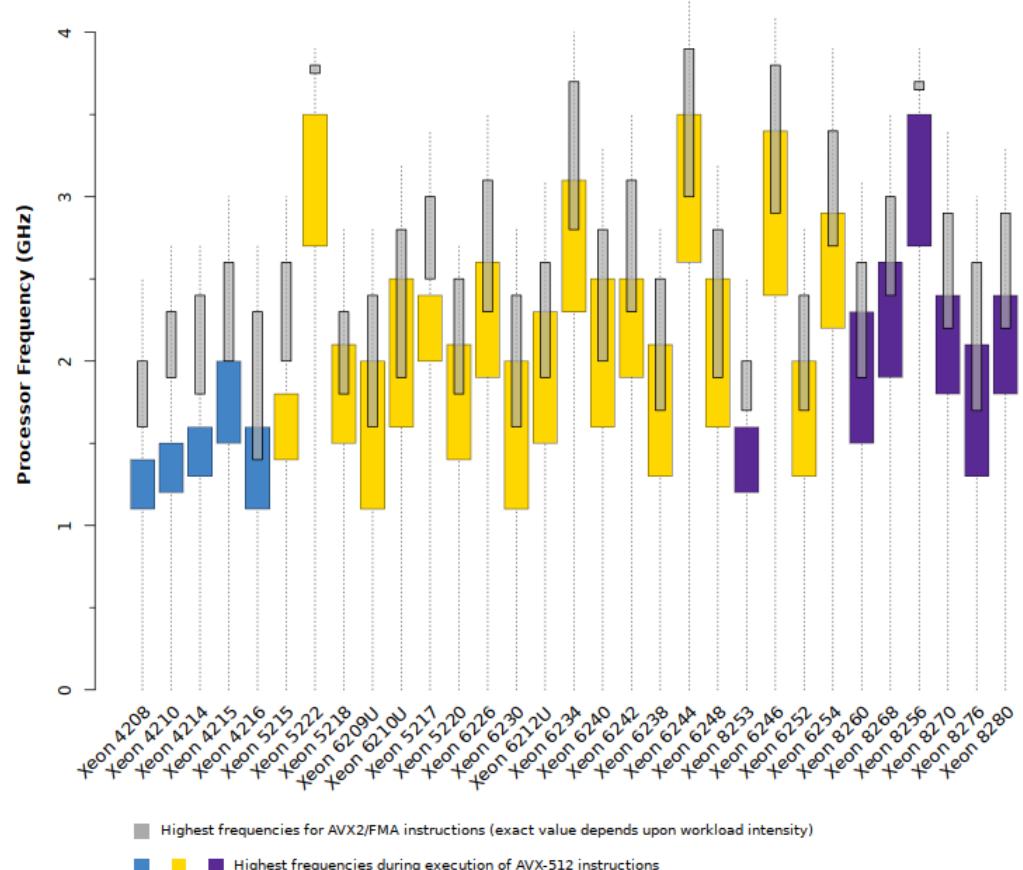
Snižování frekvence při použití AVX-512

- Cores running non-AVX, Intel® AVX2 light/heavy, and Intel® AVX-512 light/heavy code have different turbo frequency limits
- Frequency of each core is determined independently based on workload demand

| Code Type | All Core Frequency Limit |
|--|--------------------------|
| SSE
AVX2-Light (without FP & int-mul) | Non-AVX All Core Turbo |
| AVX2-Heavy (FP & int-mul)
AVX512-Light (without FP & int-mul) | AVX2 All Core Turbo |
| AVX512-Heavy (FP & int-mul) | AVX512 All Core Turbo |



Comparison of All-Core Turbo Boost Ranges (AVX-512 vs. AVX2)



VEKTORIZACE KÓDU

I Jednoduchý příklad využití registrů SSE

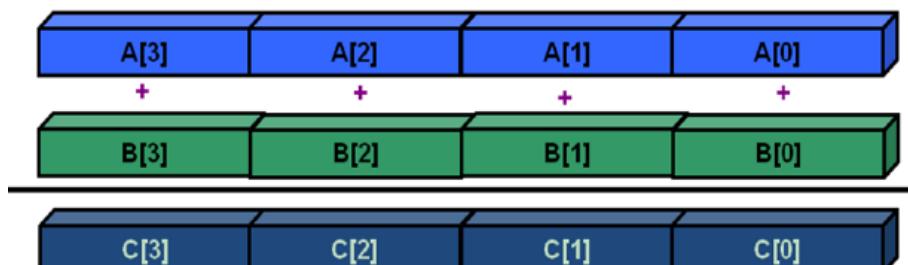
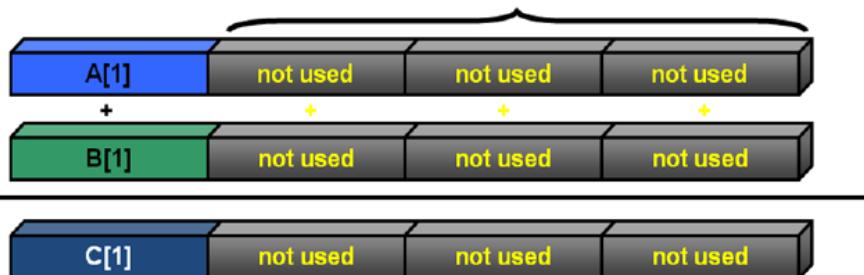
Bez vektorizace

```
for (i = 0; i < MAX; i++)  
{  
    c[i] = a[i] + b[i];  
}
```

Vektorizováno

```
for (i = 0; i < MAX; i+=4)  
{  
    c[i:4] = a[i:4] + b[i:4];  
}
```

e.g. 3 x 32-bit unused integers



Vektorizované knihovny

- Intel MKL
- Atlas, FFTW, TensorFlow, ...

Automatická vektorizace kompilátorem

- -O3 -vec

Pragma hinty kompilátoru

- OpenMP 4.0+
- Intel

Vektorové intrinsic funkce

- _mm_malloc
- _mm_add_ps

ASM kód

- addps

Automatická vektorizace komplátorem

Rozbalení smyčky je spojeno s generováním SIMD instrukcí

```
static double A[1000], B[1000],  
          C[1000];  
  
void add() {  
    int i;  
    for (i=0; i<1000; i++)  
        if (A[i]>0)  
            A[i] += B[i];  
        else  
            A[i] += C[i];  
}
```

```
.B1.2::  
    vmovaps  ymm3, A[rdx*8]  
    vmovaps  ymm1, C[rdx*8]  
    vcmpgtpd ymm2, ymm3, ymm0  
    vblendvpd ymm4, ymm1,B[rdx*8], ymm2  
    vaddpd   ymm5, ymm3, ymm4  
    vmovaps  A[rdx*8], ymm5  
    add      rdx, 4  
    cmp      rdx, 1000  
    jl       .B1.2
```

AVX

```
.B1.2::  
    movaps   xmm2, A[rdx*8]  
    xorps   xmm0, xmm0  
    cmpltpd xmm0, xmm2  
    movaps   xmm1, B[rdx*8]  
    andps   xmm1, xmm0  
    andnps  xmm0, C[rdx*8]  
    orps    xmm1, xmm0  
    addpd   xmm2, xmm1  
    movaps   A[rdx*8], xmm2  
    add     rdx, 2  
    cmp     rdx, 1000  
    jl      .B1.2
```

SSE2

```
.B1.2::  
    movaps   xmm2, A[rdx*8]  
    xorps   xmm0, xmm0  
    cmpltpd xmm0, xmm2  
    movaps   xmm1, C[rdx*8]  
    blendvpd xmm1, B[rdx*8], xmm0  
    addpd   xmm2, xmm1  
    movaps   A[rdx*8], xmm2  
    add     rdx, 2  
    cmp     rdx, 1000  
    jl      .B1.2
```

SSE4.1

- ✓ SSE2-SSE4.2
 - No native masked operations
 - "Masks" in vector registers for AND/OR blending
 - Memory operations and unsafe FP operations speculated or emulated
 - Via scalarization for memory
 - Blend w/ safe value for FP
 - Remainder vectorized unmasked
- ✓ AVX
 - vmaskmov is introduced
- ✓ AVX2
 - vgather is introduced
 - Peel/remainder vectorized unmasked
- ✓ Intel® MIC Architecture
 - Native masking in dedicated registers
 - For all operations
 - May be unsafe if masked-out memory is not paged
 - Except gathers/scatters
 - Safety lfs inserted for empty masks
 - Some operations done through gather
 - Peel/remainder/low-trip vectorized in masked mode

<https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf>

| Podpora SIMD rozšíření Intel kompilátoru -xSADA | IT FIT

| SIMD Sada | Popis |
|---------------|---|
| COMMON-AVX512 | May generate Intel AVX-512 Foundation instructions, Intel AVX-512 Conflict Detection instructions, AVX2, AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2 and SSE instructions for Intel processors. |
| MIC-AVX512 | May generate Intel AVX-512 Foundation instructions, AVX-512 Conflict Detection instructions, AVX-512 Prefetch instructions, AVX-512 Exponential and Reciprocal instructions, AVX2, AVX, SSE4.2 - SSE instructions for Intel processors. |
| CORE-AVX512 | May generate AVX-512 Foundation instructions, AVX-512 Conflict Detection instructions, AVX-512 Doubleword and Quadword instructions, AVX-512 Byte and Word instructions, AVX-512 Vector Length extensions, AVX2, AVX, SSE4.2, SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel processors. |
| CORE-AVX2 | May generate Intel AVX2, AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2 and SSE instructions for Intel® processors. |
| CORE-AVX-I | May generate Intel AVX, SSE4.2, SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel processors, |
| AVX | May generate Intel AVX, SSE4.2, SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel |
| SSE4.2 | May generate Intel SSE4.2, SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel processors. |
| ATOM_SSE4.2 | May generate Intel SSE4.2, SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel processors. |
| SSE4.1 | May generate Intel SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel processors. |
| SSSE3 | May generate Intel SSE3, SSE3, SSE2 and SSE instructions for Intel processors. |
| ATOM_SSSE3 | May generate SSE3, SSE3, SSE2 and SSE instructions for Intel processors. |
| SSE3 | May generate Intel SSE3, SSE2 and SSE instructions. |
| SSE2 | May generate Intel SSE2 and SSE instructions. |
| HOST | May generate instructions from any of the above instruction sets that are supported by the compilation host processor. |

<https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations>

- **Linux, MacOS X: -x<feature>, Windows: /Qx<feature>**
 - Může zapnout specifické optimalizace pro procesory Intel.
 - Do hlavní funkce (main) je přidán test, který vypíše chybu v případě, že daný procesor nepodporuje některé funkce.
- **Linux, MacOS X: -m<feature>, Windows: /arch:<feature>**
 - Test na intel procesory vypnut, stejně tak specifické Intel optimalizace.
 - Lze tedy použít i pro procesory AMD.
 - Pokud procesor nepodporuje nějakou funkci, vyhodí výjimku **Illegal instruction**.
- **Linux, MacOS X: -ax<feature>, Windows: /Qax:<feature>**
 - Přeloží se několik variant: [baseline](#) a [procesorově specifická](#).
 - Takto lze kompilovat pro různé instrukční sady, např. [-axSSE2,AVX, CORE-AVX2](#)
 - Baseline varianta je dnes stále [-msse2 \(/arch:sse2\)](#). Prakticky nemá smysl používat nic staršího než AVX.
- **Linux, MacOS X: -aHost, Windows: /QxHost**
 - Přeloží a optimalizuje pro procesor na kterém se překládá.

- **Nastavení úrovně detailů, které generuje Intel kompilátor**

- /Qopt-report [0|1|2|3] (Windows)
- -opt-report [0|1|2|3] (Linux, MacOS)

- **Nastavení fází, které nás zajímají**

- /Qopt-report-phase [:phase] (Windows)
- -opt-report-phase [=phase] (Linux, MacOS)
 - ipo_inl - Interprocedural Optimization Inlining Rep
 - ilo - Intermediate language Scalar Optimizatio
 - hpo - High Performance Optimization
 - hlo - High-level Optimization
 - vec - Vectorization
 - all - All optimizations

- **Uložení reportu do souboru**

- /Qopt-report-file: [file] (Windows)
- -opt-report-file= [file] (Linux, MacOS)

```
* icc -O3 -opt-report-phase=hlo -opt-report-phase=hpo

...
LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
...

Loop at line 8 blocked by 128
Loop at line 9 blocked by 128
Loop at line 10 blocked by 128
...
Loop at line 10 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
...
...(10)... loop was not vectorized: not inner loop.
...(8)... loop was not vectorized: not inner loop.
...(9)... PERMUTED LOOP WAS VECTORIZED
...
```

- **Počitatelné smyčky**

```
typedef struct{ float* data; size_t size; } vec_t;

void vec_elwise_product(vec_t* a, vec_t* b, const vec_t* c)
{
    for (auto i = 0; i < a->size; i++)
        c->data[i] = a->data[i] * b->data[i];
}
```

- **Smyčky s jedním vstupem a výstupem**

```
while (i < 100) {
    a[i] = b[i] * c[i];
    if (a[i] < 0.0) break; // data-dependent exit condition:
    i++;
} // loop not vectorized
```

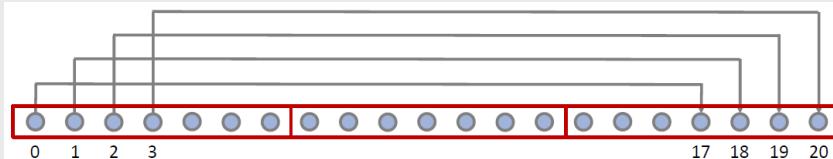
- **Přímý kód** (žádné **switch**, **if** pouze s maskováním)

```
for (int i = 0; i < length; i++) {  
    float s = b[i] * b[i] - 4 * a[i] * c[i];  
    if (s >= 0) x[i] = sqrt(s);  
    else         x[i] = 0.;  
} // loop vectorized (because of masking)
```

- **Pouze nejvnitřnější smyčky** (kompilátor může smyčky kolabovat a přehazovat)
- **Bez volání funkcí až na**
 - Intrinsic matematiku (sin, log, ...)
 - Inline funkce
 - Elementární funkce `__attribute__((vector))`
 - OMP SIMD funkce `#omp declare simd`
 - **Pozor na operátor []**, hlídání mezí polí...

- Nejednotkový rozestup** – Lze načítat pouze datové elementy uložené v paměti za sebou. Pokud je rozestup fixní, lze použít operace gather/scatter
- Nezarovnané datové struktury** – Je nutné vydávat více instrukcí load/store (gather, scatter, shuffle)
- Datové závislosti mezi iteracemi** – (RAW, WAR, WAW), některé lze ošetřit redukcí.
- Pointer aliasing** – překrývající se paměťové oblasti (je nutné dělat testy za chodu a volit různé varianty provedení – memcpy vs. memmove)

```
void add(float* a, float* b, int n)
{
    for (int i = 0; i < n - 17; i++)
    {
        a[i] += b[i + 17];
    }
}
```



- Existence závislostí mezi iteracemi
- Nejednotkový rozestup (gather / scatter)
- Míchání datových typů
- Příliš složité podmínky
- Podmínka může strážit výjimku
- **Příliš malý trip count (počet iterací)**

- Složité indexování
- Nepodporovaná struktura smyček
- **Existence nevektorizovatelného příkazu**
- Mimo vnitřní smyčku
- Vektorizace možná, ale dle heuristiky kompilátoru neefektivní
- Operátor nevhodný pro vektorizaci

<https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/vectorization/automatic-vectorization/programming-guidelines-for-vectorization.html>

- STL knihovna může být použita, ale musí si s ní kompilátor rozumět

```
std::vector<double> A, B;

void foo(int iters, int x, int y)
{
    for (int i = 0; i < iters; i++)
        A[x + i] += B[y + i];
}
```

```
$ icc -vec-report3 -c code.cpp
code.cpp(5): (col. 3) remark: loop was not vectorized:
           existence of vector dependence.
code.cpp(6): (col. 7) remark: vector dependence: assumed
           FLOW dependence
           between _M_start line 6 and _M_start line 6.
```

- Problémy se závislostmi v těle smyčky (dáno implementací STL)
- Kompilátor také nemusí rozpoznat, že A a B jsou globální invarianty.

```
#include <vector>

std::vector<double> A, B;

void foo(int iters, int x, int y)
{
    double *a = A.data();
    const double *b = B.data();

#pragma omp simd
    for (int i = 0; i < iters; i++)
    {
        a[x + i] += b[y + i];
    }
}
```

```
$ icc -vec-report3 -c code.cpp
code.cpp(9): (col. 3) remark: LOOP WAS VECTORIZED.
```

- Nesprávné zarovnání dat (na **16/32/64 bytů** pro **SSE, AVX** a **AVX-512**) je jeden z hlavních důvodů špatného výkonu.
- Proto kompilátor vkládá do kódu test na zarovnání.
 - Pokud nejsou data zarovnaná, generuje tzv. peel, body a reminder.
- **Zarovnání dat jen nutné explicitně zmínit když**
 - Alokujeme data
 - Deklarujeme nový pointer
 - Deklarujeme funkci, které má na vstupu pointer.
- **Penalizace:**
 - Nezarovnaný přístup vs. zarovnaný (ale stále v rámci **stejné cache line**) **40 %** výkonu dolů.
 - Nezarovnaný přístup vs. zarovnaný (ale **přes více cache line**) až o **500 %** horší.

Zdroj: <http://software.intel.com/en-us/articles/reducing-the-impact-of-misaligned-memory-accesses/>

- Pro nová statická pole zarovnaná na N byte:

- `__declspec(alignment(N)) type name[size];` // Intel
- `type name[size] __attribute__((aligned(N)));` // GNU
- `alignas(N) type name[bounds];` // C++-11

- Nově alokovanou dynamickou paměť:

- `_mm_malloc(size, N);` // X86 CPU linux
- `_mm_free(p);`
- `posix_memalign(void **memptr, size_t N, size_t size);` // POSIX
- `aligned_alloc(size_t N, size_t size)` // C++-11

- Lze rovněž přetížit alokátory new a delete

- Argumenty funkcí:

- `__assume_aligned(name, N);`

- Pro specifické smyčky:

- `#pragma omp simd align(a:64)`

Pokud je zarovnání porušeno, může nastat výjimka!

```
void matvec(double a[][COLWIDTH], double b[], double c[])
{
    int i, j;
    for(i = 0; i < size1; i++) {
        b[i] = 0;
    #pragma vector aligned
        for(j = 0; j < size2; j++)
            b[i] += a[i][j] * c[j];
    }
}
```

- Let's assume `a`, `b` and `c` are declared 16 byte aligned in calling routine
- **Question:** Would this be correct when compiled for Intel® SSE2?
- **Answer:** It depends on `COLWIDTH`!
 - In case `COLWIDTH` is even: Yes
 - In case `COLWIDTH` is odd: No, vectorized code would fail by alignment error after first row!
- **Solution:**
Instead of pragma use `__assume_aligned(<array>, base)` here as this refers to the start address only. It wouldn't allow vectorization, nevertheless!

Compiled both cases using **-xAVX**:

```
void mult(double* a, double* b, double* c)
{
    int i;
#pragma vector aligned
    for (i = 0; i < N; i++)
        c[i] = a[i] * b[i];
}
```

```
.B2.2:
vmovupd    (%rdi,%rax,8), %ymm0
vmulpd     (%rsi,%rax,8), %ymm0, %ymm1
vmovntpd   %ymml, (%rdx,%rax,8)
addq       $4, %rax
cmpq       $1000000, %rax
jb         .B2.2
```

```
void mult(double* a, double* b, double* c)
{
    int i;
#pragma vector unaligned
    for (i = 0; i < N; i++)
        c[i] = a[i] * b[i];
}
```

```
.B2.2:
vmovupd    (%rdi,%rax,8), %xmm0
vmovupd    (%rsi,%rax,8), %xmm1
vinsertf128 $1, 16(%rsi,%rax,8), %ymml, %ymm3
vinsertf128 $1, 16(%rdi,%rax,8), %ymm0, %ymm2
vmulpd     %ymm3, %ymm2, %ymm4
vmovupd    %xmm4, (%rdx,%rax,8)
vextractf128 $1, %ymm4, 16(%rdx,%rax,8)
addq       $4, %rax
cmpq       $1000000, %rax
jb         .B2.2
```

Compiler can create more efficient code if alignment can be guaranteed!

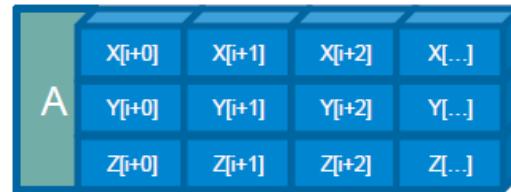
- Non-consecutive memory locations are being accessed in the loop
- Vectorization works best with contiguous memory accesses
- Vectorization might still be possible in cases of non-contiguous memory access but...
 - Data arrangement operations might be too expensive (e.g. access pattern linear/regular)
 - Vector report issued when too expensive:
Loop was not vectorized: vectorization possible but seems inefficient
- Examples:

```
for(i = 0; i <= MAX; i++) {  
    for(j = 0; j <= MAX; j++) {  
        D[i][j] += 1;                      // Unit stride  
        D[j][i] += 1;                      // Non-unit stride but linear  
        A[j * j] += 1;                     // Non-unit stride  
        A[B[j]] += 1;                      // Non-unit stride (scatter)  
        if(A[MAX - j]) == 1) last = j; // Non-unit stride  
    }  
}
```

```
struct Point;  
{  
    int x;  
    int y;  
    int z;  
};  
Point A[100]; //AoS
```

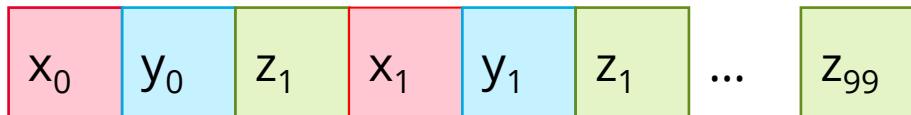


```
struct Points;  
{  
    int x[100];  
    int y[100];  
    int z[100];  
};  
Elements A; //SoA
```



- Pole struktur (AoS) vedou na špatné zarovnání v paměti a nejednotkový rozestup. Proto se špatně vektorizují.
- Struktury polí (SoA) mohou být snadno zarovnány, ale porušují OOP.
- Podpora speciálních knihoven, např. Intel SDLT
 - <https://www.intel.com/content/dam/www/public/us/en/documents/presentation/improving-vectorization-efficiency.pdf>

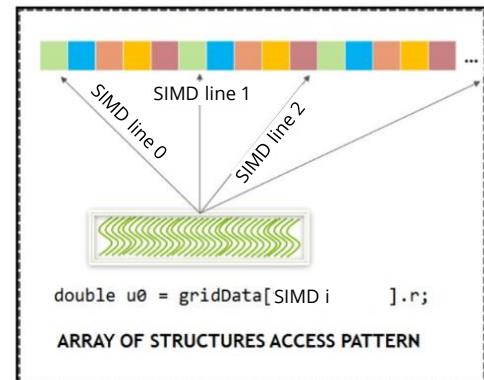
```
struct Point{  
    int x;  
    int y;  
    int z;  
};  
  
struct Point points[100];  
  
for (int i=0; i<100; i++) {  
    points[i].x += 10;  
    points[i].y += 20;  
    points[i].z += 30;  
}
```



Scalar – OK
Rozestup 1

SIMD – Problém
Rozestup 3

```
struct Coefficients_SOA {  
    int r;  
    int b;  
    int g;  
    int hue;  
    int saturation;  
    int maxVal;  
    int minVal;  
    int finalVal;  
};
```



- Všechna vlákna v SIMD čtou nejprve komponentu x, poté y a nakonec z
- Řešením je přeuspořádání struktury

```
struct Points{
    int x[4];
    int y[4];
    int z[4];
}

struct Points points;

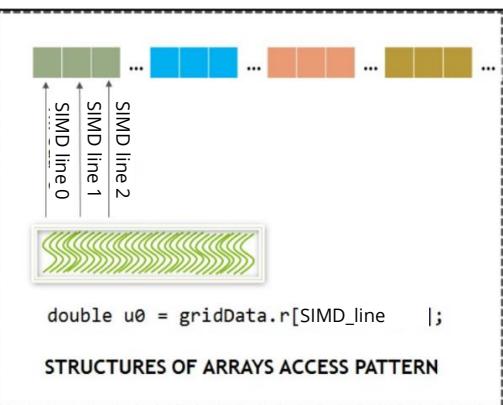
for (int i=0; i<100; i++) {
    points.x[i] += 10;
    points.y[i] += 20;
    points.z[i] += 30;
}
```



Scalar - Problém Rozestup 3

SIMD - OK Rozestup 1

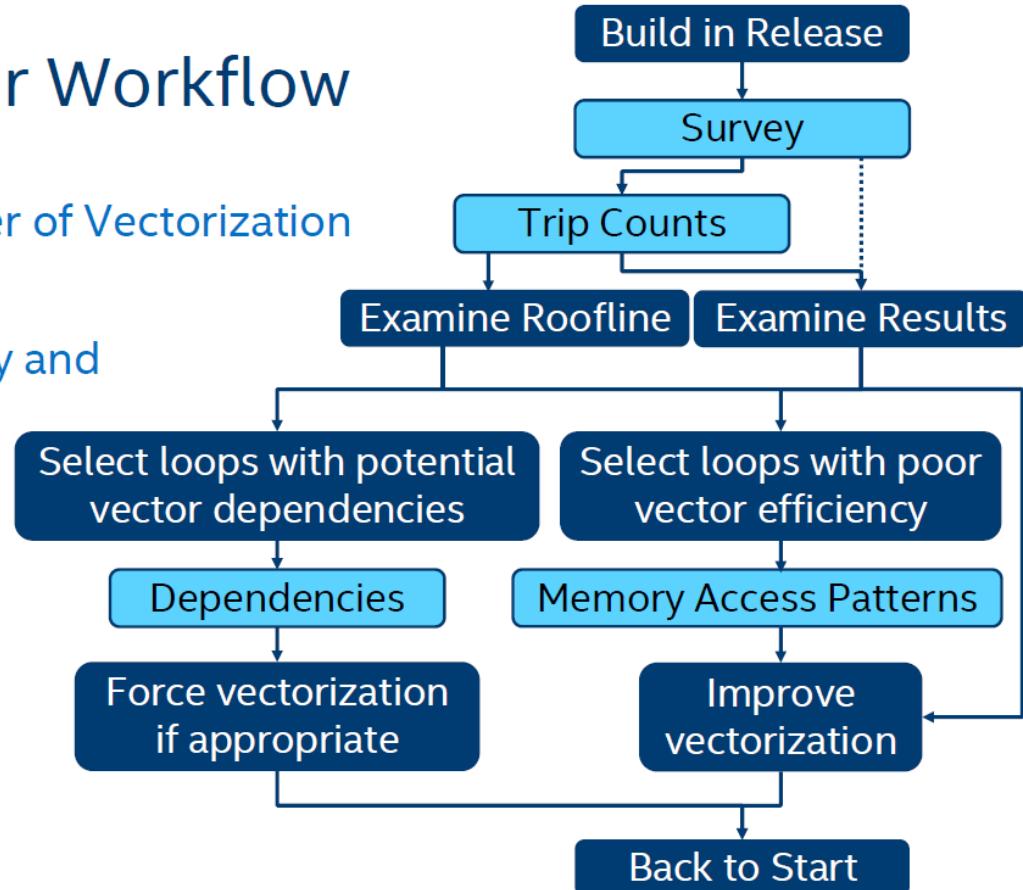
```
struct Coefficients_AOS {
    int* r;
    int* b;
    int* g;
    int* hue;
    int* saturation;
    int* maxVal;
    int* minVal;
    int* finalVal;
};
```



- Pro skalární a SIMD implementaci je nutné mít data uspořádaná jinak!
- Struktura polí rozbíjí Objektově orientovaný model

Vectorization Advisor Workflow

- **Survey** is the bread and butter of Vectorization Advisor! All else builds on it!
- **Trip Counts** adds onto Survey and enables the **Roofline**.
- **Dependencies** determines whether it's safe to force a scalar loop to vectorize.
- **Memory Access Patterns** diagnoses vectorization inefficiency caused by poor memory striding.



Survey Vectorization Advisor

Function/Loop Icons

- Scalar Function
- Vector Function
- Scalar Loop
- Vector Loop

Tip:

For vectorization, you generally only care about loops. Set the type dropdown to "Loops".

Efficiency is important!

$$\text{Speedup} = \frac{\text{Efficiency} = 100\%}{\text{Vec. Length}}$$

The black arrow is 1x. Gray means you got less than that. Gold means you got more. You want to get this value as high as possible!

Vectorizing a loop is usually best done on innermost loops. Since it effectively divides duration by vector length, you want to target loops with high self time.

| Function Call Sites and Loops | | Vector Issues | Self Time | Total Time | Type | Why No Vectorization? | Vectorized Loops | | | |
|-------------------------------------|----------------------------------|--------------------------|-----------|------------|-----------|---|------------------|------------|---------|------|
| | | | | | | | Vect... | Efficiency | Gain... | VL . |
| <input checked="" type="checkbox"/> | [loop in main at example.cpp:38] | 1 Assumed depend... | 0.391s | 0.391s | Scalar | <input checked="" type="checkbox"/> vector depen... | | 2% | 0.37x | 16 |
| <input checked="" type="checkbox"/> | [loop in main at example.cpp:64] | 1 Possible inefficien... | 0.297s | 0.297s | Vector... | | AVX2 | 8% | 1.23x | 16 |
| <input checked="" type="checkbox"/> | [loop in main at example.cpp:51] | 1 Possible inefficien... | 0.094s | 0.094s | Vector... | <input checked="" type="checkbox"/> 1 vectorizatio... | AVX2 | 100% | 7.98x | 8 |
| <input checked="" type="checkbox"/> | [loop in main at example.cpp:26] | | 0.030s | 0.030s | Vector... | | AVX2 | | | |
| <input checked="" type="checkbox"/> | [loop in main at example.cpp:14] | 3 Assumed depend... | 0.000s | 0.000s | Scalar | <input checked="" type="checkbox"/> vector depen... | | | | |
| <input checked="" type="checkbox"/> | [loop in main at example.cpp:23] | | 0.000s | 0.030s | Scalar | <input checked="" type="checkbox"/> inner loop, w... | | | | |

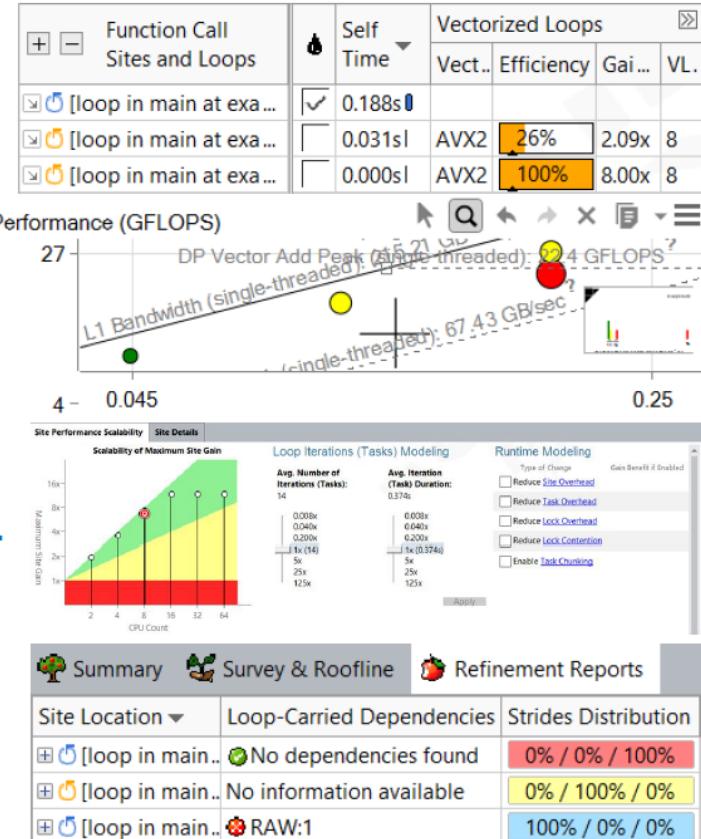
Expand a vectorized loop to see it split into body, peel, and remainder (if applicable).

Advisor advises you on potential vector issues. This is often your cue to run MAP or Dependencies. Click the icon to see an explanation in the bottom pane.

The Intel Compiler embeds extra information that Advisor can report in addition to its sampled data, such as why loops failed to vectorize.

Summary

- V T Survey** – Find the most promising sites for threading, see the meat of the vectorization information, and get recommendations from Advisor.
- V T Trip Counts & FLOPS** – Add to your Survey report to help fine-tune vector efficiency and capability, as well as unlock the powerful **Roofline** to visualize your bottlenecks and help direct your efforts.
- T Suitability** – Predict how well your proposed threading model will scale under certain conditions quickly and easily.
- V T Dependencies** – Prove or disprove the existence of parallel dependencies and learn how to fix them.
- V Memory Access Patterns** – See how you traverse your data and how it affects your vector efficiency and cache bandwidth usage.



KNIHOVNA OPENMP

- **#pragma omp simd [clause[,] clause] ...]**
 for ();
- **#pragma omp declare simd [clause[,] clause] ...]**
 function();
- **pragma omp for simd [clause[,] clause] ...]**
 for (...)
- **Clause:** safelen(length), linear(list[:linear-step]), aligned(list[:alignment]), private(list), lastprivate(list), reduction(reduction-identifier:list), collapse(n), simdlen(length), linear(argument-list[:constant-linear-step]), aligned(argument-list[:alignment]), uniform(argument-list), inbranch, notinbranch
- **Podporováno od gcc-4.9**
- Více info na <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>

- Bez direktivy SIMD se kompilátoru smyčku nepodaří vektorizovat kvůli řešení pointer aliasingu
- Pragma SIMD kompilátoru říká, že vektorizace je možná. Kompilátor tedy vypne všechny heuristiky a provede vektorizaci.
- Odpovědnost za dodržení pravidel vektorizace přebírá programátor.

```
void add(float* a, float* b, float* c,
         float* d, float* e, int n)
{
    #pragma omp simd
    for (int i=0; i<n; i++)
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

```
#pragma omp simd reduction(+:sum)
for(i = 0; i < *p; i++)
{
    A[i] = B[i] * C[i];
    sum = sum + A[i];
}
```

- **Programátor se zaručuje, že:**

- $*p$ je loop invariant, tedy hodnota na adrese p je konstanta.
- sum není aliasovaná s $B[]$ nebo $C[]$
- $A[]$ se nepřekrývá s $B[]$ nebo $C[]$
- Na sum se má pohlížet jako na redukovanou proměnnou
- Kompilátor může přeházet pořadí operací pro lepší výkon
- Kód bude vektorizován i když interní heuristiky kompilátoru říkají, že to zhorší výkon.



- **safelen (length)**
 - Maximální počet iterací, které se mohou vykonávat současně bez porušení závislostí
 $(a[i] += a[i - 10])$
 - V praxi je to maximální délka vektorového registru
- **simdlen (length)**
 - Délka preferovaného vektorového registru
- **linear (list[:linear-step])**
 - Hodnota proměnné je ve vztahu k číslu iterace
 - $x_i = x_{\text{orig}} + i * \text{linear_step}$
- **aligned (list[:alignment])**
 - Dané proměnné jsou zarovnány na daný počet bytů
 - Defaultní hodnota je dána architekturou.
- **collapse (n)**
 - Kolik smyček pod sebou se má zkombinovat do jedné.

- SIMD-enabled funkce umožnují definovat funkce, které lze volat v rámci vektorizovaného kódu (smyčky).
- Direktiva se vkládá nad deklaraci (prototyp funkce do hlavičkového souboru).
- **Zápis funkce je stejný jako pro skalární variantu (jeden element).**
- Programátor:
 - Napíše standardní funkci, která pracuje se skalárními parametry
 - Anotuje funkce pomocí pragmy a dovětků `#pragma omp declare simd`
 - Dále se nestará o to, jestli je funkce použita ve skalární nebo vektorové smyčce.
- Kompilátor:
 - Generuje skalární i vektorovou verzi/verze.
 - Volá správnou variantu funkce

Ukázka SIMD-Enabled funkcí

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}
```

```
vec8 min_v(vec8 a, vec8 b) {
    return a < b ? a : b;
}
```

```
#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}
```

```
vec8 distsq_v(vec8 x, vec8 y) {
    return (x - y) * (x - y);
}
```

```
void example() {
    #pragma omp for simd
    for (i=0; i < N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
    }
}
```

```
vd = min_v(distsq_v(
    va, vb),
    vc)
```

- **simdlen (length)**
 - Generuje funkci s podporou dané délky registru. Lze takto generovat specifické funkce pro SSE, AVX a AVX512
- **linear (list[:linear-step])**
 - Hodnota proměnné je ve vztahu k číslu iterace
 - $x_i = x_{\text{orig}} + i * \text{linear_step}$
- **uniform (list[:linear-step])**
 - Hodnota proměnné je konstantní ve všech iteracích
- **aligned (list[:alignment])**
 - Dané proměně jsou zarovány na daný počet bytů
- **inbranch**
 - Funkce je vždy volána zevnitř podmínky
- **notinbranch**
 - Funkce není nikdy volána zevnitř podmínky

Ukázka inbranch a notinbranch

```
#pragma omp declare simd inbranch  
float do_stuff(float x) {  
    /* do something */  
    return x * 2.0;  
}
```

```
vec8 do_stuff_v(vec8 x, mask m) {  
    /* do something */  
    vmulpd x{m}, 2.0, tmp  
    return tmp;  
}
```

```
void example() {  
    #pragma omp simd  
    for (int i = 0; i < N; i++)  
        if (a[i] < 0.0)  
            b[i] = do_stuff(a[i]);  
}
```

```
for (int i = 0; i < N; i+=8) {  
    vcmp_lt &a[i], 0.0, mask  
    b[i] = do_stuff_v(&a[i], mask);  
}
```

- Proč je potřebujeme?
- Protože bez nich je každý parametr funkce brán jako vektor

```
#pragma omp declare simd uniform(a) linear(i:1)
void foo(float* a, int i):
    a is a pointer
    i is a sequence of integers [i, i+1, i+2, ...]
    a[i] is a unit-stride load/store ([v]movups)
```

```
#pragma omp declare simd
void foo(float* a, int i):
    a is a vector of pointers
    i is a vector of integers
    a[i] becomes gather/scatter.
```

- Reference: <http://software.intel.com/en-us/articles/usage-of-linear-and-uniform-clause-in-elemental-function-simd-enabled-function-clause>

| Datový typ | Obsah | SSE extension | SSE2 extension | SSE4 extension |
|----------------------|-----------------------------------|---------------|----------------|----------------|
| <code>__m128</code> | 4 x float | Available | Available | Available |
| <code>__m128d</code> | 2 x double | Not available | Available | Available |
| <code>__m128i</code> | 16 x char
8 x short
4 x int | Not available | Available | Available |

- Most intrinsic names use the following notational convention:

mm <intrinsic_op> _<suffix>

- <intrinsic_op> indicates the basic operation of the intrinsic; for example, add for addition and sub for subtraction.
- <suffix> denotes the type of data the instruction operates on.
 - The first one or two letters of each suffix denote whether the data is
 - p packed
 - ep extended packed
 - s scalar
 - The remaining letters and numbers denote the type, with notation as follows:
 - s single-precision floating point
 - d double-precision floating point
 - i32 signed 32-bit integer
 - u32 unsigned 32-bit integer

Pokračování příště

Procesory s vláknovým paralelismem, Příklady x86 procesorů

AVS – Architektury výpočetních systémů

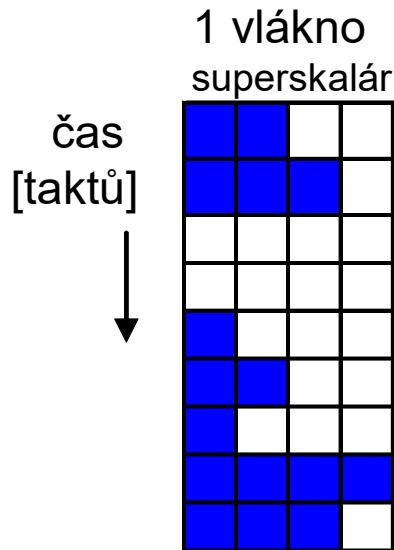
Týden 6, 2024/2025

Jirka Jaroš

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



- Použitím řady technik v mikroarchitektuře bylo dosaženo dramatické zvýšení výkonnosti CPU při zpracování jednoho kódu (vlákna).
 - Vyšší hod. kmitočet (počet stupňů linky),
 - zvýšení IPC (šířka linky),
 - superskalární zpracování,
 - odstranění konfliktů mezi instrukcemi – OOO,
 - cache L1 – L3 na čipu,
 - prediktory skoků,
 - až 6 instrukcí dokončuje v 1 taktu.
- **Pokusy pokračovat v rozvoji těchto technik nevedou již k prokazatelnému zvýšení výkonnosti.** Vadí při tom
 - paměťové latence,
 - nízké využití funkčních jednotek (~ 30 %).
- **Řešení:** technologie multithreading a multi-core.



Průměrné IPC = **16** / 9 = 1,78

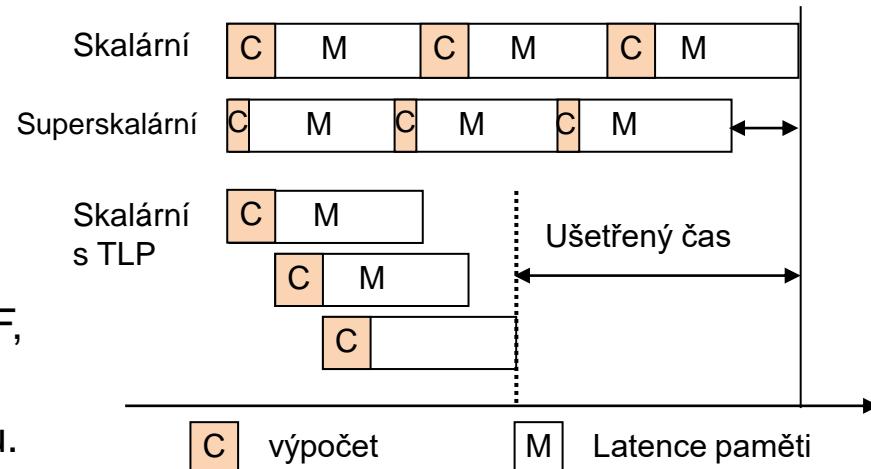
$$\begin{aligned}\text{Využití} &= \text{průměrné IPC} / \text{max. IPC} \\ &= 1,78 / 4 = 44,5 \%\end{aligned}$$

Běžné **4 cestné** superskalární CPU mají průměrné udržitelné IPC v rozmezí **1,5–2**, tedy využití < 50 %.

Důvody nízkého využití:

- špatně predikované skoky (zbytečné výpočty)
- čekání na data kvůli výpadkům, zvláště v L3 cache
- čekání na data kvůli instrukčním závislostem
- čekání na volnou funkční jednotku

- Pracovní zátěž serverů (**transakce**) je charakterizována vysokým TLP a nízkým ILP (**databáze, web servery**).
- Možnosti zvýšit výkonnost jednoho vlákna jsou omezené (vlákna dlouho čekají na vyřízení **výpadků mimo čip**)
- Proto je snaha spojit **vysoký TLP aplikací** s **podporou pro více vláken na čipu procesoru**.
- Vlákno = proud instrukcí.** Kontext: IP + SP, RF, PSW včetně myid a priority.
- Vlákna pracují ve sdíleném adresovém prostoru.
- Vlákna jsou (oproti procesům) „lehká“: rychlejší nebo okamžité přepnutí kontextu, málo nebo žádné kopírování.



1. Časový multithreading (TMT, temporal MT):

- Vlákna **se střídají** na jednom jádru **kvůli** jeho **lepšímu využití, vyššímu IPC.**
- Lze skrýt výpadky.
- Procesory TLP = procesory s podporou TMT v jádrech.

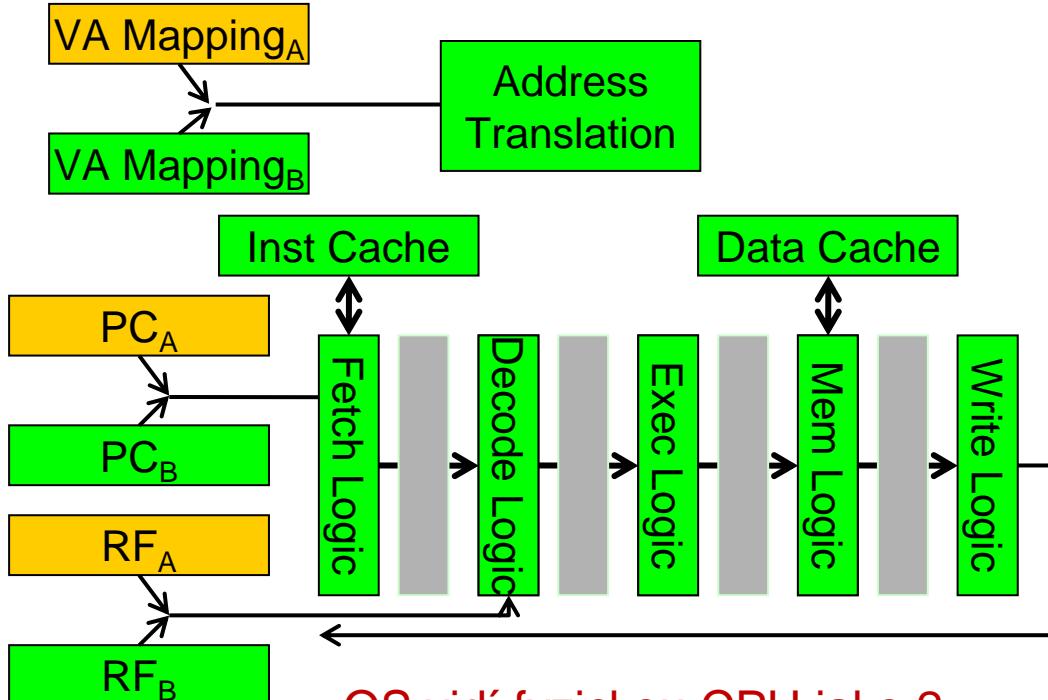
2. Prostorový MT:

- Vlákna **běží paralelně** na více-jádrovém procesoru, 1 vlákno na 1 jádru, **kvůli zvýšení výkonnosti** vzhledem k 1 jádru.

3. Kombinace 1 a 2:

- Čipový MultiProcessing / Multithreading, **CMP/MT** tj. víc vláken na každém jádru.

V každém případě je třeba zajistit co nejrychlejší přepínání vláken. Proto musí mít každé vlákno **svůj HW kontext**.

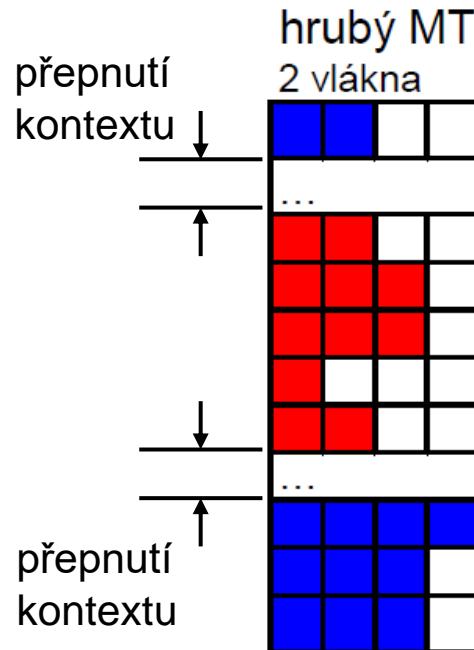


- Prostředky **sdílené**:
 - CPU
 - všechny cache
 - prediktor skoků
 - funkční jednotky
- Prostředky **replikované**:
 - sady registrů RF
 - čítače programů PC
 - ukazovátka zásobníku
 - TLB
 - RSB, return stack buffer
 - různé řídicí registry
 - řadič přerušení APIC
- Prostředky **nové**: HW pro výběr vlákna

- Velký soubor virtuálních registrů je již k dispozici, stačí
- **Oddělené tabulky RAT** přejmenování registrů pro vlákna
 - Stejný registr každého vlákna je mapován do jiného fyzického registru mapovací tabulkou RAT
- Schopnost dokončit instrukce z několika vláken
 - Oddělený ROB pro každé vlákno.
 - Levnější řešení: jen 1 ROB a v 1 taktu dokončují instrukce jen z 1 vlákna
- Je třeba zajistit stejnou **výkonnost s TLP HW jako bez něj i když běží jen 1 vlákno**
- **TMT** se jeví nejslibnější pro maximalizaci výkonnosti procesorů ILP, VLIW a vícejádrových.
- Závisí na aplikaci, IPC může vzrůst až o 30 %.

- OS pokládá jednu fyzickou CPU za několik standardních oddělených logických CPU (LCPU).
- Vše co je potřeba pro využití MT je **podpora symetrického multiprocesingu (SMP) v OS.**
- **Ale pozor!** Nemá-li plánovač v OS povědomí o MT, mohl by špatně rozložit zátěž, např. naplánovat 2 vlákna na 1 jádro (víc se zahřívá) a druhé jádro nechat bez práce.
- Plánovač proto musí rozlišovat fyzické CPU a Logické CPU.
Všechny moderní OS poběží s aktivovanou technologií MT (Intel HT) a „umí“ detekovat a řešit uvedenou situaci. (Microsoft Windows 7 a vyšší).

Jedno vlákno běží řadu taktů, k přepnutí kontextu dochází pouze při události, která zablokuje linku na mnoho taktů (např. výpadek v I/D-cache, timeout, špatná predikce).



Co s rozpracovanými instrukcemi v lince:

- zahodit (*flush*) a opakovat při další aktivaci, vyžaduje krátkou linku
- dokončit paralelně s instrukcemi nového vlákna (délka linky nehraje roli)
- dokončit, teprve pak nové.

- **Předpoklad (konzervativní):**
 - kmitočet CPU 1 GHz, výpadek v L1 I-cache 20 ns
 - bez výpadků 1 instr/takt, výpadek 1x za 100 instrukcí
- **Bez MT** je doba vykonání 100 instrukcí + 1 výpadku 100 + 20 taktů -> $IPC = 100 / 120 = 0,83$.
- **S hrubým MT (přepnutí kontextu 3 takty)** je doba vykonání 100 instrukcí + 2 přepnutí = 100 + 6 taktů (pokud jde kód rozdělit na 2 vlákna)
 $IPC = 100 / 106 = 0,94$ (zlepšení o $0,94 / 0,83 = 1,14$
tj. o 14%)

- **Kdy dojde k přepnutí vlákna:**
 - kromě výpadku v aktuálním vláknu musí existovat jiné připravené vlákno **nebo**
 - je připraveno vlákno s vyšší prioritou než je priorita aktuálního vlákna **nebo**
 - existuje vlákno, které posledních n cyklů neprovědlo žádnou instrukci (eliminace hladovění).
- **Vlastnosti hrubého MT:**
 - vhodný jen pro dlouhá blokování (např. výpadky LLC)
 - důležitá je co nejmenší doba přepnutí kontextu
 - spravedlivost u vláken s různou četností výpadků: vlákno s malým počtem výpadků je třeba **omezit časovým kvantem**.
- **Poznámka:** připravené vlákno čekající ve smyčce (např. na kritickou sekci) nemá smysl přepínat na běh. Lze je odstavit spec. instrukcemi (viz dále) nebo mu snížit prioritu.

- **1 vláknو:**

R – průměrný počet taktů při zpracování instrukcí mezi paměťovými operacemi
 L – latence (paměťové) operace [taktů]

účinnost $E_1 = \frac{R}{R + L} = \frac{1}{1 + L/R}$

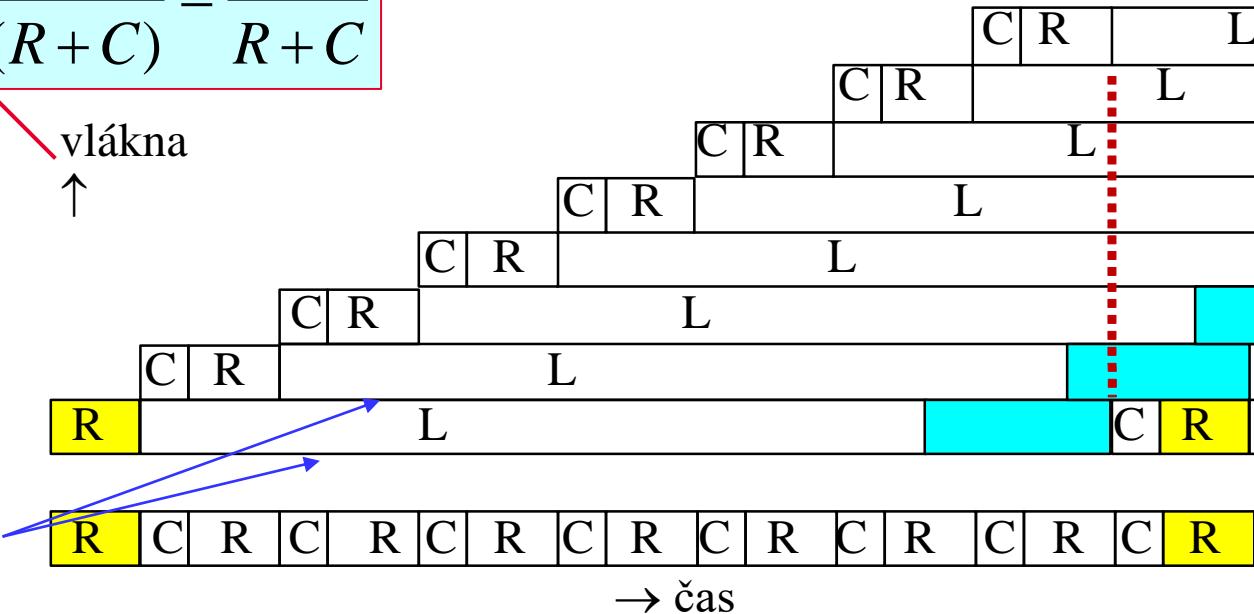
- **N vláken:** jedno přepnutí kontextu stojí C [taktů]

$$E = \frac{\text{doba práce}}{\text{doba}(práce + režie)} = \frac{\text{doba práce}}{\text{doba}(práce + přepínání + čekání)}$$

$$E_{sat} = \frac{NR}{N(R+C)} = \frac{R}{R+C}$$

vlákna
↑

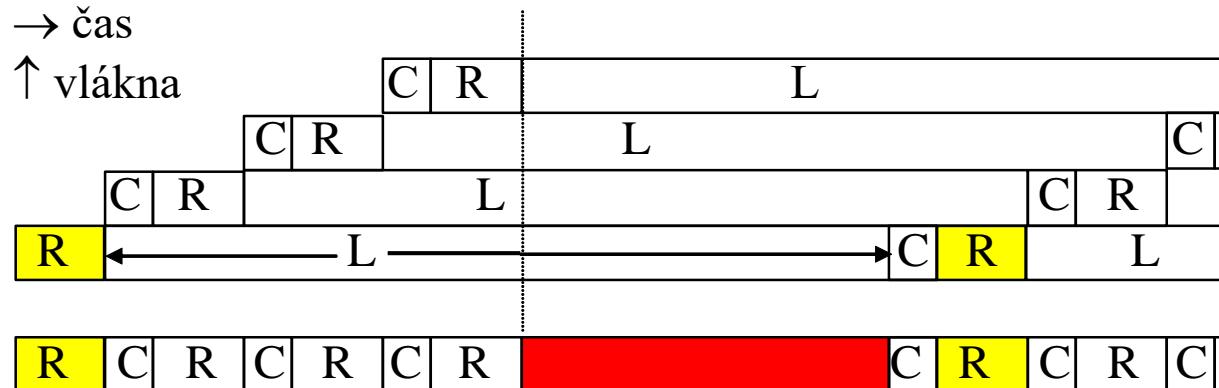
Předpoklad:
výpadky
neblokují
přístupy
dalších
vláken do
cache



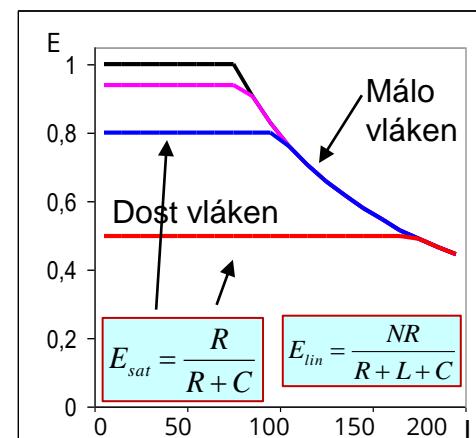
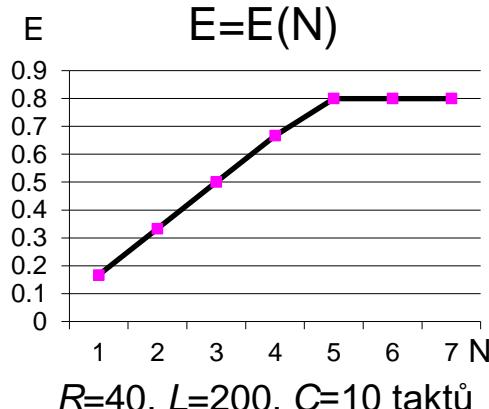
$$(N-1)(R+C) \geq L, \quad N \geq \frac{L}{R+C} + 1 = N_{sat}$$

Málo vláken – účinnost pod maximem

$$E_{lin} = \frac{NR}{R + L + C}$$



$$(N-1)(R+C) < L, \quad N < N_{sat} = \frac{L}{R+C} + 1$$



Multivláknová CPU pracuje s N kontexty

- přepnutí kontextu trvá 10 taktů
- doba práce v jednom kontextu činí průměrně 80 taktů.
- Přepnutí kontextu se provádí při každém výpadku v paměti cache, přičemž vyřízení výpadku trvá 270 taktů.

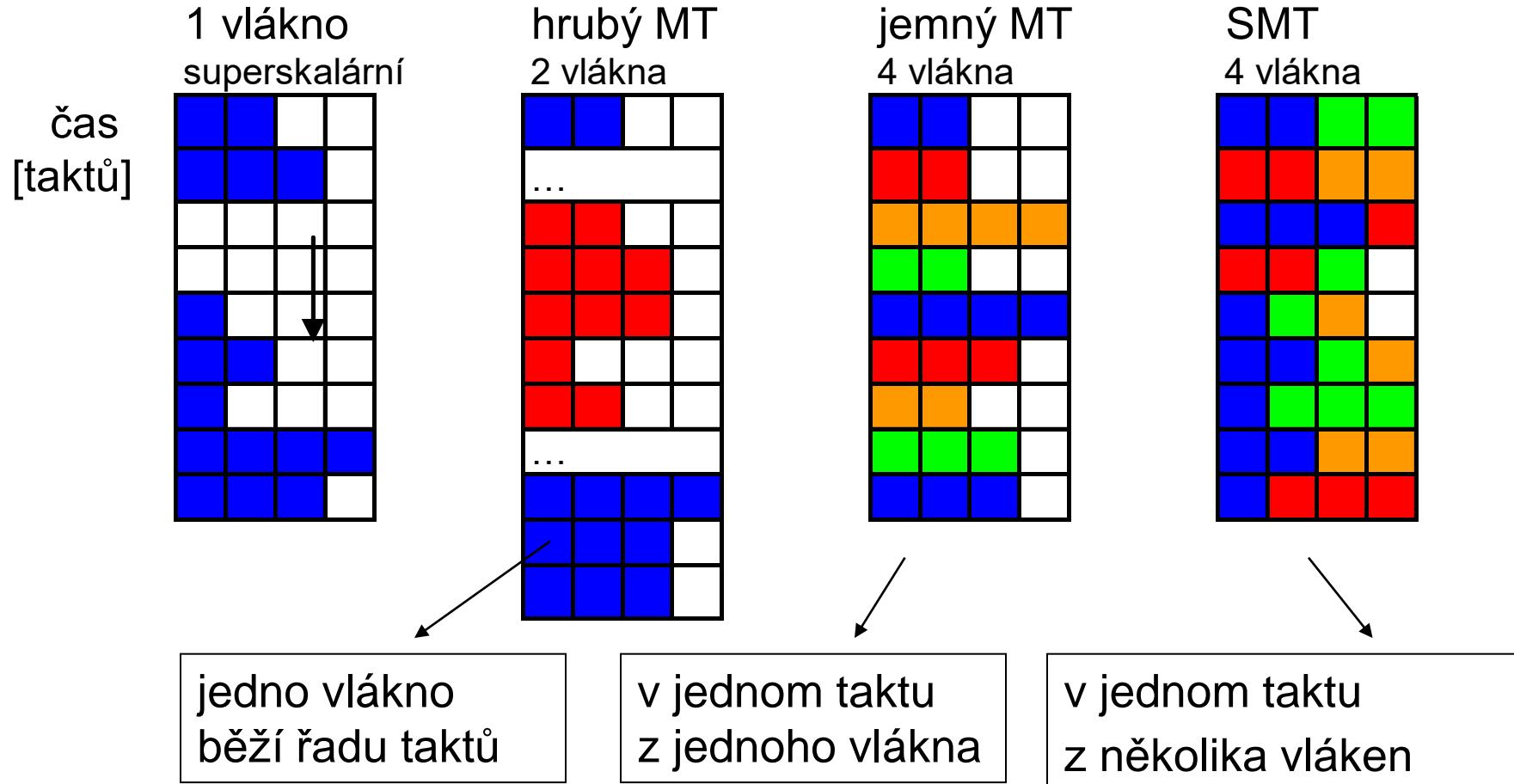
Najděte

1. kritický počet kontextů na hranici mezi lineární a saturovanou oblastí
2. využití CPU (tj. účinnost E) pro $N = 8$.

$$N \geq N_{sat} = \frac{L}{R + C} + 1 = \frac{270}{80 + 10} + 1 = 4$$

$$E_{sat} = \frac{R}{R + C} = \frac{80}{80 + 10} = 88,89\%$$

- **Jemný MT.** V každém taktu se přepíná na jiné vlákno (prokládání vláken).
 - výběr vláken např. cyklicky, vynechají se zablokovaná vlákna
 - umí krátká i dlouhá blokování
 - ale zpomalí provedení individuálních vláken
 - nulová doba přepnutí kontextu.
- **SMT**, simultánní MT (*Simultaneous MultiThreading*).
 - v jednom taktu se zpracovávají instrukce z několika vláken.
 - kontext se přepíná každý takt, podskupina připravených vláken dodá instrukce současně, opět s nulovou režií.



- SMT se kromě překrývání latence některých instrukcí snaží i o lepší využití funkčních jednotek (vyšší ILP).
- Tomu odpovídá i strategie výběru vláken:
 - cyklicky v každém taktu pevný počet instrukcí z pevného počtu vláken
 - upřednostní vlákna s nejmenším počtem rozpracovaných výpadků nebo
 - s nejmenším počtem instrukcí ve stupních dekódování, přejmenování a v RS
 - upřednostní vlákna (s nejvyšší pravděpodobností) na správné větví.

IMPLEMENTACE MULTITHREADINGU

- Hyperthreading je firemní název pro **dvouvláknový SMT**; zaveden u Pentia III a Pentia 4 (mikroarchitektura **NetBurst**).
- HT byl kritizován pro velkou spotřebu a vynechán u 1. gen. mikroarchitektury **Core**, nicméně u pozdějších vícejádrových **Nehalem**, **Sandy Bridge**, **Haswell** i **Atom** je znova zaveden.
- HT lze povolit nebo zakázat v BIOSu
- První procesory **Itanium** používaly hrubý 2 vláknový MT nazývaný SoEMT–Switch on Event Multithreading.
- Procesory **Tukwila** a **Poulson** používají 2 + 2 vláknový SMT (2 vlákna v přední a 2 v zadní části linky).
- Intel **Xeon Phi** má 4 cestný jemný MT (vlákna se střídají). Na rozdíl od HT jej nelze vypnout.

1. Replikace, zdvojení stejných prostředků pro každé vlákno

- Registry, RSB, ITLB (velké stránky)

2. Rozdělení prostředků na vlákna (staticky, s označením čísla vlákna)

- Load buffer, store buffer, ROB, ITLB (malé stránky)

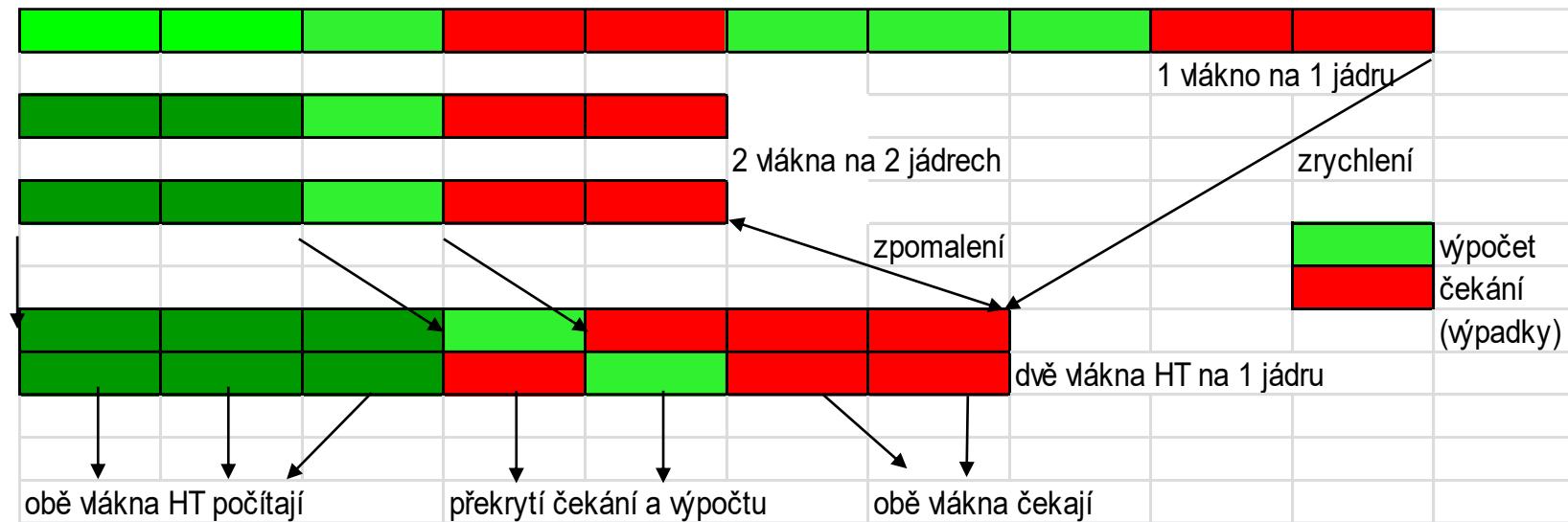
3. Soutěživé sdílení, závisí na dynamickém chování vláken.

- Rezervační stanice, cache L1 až L3, datový TLB, TLB druhé úrovně

4. Sdílené prostředky, které si nejsou vědomy existence vláken

- Funkční jednotky

- Logické CPU se jeví jako standardní oddělené CPU, ale mají jeden virtuální adresový prostor, rozdělený D-TLB (položky označeny ID bitem vlákna) a společnou virtuálně adresovanou L1 cache (virtuální index, fyzický tag).
- Položky v L1 cache nejsou označeny identitou vlákna, takže je žádoucí, aby jedno vlákno HT nepřepisovalo položky druhého.
 - Např. privátní zásobníky vláken musí začínat na jiných VA, aby se mapovaly na jiná místa v L1 cache (zařídí OS). Jinak by docházelo k vzájemnému vyhazování bloků z L1 do L2 (write back).
- 2 operační režimy: sdílený a adaptivní (nastaví se v BIOSu).
 - Sdílený: soutěživé sdílení, každé vlákno má svou exkluzivní část L1 cache podle potřeby.
 - Adaptivní (default): každé vlákno má přístup do celé L1 cache.

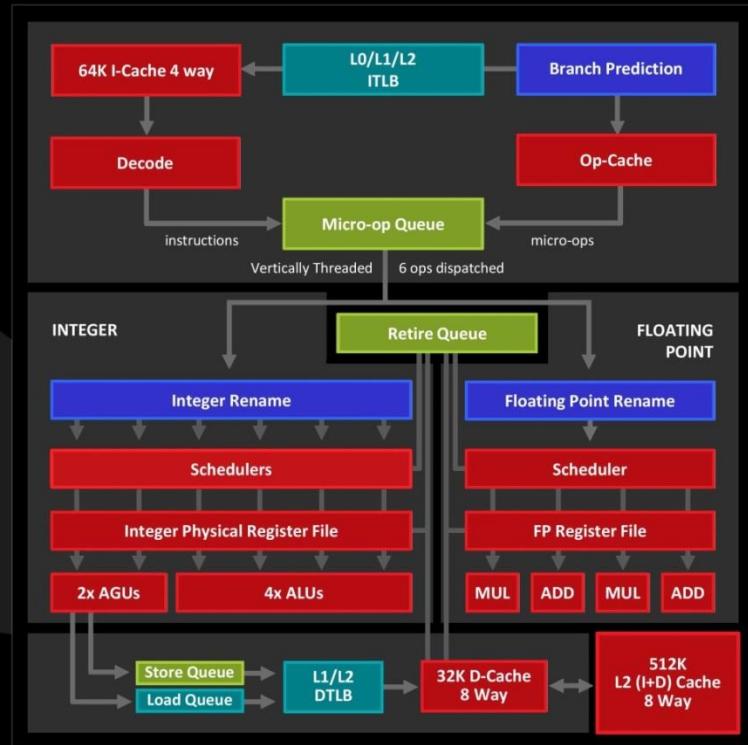


2 vlákna v HT běží pomaleji než 2 vlákna na 2 jádrech.

Důvody:

1. Dvě vlákna HT počítají déle protože sdílí FJ a obě třeba požadují stejnou FJ, takže jedno vlákno musí čekat na volnou FJ.
2. Dvě vlákna v HT čekají déle – úspěšnost 2 vláken v jedné cache je horší než úspěšnost každého vlákna ve vlastní cache.

- **IBM POWER7** (2010) má 8 jader, každé 4 vlákna.
MT lze přepínat mezi SMT2 nebo SMT4 a tak optimalizovat jádro na nejkratší odezvu nebo maximální propustnost (tzv. **dynamický MT**).
- **IBM POWER8** (2013) má až 8 vláken na 1 jádro. Může běžet v 1-, 2-, 4- nebo 8-SMT módu a přepínat mezi nimi dynamicky.
- **Oracle/Sun Sparc M7** server procesor (2015, 20 nm, 32 jader, 10 miliard tranzistorů, 3,6 GHz); 1–8 vláken na jádro.
- Od r. 2016 **AMD** ve svých procesorech **Zen** opouští CMT (Clustered MultiThreading) a nasazuje SMT.

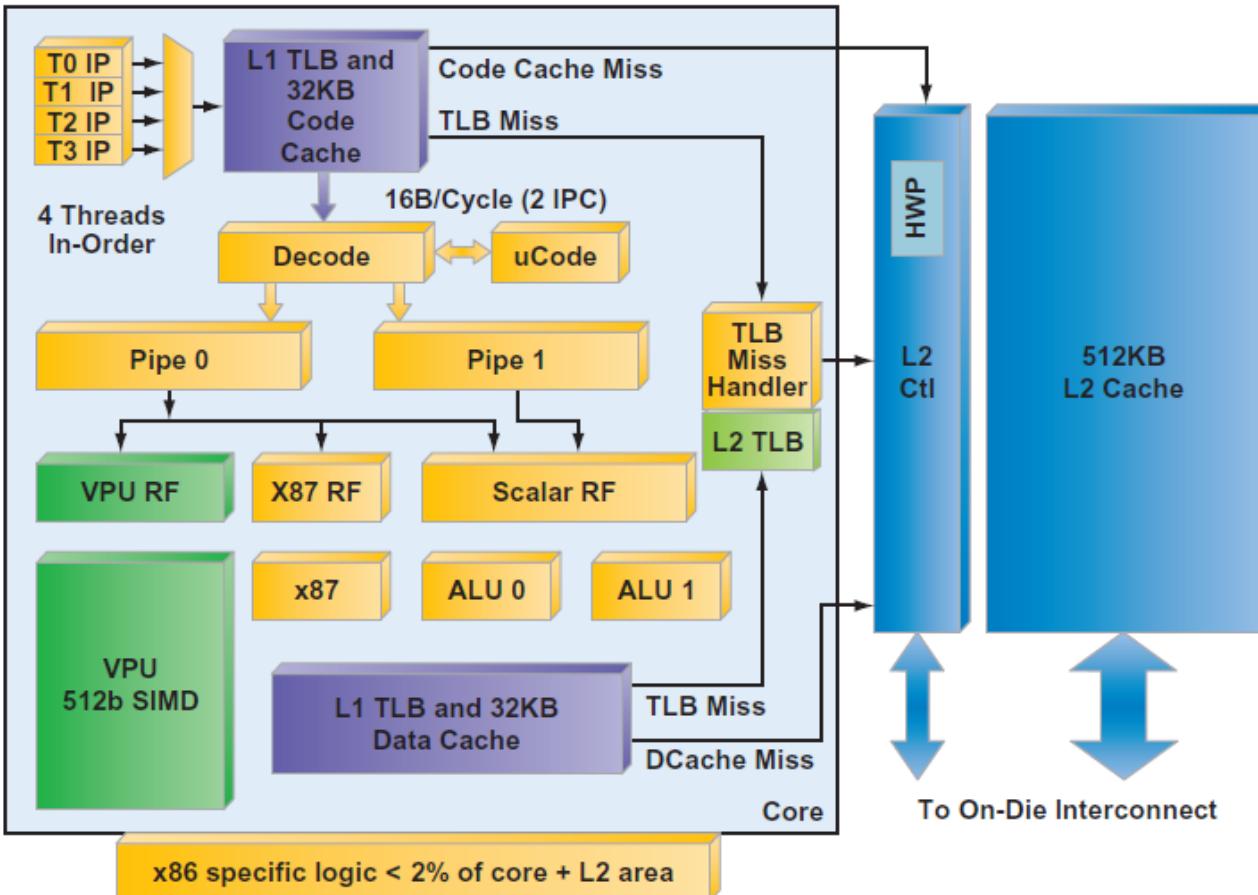


SMT OVERVIEW

- ▲ All structures fully available in 1T mode
- ▲ Front End Queues are round robin with priority overrides
- ▲ Increased throughput from SMT

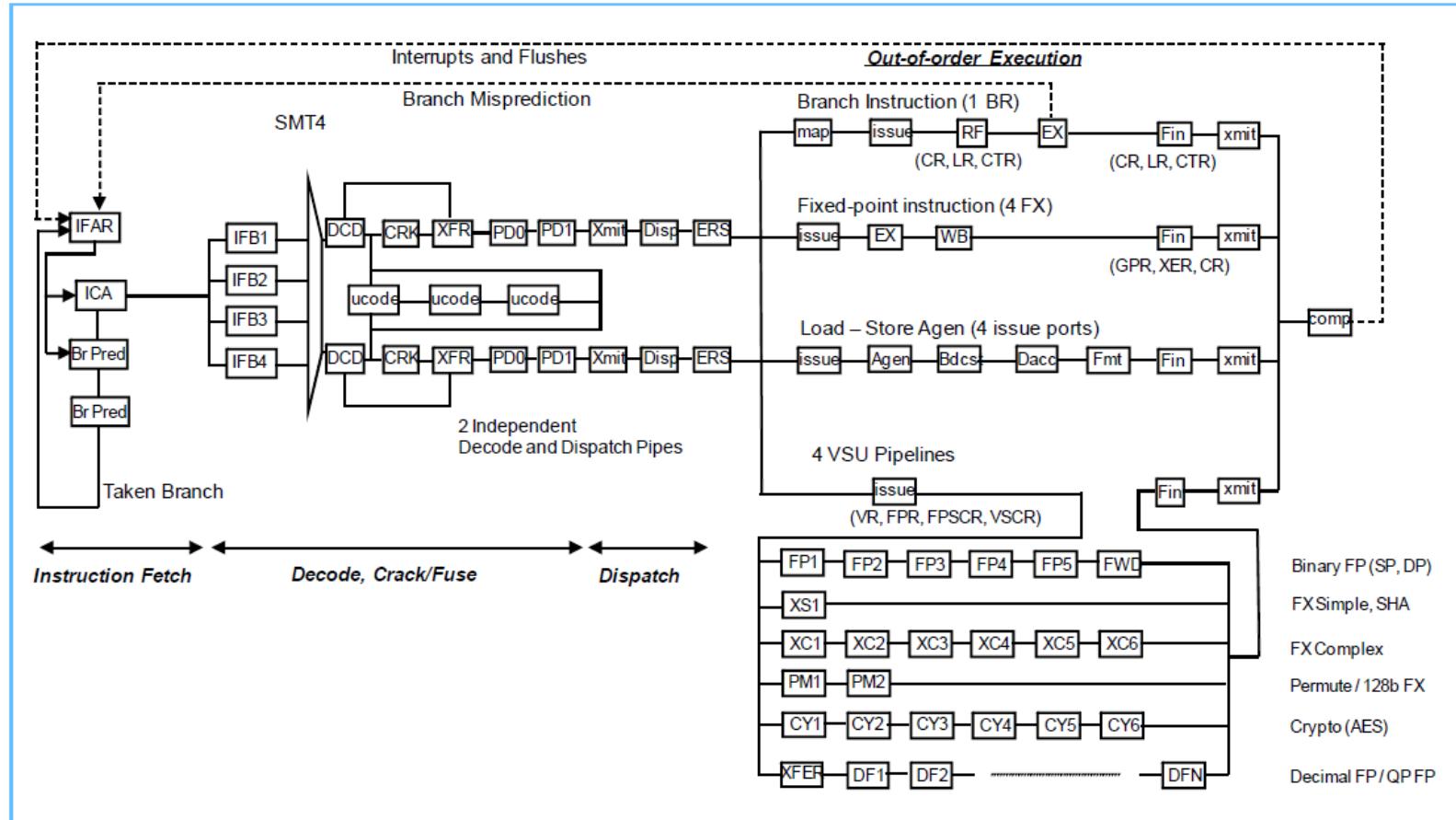
- Competitively shared structures
- Competitively shared and SMT Tagged
- Competitively shared with Algorithmic Priority
- Statically Partitioned

Intel Xeon Phi (4-way TLP)



- Pentium ISA včetně x87
- In-order zpracování
- 64b adresování
- 4 HW vlákna/jádro
- 2 cykly na dekódování instrukcí
- 2 instrukce za takt (vektorová + skalární)
- Latence vektorových operací je 4 takty
- Dvě pipeline

IBM Power 9 – Pipeline



HISTORIE A PŘÍKLADY SUPERSKALARNÍCH ARCHITEKTUR

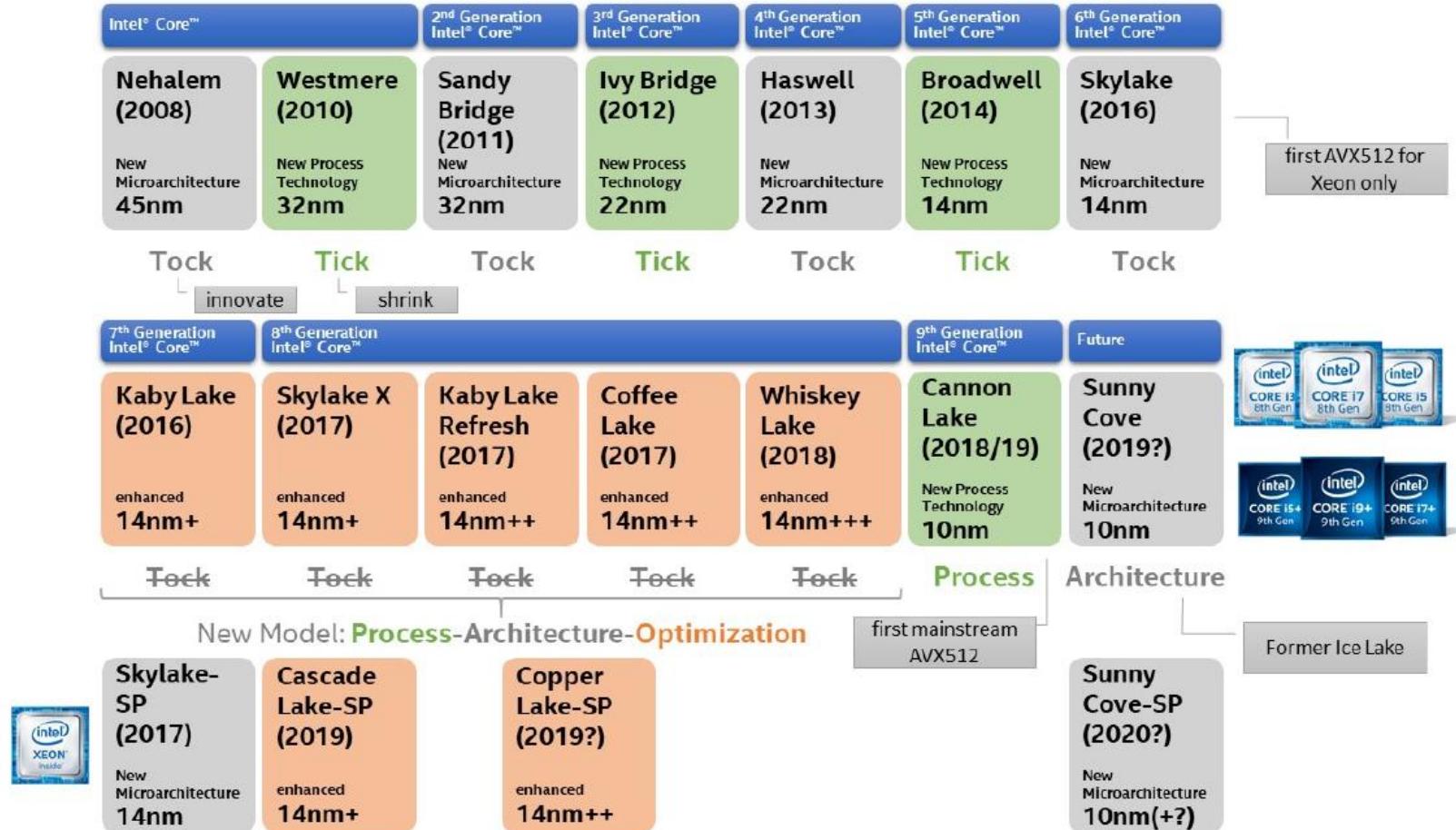
- P5 (1993):
 - první superskalární IA-32 mikroarchitektura – 1993:
 - In Order, dvojitá integer pipeline (**U** a **V**) 5 stupňů
 - Dokončují až 2 instrukce/takt. Kompilátor plánoval dvojice staticky.
- P6 (1995):
 - OOO, zavedeno super-řetězení (14 stupňů).
 - Procesory: Pentium Pro, Pentium II, III; MMX a SSE.
 - Modernizovaná P6: Pentium M, Core Solo, Core Duo.
- NetBurst (2000):
 - Trace cache, 31 stupňů, SSE2, SSE3, hyper-threading HT, EM64T.
 - Procesory: Pentium 4, Pentium D, Xeon
- Core (2006):
 - příkon ↓, 14 stupňů pipeline, 65 nm, multi-core, SSE3, Intel 64
 - Procesory: Pentium dual core, Celeron, Xeon, Core 2
- Nehalem (2008): řady: i3, i5, i7
 - 45 nm, HT, L3C, Quick Path, integrované MemCtrl, bufer μops.
 - 32 nm Nehalem = Westmere: IGP (Integrated GPU).

- Sandy Bridge 2010:
 - 32 nm, AVX 256 bitů, μop-cache, HT.
 - 22 nm Sandy Bridge = Ivy Bridge: 3D-tranzistor.
- Haswell 2013:
 - 22 nm, 4 ALU, 3 AGU, 2 jednotky predikce skoků, AVX2, FIVR (Fully Integrated Voltage Regulator)
 - 35–40 MB LLC. Server procesory až 20 jader, možnost rozdělit jádra do 2 uzelů NUMA (COD, cluster on die)
 - 4 verze integrované GPU (až 40 EU), TDP 35–140 W.
 - 14 nm Sandy Bridge = Broadwell
- Skylake 2015:
 - 14 nm, 4 typy Y, U, H a S (TDP 4–95W) integrovaná L4 eDRAM cache (64/128 MB), podpora DDR3/4,
 - Kabylake 14 nm, optimalizované Skylake, podpora kódování a dekódování 4K videa odlehčí CPU.

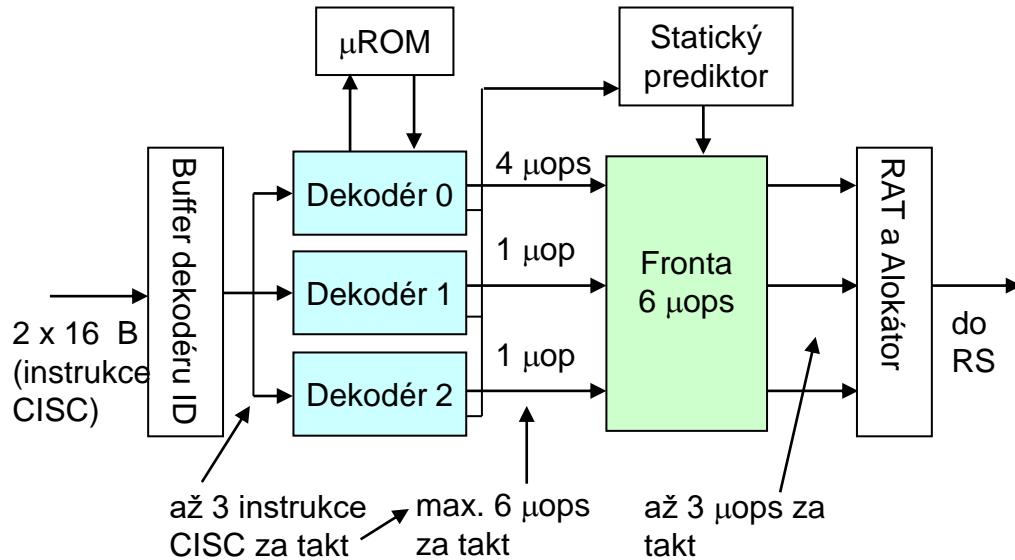
| | | |
|-----------------------------------|---------------------------|-------|
| P5 (Pentium) | superskalární, „in-order“ | 5 |
| P6 (Pentium Pro) | | 14 |
| P6 (Pentium III) | | 10 |
| NetBurst Pentium 4 (180 a 130 nm) | | 20 |
| NetBurst Pentium 4 (90 a 45 nm) | | 31 |
| Core | superskalární | 14 |
| Nehalem | „out-of-order“ | 16 |
| Sandy Bridge | | 14–19 |
| Ivy Bridge | | 14–19 |
| Haswell | | 14–19 |
| Bonnell (Atom) | | 16 |
| Quark | skalární | 5 |

Intel Roadmap

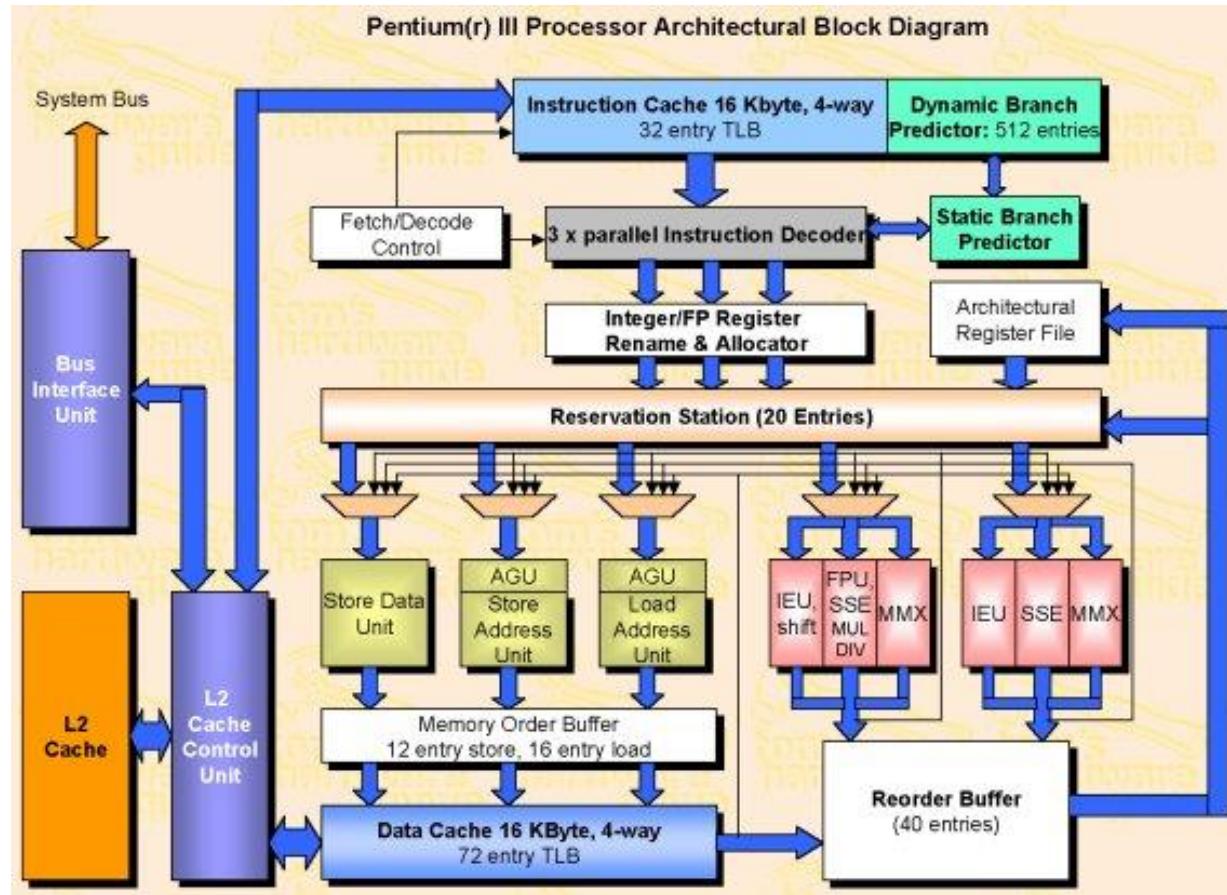
IT FIT



- Řetězené zpracování CISC-ových instrukcí x86 se řeší transformací (dekódováním) na RISC-ové mikrooperace délky 72 bitů.
- Délka instrukcí x86: 1–15 B, **dekodér délky instrukcí** posílá až 3 instrukce x86 na 3 dekodéry:
 - D0 zpracovává 1. instrukci, která generuje až 4 µop/takt.
 - D1 a D2 zpracovávají jednodušší 2. a 3. instrukci, které nejsou delší než 8 B a generují jen 1 µop.
 - 2. a 3. instrukce musí čekat na D0, pokud to nesplňují.
 - Pro dekódování instrukcí, které generují více než 4 µop je použita paměť mikrokódu a generování trvá 2 nebo více taktů.



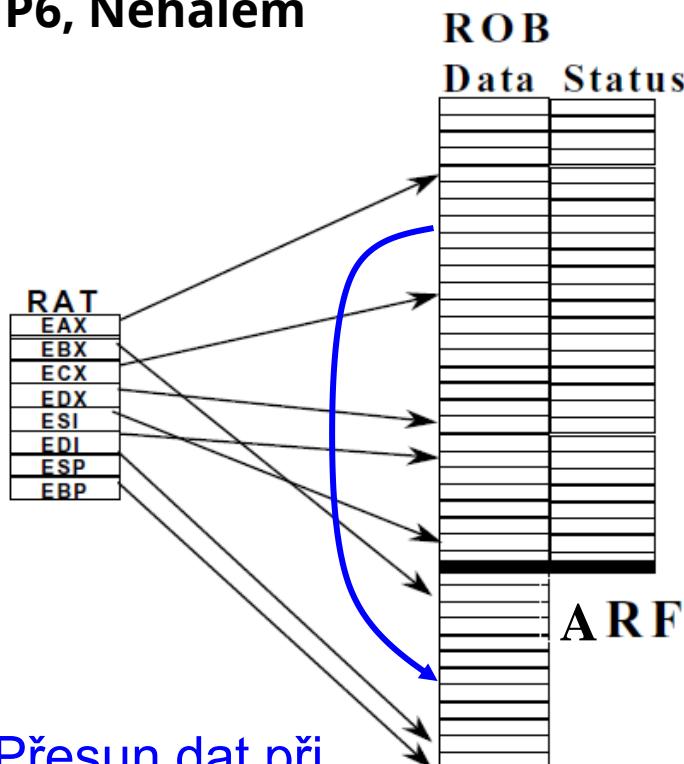
- Prediktor podmínky je 2 úrovňový adaptivní (autoři Yeh a Patt) s $k = 4$ -bitovým lokálním BHSR
 - perfektně predikuje libovolné periodické sekvence až $k + 1 = 5$ bitů
 - na 1 skok je třeba 36 bitů (= 16 dvoubitových prediktorů + 4 bity BHSR).
- BTB je organizován jako skupinově asociativní cache (128 skupin, 4 cesty, tj. 512 položek)
- Položka obsahuje adresu skokové instrukce b , cílovou adresu skoku t a 4 bitový lokální BHSR. Jeden index do PHT je část adresy skoku b , jako druhý index se použije obsah BHSR.
- **Pokuta za špatnou predikci je 10–20 taktů.**
- Není-li skok v BTB, použije se statická predikce (skok v kódu dopředu -, skok dozadu +)



- r. 2000, IA-32 procesor (adresa 32 bitů, instrukce x86)
- SSE2 (Streaming SIMD Extension 2)
- Trace Cache (kapacita 12k μ ops, cca 64 bitů / μ op
 - TC může rozeslat do RS 3 μ ops/takt, vydat do FJ se může až 6 μ ops/takt a propustit opět 3 μ ops/takt.
- Přejmenování mapuje 8 standardních registrů x86 na 128 vnitřních fyzických registrů PRF, 2 tabulky RAT (front-end a propouštěcí) → není nutno kopírovat registry při propouštění.
- ROB: až 126 μ ops v pořadí bez hodnot dst operandů
- **Co bylo špatné:** chyběla L3 cache na čipu, malá L1 D-cache (8 KiB), výkonnost \approx Pentium III, velký příkon

Přejmenování registrů v P6 a NetBurst

P6, Nehalem



Přesun dat při
propuštění instrukce

Sandy Bridge

NetBurst

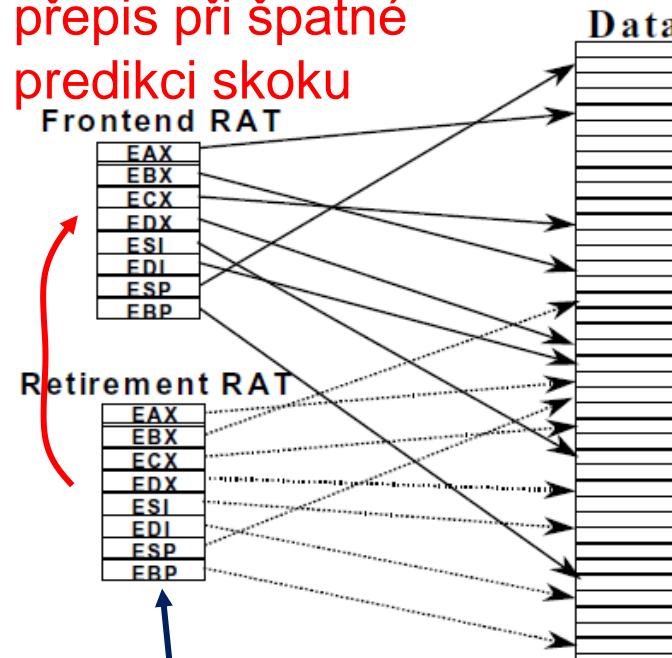
přepis při špatné
predikci skoku

Frontend RAT

| |
|-----|
| EAX |
| EBX |
| ECX |
| EDX |
| ESI |
| EDI |
| ESP |
| EBP |

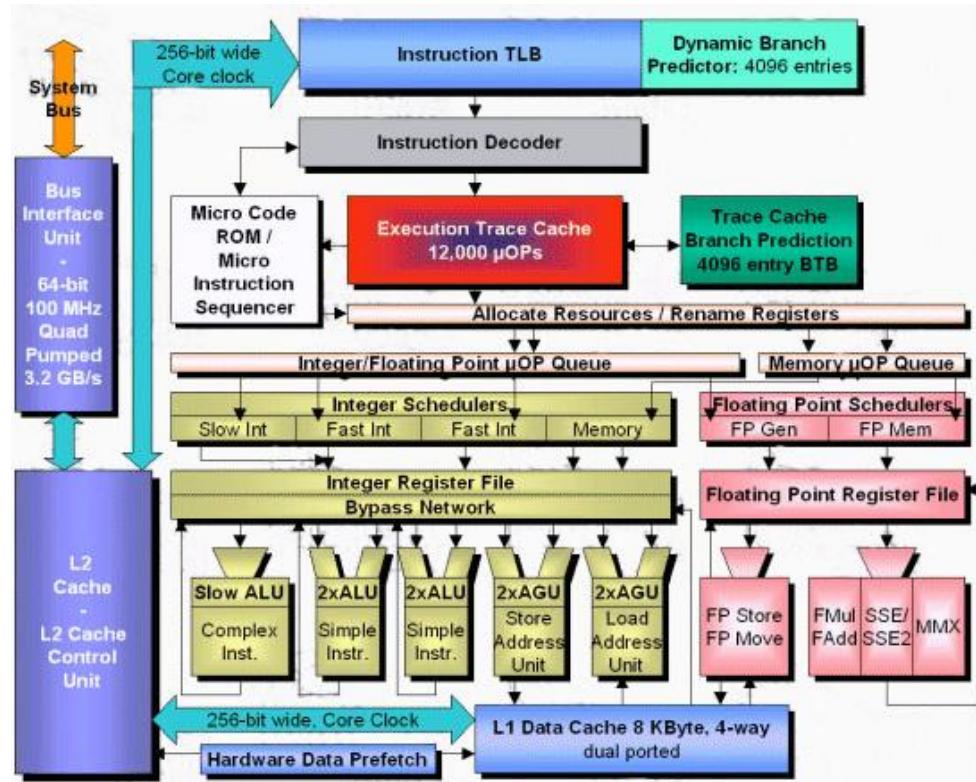
Retirement RAT

| |
|-----|
| EAX |
| EBX |
| ECX |
| EDX |
| ESI |
| EDI |
| ESP |
| EBP |



Při propuštění instrukce se sem
vloží mapování jejího dst registru

Pentium 4 – architektura NetBurst

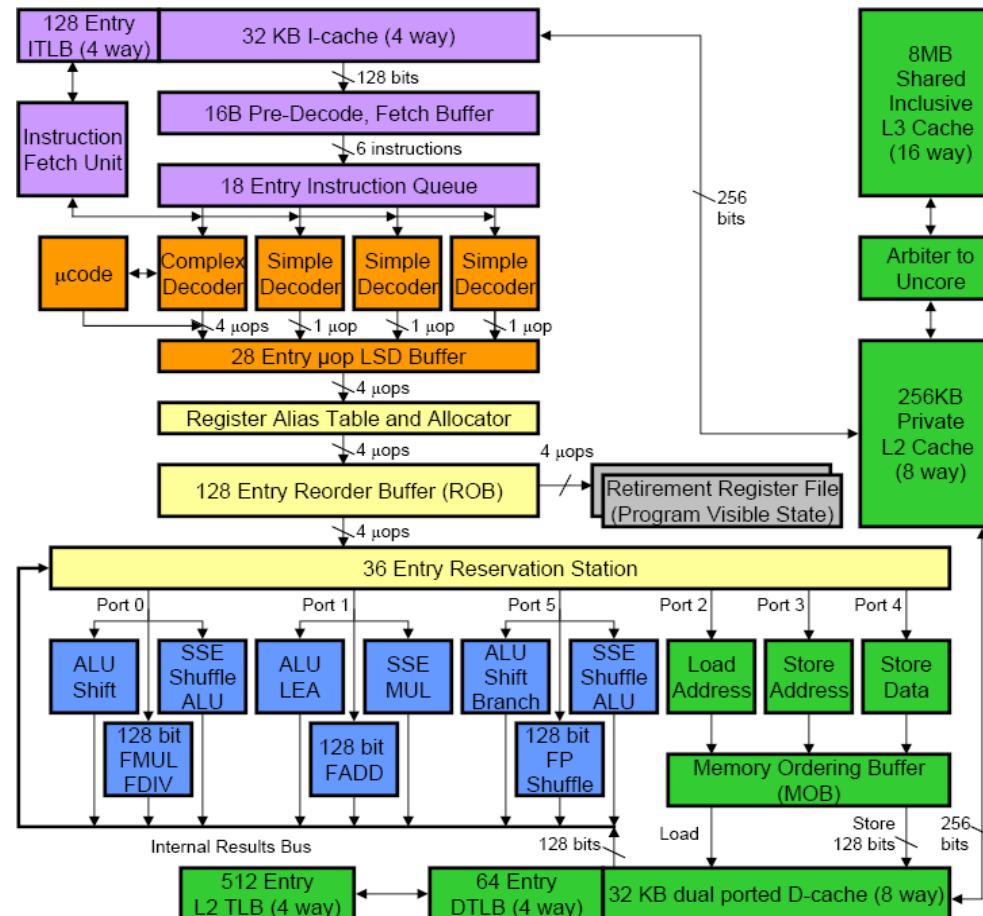


| | | | | | | | | | | | | | | | | | | | |
|-----------|----------|-------------|---|--------|-----|-----|-----|-----|------|------|----|----|----|------|-------|-------|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| TC Nxt IP | TC Fetch | Drive Alloc | | Rename | Que | Sch | Sch | Sch | Disp | Disp | RF | RF | Ex | Flgs | Br Ck | Drive | | | |

- Použita u vícejádrových procesorů se sdílenou pamětí cache L2. Snížení příkonu a zvýšení výkonnosti činí kolem 40 %. **Vychází z P6.**
- **Široká instrukční linka.**
 - Dekóduje a propouští až **4 instrukce za takt**, rozesílat a provádět může až **5 µops**.
 - Umí sdružovat instrukce x86 (*macrofusion*) a také sdružovat µops vzniklé z jedné x86 (*microfusion*), čímž lze dosáhnout až 6 µops za takt.
 - Ukazovatel zásobníku je modifikován speciálním HW. To dovoluje **načítání dat ze zásobníku již na začátku linky** (25 % všech načítání je ze zásobníku).
 - Tyto inovace zachovány i v novějších mikroarchitekturách
- **Pokročilá práce s multimédii.** Instrukce **MMX, SSE, SSE2, SSE3** se **128 bity** provedené **za 1 takt** znamenají výkonnost až 24 GFLOP/s (1 jádro na 3 GHz, SP).
- **Inteligentní napájení.** Dynamické odpojování subsystémů dle potřeb nebo přepojování do úsporného režimu neovlivňuje responzivitu.
- **Pokročilá chytrá cache.** Sdílená sjednocená cache úrovně 2 může být celá k dispozici jen 1 jádru, když druhé není aktivní. Špičková přenosová rychlosť je 96 GB/sec @ 3 GHz.
- **Chytrý přístup do paměti.** Je zavedena podpora pro **RPW** i pro případ dosud neznámé adresy zápisu (dynamické rozlišování adres, *memory disambiguation*)
- **Přednačítání dat** do L1/L2 D-cache pomocí tabulky historie čtení

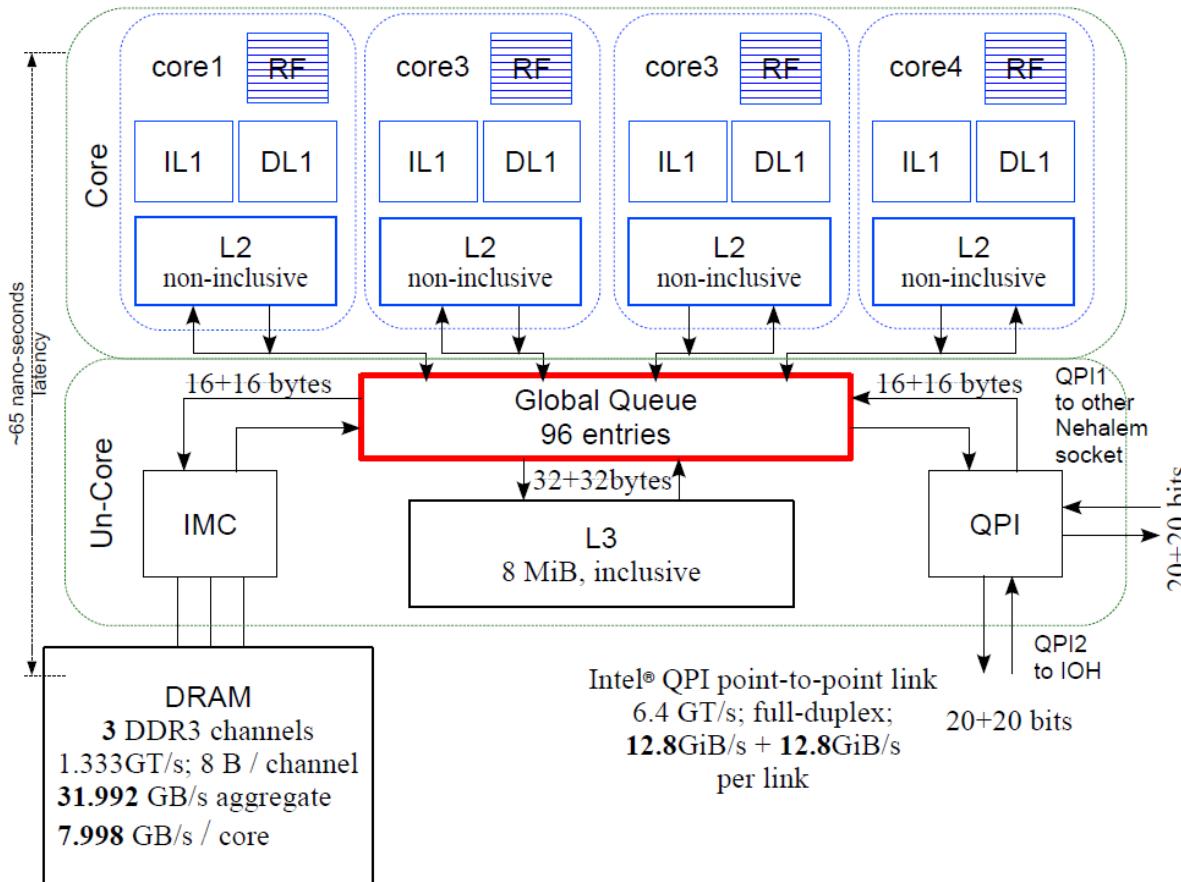
- Druhá generace architektury Core, orientace na výkonnost: 2, 4, 6 nebo 8 jader (45 nm, 4 jádra: 731 M tranzistorů)
- 16 stupňová linka, 6 FJ (3 paměťové, 3 výpočetní) a **HyperThreading (HT)**
- Větší cache a vyšší propustnost pamětí 32 KiB L1 I-cache, 32 KiB L1 D-cache, L2C: 256 KB.
- **Register Alias Table (RAT)** může přejmenovat až 4 μop za takt a každé přidělit dst. registr v ROB více rozpracovaných mikrooperací.
- **Dvouúrovňový prediktor skoků i dvouúrovňový TLB**.
- **Loop Stream Detector LSD**: ve frontě 18 před-dekódovaných instrukcí detekuje každé tělo smyčky, uloží je dekódované do LSD buferu (až 28 μop) a pak opakovaně používá (bez načítání a dekódování) až do špatné předpovědi skoku (malá náhrada Trace cache).
- **Turbo režim**: kmitočet hodin se zvyšuje, pokud není překročena teplotní mez.

Mikroarchitektura Nehalem

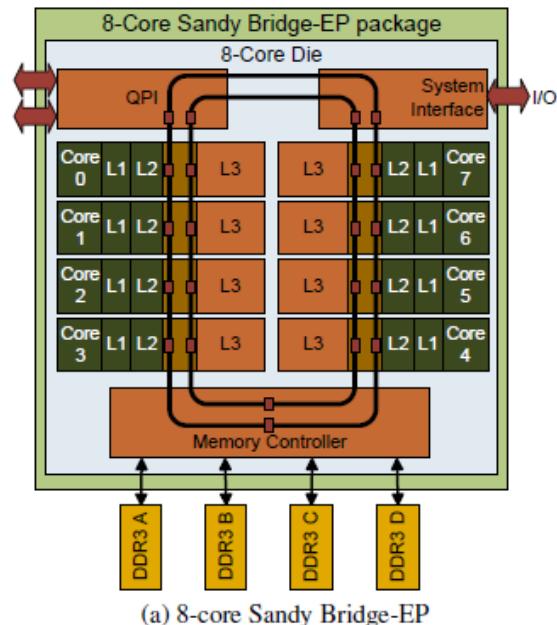


- **Inovace:** nové propojení soketů: front-side bus FSB nahrazen linkami (QPI, Quick Path Interconnect).
- **Inovace:** integrovaný řadič paměti, podporuje 3 paměťové kanály DDR3 SDRAM nebo 4 FB-DIMM, severní most eliminován.
 - Firma AMD zavedla linky HyperTransport (HT) a integrované řadiče paměti již v roce 2003, o 5 let dříve.
- **Sdílená L3C:** 4–8 MB je inkluzivní, obsahuje data z L1 i L2 a info, kde jsou lokálně (menší komunikace).
- **Čip procesoru grafiky** v témže pouzdro jako CPU.
- **Řízení příkonu:** vestavěný mikrořadič a senzory teploty, proudu a příkonu, odepínání jader, možnost redukce příkonu pamětí a QPI.
- Global Queue (GQ) uchovává, spravuje a plánuje tok dat v „uncore“. Má 3 fronty požadavků:
 - WQ, žádosti zápisu z lokálních jader, 16 položek
 - LQ, žádosti čtení z lokálních jader, 32 položek
 - QQ, fronta QPI, žádosti jdoucí mimo čip, 12 položek
- Obsahuje **křížový přepínač** pro výměnu dat mezi propojenými částmi (L2, L3, IMC, QPI).
- **Funkce:**
 - lokální žádost jádra o čtení: GQ sonduje další jádra. Z více vlastníků kopí jedno jádro dodá data.
 - Když nikdo nemá kopii a L3 ano, dodá data inkluzivní L3
 - Výpadek v L3: data dodá lokální IMC (Integrated Memory Controller) za 65 ns, popřípadě vzdálený IMC přes QPI za 105 ns

Celkový diagram procesoru Intel Nehalem



- 4, 6 a 8 jader na 3,0–3,8 GHz s podporou HyperThreadingu (HT) a s technologií Turbo Boost
- Jádra, grafika, L3 cache a systémový agent jsou propojeny **kružnicovým propojením** s propustností 256 bitů/takt.
- Podpora **Advanced Vector Extension (AVX): 256 bitů, 32 GFLOPS/jádro (8 FP/takt),**
- Každé jádro: 32 KiB L1 D-cache + 32 KiB L1 I-cache (3 taktů), 256 KiB L2 cache (8 taktů).
- 8 MiB **sdílená L3 cache** (25 taktů). Je též sdílena s integrovaným grafickým jádrem. Blok cache 64 byte.
- **Integrované jádro grafiky na 1–1,4 GHz, 16 ex. jednotek.**
- **Integrovaný řadič paměti** s max. propustností 25,6 GB/s, podporuje DDR3-1600 dual channel RAM.
- CPU ↔ L3 cache: průměrně jen 1,5 skoků (do lokálního bloku cache není třeba jít po kružnici). Latence sdílené L3 cache je 26 až 31 taktů.

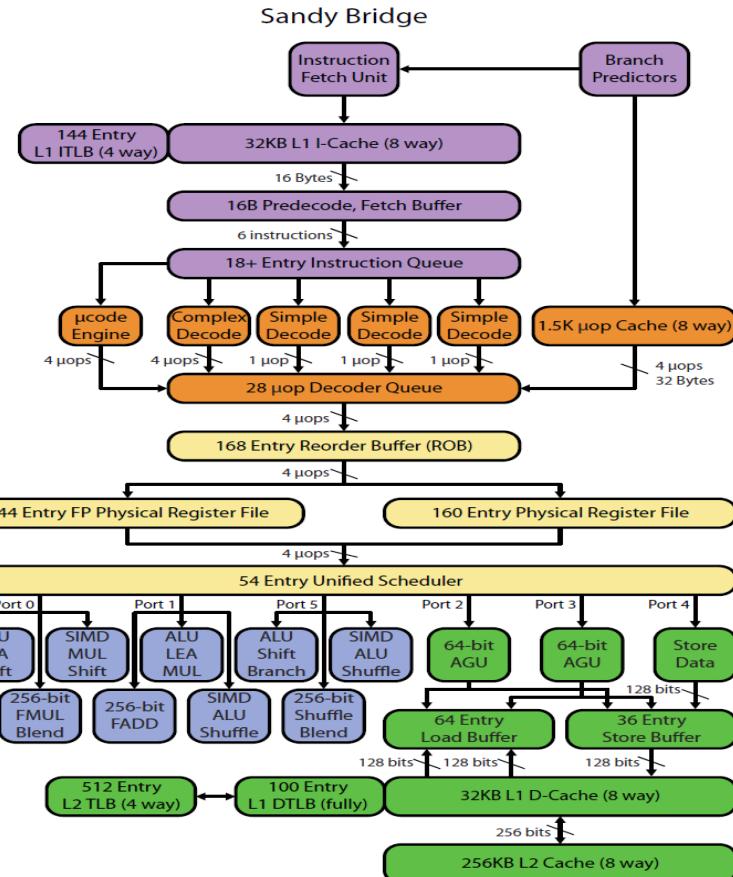


Dvě propojovací kružnice, jedna pro směr nahoru, druhá dolů. Pro každý přenos je vybrán směr kratší cesty do cíle, nejvíce 4 skoky.

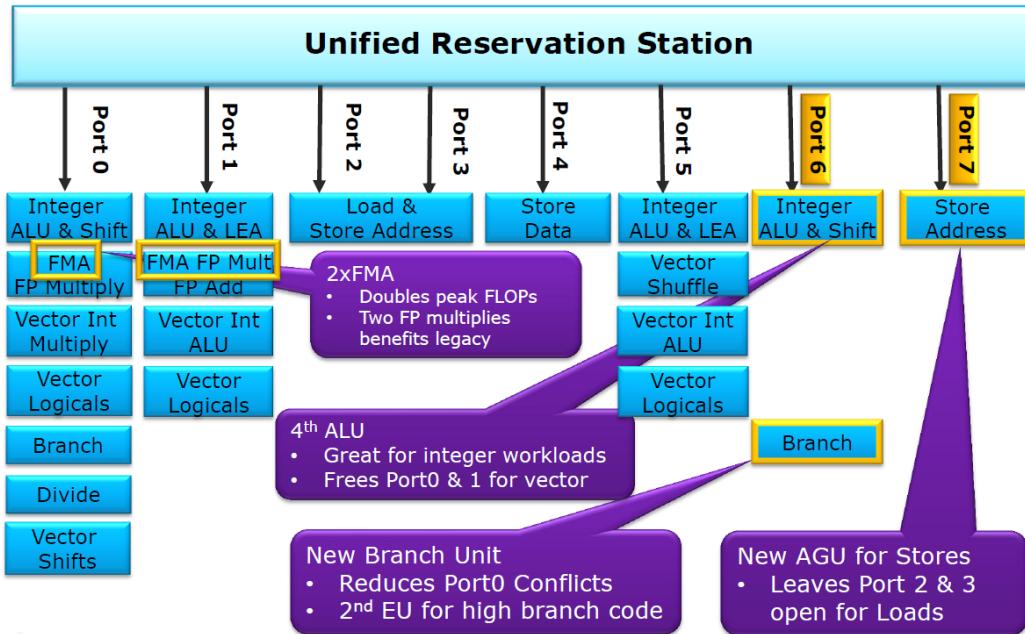
μ op-cache (dekódovaná I-cache) v Sandy Bridge

IT FIT

- Je částí L1 I-cache, zachovává výhody Trace cache, eliminuje složité dekódování při mnohem nižším příkonu.
- μ op-cache má kapacitu 1536 μ ops, 10 % velikosti Trace cache Pentia 4.
- Mapování instrukcí do μ op-cache probíhá po blocích 32 B instrukcí, 1 blok může zabrat až 18 μ ops. Každý blok μ op-cache uchovává „metadata“ včetně počtu platných μ ops v bloku a délku odpovídajících x86 instrukcí.
- Jestliže okénko 32 B instrukcí má více než 18 μ ops, musí jít přes tradiční front-end.
- **Mikrokódované** instrukce nejsou v μ op-cache – jsou reprezentovány ptr do ROM mikrokódu a případně několika prvními μ ops.

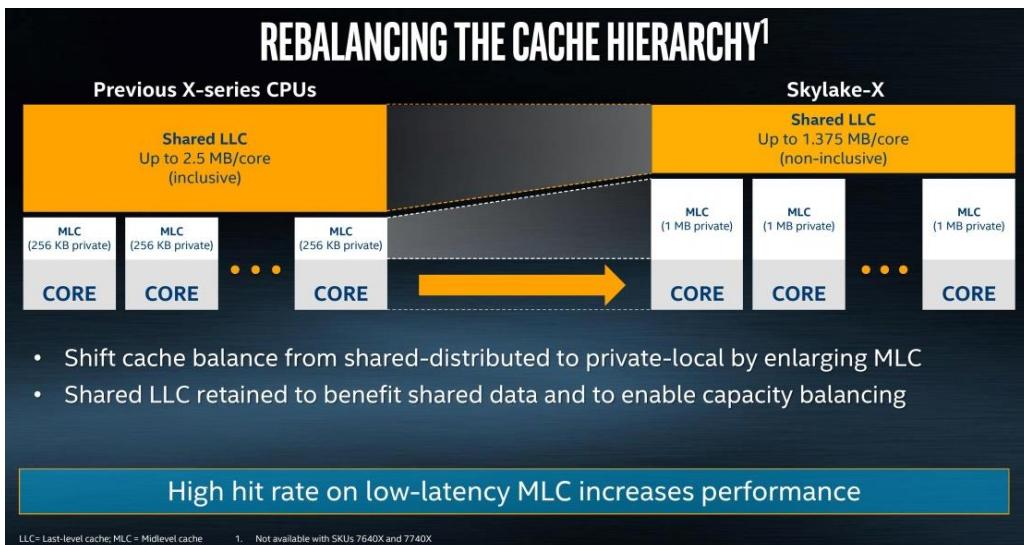


- Haswell je zaměřen na **nižší příkon pro mobilní zařízení** (hybridní laptop-tablety) ale i pro superpočítáče. Dřívejší TDP (Thermal Design Power) 35 až 45 W pro mobilní procesory je redukován na ULT: 13,5 W a 15 W TDP, ULTX: 10 W TDP.
- Superpočítáč v Ostravě obsahuje 24 192 jader Haswell-EP!
- Podpora pro AVX2 a MAD operace.**
- Haswell má výkonnější grafiku GT3e, ze 16 jednotek na 1150 MHz (GT1 u Sandy Bridge) narostla na 40 ex. jednotek a 1300 MHz.
- eDRAM (embedded DRAM) 128 MiB je na vlastním čipu, ale ve stejném pouzdru jako procesor. Pracuje jako sdílená L4 cache jak pro grafiku, tak pro jádra procesoru. Vylepšuje paměťovou propustnost.



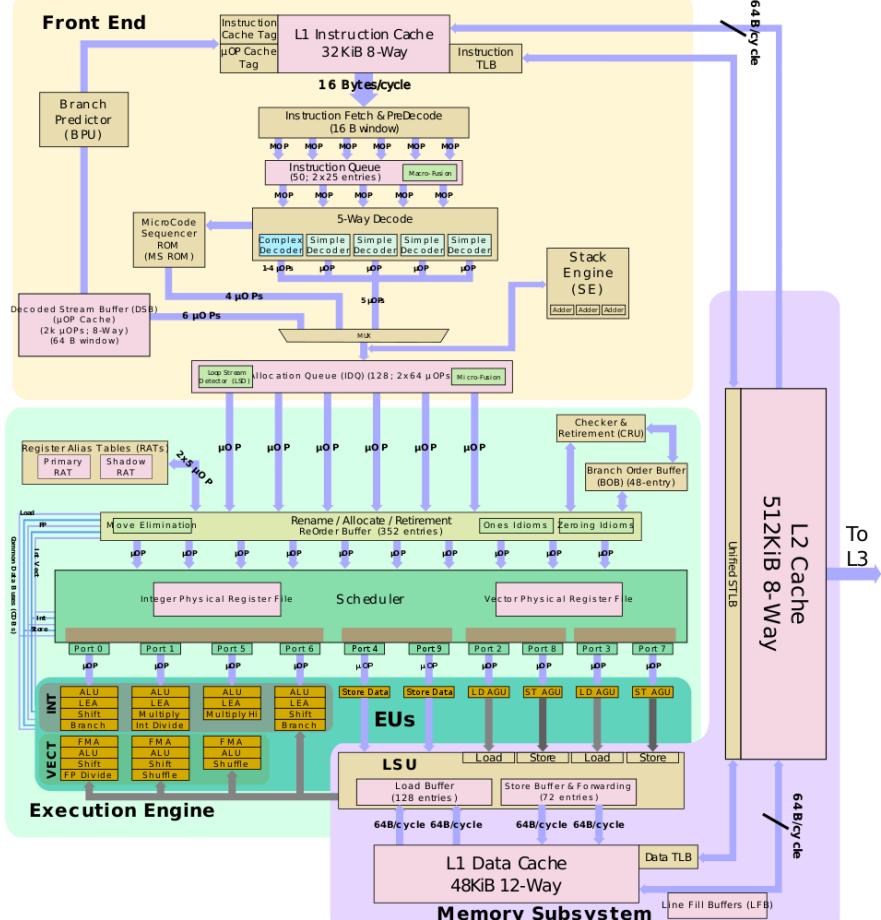
- Broadwell = Haswell předělaný na 14 nm (2014).

- Podpora pro AVX-512
- Nová organizace cache
 - Zvětšení L2 cache na úkor L3 cache
- Změna propojovací sítě
 - Z hierarchických kruhů na 2D mřížku



| Comparison: Skylake-S and Skylake-SP Caches | | |
|--|----------|--|
| Skylake-S | Features | Skylake-SP |
| 32 KB
8-way
4-cycle
4KB 64-entry
4-way TLB | L1-D | 32 KB
8-way
4-cycle
4KB 64-entry
4-way TLB |
| 32 KB
8-way
4KB 128-entry
8-way TLB | L1-I | 32 KB
8-way
4KB 128-entry
8-way TLB |
| 256 KB
4-way
11-cycle
4KB 1536-entry
12-way TLB
Inclusive | L2 | 1 MB
16-way
11-13 cycle
4KB 1536-entry
12-way TLB
Inclusive |
| < 2 MB/core
Up to 16-way
44-cycle
Inclusive | L3 | 1.375 MB/core
11-way
77-cycle
Non-inclusive |

Architektura SunnyCove (2019)



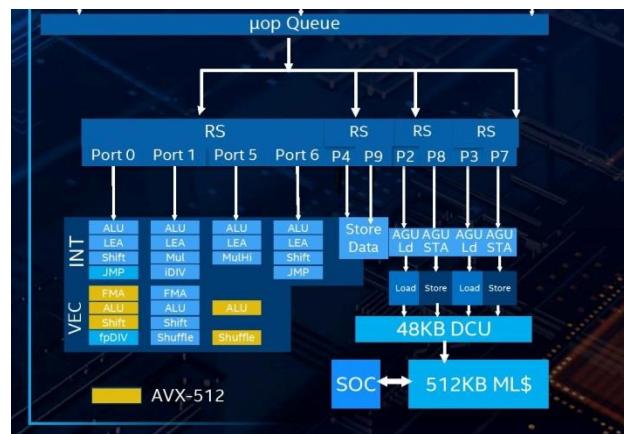
Změna velikostí cache

- L1 z 32 KB → 48 KB
- L2 z 256 KB → 512 KB
- Zvětšena TraceCache (2,25k)

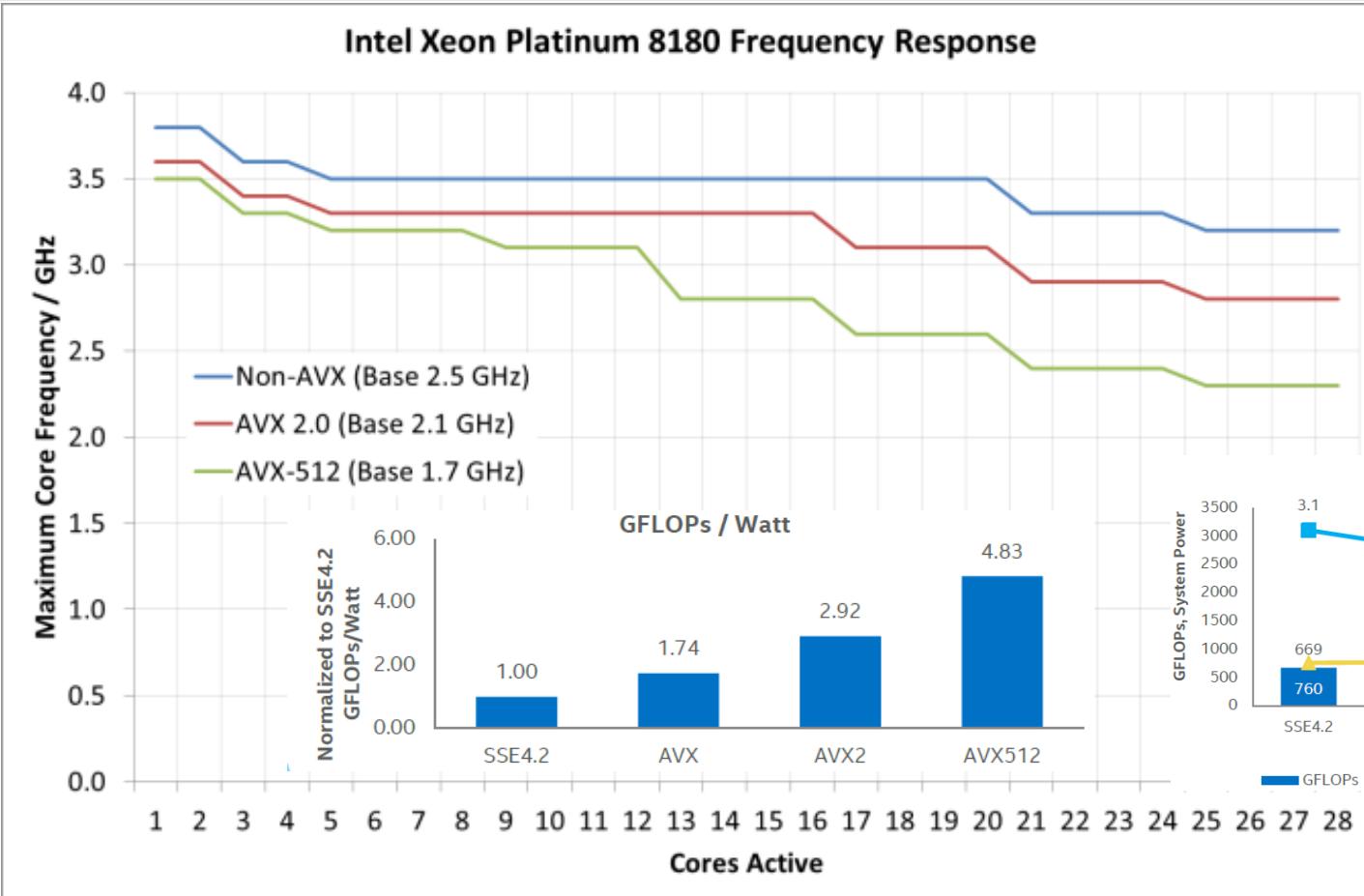


Back-end

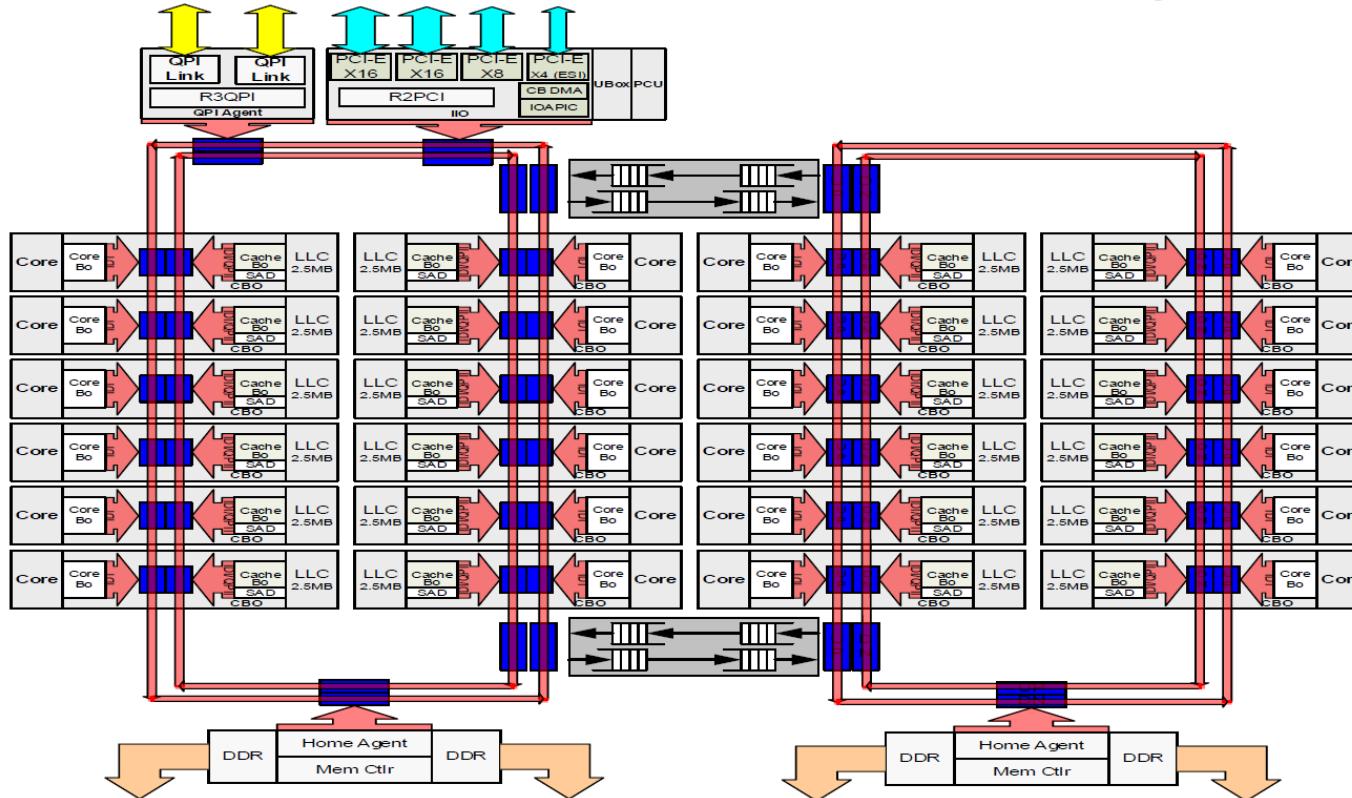
- Zvýšeno IPC o 18 %
- 8 → 10 výpočetních linek
- ROB zvětšen z 224 na 352 záznamů
- Nová AGU jednotka (4)
- Výrazně zvětšeny load/store buffery



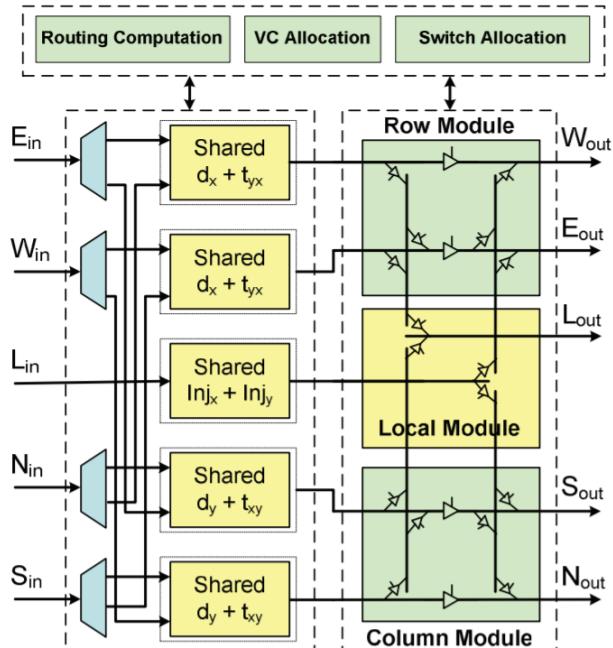
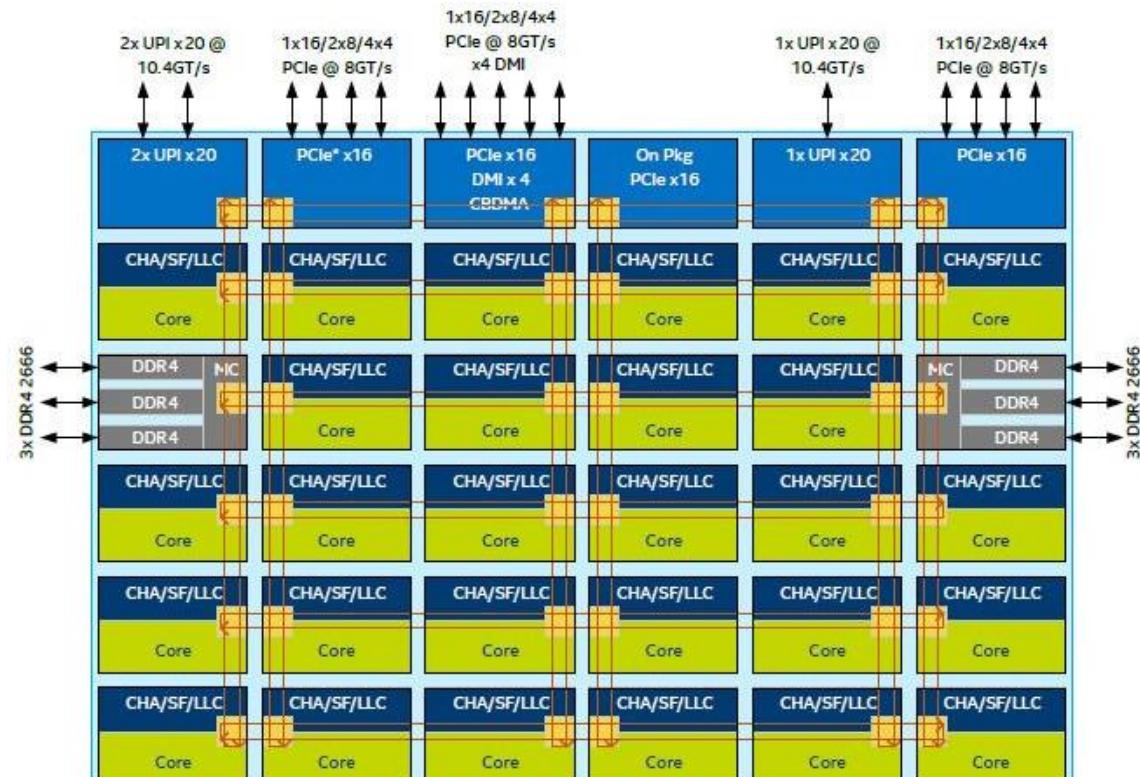
Frequency Scaling with Instruction Types



Intel® Xeon® Processor E5 v4 Product Family HCC



Zapojení do mřížky 6x6 (28 jader)

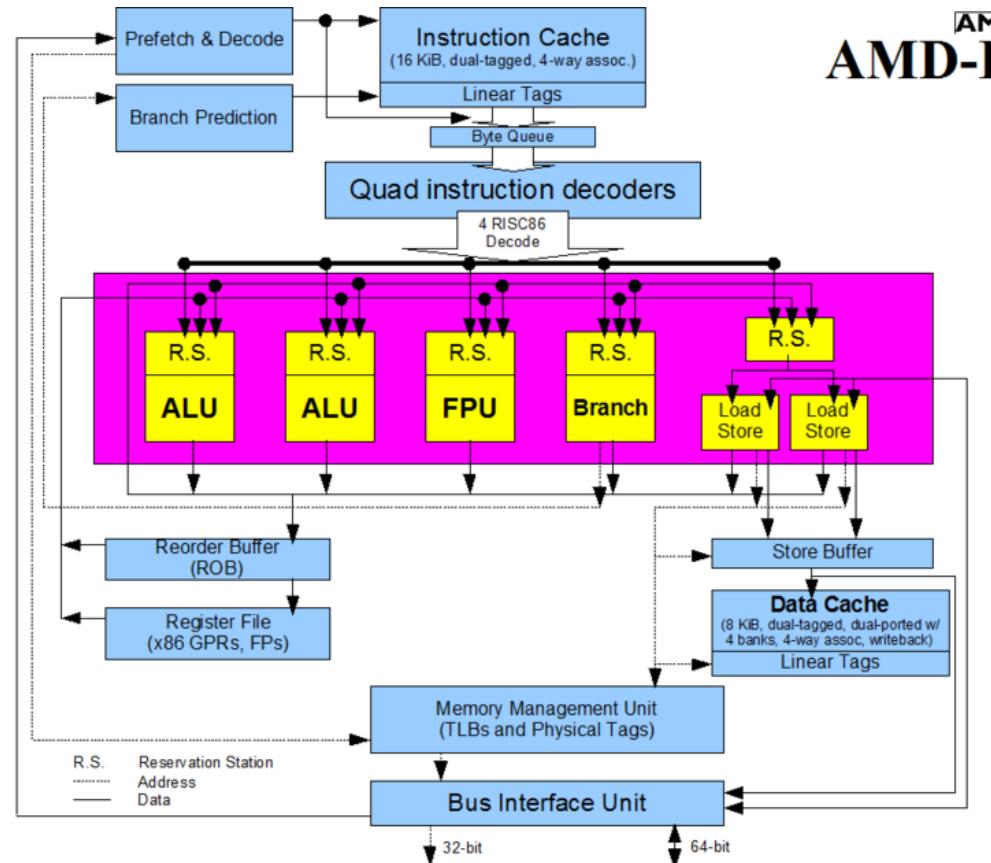


(c) 5x5 MoDe-X-Single Router Design (Single Injection)

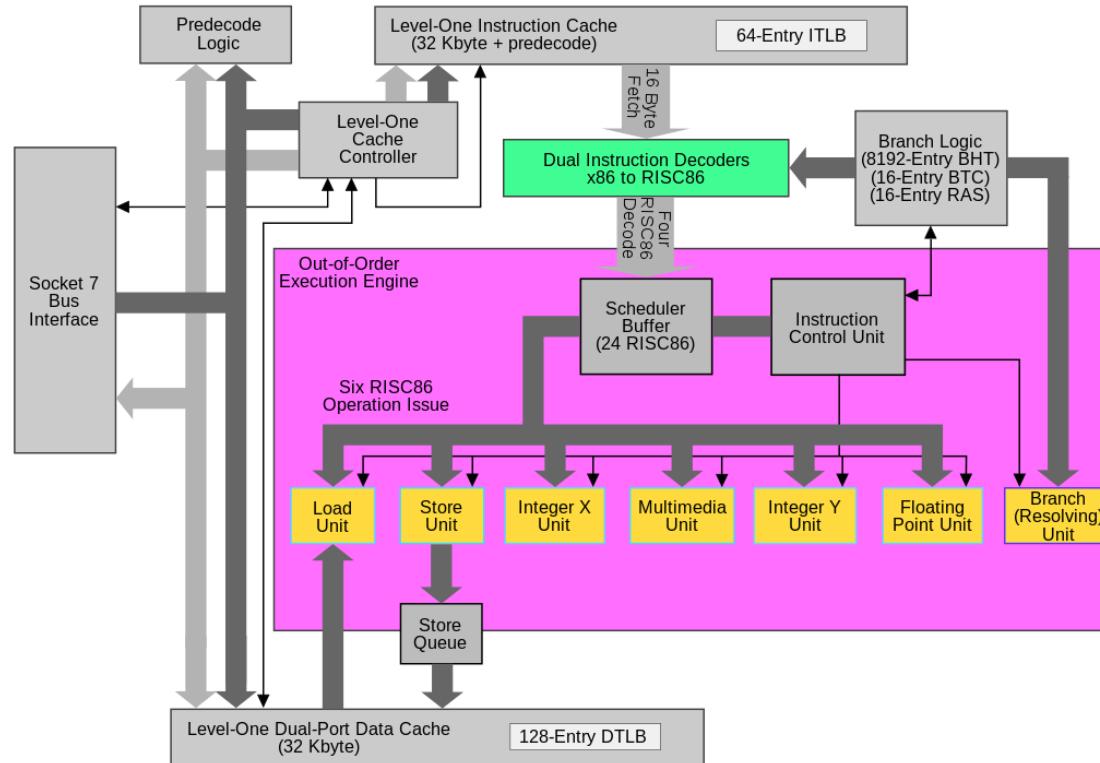
PROCESORY AMD

- **AMD K5 (1996)**

- První vlastní OOO procesor AMD
- 6 výpočetních jednotek, 4 vydání instrukce za takt, 5 stupňů linky.
- Spekulativní provádění podél 3 predikovaných větví
- Penalta 3 takty při špatné predikci
- Přejmenování registrů
- 16 KB L1, přístup do L1 v 1 taktu!
- Podpora MESI cache coherent protokolu

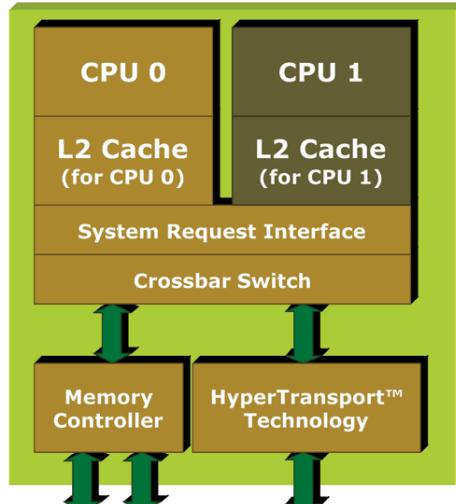
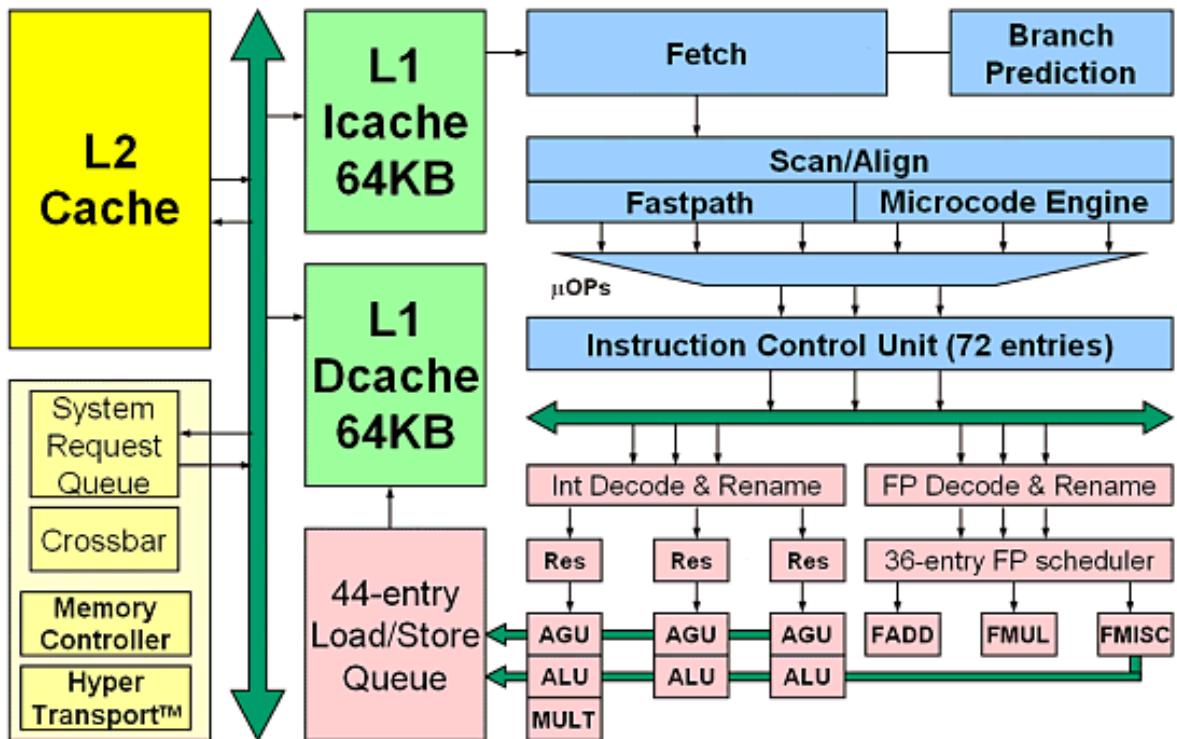


- Uvedena na trh v roce 1997
- Přináší instrukce MMX, později 3DNow

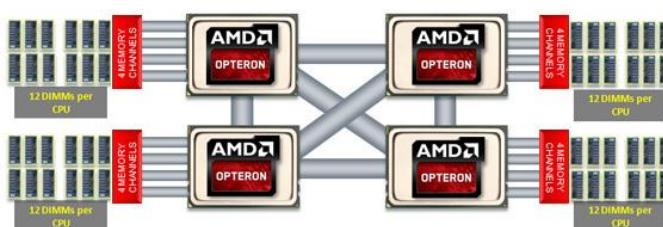


První CPU s instrukcemi x86 na 64 bitech, kompatibilní s Windows (2003), 32 bitové i 64 bitové aplikace, SW investice neznehodnoceny. Reakce Intelu: Extended Memory 64-bit Technology) EM64T a pak Intel® 64.

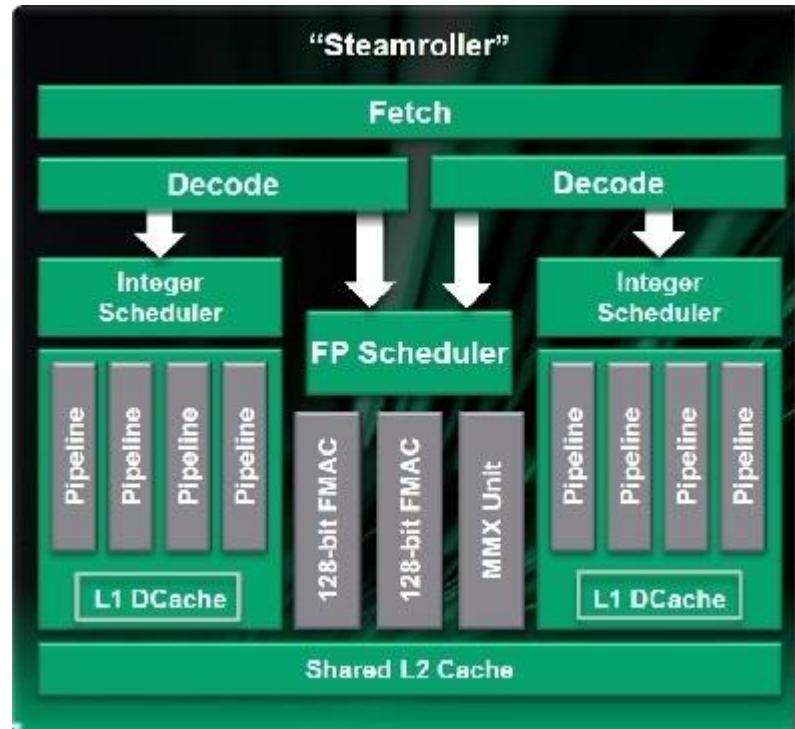
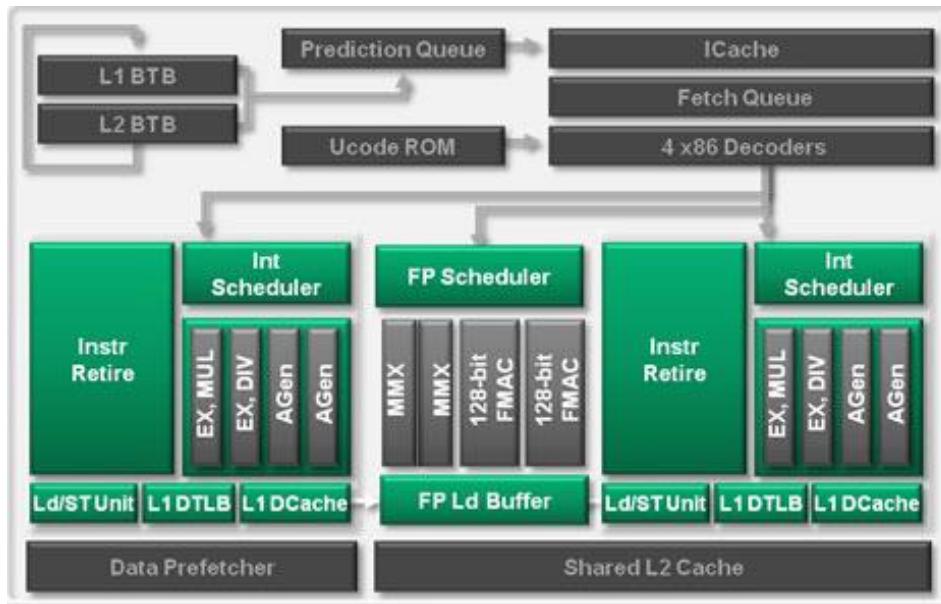
- **2, 4, 6, 8, 12, nebo 16 jader**, linka 12 stupňů, technologie SOI (Silicon on Insulator)
- **Linky Hyper Transport – 2003** (point-to-point) pro propojení s dalšími CPU (nahradily sběrnici) nebo I/O. Šířka 16 bitů, při 800 MHz to znamená 3,2 GB/s. Umožněna stavba multiprocesorů bez dalších součástek.
- **Řadič paměti DDR na čipu – 2003**, 128 bitové rozhraní na 333 MHz paměť.
- **Mikroarchitektura K10: Phenom II, 2,3,4 nebo 6 jader, 2008–12.**

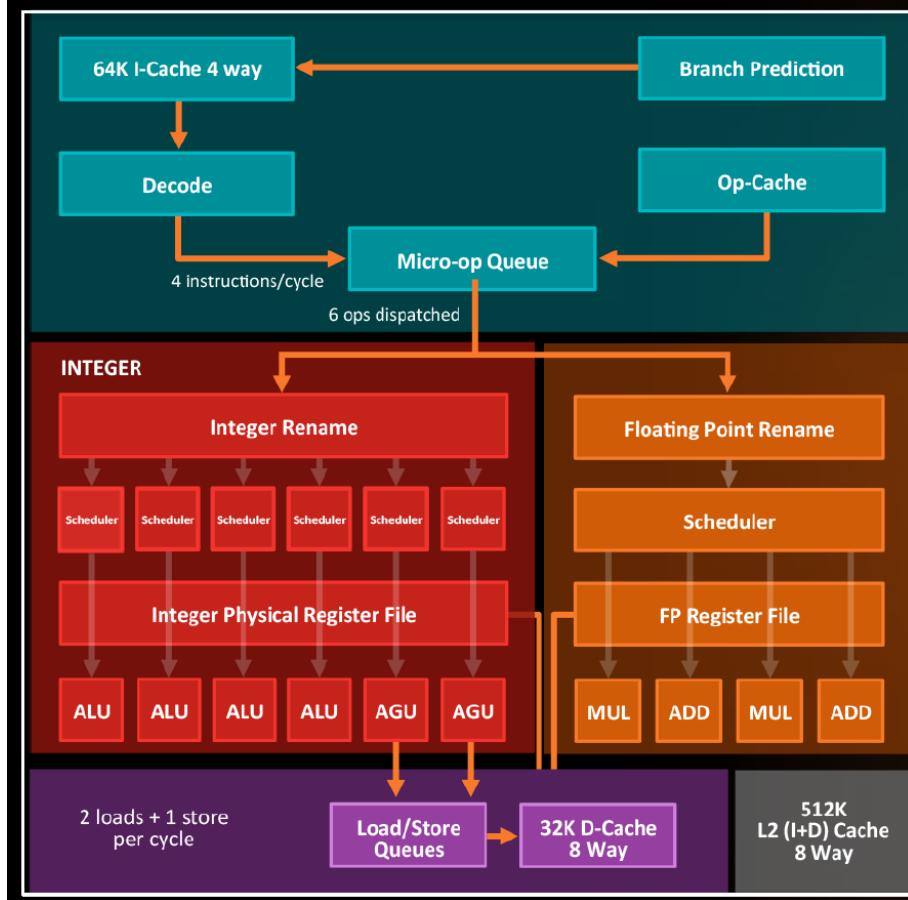


AMD Athlon™ 64 X2
Dual-Core Processor Design



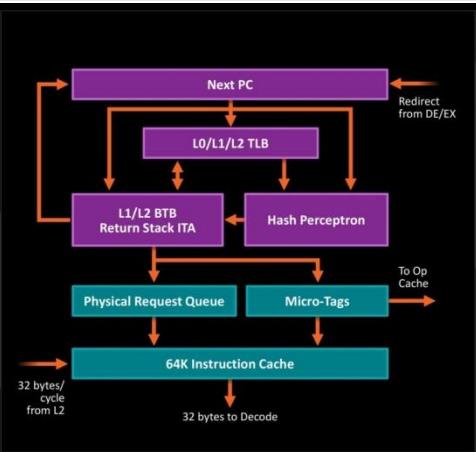
- Mikroarchitektura Bulldozer: 1 až 4 moduly se 2 jádry (tzv. CMT, Clustered MultiThreading, 1 až 2 vlákna/modul).
 - 10 až 100 W, 3,6–4 GHz, 2012.
 - Každý 2-jádrový modul sdílí L1-I cache, stupně načítání a dekódování, L2 cache, FPU.
 - Později přidány dedikované dekodéry (Steamroller)



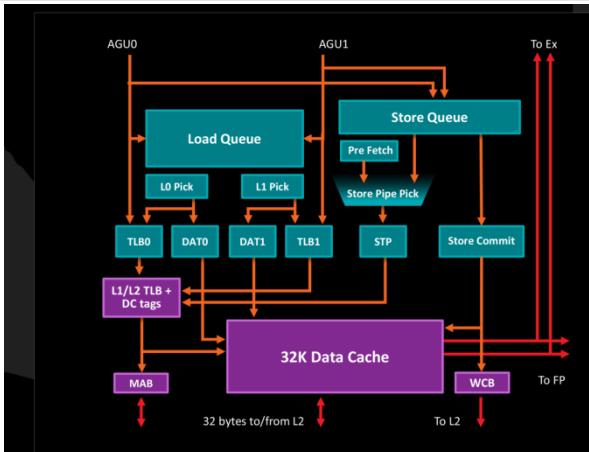


ZEN MICROARCHITECTURE

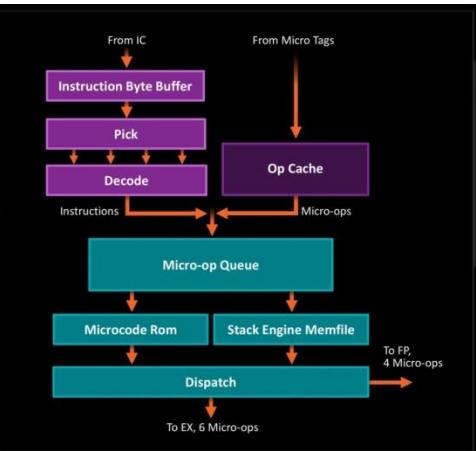
- ▲ Fetch Four x86 instructions
- ▲ Op Cache instructions
- ▲ 4 Integer units
 - Large rename space – 168 Registers
 - 192 instructions in flight/8 wide retire
- ▲ 2 Load/Store units
 - 72 Out-of-Order Loads supported
- ▲ 2 Floating Point units x 128 FMACs
 - built as 4 pipes, 2 Fadd, 2 Fmul
- ▲ I-Cache 64K, 4-way
- ▲ D-Cache 32K, 8-way
- ▲ L2 Cache 512K, 8-way
- ▲ Large shared L3 cache
- ▲ 2 threads per core

**FETCH**

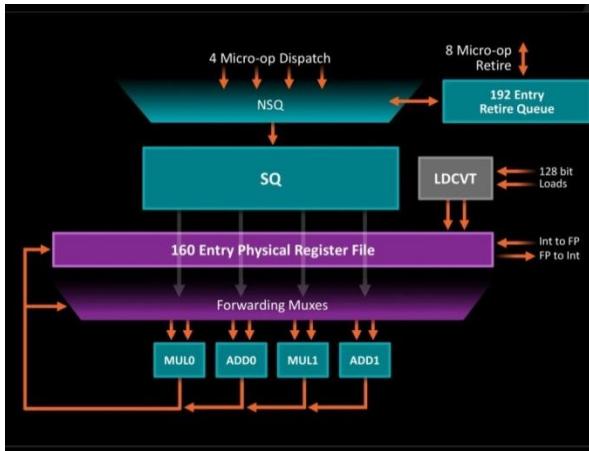
- ▲ Decoupled Branch Prediction
- ▲ TLB in the BP pipe
 - 8 entry L0 TLB, all page sizes
 - 64 entry L1 TLB, all page sizes
 - 512 entry L2 TLB, no 1G pages
- ▲ 2 branches per BTB entry
- ▲ Large L1 / L2 BTB
- ▲ 32 entry return stack
- ▲ Indirect Target Array (ITA)
- ▲ 64K, 4-way Instruction cache
- ▲ Micro-tags for IC & Op cache
- ▲ 32 byte fetch

**LOAD/STORE AND L2**

- ▲ 72 Out of Order Loads
- ▲ 44 entry Store Queue
- ▲ Split TLB/Data Pipe, store pipe
- ▲ 64 entry L1 TLB, all page sizes
- ▲ 1.5K entry L2 TLB, no 1G pages
- ▲ 32K, 8 way Data Cache
 - Supports two 128-bit accesses
- ▲ Optimized L1 and L2 Prefetchers
- ▲ 512K, private (2 threads), inclusive L2

**DECODE**

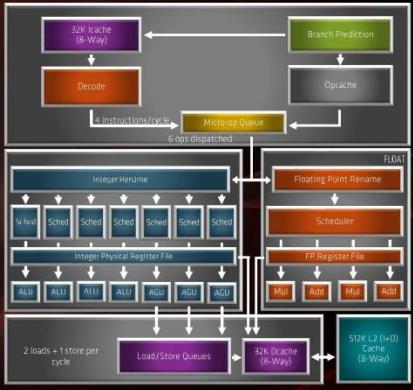
- ▲ Inline Instruction-length Decoder
- ▲ Decode 4 x86 instructions
- ▲ Op cache
- ▲ Micro-op Queue
- ▲ Stack Engine
- ▲ Branch Fusion
- ▲ Memory File for Store to Load Forwarding

**FLOATING POINT**

- ▲ 2 Level Scheduling Queue
- ▲ 160 entry Physical Register File
- ▲ 8 Wide Retire
- ▲ 1 pipe for 1x128b store
- ▲ Accelerated Recovery on Flushes
- ▲ SSE, AVX1, AVX2, AES, SHA, and legacy mmx/x87 compliant
- ▲ 2 AES units

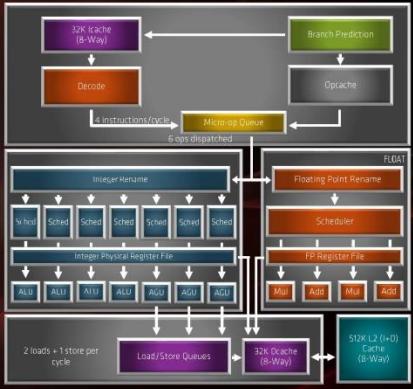
"ZEN 2" MICROARCHITECTURE OVERVIEW

- 2 threads per core (SMT) carried forward
- New TAGE branch predictor
- Larger Micro-Op Cache, now 4K instructions
- Larger L3 cache, now 2X "Zen" and "Zen+"
- 4 integer units
 - Large rename space ~ 180 registers
 - Increased AGUs from 2 to 3
- 3 AGENs per cycle
- 2 loads and 1 store per cycle
- 2 floating point units x 256 Fmacs
 - built as 4 pipes, 2 Fadd, 2 Fmul
 - Now supports single-op AVX256



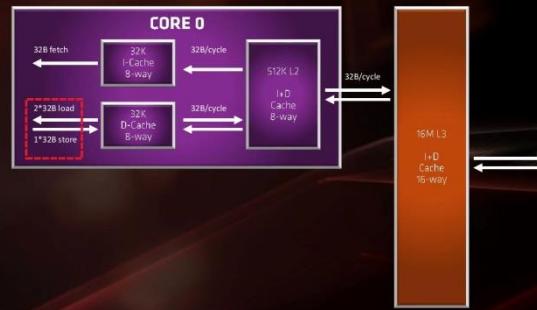
"ZEN 2" MICROARCHITECTURE OVERVIEW

- I-cache 32k, 8-way
- D-cache 32k, 8-way
- L2 cache 512k, 8-way
- TLBs
 - L1 64 entries I & D, all page sizes
 - L2 512 I, 2K D, everything but 1G
- Faster Virtualization Based Security
 - With Guest Mode Execute Trap
- Hardware-enhanced Security mitigations



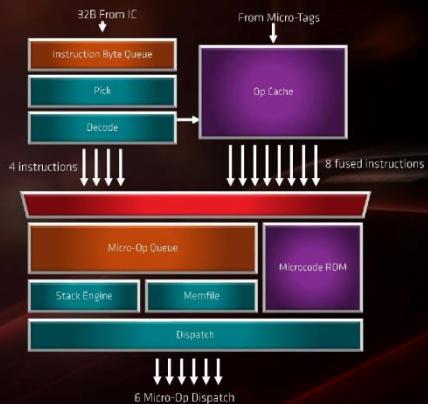
"ZEN 2" CACHE HIERARCHY

- Doubled L1 load/store bandwidth over Zen
- Improved L1 and L2 prefetch throttling
- Fast private 512k L2 cache
- Fast shared L3 cache
- High bandwidth enables prefetch improvements
- L3 is filled from L2 victims
- Fast cache-to-cache transfers
- Large Queues for Handling L1 and L2 misses



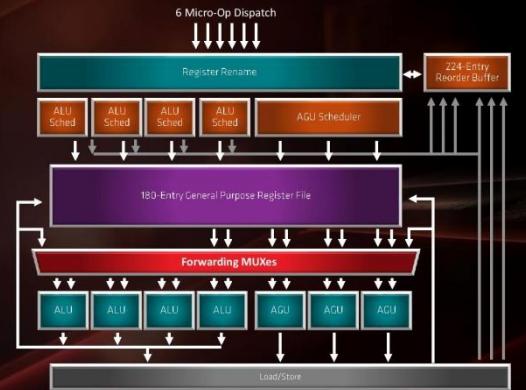
DECODE

- Op cache improvements
- Doubled capacity to 4K fused instructions
- Better instruction fusion
- Increased effective throughput



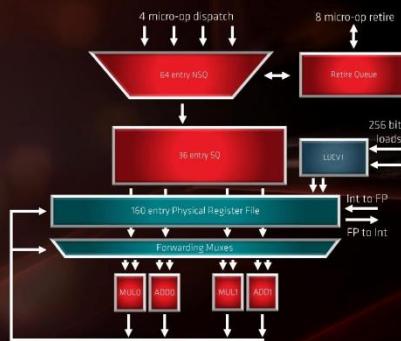
INTEGER EXECUTION

- 92 entry integer scheduler, up from 84
- 4, 16-entry ALU queues
- 1, 28-entry AGU queue
- 180 entry physical register file (up from 168)
- 7 issue per cycle, up from 6
- 4 ALUs, 3 AGUs
- 224 entry ROB, up from 192
- Improved SMT fairness for ALU and AGU schedulers
- Watermarked ALU tokens to manage spinlocks



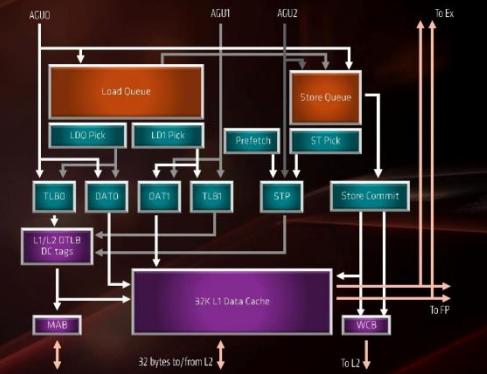
FLOATING POINT UNIT

- Doubled Floating Point & Load Store bandwidth from 128b to 256b
- Improved performance for instructions using 256b ymm registers which are generated by AVX intrinsics or /arch:[AVX|AVX2] compiler flags
 - Faster inline memcpy & memset
 - Faster physics simulation
 - Faster audio effects processing (Microsoft™ XAudio2_9)
- Improved mul latency from 4 to 3 cycles



LOAD/STORE

- 48 entry store queue, was 44
- 2K entry L2 DTLB, 1G as 2M, was 1.5K no 1G
- Improved L2 DTLB latency
- 32KB, 8-way L1 data cache
 - Two 256-bit reads
 - One 256-bit write
 - 64B load, 32B store alignment boundaries
- Increased Load/Store bandwidth to 32B/clk (up from 16B/clk)
- Faster string copy and float-point point performance
- Improved write-combining buffer performance
 - While using multiple streams, the hardware avoids closing buffers before they are completely full
- Improved prefetch throttling



Pokračování příště

Programování se sdílenou pamětí

Úvod do OpenMP a smyčky for

AVS – Architektury výpočetních systémů

Týden 7, 2024/2025

Jirka Jaroš

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



- Tři paralelní programovací modely
 - Abstrakce předkládané programátorovi
 - Ovlivňují jak programátoři přemýšlejí při psaní programů
- Tři architektury strojů
 - Abstrakce hardware pro software na nízké úrovni
 - Typicky odpovídají implementaci

1. Sdílený adresový prostor (MIMD, SPMD)

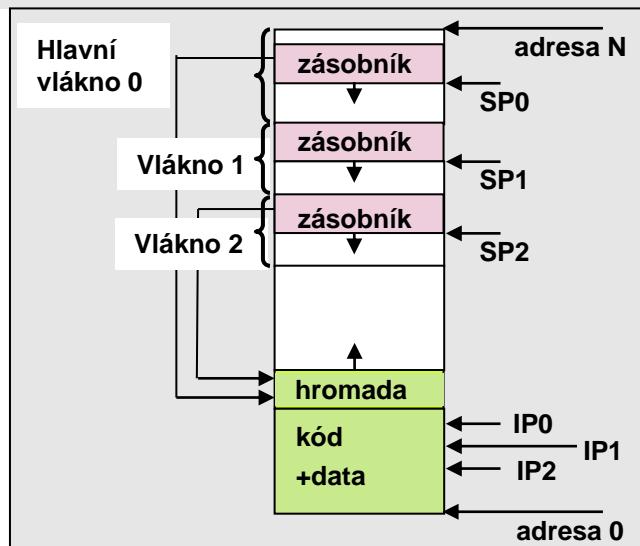
(Shared Address Space, SAS)

2. Zasílání zpráv (MIMD, SPMD)

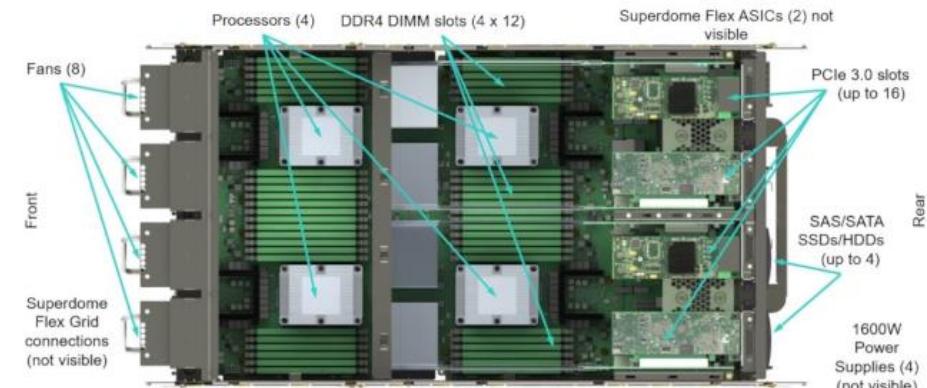
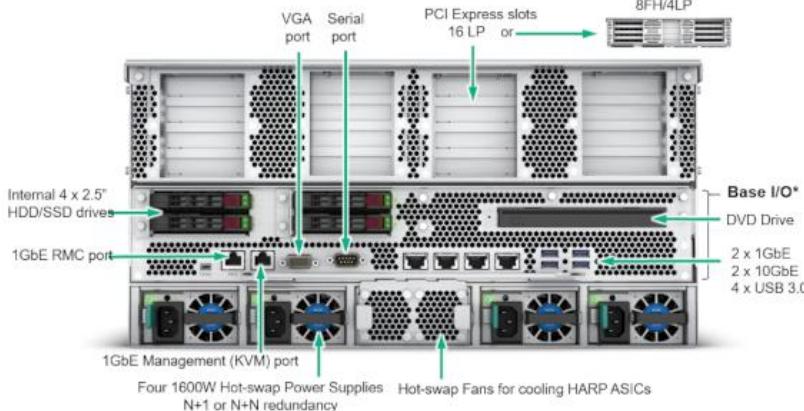
(Message Passing, MP)

3. Datově-paralelní model (SIMD, SIMT)

- Vlákna **komunikují**:
 - **čtením/zápisem do sdílených proměnných**
 - Komunikace je implicitní v paměťových operacích – snadnější programování.
 - Vlákno 1 zapíše do X, vlákno 2 čte X (vidí update).
 - To vyžaduje synchronizaci.
 - **manipulací synchronizačních primitiv**
 - např. vzájemné vyloučení použitím zámků
- Přirozené rozšíření sekvenčního programovacího modelu
- Sdílené proměnné jako velká nástěnka
 - každé vlákno může číst nebo zapisovat
 - problém se **současným přístupem R+W** ke sdíleným datům



- HPE Superdome Flex server
- 32x Intel® Xeon® Platinum, 24-core, 2.9 GHz, 205W
- 24 TB DDR4 2993MT/s of physical memory per node
- 2x 200 Gb/s IB port
- 71.2704 TFLOP/s
- <https://www.youtube.com/watch?v=a6XBOCQfcF4>



1. **Sekvenční jazyk** + **příkazy** paralelního zpracování, komunikace a synchronizace (např. Universal Parallel C)
2. **Sekvenční jazyk** + **dynamické knihovny**:
 - programování s vlákny (Pthreads, Java threads, WindowsThreads ..., Intel Threading Building Blocks)
 - nebo zasílání zpráv MPI (Message Passing Interface)
3. **Sekvenční jazyk** + **direktivy pro kompilátor** (pragma).
 - Umožňuje sekvenční i paralelní zpracování a inkrementální postup při paralelizaci
4. **OpenMP API**: kombinace direktiv a knihovních programů.

- **Vlákna jsou ovládaná a použitá kernelem OS**
 - Uživatel má k dispozici knihovny vláken, např. POSIX threads, Windows threads.
 - nebo API (**OpenMP**, OpenACC, CUDA, OpenCL).
 - OS mapuje (plánuje) SW vlákna na HW vlákna.
- **HW vlákno**
 - Prostředky provádějící vlákno nezávislé na jiných HW vláknech.
 - Vlákna 1 procesu mohou běžet na různých jádrech.
 - **Volání systému nebo knihovních programů** musí být pro vlákna bezpečné (**thread-safe**), tj. správnost je zajištěna i při volání z více vláken současně.

- Vyvinuto konsorciem (OpenMP ABR) hlavních výrobců HW a SW pro paralelní výpočty se sdílenou pamětí (procesory i akcelerátory)
- Existuje API pro C/C++ a pro Fortran
 - První verze v roce 1997.
 - Současná verze 5.0 (2018).
 - Verze 5.2 ve stádiu implementace.
- Anglické tutoriály:
 - OpenMP 3.0:
 - <https://computing.llnl.gov/tutorials/openMP/>
 - OpenMP 4.0:
 - <http://wiki.scinethpc.ca/wiki/images/9/9b/Ds-openmp.pdf>
 - Kompletní dokumentace 5.x, poslední vývoj, user's group:
 - <http://www.openmp.org/specifications/>
 - <http://community.org>



- **OpenMP podporuje:**

- jemný a hrubý paralelismus (smyčky, paralelní sekce)
- datový a funkční paralelismus (SPMD, paralelní sekce)

- **OpenMP dovoluje:**

- inkrementální parallelizaci programů se sdílenou pamětí
- psát přenosné a částečně škálovatelné programy
- psát paralelní programy se zabudovanou sekvenční verzí
- ověřit správnost programů

- **OpenMP zjednodušuje:**

- psaní vícevláknových programů ve Fortranu, C a C++
- ve verzi 4.0 existuje přidává vektorizaci SIMD (AVX)
- od verze 4.0 podporuje programování akcelerátorů (Xeon Phi, GPU)

- Vlákna OpenMP jsou abstrakcí
 - implementace je může mapovat na vlákna kernelu OS, lehká vlákna POSIX (P-threads), Win32* threads apod.
- OpenMP je nezávislé na stroji a OS
 - přenos správného programu jinam vyžaduje „jen“ rekompilaci
 - OpenMP-kompatibilní (-aware, -compliant) kompilátory existují pro všechny hlavní verze Unixu, Linuxu, Windows
 - Zapnutí OpenMP (gcc -fopenmp, intel -qopenmp)
- Implementace OpenMP pro C/C++ poskytuje hlavičkový soubor soubor **omp.h**
s definicemi a prototypy funkcí.

- Direktivy pro kompilátor

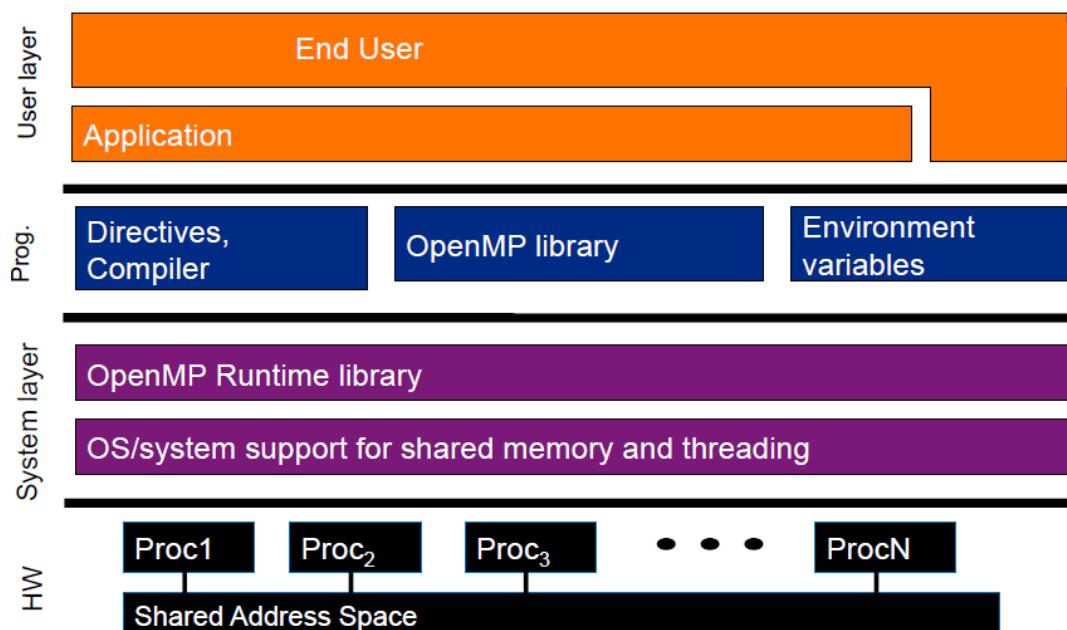
- `#pragma omp ...`
- vytvoření týmu vláken
- sdílení práce mezi vlákny
- synchronizace vláken

- Knihovní rutiny

- pro nastavení atributů vláken a dotazy na ně
 - `omp_set_num_threads(...)`,
 - `omp_get_thread_num()`,
 - `omp_get_num_threads()`,

- Proměnné prostředí

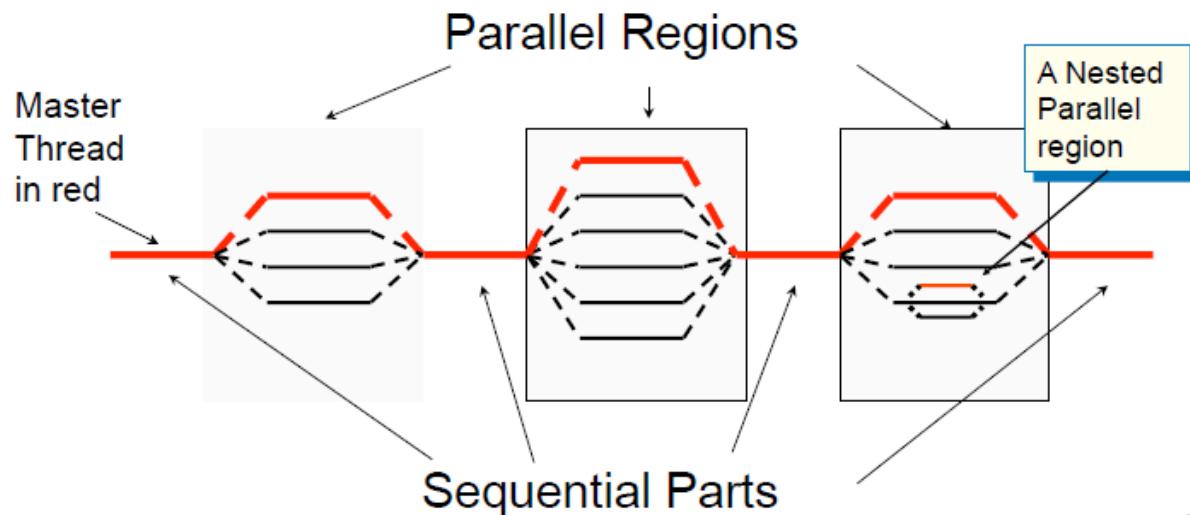
- řízení chování paralelního programu za jeho běhu (`OMP_NUM_THREADS`, `OMP_PLACES`, `OMP_PROC_BIND`, `OMP_ENV_DISPLAY`, ...)
- <https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf>



- **Vlákno**
 - IP + SP, registry, privátní zásobník.
 - Na svou identitu se může dotázat.
- **Je-li více vláken než logických jader** (oversubscription)
 - Pak se vlákna mapují na jádra dynamicky (Web servery).
 - Lépe 1 vlákno na 1 (logické) jádro, a staticky bindovat. Odpadá pak režie plánování vláken, vyplachování cache a další problémy.
- **Program OpenMP**
 - Se začne vykonávat jedním (hlavním) vláknem (**master thread**).
 - Jakmile dojde k **paralelnímu příkazu**, vytvoří se tým vláken (**worker threads**).
 - Členové týmu provádějí příkazy paralelně a synchronizují se na **bariéře**.
 - Program se může při běhu takto **větvit a zase spojit mnohokrát** (model fork–join).

- **Strukturovaný blok:**

- blok jednoho nebo { více příkazů; } s jedním vstupem nahoře a jedním výstupem dole.
- Může obsahovat exit() uvnitř.
- Každá OpenMP direktiva platí pouze pro následující strukturovaný blok.



Direktivy kompilátoru

VYTVOŘENÍ TÝMU VLÁKEN

- **Direktiva parallel**

- definuje paralelní oblast (region)
- vytvoří tým vláken
- zahájí paralelní zpracování
 - strukturovaný blok provádějí **všechna vlákna**, každé vlákno specificky podle svého ID (\rightarrow SPMD)
- hlavní (master) vlákno má číslo 0
- na konci paralelní oblasti je **automaticky bariéra**

nastavení počtu vláken

```
01  omp_set_num_threads(4);  
02  double a[1000];  
03  #pragma omp parallel [clause[clause]... ]  
04  {  
05      int id = omp_get_thread_num();  
06      work(id, a);  
07  }
```

Dovětky

Strukturovaný blok
vykonávaný 4 vlákny

```
private(list)
shared(list)
default(shared|none)
reduction(operator:list)
if(logical expression)
copyin(list)
num_threads(thread_count)
```

- Proměnné deklarované před paralelní oblastí musí být v jejím lexikálním (textovém) rozsahu označeny private nebo shared
 - bez označení budou shared (pozor na hazardy -> vzájemné vyloučení)
- if: vytvoření týmu vláken lze podmínit
 - podle výsledku nějakého testu uživatele (např. počet iterací), se pracuje sekvenčně if (FALSE) nebo paralelně if (TRUE)

- **Počet je určen následujícími faktory, pořadí podle priority:**

- Vyhodnocení dovětku `if` (jestli vůbec paralelně).
- Nastavení dovětkem `num_threads (thread_count)` v dané paralelní oblasti.
- Použitím knihovní funkce `omp_set_num_threads (int)` pro všechny následující paralelní oblasti.
- Nastavením proměnné prostředí pro celý program
`export OMP_NUM_THREADS=12`
- Implementační implicitní hodnota = obvykle počet CPU (v tomto počtu jsou tvořena i „dynamická“ vlákna).
- Počet vláken v týmu zůstává uvnitř paralelní oblasti konstantní. Pro další oblast jej může změnit:
 - uživatel (`void omp_set_num_threads (int)`)
 - obslužný systém (dynamická vlákna)

- Dovětek **private (var)**
 - Vytvoří novou lokální kopii proměnné *var* pro každé vlákno
 - Hodnoty privátních proměnných **nejsou** v paralelní oblasti **inicializovány** na hodnotu původní sdílené proměnné *var* (C++ volá **defaultní constructor**)
 - Pokud potřebujeme okopírovat původní hodnotu do lokálních proměnných (C++ **copy constructor**), použijeme direktivu **firstprivate**
 - Za paralelní oblastí nejsou privátní proměnné dostupné – obnoví se hodnota původní vnější proměnné před paralelní oblastí
 - Pokud potřebujeme přenést hodnotu privátních proměnných zpět do původní, můžeme využít buď **reduction** nebo **lastprivate**
- **Privatizovat** se mohou **jen úplné objekty**, nikoliv prvky polí nebo části datových struktur.
- **Proměnné** deklarované **uvnitř paralelní oblasti** jsou také **privátní** (bez dovětku)!

```
01 #include <omp.h>
02 #include <stdio.h>
03 void main() {
04     int tid, nt ;
05     #pragma omp parallel private(tid, nt)
06     {
07         nt = omp_get_num_threads();
08         tid = omp_get_thread_num();
09         printf("Hello from thread %d \
10             out of %d \n", tid, nt);
11     }
12 }
```

Direktiva paralelní oblasti,
počet vláken implicitní

Kolik nás je v týmu?

nt = omp_get_num_threads();
tid = omp_get_thread_num();

Moje ID

Tento strukturovaný
blok provádí každé
vlákno! (SPMD)

Konec paralelní oblasti

možné výstupy (4 vlákna)

Hello from thread 1 out of 4
Hello from thread 2 out of 4
Hello from thread 0 out of 4
Hello from thread 3 out of 4

Hello from thread 3 out of 4
Hello from thread 1 out of 4
Hello from thread 2 out of 4
Hello from thread 0 out of 4

Vlákna vypisují v náhodném pořadí, tak jak dospěly k funkci printf.

Příklad: Privátní proměnné

```
int x = 5, y = 6, z = 7;  
float a[10], b[10], c[10];  
#pragma omp parallel num_threads(5) \  
    private(x, a)           \  
    firstprivate(y, b)       \  
    shared(z, c)           \  
{  
    int thread_id = omp_get_thread_num();  
    x++; y++; z++;  
  
    a[thread_id] = 0;  
    b[thread_id] = 1;  
    c[thread_id] = 2;  
  
    a += thread_id; *a = 5;  
    b += thread_id; *b = 5;  
    c += thread_id; *c = 5;  
}
```

Co bude v proměnných?

Kam zapíši?

Kam zapíši?

- Program v OpenMP v sobě může mít zabudovánu sekvenční verzi:

Kompilátor s OpenMP

`#pragma omp`

se bere jako direktiva OpenMP

`#ifdef __OPENMP`

příkazy, které se mají provést jen v paralelní verzi (např. knih. fce)

`#endif`

Kompilátor bez OpenMP

`#pragma omp`

pragma je ignorováno

`#ifdef __OPENMP`

tyto příkazy se v sekvenční verzi vyneschají

`#endif`

- Makro `__OPENMP` = yyyy-mm = rok a měsíc schválené specifikace OpenMP.

```
#include <stdio.h>
#ifndef _OPENMP
    #include <omp.h>
#endif

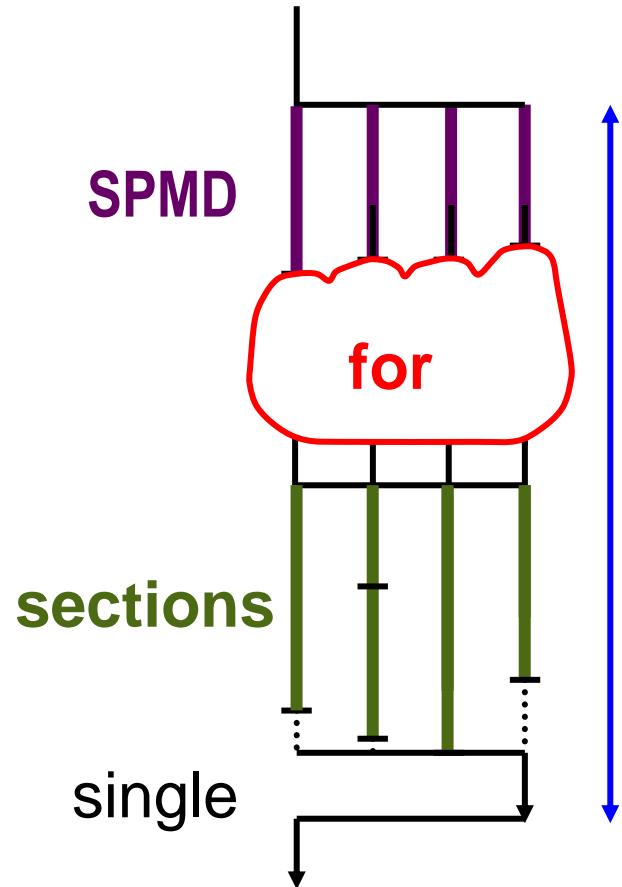
void main() {
    int tid = 0, nt = 1;
    #pragma omp parallel private(tid, nt)
    {
        #ifdef _OPENMP
            nt = omp_get_num_threads();
            tid = omp_get_thread_num();
        #endif
        printf("Hello from thread %d out of %d \n", iam, nt);
    }
}
```

- Direktivy platí pouze v jejich statickém, tj. **lexikálním rozsahu**
 - **Statický nebo lexikální rozsah**: kód textově uzavřen mezi začátek a konec strukturovaného bloku následujícího za direktivou
 - **Dynamický rozsah**: statický rozsah rozšířený o procedury a funkce volané zevnitř statického rozsahu
 - Direktiva, která je v dynam. rozsahu jiné direktivy, ale ne v jejím statickém rozsahu, se nazývá **sirotek** (orphan).

```
#pragma omp parallel
{
    void dowork();
}
```

```
void dowork ()
{
    #pragma omp for
    for (int i=0; ... )
    ...
}
```

- Direktiva **parallel** sama o sobě představuje program **SPMD**, tj. každé vlákno provádí redundantně stejný kód.
- **Jak diverzifikovat práci vláken v týmu?**
Pomocí direktiv **uvnitř** paralelní oblasti pro sdílení práce (work sharing):
 - ve smyčce **for**
 - v **sekcích**
 - 1 vláknem (single)
 - v úlohách (task)



Direktivy kompilátoru

PARALELIZACE SMYČEK

Sekvenční kód

```
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

OpenMP,
jen direktiva parallel

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if (id == Nthrds-1)iend = N;
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
}
```

OpenMP,
Direktiva parallel a for

```
#pragma omp parallel
#pragma omp for
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

#pragma omp for [clause[clause] ...]
blok

```
private(list)
firstprivate(list)
lastprivate(list)
nowait ... na konci smyčky není implicitní bariéra
```

reduction(operator: list)
schedule(kind[, chunk_size])
ordered

- počet iterací musí být znám na vstupu do smyčky
- všechny iterace se musí dokončit, blok: 1 vstup, 1 výstup
- schedule kind: static, dynamic, guided, runtime, auto
 - Norma OpenMP 4.5 zavádí modifikátor chunku simd, monothonic, nonmonothonic.
Např. **schedule (simd:static)**
- **vnořené smyčky:**
 - paralelně se dělá jen smyčka nejbližší direktivě
 - její index je implicitně privátní

```
#pragma omp parallel for
for (int y = 0; y < 25; ++y)
{
    #pragma omp for
    for (int x=0; x < 80; ++x)
    {
        tick(x, y);
    }
}
```

vnoření paralelních
smyček je nelegální

```
#pragma omp parallel for collapse(2)
for (int y = 0; y < 25; ++y)
    for (int x = 0; x < 80; ++x)
    {
        tick(x, y);
    }
}
```

novější dovětek **collapse**
vytvoří jednu smyčku ze
2 vnořených a tu paralelizuje.

Pozor: Zavádí režii výpočtu
indexů x a y

Tyto zápisy jsou ekvivalentní:

```
double res[MAX]; int i;  
#pragma omp parallel  
{  
    #pragma omp for  
    for (i = 0; i < MAX; i++)  
        res[i] = huge();  
}
```

```
double res[MAX]; int i;  
#pragma omp parallel for  
for (i = 0; i < MAX; i++)  
    res[i] = huge();
```

když paralelní oblast obsahuje
jen `#pragma omp for`

Podobně lze zkrátit

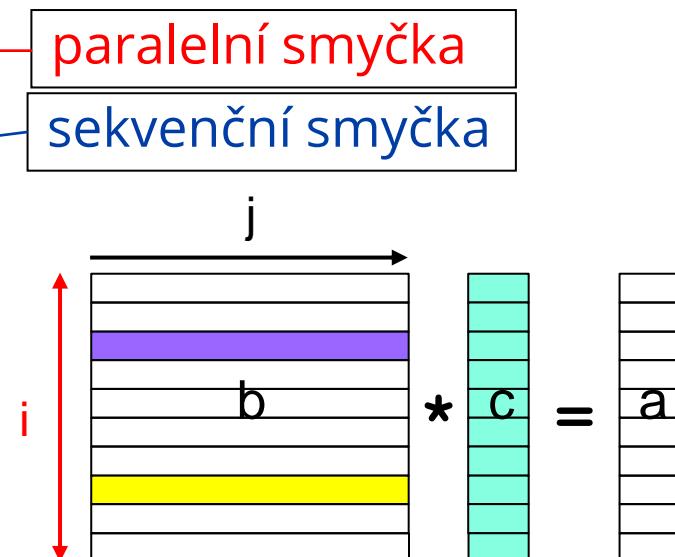
`#pragma omp parallel sections`

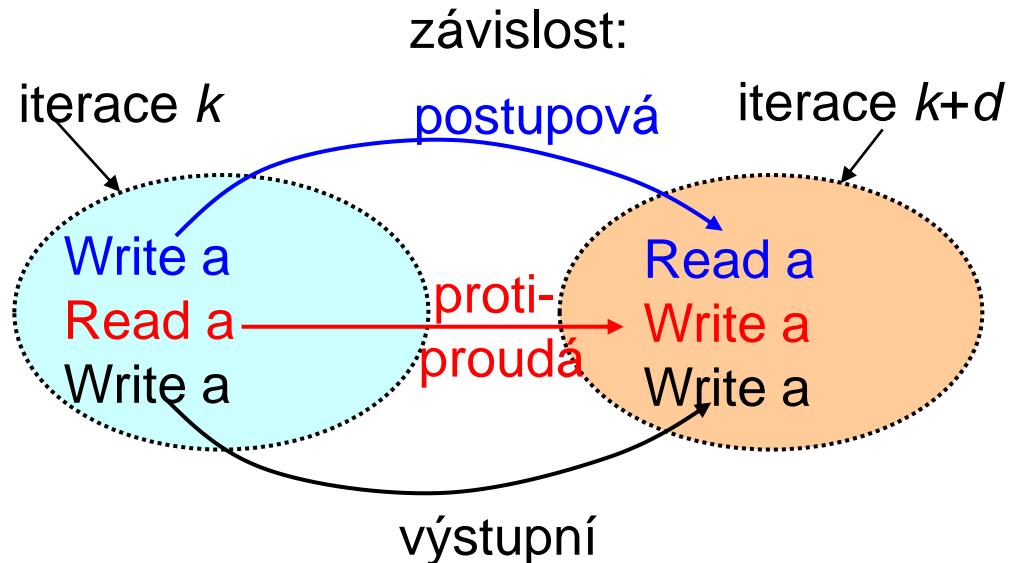
```

void mxv(int n, int m, double* a, double* b[],
         double* c)
{
    #pragma omp parallel for default(none) \
        shared(m,n,a,b,c)
    for (int i=0; i < n; i++) ← paralelní smyčka
    {
        double sum = 0.0;
        for (int j=0; j < m; j++) ← sekvenční smyčka
            sum += b[i][j]*c[j];
        a[i] = sum;
    }
}

```

Bude-li $n = 10$ řádků a 2 vlákna,
pak thread 0 si vezme řádky $i = \{0, 4\}$
a thread 1 řádky $i = \{5, 9\}$.





- Neznáme pořadí provedení iterací!
 - Iteraci k provádí jedno vlákno v čase t , ale iteraci $k+d$ může provádět jiné vlákno před nebo po iteraci k (nejisté!)
 - Kompilátory OpenMP **nekontrolují závislosti** mezi iteracemi.
 - Když výsledek **iterace závisí na jiných iteracích**, nelze přímo parallelizovat.

Jak udělat iterace smyčky nezávislé, aby mohly být prováděny v libovolném pořadí bez závislostí?

Příklad:

```
for (i=2; i<=m; i++)
    for (j=1; j<=n; j++)
        a[i][j] = 2 * a[i-1][j];
```

Závislost v indexu **i** přemístíme do sekvenční smyčky, tam nevadí:

```
int i, j;
#pragma parallel for private(i)
for (j=1; j<=n; j++) ← paralelní, j je impl. privátní
    for (i=2; i<=m; i++) ← sekvenční, i je privátní
        a[i][j] = 2 * a[i-1][j];
```

Pozor ale na lokalitu dat!

- **reduction** (*op: list*) provádí redukci skalárních proměnných na seznamu *list*, operátorem *op*
- je vytvořena **privátní** kopie každé proměnné ze seznamu *list* (jedna pro každé vlákno) a inicializována.
- Na konci je **atomicky** aktualizována každým vlákнем **stejnojmenná sdílená proměnná**.

```
double ave = 0.0, A[MAX]; int i;  
#pragma omp parallel for \  
    reduction (+: ave)  
for (i = 0; i < MAX; i++)  
    ave += A[i];  
ave = ave / MAX
```

může být
seznam

lze specifikovat
více redukcí
najednou

| Operátor | Poč. hodnota |
|----------|--------------|
| + | 0 |
| * | 1 |
| - | 0 |
| min | max. možná |
| max | min. možná |

| Operátor | Poč. hodnota |
|----------|--------------|
| & | ~0 |
| | 0 |
| ^ | 0 |
| && | 1 |
| | 0 |

OpenMP 4.0:

- Rozšíření o redukci definované uživatelem
#pragma omp declare reduction ...

```
int mini = a[0];
int maxi = a[0];
for (i=1; i<n; i++)
{
    if (a[i] < mini)
        mini = a[i];
    if (a[i] > maxi)
        maxi = a[i];
}
```

```
int mini, maxi;
#pragma omp parallel for reduction \
    (min:mini, max:maxi)
```

OpenMP smyčky

PLÁNOVÁNÍ ITERACÍ

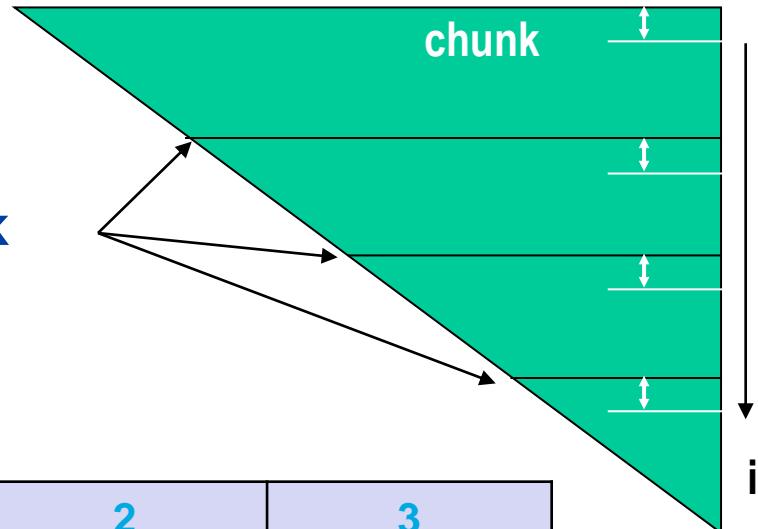
```
#pragma omp for schedule(static[, chunk_size])
for (i=0; i<n; i++) {
    the_same_work(i);
}
```

- nejméně režie při běhu, naplánování provedeno v době komplikace.
- když není udán chunk_size, dostane každé vlákno cca stejnou porci iterací q .
 - Je-li $n = t * q - r$, pak některá vlákna dostanou méně než q
(r -krát $q - 1$ nebo 1 krát $q - r$)
- je-li udán chunk_size, jsou porce přiděleny vláknům cyklicky (prokládané plánování). Užitečné, když se práce v iteracích mění lineárně.
- Jeli udán modifikátor SIMD, je chunk_size zarovnán na velikost SIMD registru.

```
#pragma omp for schedule(static[, chunk_size])
```

```
for (i=0; i < n; i++) {  
    the_same_work(i);  
}
```

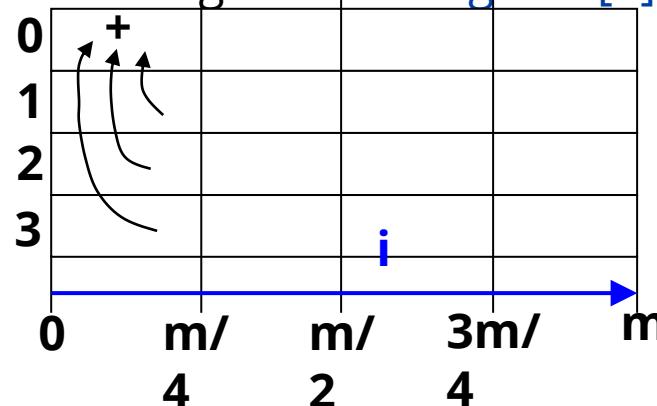
4 vlákna bez chunk



Příklad: 16 iterací, 4 vlákna:

| Thread ID | 0 | 1 | 2 | 3 |
|-----------|------------|--------------|--------------|--------------|
| no chunk | 0-3 | 4-7 | 8-11 | 12-15 |
| chunk=2 | 0-1
8-9 | 2-3
10-11 | 4-5
12-13 | 6-7
14-15 |

- Vektor $a[n]$ má prvky integer v intervalu $<0, m-1>$. Četnost výskytu různých hodnot se má znázornit histogramem **histogram[m]**.
 - Nejdřív každé z P vláken vytvoří vlastní histogram **histogram[myid][m]**, $myid = 0, \dots, nt-1$ ze svého segmentu vektoru $a[n]$
 - pak každé vlákno posčítá jednu část všech P histogramů do části jednoho globálního histogramu **histogram [0][m]**.



= histogram[0][m] (t0)
= histogram[1][m] (t1)
= histogram[2][m] (t2)
= histogram[3][m] (t3)

```
int histogram [10] [m];           // až 10 vláken
#pragma omp parallel shared (a, m, n)
{
    int nt, myid, i, k;
    nt = omp_get_num_threads();
    myid = omp_get_thread_num();
#pragma omp for schedule(static) num_threads(10)
    for (i = 0; i < n; i++)          // segmenty vektoru
        histogram[myid][a[i]]+=1;
#pragma omp for schedule(static) num_threads(2)
    for (i = 0; i < m; i++) {
        // segmenty histogramů
        for (k = 1; k < nt; k++)
            // vláken k > 0 do k = 0
            histogram[0][i]+=histogram[k][i];
    }
}
```

```
#pragma omp for schedule(dynamic[, chunk_size])
for (i = 0; i < n; i++) {
    unpredictable_amount_of_work(i);
}
```

- iterace jsou přidělovány po blocích ($\text{chunk_size} \geq 1$), se **synchronizací (exkluzivním přístupu k indexu)** při každém přidělení → největší režie za běhu
- default $\text{chunk_size} = 1$; čím je chunk_size větší, tím je menší synchronizační režie ale hrubší vyvážení zátěže.
- vlákno čeká na bariéře nejvýš dobu kterou trvá jinému vláknu dokončit jeho posledních chunk_size iterací.

```
#pragma omp for schedule(guided[, chunk_size])
for (i = 0; i < n; i++) {
    the_similar_work(i)
}
```

- menší synchronizační režie než **dynamic**;
- typická implementace přidělí prvnímu volnému vláknu porci
 $q_0 = \lceil n / t \rceil$ iterací
- následující porce se přidělují podle vztahu
 $q_i = \lceil q_{i-1} (1 - 1/t) \rceil$
(minimálně `chunk_size`, default `chunk_size` je 1);
- pro další iterace jde vlákno, které právě dokončí předchozí porci iterací.

Příklad: 200 iterací. Postupně odebrané porce:

$$50 = \lceil 200 / 4 \rceil,$$

$$38 = \lceil 50(1 - 1 / 4) \rceil,$$

$$29 = \lceil 38(1 - 1 / 4) \rceil,$$

22 = $\lceil 29(1 - 1 / 4) \rceil$, vlákno skončilo 1. porci iterací,

17, ... dostane 2. porci

13, vlákno přichází pro 2. porci,

10, vlákno přichází pro 2. porci,

8, ... vlákno přichází pro 3. porci,

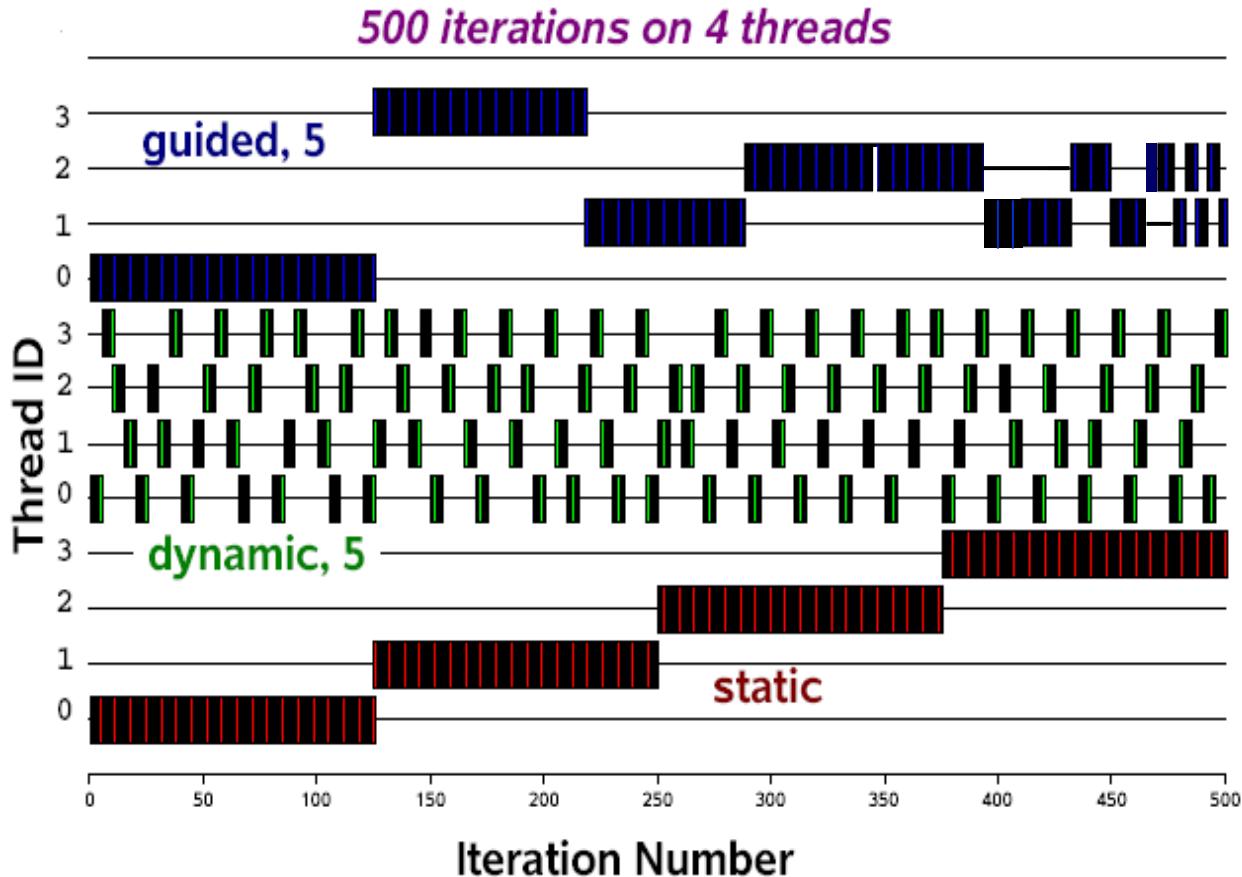
6, ... atd.

5,

2.

Celkem iterací na vlákna: **50, 50, 48, 52**
a 11 synchronizovaných přístupů
k indexu smyčky

I | Porovnání průběhu smyček



- Mějme 35 stejně náročných iterací 0 až 34, 3 vlákna 0 až 2. Najděte všechny dávky iterací přidělené vláknům 0–2.

| | vlákno 0 | vlákno 1 | vlákno 2 |
|-----------------------|-------------|----------|-------------|
| static,
no chunk | 12 | 11 | 11 |
| static,
chunk = 5 | $5 + 5 + 4$ | $5 + 5$ | $5 + 5$ |
| dynamic,
chunk = 7 | $7 + 7$ | $7 + 6$ | 7 |
| guided,
no chunk | 12 | $8 + 3$ | $6 + 4 + 1$ |

#pragma omp for schedule(runtime)

- Způsob plánování (schedule kind) a chunk_size mohou být zvoleny pomocí proměnné prostředí **OMP_SCHEDULE** před provedením programu, např. v Unixu příkazem shellu

```
setenv OMP_SCHEDULE "dynamic,3"
```

To je pohodlné pro experimentování, není nutná rekompilace.

#pragma omp for schedule(auto)

- Plánování je ponecháno na obslužném (runtime) systému. Vhodné, když se obslužný systém může učit z předchozích běhů též smyčky.

- `schedule(static)` má **nejmenší režii**
 - vhodné, když je práce v iteracích stejná (bez `chunk_size`) nebo se mění lineárně (pak je `chunk_size` nutný).
- `schedule(dynamic, chunk_size)` je vhodná pro **vyvážení zátěže**, když provedení iterací trvá různě dlouhou dobu
 - bez `chunk_size` je sync. režie **příliš velká** (implicitně `chunk_size = 1`)!
- `schedule (guided)` je vhodná pro iterace s ne příliš rozdílnou dobou provedení
 - má nižší počet porcí a tím nižší synchronizační režii než `dynamic`.

- **simd – schedule(simd:static)**

- Velikost chunku se vždy zaokrouhlí na nejbližší vyšší velikost simdlen.

```
int dot;  
#pragma omp parallel for simd \  
    schedule(simd:static:100) reduction(+: dot) aligned(a, b: 64)  
{  
    for (int i = 0; i < N; i++)  
        dot += (a[i] * b[i]);  
}
```

- **monotonic – schedule(monotonic:dynamic)**

- Pokud již vlákno vykonalo iterací i , musí být následující iterace větší než i (monotoně rostoucí posloupnost)

- **nonmonotonic – schedule(nonmonotonic:dynamic)**

- Iterace se mohou přidělovat v libovolném pořadí

```
#pragma omp parallel default(none) \
    shared(n, a, b, c, d) private(i)
{
    #pragma omp for nowait
    for (i = 0; i < n-1; x++)
    {
        b[i] = (a[i] + [a[i+1]]) / 2;
    }

    #pragma omp for nowait
    for (i = 0; i < n; y++)
    {
        d[i] = 1.0 / c[i];
    }
}
```

Zrušena
implicitní
bariera

OpenMP

IMPLICITNÍ CHOVÁNÍ PROMĚNNÝCH

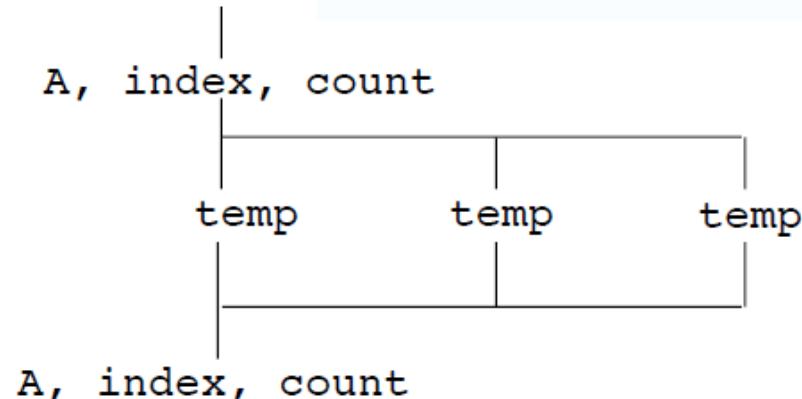
- Proměnné deklarované/použité **před příkazem parallel** jsou v paralelní oblasti implicitně **sdílené**.
 - např. indexové proměnné sekvenčních smyček musí být v dovětku označeny jako privátní.
 - **indexové proměnné paralelních smyček** nemusí být na seznamu v dovětku `private()`, jsou implicitně privátní.
- Automatické proměnné deklarované v bloku příkazů **uvnitř paralelní oblasti** nebo lokální proměnné ve funkcích volaných z paralelní oblasti jsou **privátní (na zásobníku)**.
 - Proměnné **static** jsou **sdílené**, a jsou staticky alokovány kompilátorem v předurčené oblasti sdílené paměti.

```
extern double A[10];
void work(int * index)
{
    double temp[10];
    static int count;
}
```

A, index a count
jsou **sdílené**
všemi vlákny.

temp je lokální
(privátní) v každém
vláknu.

```
double A[10];
int main() {
    int index[10];
#pragma omp parallel
    work(index);
    printf("%d\n", index[0]);
}
```



- **firstprivate**(list)... pro inicializaci stejnojmenných privátních proměnných všech vláken ve smyčce jsou použity původní hodnoty proměnné hlavního vlákna 0 (konstrukce C++ copy konstruktorem).
- **lastprivate**(list) ... hodnota privátní proměnné patřící vláknu, které provedlo **poslední sekvenční iteraci** (resp. sekci označenou jako poslední) je přiřazena kopii proměnné vlákna 0.
- proměnná může být případně jak firstprivate, tak lastprivate.
- když je použit **default(none)**, musí být každá proměnná v některém seznamu (list) – **shared, first/last/private**.
 - Výhodné pro debug kódu.

```
main() {  
    int C, B; int A = 20, n = 100, idx, *data;  
    ... // alokace a načtení vektoru data  
  
    #pragma omp parallel  
    {  
        #pragma omp for firstprivate(A) \  
                     lastprivate(B, idx)  
        for (int i = 0; i < n; i++) {  
            B = A + i; /* A: je-li jen private, není def. */  
            if (data[i] == 0) idx = i;  
        }  
        C = B; /* B: je-li jen private, není hodnota B a tedy C def. */  
        /* sdílené C = lastprivate B = 20 + n - 1 */  
    }  
}
```

- } /* konec paralelní oblasti: idx je náhodně nastaveno některými vlákny v některých iteracích. Lastprivate je zde nesmysl, lépe je použít redukci (např. když hledáme nejvyšší index nulového prvku ve vektoru data). */

- Globální proměnné, které mají být privátní pro každé vlákno v rámci celého programu, bez ohledu na lexikální rozsah direktivy parallel, lze definovat pomocí:
#pragma omp threadprivate (list)
- **Threadprivate** proměnné všech vláken mohou být inicializovány na začátku paralelní oblasti hodnotami hlavního vlákna pomocí **copyin (list)** nebo v době jejich definice.
- **Příklad:** vytvoření čítače pro každé vlákno:

```
int counter = 0;  
#pragma omp threadprivate (counter)
```

```
int increment_counter()  
{  
    counter++;  
    return (counter);  
}  
← volají vlákna v paralelní oblasti,  
master i v sekvenční oblasti  
← vlákno t(id) inkrementuje svůj čítač
```

Pokračování příště

Programování se sdílenou pamětí

OpenMP sekce a tasky

AVS – Architektury výpočetních systémů

Týden 8, 2024/2025

Jirka Jaroš

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



OpenMP Direktivy kompilátoru

PARALELNÍ SEKCE

- Sekce určují úseky kódu, které mohou běžet paralelně.
- Každá sekce je provedena **jen jednou** nějakým vlákнем v týmu.
- Sekcí může být více než vláken, ale pak nemůžeme mezi sekci komunikovat stylem producent -> konzument!
- Počet sekcí nelze měnit dynamicky za chodu programu.

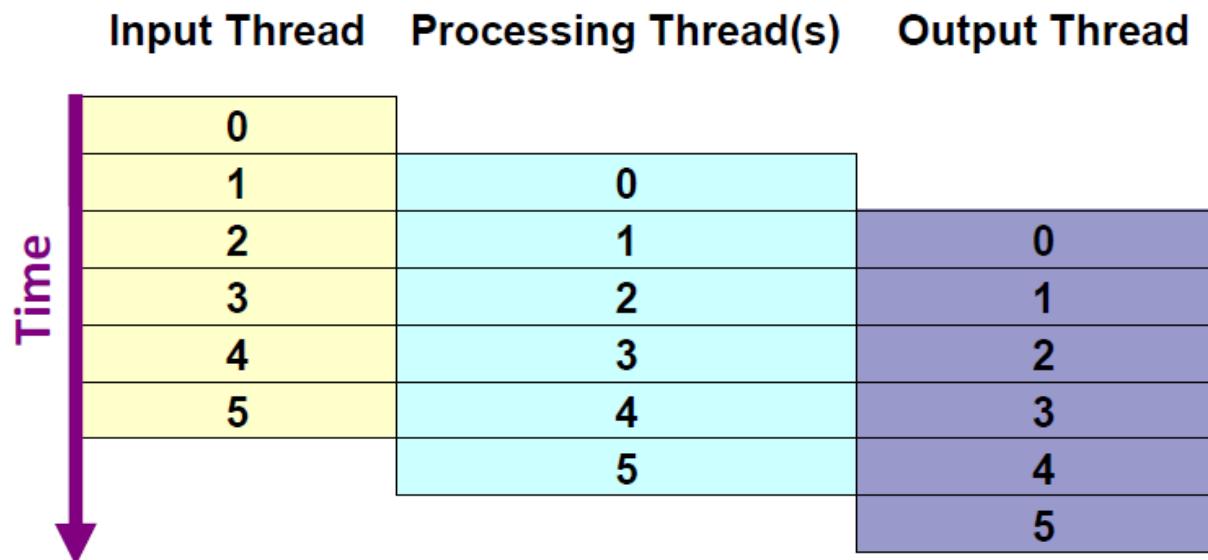
```
#pragma omp sections [clause[ clause] ...]
{
    #pragma omp section
    {work1(); }
    #pragma omp section
    {work2();
     work3(); }
    #pragma omp section
    {work4(); }
}
```

/* implicitní bariéra pokud se nepoužije **nowait** */



```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait
```

- Pipeline, producent/konzument, Master-Workers, offload zpracování na GPU/XeonPhi...
- Data mezi sekczemi předávána pomocí **sdílených bufferů strážených kritickými sekczemi**.
- Pomocí **nested parallelizmu** je možné přiřadit více vláken jedné sekci.



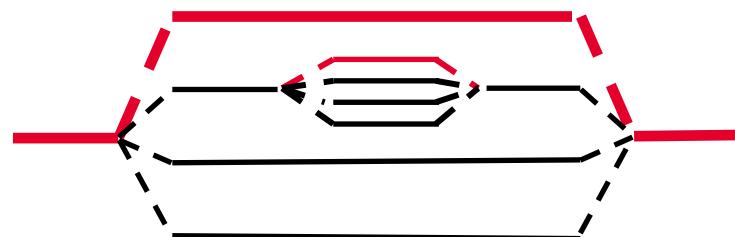
```
#pragma omp parallel sections num_threads(3)
{
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) read_input(i);
            (void) signal_read(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_read(i);
            (void) process_data(i);
            (void) signal_processed(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_processed(i);
            (void) write_output(i);
        }
    }
}
/*-- End of parallel sections --*/
```

Input Thread

Processing Thread(s)

Output Thread

- Direktiva **parallel** uvnitř jiného **parallel** stanoví dynamicky nový vnořený tým vláken. U současných implementací je tým omezen na 1 vlákno (default) a úroveň zanoření na 1.
- **Toto chování lze změnit**
 - nastavením proměnné prostředí **OMP_NESTED** na **TRUE**
 - voláním funkce dyn. knihovny **omp_set_nested()** ;
 - aktuální stav lze zjistit pomocí **omp_get_nested()** ;
- **Vlákno které dorazí k direktivě parallel se stává novým master vlákнем pro vnořenou část.**
 - Pokud se ve vnořené paralelní oblasti zeptám na počet vláken, získaná hodnota je pro tuto úroveň!
 - Abych zjistil kolik je celkem všech vláken na všech úrovních, musím tuto funkci zavolat všude a pak redukcí summarizovat, případně pomocí prefix-scanu zjistit svoje globální ID



- Není to nutně hlavním vláknem (master), ale prvním vláknem, které dojde k direktivě **single**.
- Užitečné pro tisk zpráv o postupu řešení, vstup dat, přidělování tasků.
- **Obsahuje implicitní bariéru** pokud se nepoužije **nowait**.

```
#pragma omp single [clause[ clause] ...]
```

```
    printf ("Work done.\n");
```



```
private(list) , firstprivate(list) ,  
nowait, copyprivate(list)
```



vlákno single zpřístupní svoje privátní proměnné ostatním vláknům v týmu (broadcast)

```
#include <omp.h>

void input_params(int, int); // Read commandline params
void do_work(int, int);

void main() {
    int Nsize, choice;

#pragma omp parallel private(Nsize, choice)
{
    #pragma omp single copyprivate(Nsize, choice)
        input_params(Nsize, choice);
        // implicitní bariera
        do_work(Nsize, choice);
}
}
```

Broadcast private
proměnných všem

#pragma omp master

strukturovaný blok

- Označuje blok kódu **v rámci paralelní oblasti**, který je proveden jen **hlavním vlákнем**.
- Ostatní vlákna blok přeskočí a nemusí ani dojít k této direktivě.
- Na rozdíl od single **neobsahuje** implicitní bariéru na konci bloku.
- **Použití:**
 - pro omezení I/O operací jen na hlavní vlákno
 - pro přístup do jeho threadprivate proměnných
 - generování tasků,...

- K příkazu musí dojít všechna vlákna nebo žádné.
- Každé vlákno musí projít sérii příkazů sdílení práce ve stejném pořadí.
- Není dovoleno skočit dovnitř nebo ven z bloku spojeného s tímto příkazem (**jedině exit()**).
- Vnořování příkazů sdílení práce **je nelegální**.
- Příkazy sdílení práce **nesmí uvnitř obsahovat bariéru**.
- Mají implicitní bariéru nebo nowait na konci.
- Direktivy sdílení práce mohou být **sirotci**.

- Dovolují během **sériové části** programu automaticky přenastavit **počet vláken tak, aby byl roven počtu použitelných jader** a to dynamicky, podle aktuální zátěže systému např.
v multiprogramovém prostředí, kde běží více aplikací/uživatelů současně.
 - Počet vláken se nemůže změnit v době provádění paralelní oblasti.
-
- Nastavení **počet vláken = počet jader:**
`omp_set_num_threads (omp_get_num_procs())`
 - Nastavení dynamických vláken pro celý program:
`setenv OMP_DYNAMIC (TRUE, FALSE)`
 - Povolení/zákaz dynamických vláken za běhu:
`void omp_set_dynamic(int) (0, 1)`
 - Dotaz jsou-li dynamická vlákna povolena/zakázána:
`int omp_get_dynamic(void) (0, 1)`

| Name | Functionality |
|----------------------------------|---|
| <code>omp_set_num_threads</code> | <i>Set number of threads</i> |
| <code>omp_get_num_threads</code> | <i>Return number of threads in team</i> |
| <code>omp_get_max_threads</code> | <i>Return maximum number of threads</i> |
| <code>omp_get_thread_num</code> | <i>Get thread ID</i> |
| <code>omp_get_num_procs</code> | <i>Return maximum number of processors</i> |
| <code>omp_in_parallel</code> | <i>Check whether in parallel region</i> |
| <code>omp_set_dynamic</code> | <i>Activate dynamic thread adjustment
(but implementation is free to ignore this)</i> |
| <code>omp_get_dynamic</code> | <i>Check for dynamic thread adjustment</i> |
| <code>omp_set_nested</code> | <i>Activate nested parallelism
(but implementation is free ignore this)</i> |
| <code>omp_get_nested</code> | <i>Check for nested parallelism</i> |
| <code>omp_get_wtime</code> | <i>Returns wall clock time</i> |
| <code>omp_get_wtick</code> | <i>Number of seconds between clock ticks</i> |

| | |
|------------------------------|---|
| <code>OMP_NUM_THREADS</code> | positive number |
| <code>OMP_DYNAMIC</code> | TRUE or FALSE |
| <code>OMP_NESTED</code> | TRUE or FALSE |
| <code>OMP_SCHEDULE</code> | "static,2" |
| <code>OMP_PROC_BIND</code> | TRUE, FALSE, CLOSE,...
(obslužný systém ne/bude přesunovat vlákna mezi jádry) |
| <code>OMP_DISPLAY_ENV</code> | TRUE or FALSE |
| plus další ... | |

OpenMP Direktivy kompilátoru

TASKY

Direktiva sdílení práce task

- Task (úloha) má kód, datové prostředí (svoje data), interní řídicí proměnné a přiřazené vlákno.
- Jedno vlákno **může** generovat tasky a ty jsou pak prováděny vlákny týmu v neurčeném pořadí (nezávisle na sobě).
- Umožňuje paralelizaci smyček while, rekurzivních výpočtů, ...

```
#pragma omp task [clause[ clause] ...]
```

strukturovaný blok



dovětky definují datové prostředí tasku:

```
default (shared | none)
private (list)
firstprivate (list)
shared (list) a další... (mergeable, final, untied)
```

- Vlákno, **které dojde k direktivě task** vygeneruje novou instanci tasku, zabalí kód a data pro provedení.
- Pro každou **private** a **firstprivate** proměnnou **je alokována paměť**, hodnota firstprivate proměnné je inicializována hodnotou původní proměnné před direktivou task.
- **Nějaké vlákno v paralelní oblasti pak provede task**, obslužné prostředí zařídí okamžité nebo pozdější provedení.

- Proměnné, které jsou **privátní** na vstupu do tasku, jsou uvnitř tasku implicitně **firstprivate**.
- **Sdílené** proměnné před direktivou task zůstávají **sdílené** i uvnitř tasku.
- Chceme-li dostat nějaký výsledek z tasku ven, musíme přes sdílenou proměnou.
- Proměnné deklarované v tasku jsou **privátní**, v tasku – sirotku **firstprivate**.

```
int b , c ;  
#pragma omp parallel private ( b )  
{  
    int d ;  
    #pragma omp task  
    {  
        int e ;  
        b = firstprivate  
        c = shared  
        d = firstprivate  
        e = private  
    }  
}
```

Příklad tvorby tasků

```
#pragma omp parallel ← Zde se vytvoří vlákna.  
{  
    #pragma omp master ← Pouze master vlákno projde tuto část  
    {  
        #pragma omp task ← 3 nezávislé úlohy.  
        fred();  
  
        #pragma omp task ← Ostatní vlákna nečekají a hned jdou  
        daisy();  
  
        #pragma omp task ← zpracovávat.  
        billy();  
    } ← Zde je fronta čekajících tasků. V  
} ← tomto bodě je garantováno  
      dokončení všech tasků.
```

Příklad na tasky: Průchod vázaným seznamem

```
my_pointer = listhead;  
#pragma omp parallel  
{  
    #pragma omp single nowait  
    {  
        while (my_pointer)  
        {  
            #pragma omp task firstprivate(my_pointer)  
            {  
                do_independent_work (my_pointer);  
            }  
            my_pointer = my_pointer->next;  
        }  
    } // end of single - bariéra potlačena (nowait)  
} // end of parallel region - implicitní bariéra
```

Jedno vlákno bude řídit smyčku while,
generovat tasky pro další vlákna
týmu

my_pointer musí být firstprivate,
aby každý task měl def. svou hodnotu

blok 1

blok 2

blok 3

Všechny tasky dokončí zde

I Kdy je task dokončen?

- Na vláknové barieře (implicitní nebo explicitní)
 - Všechny tasky vygenerované v jednom paralelním regionu, dokončí v tomtéž regionu.
- Na direktivě taskwait
 - Čeká se na dokončení všech tasků definovaných v aktuálním tasku.
 - Platí pouze pro tasky vygenerované v tomto tasku, tedy neplatí pro následníky.
- Na konci taskgroup regionu
 - Čeká se na všechny tasky generované z tohoto tasku, tedy i následníky.

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        fred();

        #pragma omp task
        daisy();

        #pragma omp taskwait
        #pragma omp task
        billy();
    }
}
```

fred() a **daisy()** musí dokončit dříve než **billy()** začne

| | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|----|----|----|----|
| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| fib(n) | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 |

```
if (n<2) return n;
```

Sekvenčně:

```
int seqfib(const int n)
{
    int x, y;
    if (n < 2) return n;

    x = seqfib(n - 1);
    y = seqfib(n - 2);
    return x + y;
}
```

Jakmile je dosaženo jisté hodnoty n , je lépe počítat $\text{fib}(n)$ sekvenčně a režii tasků v OpenMP vyněchat:

```
if (n≤ 30) return seqfib(n);
```

I Fibonacci čísla paralelně s tasky

```
int main (int argc,  
          char **argv)  
{  
    int n, result;  
    n = atoi (argv[1]);  
    #pragma omp parallel  
    {  
        #pragma omp single  
        {  
            result = fib(n);  
        }  
    }  
    printf ("fib(%d)=%d\n",  
           n, result);  
}
```

```
int fib (int n)  
{  
    int x, y;  
    if (n< 2 ) return n;  
    if (n≤ 30) return seqfib(n);  
  
    #pragma omp task shared(x)  
        x = fib(n-1);  
    #pragma omp task shared(y)  
        y = fib(n-2);  
    #pragma omp taskwait  
    return x+y;  
}
```

pozastav rodič.
task, až dokončí
dceřiné tasky

x+y musí být přístupné –
shared, default je firstprivate

- Závislosti mezi tasky lze specifikovat pomocí sdílených proměnných a klauzule **depend**.
- **Depend (in: var)** – task se nespustí dokud se nevykonají přechozí tasky, které se odkazují na proměnou var v klauzuli **depend (out: var)** – závislosti na které se má čekat.

```
#pragma omp task depend (out: a)
{ ... } // writes a
```

```
#pragma omp task depend (out: b)
{ ... } // writes b
```

```
#pragma omp task depend (in: a, b)
{ ... } // reads a and b
```

- v OpenMP 4.0 lze čekat i na části polí

```
#pragma omp task depend (in: a[1:10]))
```

- Umožnuje suspendovat právě probíhající task a nechat běžet jiné

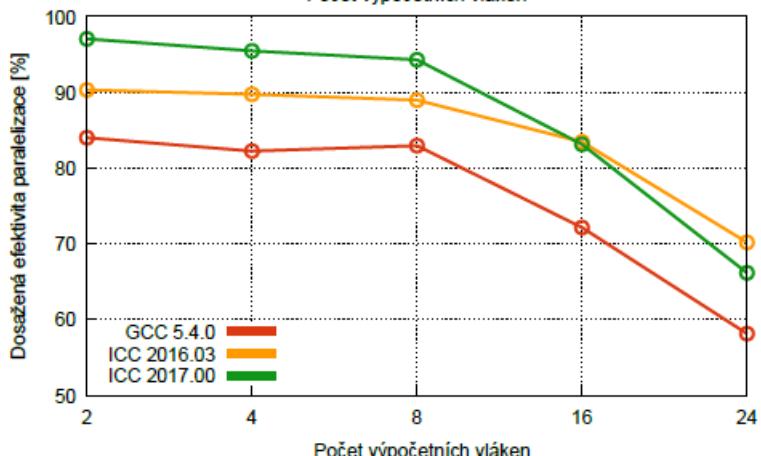
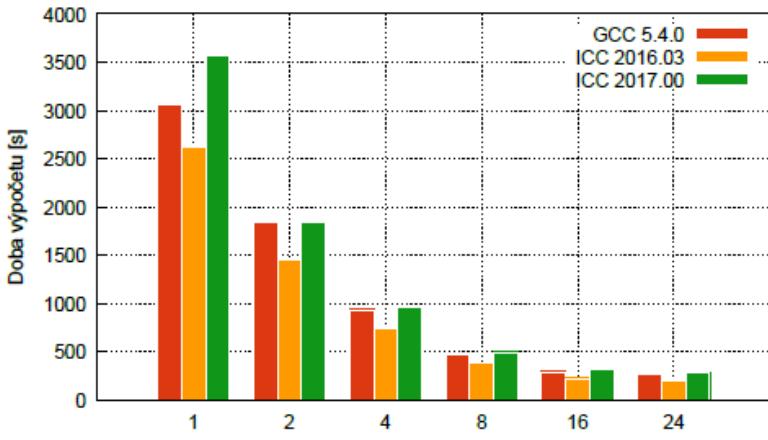
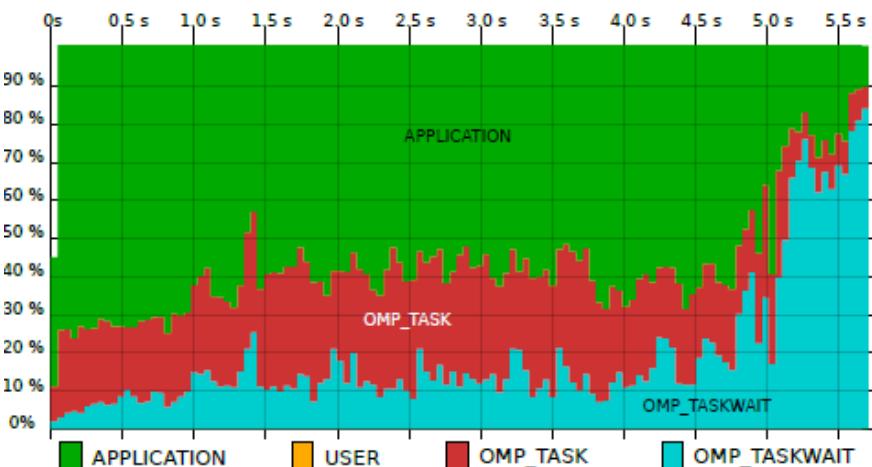
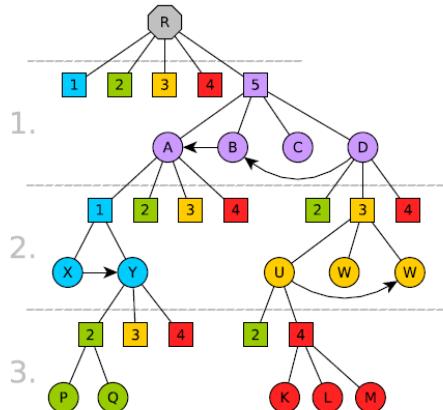
```
#include <omp.h>

void something_useful();
void something_critical();

void foo(omp_lock_t * lock, int n)
{
    for(int i = 0; i < n; i++)
        #pragma omp task
        {
            something_useful();
            while( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
}
```

The waiting task may be suspended here and allow the executing thread to perform other work. This may also avoid deadlock situations.

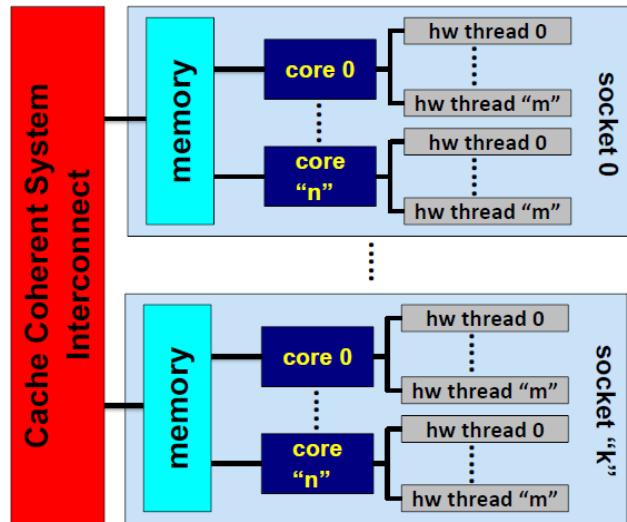
- Tasky jsou implicitně svázány (**tied**) s vlákнем, které je jako první začne vykonávat – ne to které ho vytvoří!
- Platí následující omezení pro plánování vykonání
 - Pouze vlákno které je svázáno s taskem jej může vykonávat.
 - Task může být suspendován (přerušen) pouze v tzv. scheduling points
 - Vytvoření tasku, ukončení tasku, taskwait, barrier, taskyield.
 - Pokud není vlákno suspendováno na bariéře, může vykonávající vlákno přeponout pouze k přímému potomku všech tasků svázaných s tímto vlákнем.
- **Tasky lze vytvářet i v režimu untied**
 - Task může být převzat jiným vlákнем na scheduling pointu.
 - Umožnuje vyšší flexibilitu při implementaci a vyvažování zátěže.
 - Ale:
 - threadprivate proměnné mohou být nedefinované
 - Ptát se na ID vlákna nedává smysl
 - Pozor na kritické sekce



OpenMP Direktivy kompilátoru

NOVINKY VERZE 4.0/4.5

- Jak mapovat OpenMP vlákna na HW vlákna v systému?
- HW vlákna jsou číselována (/proc/cpuinfo)
- OS může libovolně migrovat OpenMP vlákno přes HW vlákna.
- Vlákna lze i **zamknout** v rámci tzv. `OMP_PLACES` pomocí `OMP_PROC_BIND` nebo `proc_bind` klauzule.
- Existují 3 abstraktní místa (thread, core, socket)
- Příklady:
 - `OMP_PLACES = "{0, 1, 2}, {5, 6, 7}"`
 - `OMP_PLACES = "{0-7}, {8-15}"`
 - `OMP_PLACES = threads | cores | sockets`



- Proměnná OMP_PROC_BIND udává jak se OpenMP vlákna distribuuují přes OMP místa.
 - **Master** – Každé vlákno v týmu je přiřazeno na stejné místo jako master vlákno.
 - **Close** – Přiřadí vlákna z týmu na místa blízká rodičovskému vláknu (blízkost je definována pořadím míst v OMP_PLACES).
 - Využití – využít nejprve vlákna na jednom HW jádře/socketu.
 - **Spread** – Round robin distribuce vláken přes všechna místa.
 - Použití všech jader/socketů v systému

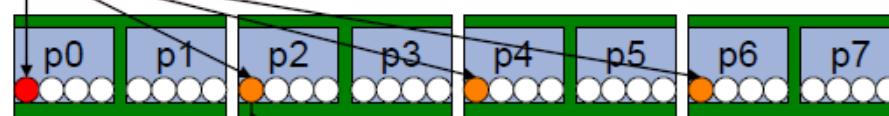
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4    = cores  
#pragma omp parallel proc_bind(spread)  
#pragma omp parallel proc_bind(close)
```

Example

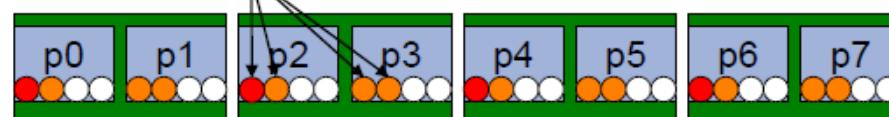
→ initial



→ spread 4



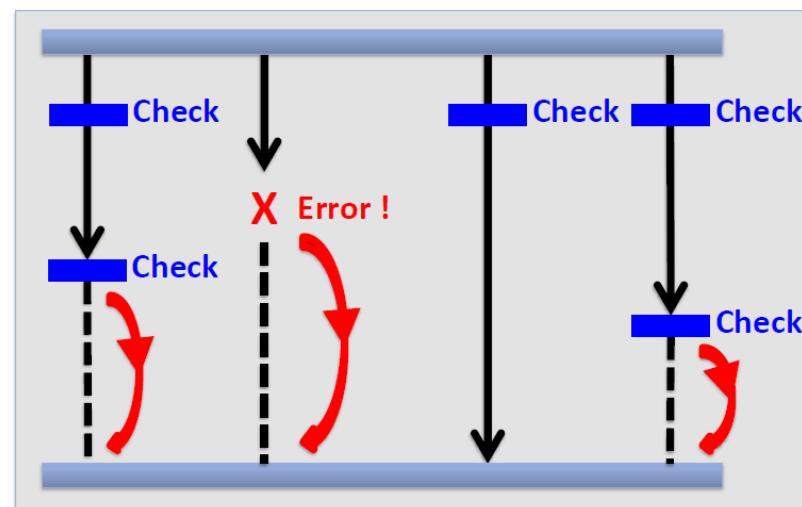
→ close 4



- **Cancel – provede přerušení OpenMP regionu**
 - Vlákno (úloha), které je přerušeno pokračuje ke konci přerušeného regionu
 - `#pragma omp cancel <construct> [if-clause]`
- **Cancellation point – Bod, kde vlákna / úloha testuje jestli má být vykonávání přerušeno**
 - Buď přerušení nebo pokračování
 - Mezi tyto body patří implicitní bariéry, regiony mezi bariérami, cancel regiony, cancellation pointy
 - `#pragma omp cancellation point <construct>`

Podporované konstrukty

- Parallel
- Sections
- For
- Taskgroup



Příklad na OpenMP cancel – Je v matici nulový prvek?

```
bool has_zero = false;  
#pragma omp parallel default(none) shared(matrix, has_zero)  
{  
    #pragma omp for  
    for (int row = 0; row < rows; row++)  
    {  
        for (int col = 0; col < cols; col++)  
        {  
            if (matrix(row, col) == 0)  
            {  
                #pragma omp critical  
                {  
                    has_zero = true;  
                }  
                #pragma omp cancel for  
            }  
        }  
    }  
    #pragma omp cancellation point for  
}
```

Zde přerušíme výpočet

Zde se ostatní vlákna dozví, že mají ukončit výpočet

```
void search_parallel(btreet *t,  
element e, bool* present)  
{  
    #pragma omp parallel  
    {  
        #pragma omp master  
        {  
            #pragma omp taskgroup  
            search(t, e, present);  
        }  
    }  
}
```

Taskgroup zajistí
zrušení všech
vygenerovaných tasků
v této oblasti

```
void search (btreet *t, element e, bool*  
present)  
{  
    if (t->element == e)  
    {  
        #pragma omp cancel taskgroup  
        *present = true; Task se vždy dokončí celý.  
    }  
    if (t->left)  
    {  
        #pragma omp task  
        search(t->left, e, present);  
    }  
    if (t->right)  
    {  
        #pragma omp task  
        search(t->right, e, present);  
    }  
}
```

Pokračování příště

Programování se sdílenou pamětí

OpenMP synchronizace

AVS – Architektury výpočetních systémů

Týden 9, 2024/2025

Jirka Jaroš

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



- **Synchronizace slouží**

- k ochraně přístupů ke sdíleným datům,
- k čekání na nějakou událost nebo
- k vynucení pořadí akcí.

- **OpenMP direktivy:**

- **Vysoká úroveň**

critical,
atomic,
barrier,
master,
ordered

} **vzájemné vyloučení**

} **synchronizace událostmi**

- **Nízká úroveň** (synchronizace na míru):

flush

locks (jednoduché a vnořené)

barrier, flush
jsou implicitně
součástí
jiných direktiv!

1. Několik vláken čte a modifikuje **disjunktní prvky** sdíleného objektu
= **bez problému**
2. Několik vláken čte a zapisuje **do téže sdílené proměnné** (datové struktury)
= **problém**

Případ 2 vyžaduje synchronizaci.

```
#pragma omp parallel for \
shared(a, b)
for (i = 0; i < n; i++)
{
    a[i] += b;
}
```

~~```
s = 0;
#pragma omp parallel for \
shared(a)
for (i = 0; i < n; i++)
{
 s += a[i];
}
```~~

**THREAD 1: increment(x)**

```
{
 x++;
```

**THREAD 1:**

```
1 lw r2, 0(r1)
2 addi r2, #1
3 sw r2, 0(r1)
```

Není  
atomická  
operace

**THREAD 2: increment(x)**

```
{
 x++;
```

**THREAD 2:**

```
1 lw r2, 0(r1)
2 addi r2, #1
3 sw r2, 0(r1)
```

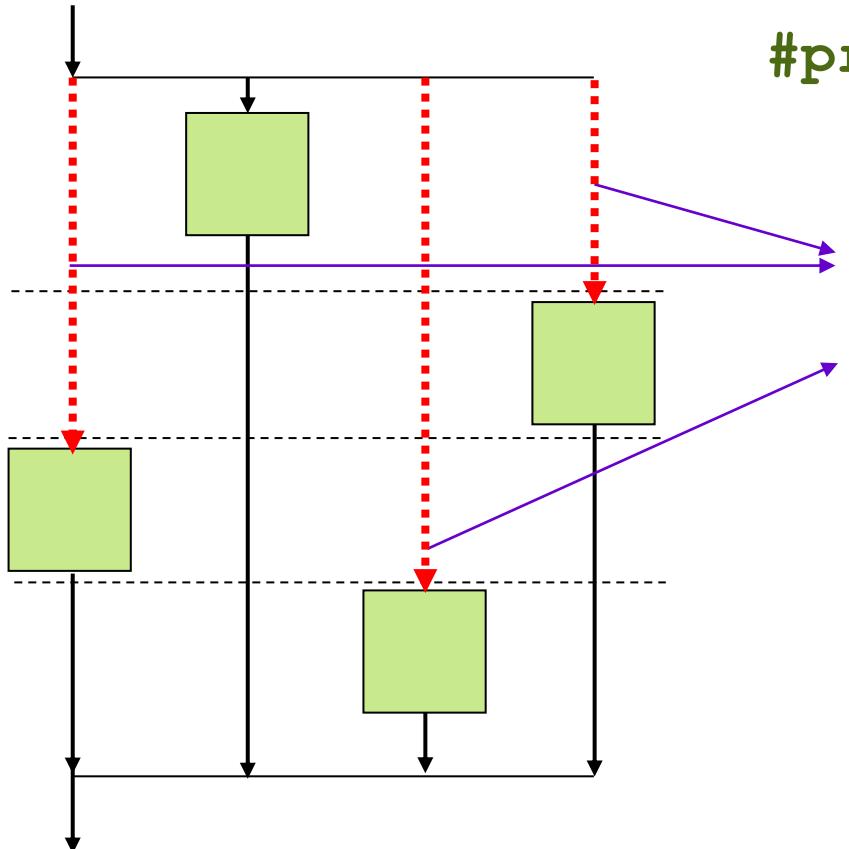
- Proložení 1-1-2-2-3-3 dává výsledek  $x = 1$
- Proložení 1-2-3-1-2-3 dává zamýšlený výsledek  $x = 2$
- Adresa  $x$  je v registru  $r1$ , přičte se k ní offset 0

```
/*nalezení počtu nulových prvků vektoru*/
int count =0;
pragma omp parallel for
 for (i = 0; i < n; i++)
 if (a[i] == 0) count++;
```

- **špatně**, modifikace sdílené proměnné **count** souběžně více vlákny.
- Vlákna musejí **přistupovat ke count exkluzivně**, přístup jednoho vlákna vylučuje přístup jiného!  
(Zde se snadno vyřeší dovětkem **reduction(:count)**)

**OpenMP direktivy kompilátoru**

# **VZÁJEMNÉ VYLOUČENÍ**



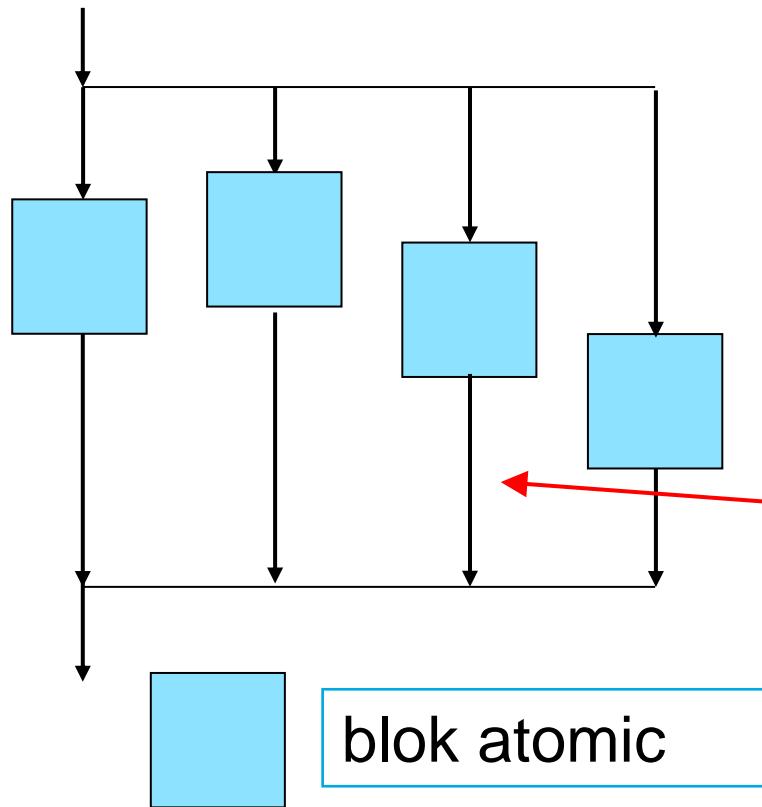
`#pragma omp critical[ (name) ]  
blok`

Čekání na vstup do  
kritické sekce

Každé vlákno provádí kritickou sekci,  
ale v určitém okamžiku pouze jediné.  
**Pořadí není zaručeno!**



## #pragma omp atomic



- Každé vlákno provádí blok **atomic**, např. inkrementaci prvku vektoru, **nerozdělitelně**.
- Pokud každé vlákno modifikuje
  - **jiný prvek vektoru**, mohou běžet souběžně
  - **stejnou proměnnou**, běží jedno vlákno po druhém (serializace jako u critical)

**1. Vzájemné vyloučení** pomocí **kritické sekce (CS)** znamená, že CS se provádí postupně jednotlivými vlákny (serializace, opak paralelizace).

**Implementace:** pomocí sdílené binární proměnné **zámek (lock)**.

**2. Atomicita** znamená nepřerušené zpracování sekvence instrukcí.

**Implementace:** pomocí atomických instrukcí (čtení – modifikace – záznam). Několik vláken může pracovat v atomické sekci **souběžně**, pokud

- pracují **s různými daty** a pokud
- atomické instrukce z více vláken **nejsou serializovány** architekturou (sběrnice).

```
s = 0;
#pragma omp parallel for \
 shared(a)
for (i = 0; i < n; i++)
{
 #pragma omp critical
 s += a[i];
}
```

Výsledek je správně, ale výpočet je sekveční.

Výpočet je tedy velice pomalý (serializace + režie CS)

Atomická sekce zde příliš nepomůže, jen sníží režii.

OpenMP přece podporuje redukci...

```
s = 0;
#pragma omp parallel for reduction(+:s)
for (i = 0; i < n; i++)
 s += a[i];
```

```
int first = n;
#pragma omp parallel for
for (int i = 0; i < n; i++)
{
 if (a[i] == 0 && i < first)
 {
 #pragma omp critical
 if (i < first) first = i;
 }
}
```

- Hledá se index `first` prvního nulového prvku vektoru.
- `first = n` když neexistuje.

## Proč dva testy `i < first`?

1. Abychom vstupovali do kritické sekce jen když je to nutné, co nejméně krát!
2. Aby bylo jisté, že v mezičase nenalezlo jiné vlákno nulový prvek s nižším indexem

Lépe: `#pragma omp parallel for reduction(min:first)`

# Příklad programu SPMD: fronta úloh

- Sdílená datová struktura obsahuje seznam úloh různé velikosti a složitosti.
- Daná položka (item) může být zpracována libovolným vlákнем, více položek souběžně více vláknů.

```
int get_next_item();
void process_item(int item);

int main()
{
 int item;
 #pragma omp parallel private(item)
 {
 item = get_next_item();
 while (item != -1)
 {
 process_item(item);
 item = get_next_item();
 }
 }
}
```

Fronta obsahuje úlohy označené indexy 0 až MAX-1

```
int get_next_item()
{
 static int head = 0;
 int new_item;
 #pragma omp critical
 {
 if (head == MAX)
 new_item = -1;
 else
 new_item = head++;
 }
 return new_item;
}
```

- **Nepojmenované CS jsou globální.**
  - V jednom okamžiku může běžet kdekoli v programu jen jedna CS.
- **OpenMP dovoluje kritické sekce pojmenovat a tím snížit čekání vláken:**
  - ze stejně pojmenovaných sekcí může v daném okamžiku běžet jen jedna,
  - ale sekce pojmenované jinak mohou běžet souběžně!
- Vnoření **critical** stejného jména = deadlock.
- Vnoření **critical** různého jména: musí být zajištěno stejný pořadí vstupu do kritických sekcí, jinak deadlock.
- **Proto je třeba CS sekci vždy pojmenovat!**

- Direktiva chrání aktualizaci (tj. čtení – modifikaci – zápis, angl. update) **jednoho** paměťového místa.

```
#pragma omp atomic
```

```
 x++; | ++x; | x--; | --x;
```

```
 x binop = expr; | x = x binop expr;
```

- kde *x* je skalár a *binop* je:

**+, \*, -, /, bitwise AND, XOR, OR (&, ^, |), <<, >>**

- Dovětky direktivy atomic byly přidány v OpenMP 3.1:

**[read | write | update | capture] [seq\_cst]**  
default

```
pragma omp atomic read
```

```
v = x;
```

```
pragma omp atomic write
```

```
x = expr;
```

**Žádná část *x* se nemůže změnit dokud R/W není hotové.**

- Dovětek capture umožňuje atomickou aktualizaci se záchytém (přiřazením) hodnoty před nebo po aktualizaci do privátní proměnné (fetch-and-increment).

**#pragma omp atomic [capture]**  
příkaz nebo strukturovaný blok

```
#pragma omp atomic capture
{
 old_value = *p;
 (*p)++;
}
// zde se dá použít old_value
```

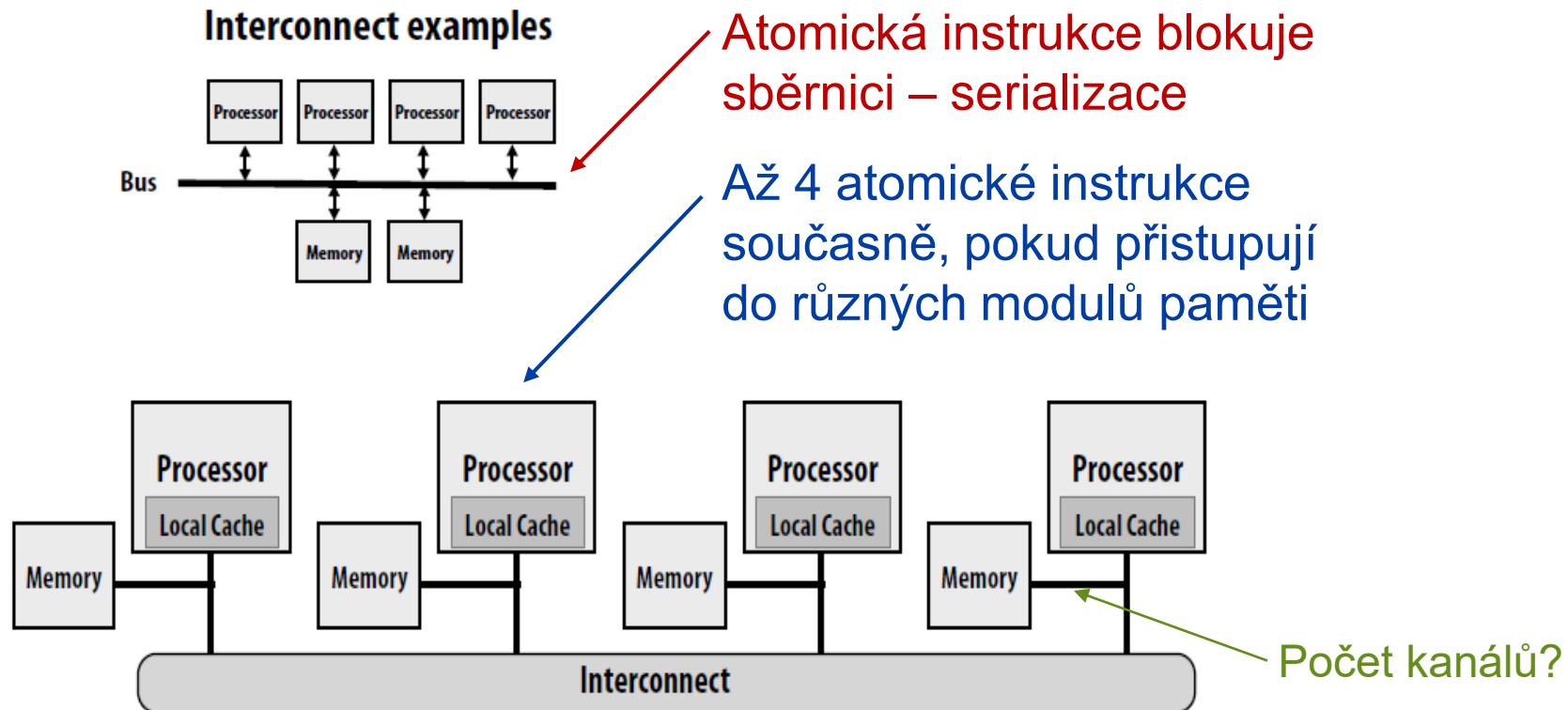
- Dovětek **seq\_cst** = sequential consistency, garantuje proložené provedení operací všech vláken a současně zachování pořadí operací v kódu jednotlivých vláken. Obsahuje implicitní operaci flush (viz dále).

- Vytvoření histogramu z vektoru n čísel a[i]

```
#pragma omp parallel for shared(histogram, a, n)
for (i=0; i < n; i++)
{
 #pragma omp atomic
 histogram[a[i]] += 1;
}
```

- Na rozdíl od **critical** může zpracování atomického bloku využít přídavný paralelismus. Avšak
  - současná aktualizace **různých paměťových míst** může probíhat **jen když atomické instrukce nebudou serializovány** (např. sběrnicí);
  - výkonnost může klesnout kvůli falešnému **sdílení (false sharing)**, kdy je generováno mnoho výpadků (viz dále).

Kolik atomických instrukcí může běžet současně:



```
#pragma omp parallel for \
 reduction(+: s)
for (int i = 0; i < n; i++)
{
 s += a[i];
}
```

```
s = 0;
#pragma omp parallel for
for (int i = 0; i < n; i++)
{
 #pragma omp atomic
 s += a[i];
} /*pomalé, serializované*/
```

```
s = 0;
#pragma omp parallel
{
 int mysum = 0;
 #pragma omp for nowait
 for (int i = 0; i < n; i++)
 {
 mysum += a[i];
 }
 #pragma omp atomic
 s += mysum;
} /*velmi dobré provedení*/
```

- Kritická sekce CS je časově drahá jen když o ni silně soupeří mnoho jader.

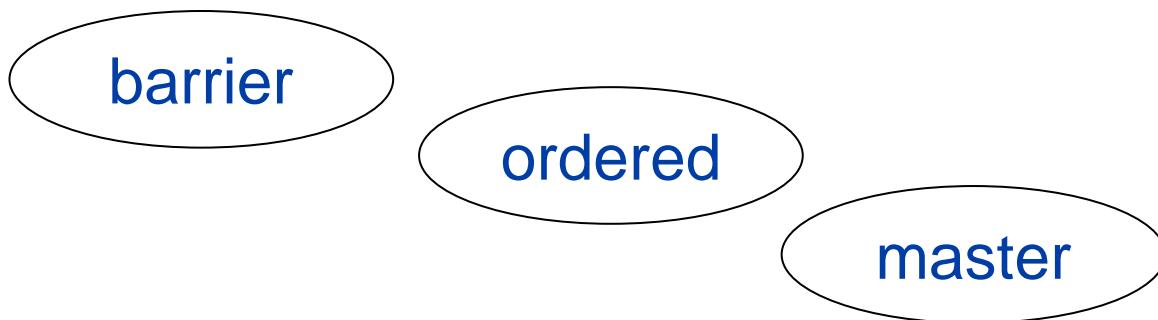
## Snížení režie:

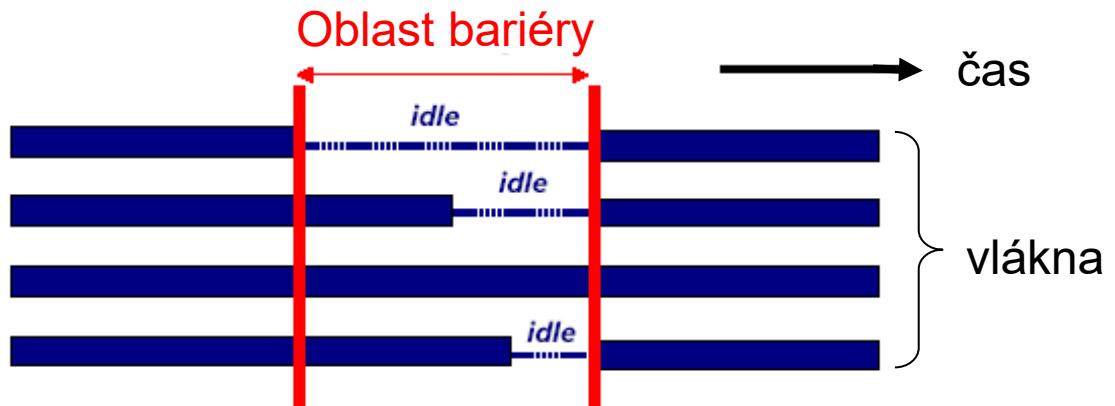
- Kde je to možné, použij direktivu atomic místo critical.
  - Důvod: zápis do jedné sdílené proměnné jsou sice serializovány oběma direktivami, ale režie atomic je mnohem menší – je implementována jen atomickými instrukcemi bez spin zámků.
- Místo jedné CS je často lepší použít mnoho zámků a tak značně snížit soupeření o jeden každý zámek (tzv. vektorový zámek).  
Např. 1 zámek na každou 1/10 intervalu hodnot histogramu.

**OpenMP direktivy kompilátoru**

# **SYNCHRONIZACE UDÁLOSTMI**

- Jedno vlákno (proces) nastaví booleovskou podmínu, další vlákna (procesy) ji testují a čekají na jej nastavení.
- Sem patří bariéra, synchronizace 2 vláken pomocí booleovského návěští (flag), aj.





- Bariéra způsobí, že vlákna co přišla k bariéře čekají až k bariéře dorazí všechna vlákna týmu.
- Vícenásobné bariéry musí procházet všechna vlákna ve stejném pořadí.
- `#pragma omp barrier` má smysl pouze v rozsahu direktivy parallel, může být i sirotek.

- Bariéry nejsou dovoleny v **dynamickém** rozsahu **for, ordered, sections, single, master** a v sekcích **critical**.
- Implicitní bariéry jsou na konci paralelních oblastí **parallel**, paralelních smyček **for**, sekcí **sections**, a skupin tasků **taskgroup** (pokud není potlačeno dovětkem **nowait**)
- Bariéry mají značnou časovou režii. Je třeba je používat obezřetně!

- Oblast **ordered** se provádí v sekvenčním pořadí (serializace, velká režie)
- Užitečné pro sekvenční uspořádání výstupu výpočtu (debug):

```
#pragma omp for ordered schedule(dynamic)
for (i = low; i < high; i += step)
 work(i);
```

dovětek **ordered** avizuje přítomnost  
direktivy **ordered** někde v paralelní oblasti

```
void work(int k)
{
 #pragma omp ordered
 printf(" %d", k);
}
```

direktiva **ordered** (sirotek)

## #pragma omp master strukturovaný blok

- Označuje blok kódu v rámci paralelní oblasti, který je proveden jen hlavním vláknem.
- Ostatní vlákna blok přeskočí a nemusí ani dojít k této direktivě.
- Na rozdíl od single **neobsahuje** implicitní bariéru na konci bloku
- Použití:
  - pro omezení I/O operací jen na hlavní vlákno
  - pro přístup do jeho threadprivate proměnných.
  - generování tasků,...

**OpenMP direktivy a funkce**

# **SYNCHRONIZACE PROGRAMOVANÁ UŽIVATELEM**

|                                  |                                        |
|----------------------------------|----------------------------------------|
| vlákno p1<br>(producent)         | vlákno p2<br>(konzument)               |
| data = ... ;<br><b>flag = 1;</b> | <b>while (flag==0);</b><br>... = data; |

Takto jednoduše komunikovat NELZE !

- Předpokládá provedení zápisů (producent) a čtení (konzument) v pořadí uvedeném v kódu.
- Moderní procesory/kompilátory **mění pořadí jak čtení, tak zápisů** (je to nezbytné pro zlepšení výkonnosti). Případně mohou být paměťové transakce zviditelněny v opačném pořadí vlivem propojovací sítě.
- **Pro obnovení správného pořadí použijte direktivu flush!**

- Paměťová **konzistence** se týká pořadí, ve kterém vidí **různá vlákna** aktualizace **různých** paměťových míst, tj. **kdy** jsou zápisy vidět.
  - **Zápisy** na adresu **x** a pak na **y** mohou dvě vlákna vidět **v jiném pořadí** (jedno vlákno **xy**, druhé **yx** nebo obě **yx**).
- Modely konzistence jsou založeny na uspořádání R, W a S (S = synchronizační instrukce **fence**).
- Základní je **sekvenční konzistence**, SC. Předpokládá, že
  - paměťové operace jsou atomické,
  - pořadí přístupů do paměti specifikované v programu každého vlákna nemění kompilátor ani HW CPU.
- Operace R, W, S jsou v multiprocesoru sekvenčně konzistentní když **globální sekvence** přístupů do paměti
  1. **zachovává pořadí** přístupů každého vlákna
  2. je **proložením** kódů všech vláken.

- **Sekvenční konzistence** je v současnosti nemoderní, je totiž překážkou modernímu hardware a optimalizujícím kompilátorům.
- Moderní procesory vykazují **relaxovanou** (uvolněnou) **paměťovou konzistenci**, kdy se čtení a zápisy mohou vzájemně předbíhat, i když je to proti intuici. Je to kvůli zlepšení výkonnosti. **OpenMP ji podporuje**.
- Modely konzistence paměti jsou často exportovány z procesorů do multiprocesorů, kdy různé CPU mohou vidět paměťové operace v jiném pořadí i vlivem propojovací sítě.
- Programátor musí použít synchronizační příkazy a primitiva (např. fence, **flush**) aby **vymezil oblasti relaxovaného chování**.

```
#pragma omp flush [(list)]
```

- Definuje bod, v němž má vlákno garantováno, že hodnoty
  - všech proměnných viditelných vláknu (když chybí list)
  - nebo jen proměnných v seznamu list
    - jsou konzistentní s hlavní pamětí a tak viditelné všem vláknům.
- Kompilátor může totiž některé proměnné dočasně udržovat v registrech místo v paměti kvůli optimalizaci, HW zase v Load nebo Store buferu.
- Paměťové operace mohou být jen částečně dokončené, jejich výsledky na cestě. Takže některá vlákna mohou vidět dočasně sdílenou paměť jinak než jiná vlákna.
- Flush vynutí v místě svého výskytu shodné (konzistentní) vidění všech vláken.

**Doporučení: na seznamu (list) používat jen jednu proměnnou!**

- Flush se chová se jako **paměťová zábrana (memory fence)** – brání přesunům paměťových přístupů R a W (se všemi proměnnými nebo těmi uvedenými v seznamu list) přes flush:
  - všechny operace R, W před flush se dokončí;
  - nové op. R, W za flush se zahají až po návratu z flush;
  - dvě operace flush s překrývajícími se množinami proměnných v seznamech list nemohou být přehozeny.
- **flush je implicitně přítomen mj.**
  - na vstupu nebo výstupu z paralelních oblastí
  - na implicitních a explicitních bariérách
  - na vstupu/výstupu oblastí critical
  - kdykoli je zámek „set“ nebo „unset“

Jaká přeskládání jsou možná v následujícím kódu:

```
a = . . . ; // (1)
b = . . . ; // (2)
c = . . . ; // (3)
#pragma omp flush(c) // (4)
#pragma omp flush // (5)
. . . a . . . b . . . ; // (6)
. . . c . . . ; // (7)
```

- (1) a (2) se nesmí přesunout za (5),
- (6) se nesmí přesunout před (5),
- (4) a (5) se nemohou vyměnit, (c) a () se překrývají
- (3) se nesmí přesunout za (4),
- (7) se nesmí přesunout před (4), atd.

# Synchronizace dvojice vláken v OpenMP

- OpenMP nemá synchronizační konstrukty pro tento účel
- Když je to potřeba, musí to zařídit programátor sám:

```
int main()
{
 double *A, sum;
 int flag = 0, flg_tmp;
 A = new double[N];

#pragma omp parallel sections
{
 #pragma omp section ← producent
 {
 fill_rand(N, A);
 #pragma omp flush
 #pragma omp atomic write
 flag = 1;

 #pragma omp flush (flag)
 } // end of producent
```

```
#pragma omp section ← konzument
{
 while (true)
 {
 #pragma omp flush(flag)
 #pragma omp atomic read
 flg_tmp = flag;
 if (flg_tmp == 1)
 break;
 }

 #pragma omp flush
 sum = sum_array(N, A);
} // end of consument
} // end of parallel
delete[] A;
} // main
```

# Synchronizace 2 vláken bez flush

- flush není třeba, jelikož seq\_cst jej přidává k atomickým operacím.

```
int main()
{
 double *A, sum;
 int flag = 0, flg_tmp;
 A = new double[N];

 #pragma omp parallel sections
 {
 #pragma omp section ← producent
 {
 fill_rand(N, A);
 #pragma omp atomic write seq_cst
 flag = 1;
 } // end of producent
 } // end of consumer
```

```
#pragma omp section ← konzument
{
 while (true)
 {
 #pragma omp atomic read seq_cst
 flg_tmp = flag;
 if (flg_tmp == 1)
 break;
 }
 sum = sum_array(N, A);
} // end of consumer
} // end of parallel
delete[] A;
} // main
```

- Proměnná zámku umožňuje synchronizovat vlákna
  - je typu `omp_lock_t` a má 64 bitovou adresu.

### Knihovní rutiny:

- `omp_init_lock` - inicializuje (alokuje) zámek.
- `omp_set_lock` - čeká na volný zámek (blokuje) až se dočká, nastaví jej.
- `omp_test_lock` - pokusí se získat zámek, ale nečeká (neblokuje); vrátí úspěch (1), nezdar (0)
- `omp_unset_lock` - uvolní zámek
- `omp_destroy_lock` - dealokace

```
#include <omp.h>
int main()
{
 int id;
 omp_lock_t lck; // vytvoří proměnnou zámku lck
 omp_init_lock(&lck); // inicializace požaduje pointer na lck

#pragma omp parallel shared(lck) private(id)
{
 id = omp_get_thread_num();
 while (!omp_test_lock(&lck)) // 0 = nezdár, neblokuje
 {
 work2(id); // pokud test_lock vrací 0
 work2(id); // dělej něco jiného work2
 }
 work1(id); // volný lock získán a nastaven, dělej work1(id) v CS
 omp_unset_lock(&lck); // lock uvolněn
}
omp_destroy_lock(&lck);
}
```

# | Explicitní zámek místo direktivy critical

```
omp_lock_t maxlock; // vytvoř zámek maxlock
omp_init_lock(&maxlock);
...
#pragma omp parallel for
{
 for(i = 0;i < N; i++)
 {
 ...
 omp_set_lock(&maxlock);
 kritická sekce
 omp_unset_lock(&maxlock);
 }
 omp_destroy_lock(&maxlock);
}
```

blokuje, (čeká na volný maxlock)

kdo získal maxlock,  
ten jej uvolní

- Kdykoliv se vlákno pokouší získat exkluzivní přístup ke dvěma nebo více sdíleným prostředkům, může vzniknout deadlock.
- **Deadlock vznikne pokud platí když platí 4 podmínky:**
  1. Přístup ke každému prostředku je exkluzivní.
  2. Vláknu je dovoleno užívat jeden prostředek a zároveň žádat jiný.
  3. Žádné vlákno není ochotno se vzdát prostředku, který získalo.
  4. Existuje cyklická posloupnost vláken snažících se získat prostředky, přičemž každý prostředek je užíván jedním vláknem a žádán jiným.

Deadlocku se lze vyhnout negací jedné z těchto podmínek.

| direktiva / dovětek | režie [us] |
|---------------------|------------|
| parallel            | 1,5        |
| barrier             | 1,0        |
| schedule static     | 1,0        |
| schedule guided     | 6,0        |
| schedule dynamic    | 50 *)      |
| ordered             | 0,5        |
| single              | 1,0        |
| reduction           | 2,5        |
| atomic              | 0,5        |
| critical            | 0,5        |
| lock/unlock         | 0,5        |

Intel Xeon quad-core,  
3 GHz, kompilátor Intel

\*) default chunk size = 1;  
pro chunk size = 16 jen  
5  $\mu$ s .

S počtem vláken roste  
režie direktiv parallel  
a barrier lineárně.

# Pokračování příště

# Multiprocesory se sdílenou pamětí: Koherence pamětí cache

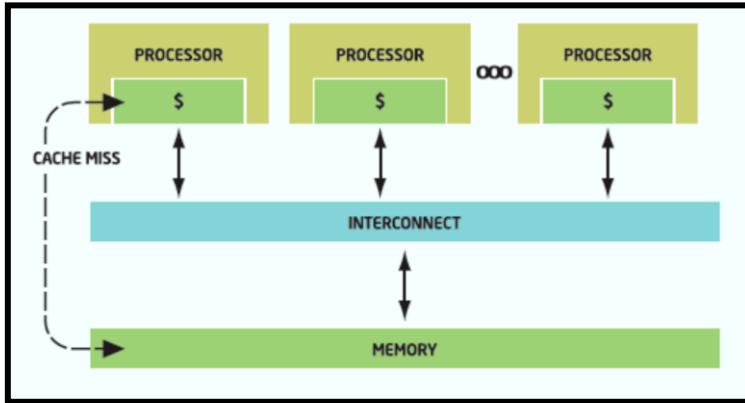
## AVS – Architektury výpočetních systémů

### Týden 10, 2023/2024

**Jirka Jaroš**

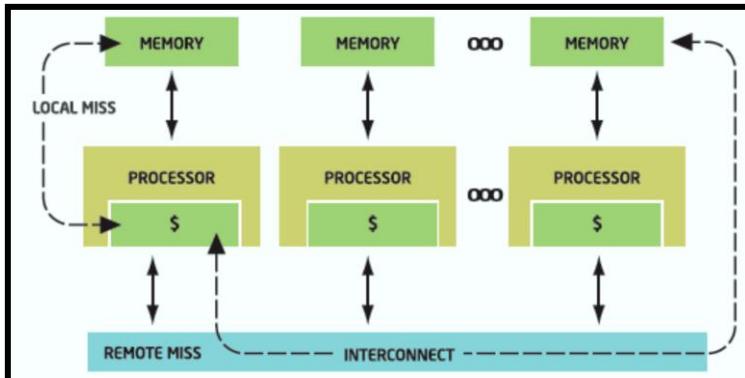
Vysoké učení technické v Brně, Fakulta informačních technologií  
Božetěchova 1/2, 612 66 Brno - Královo Pole  
[jarosjir@fit.vutbr.cz](mailto:jarosjir@fit.vutbr.cz)





## SAS – UMA (CMP)

- uniformní doba přístupu do hlavní paměti. Implementace:
  - výkonné sběrnice: souběžné přenosy
  - X-bar: paralelní přenosy

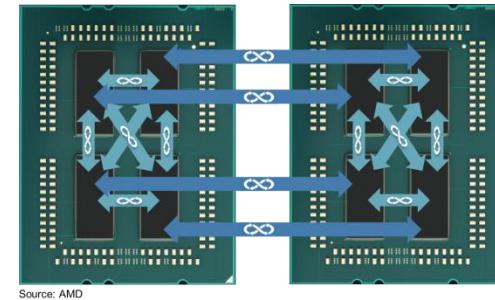


## SAS – NUMA (DSM)

- neuniformní doba přístupu do hlavní paměti (např. lokální a vzdálený výpadek v cache).
  - Vzdálený přístup: automatické zaslání zprávy tam a zpět

## 1. Čipové multiprocesory CMP

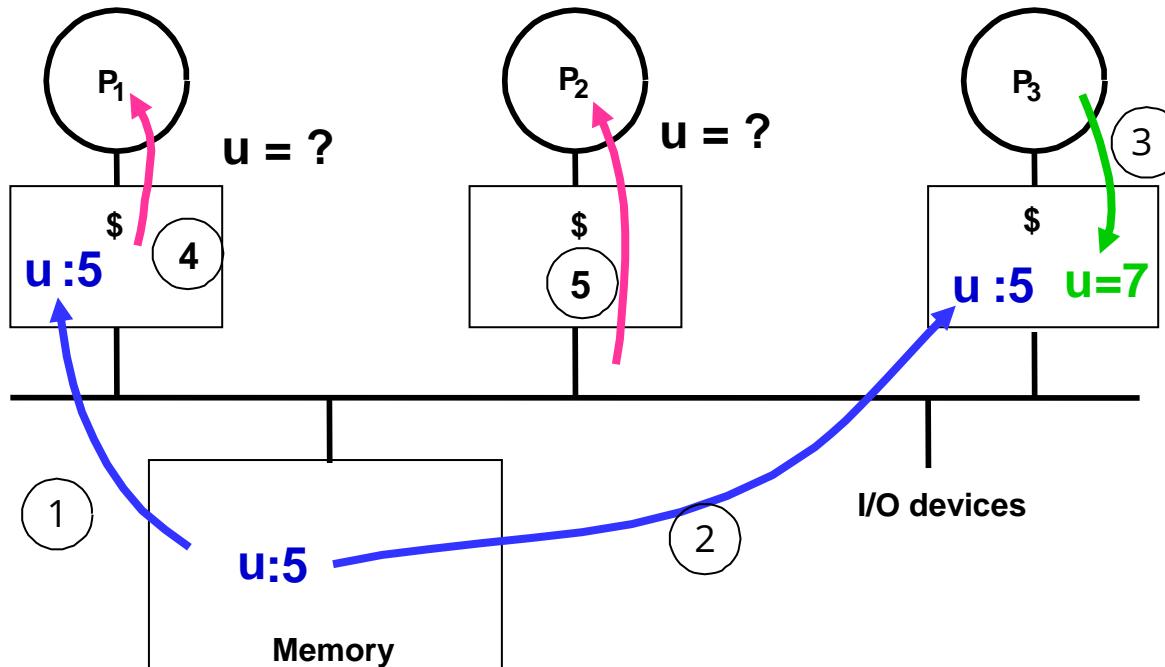
- centrální sdílená paměť (UMA)
- s plným propojením (2–8 jader nebo procesorů)
- se sběrnicí (do 16 jader)
- s křížovým přepínačem (8 a 16 jader Sparc T2, T3)
- s kruhovým propojením (lepší než sběrnice, levnější než X-bar).  
8–16 jádrový CPU Sandy Bridge, Haswell (Intel)
- s propojovací sítí – dnešní procesory Skylake X (2D mřížka), EPYC (Infinity path)



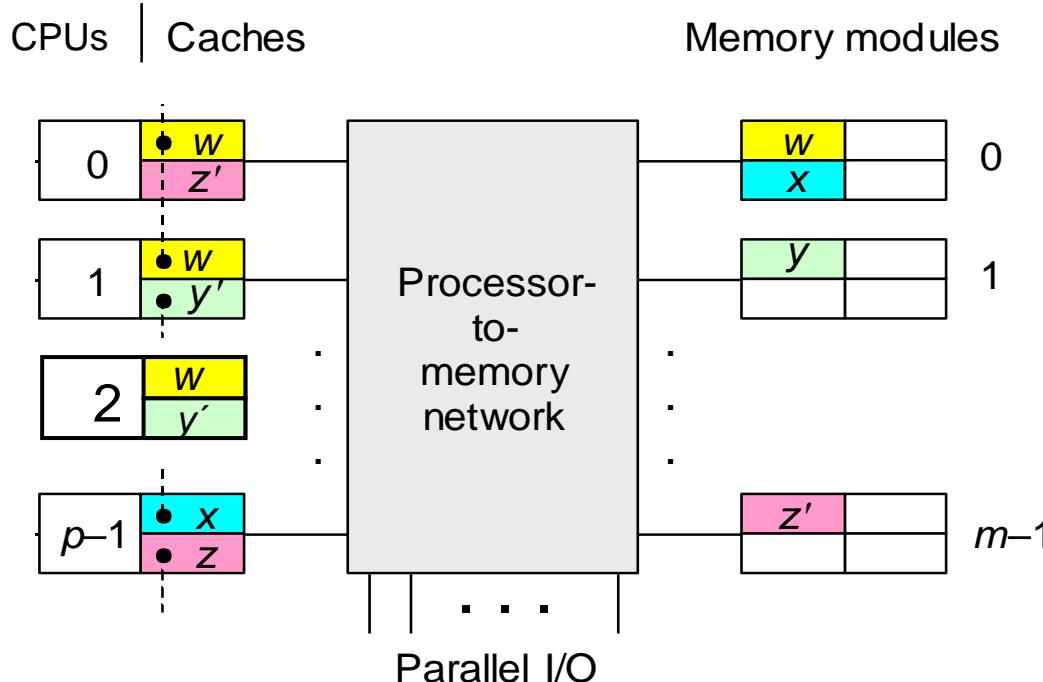
## 2. Distribuovaná sdílená paměť DSM

- sdílená paměť fyzicky distribuovaná → NUMA
- propojovací síť, např. 2D-torus, tlustý strom, hyperkostka
- někdy oddělená adresová a datová síť
- rozšiřitelnost až do 2048 jader a 64 terabajtů (TB)

- Jsou většinou **symetrické multiprocesory (SMP)**, tj.
  - jsou složeny ze shodných CPU (jader), která
  - sdílí hlavní paměť (SAS), buď UMA nebo NUMA
  - jsou řízeny rovným způsobem jednou instancí OS.
- Jakmile do **sdílené paměti** může přistupovat více vláken, je nutné zajistit, aby **na jedné adrese** našla při čtení vždy **stejná data**, ať jsou data v kterékoliv úrovni cache nebo v hlavní paměti (**tj. koherentní kopie**).
- **Proto je kritická**
  - aktualizace (zápis) dat na jednu adresu 1 vláknem (správný postup je dán **protokolem koherence cache**)
  - aktualizace **dat na různých adresách** více vlákny (pořadí je určeno **modelem paměťové konzistence**).



Koherence pamětí cache znamená, že pro danou adresu existuje jediná (koherentní) verze sdílených dat v jedné, několika nebo i ve všech pamětech cache. Na kopii v paměti nezáleží, ta může být zastaralá (neplatná).



Možné stavy bloku v cache:

- čistý/špinavý
- jediný/sdílený
- platný/neplatný

Které kopie (s čárkou je nejnovější) se nesmí v cache vyskytovat?

x, w = čistá kopie, y' = špinavá, z = nekoherentní

- **Sekvenční počítač:**
  - vidí vlastní čtení a zápisy v pořadí, jak je vydává
  - při čtení vidí **poslední hodnotu** zapsanou na danou adresu.
  - u paralelního počítače ale různé CPU nemusí vidět paměťové akce ve stejném pořadí.
- **U paralelního počítače** musíme zajistit
  - **šíření zápisů** – každý zápis se stane viditelný všem CPU (tj. jádrům všech procesorů)
  - **serializaci zápisů** – všechny CPU vidí **zápis** na adresu **x** v **témže pořadí** (u sběrnice splněno triviálně).
- **Koherence** se týká jednoznačnosti zápisů **na 1 adresu**
- **Konzistence** se týká pořadí přístupů **na různé adresy**.

Při výpadku čtení nebo zápisu v místní cache vyšle jádro (vlákno) žádost s adresou bloku přes rozhraní do sdílené paměti (cache) vyšší úrovně.

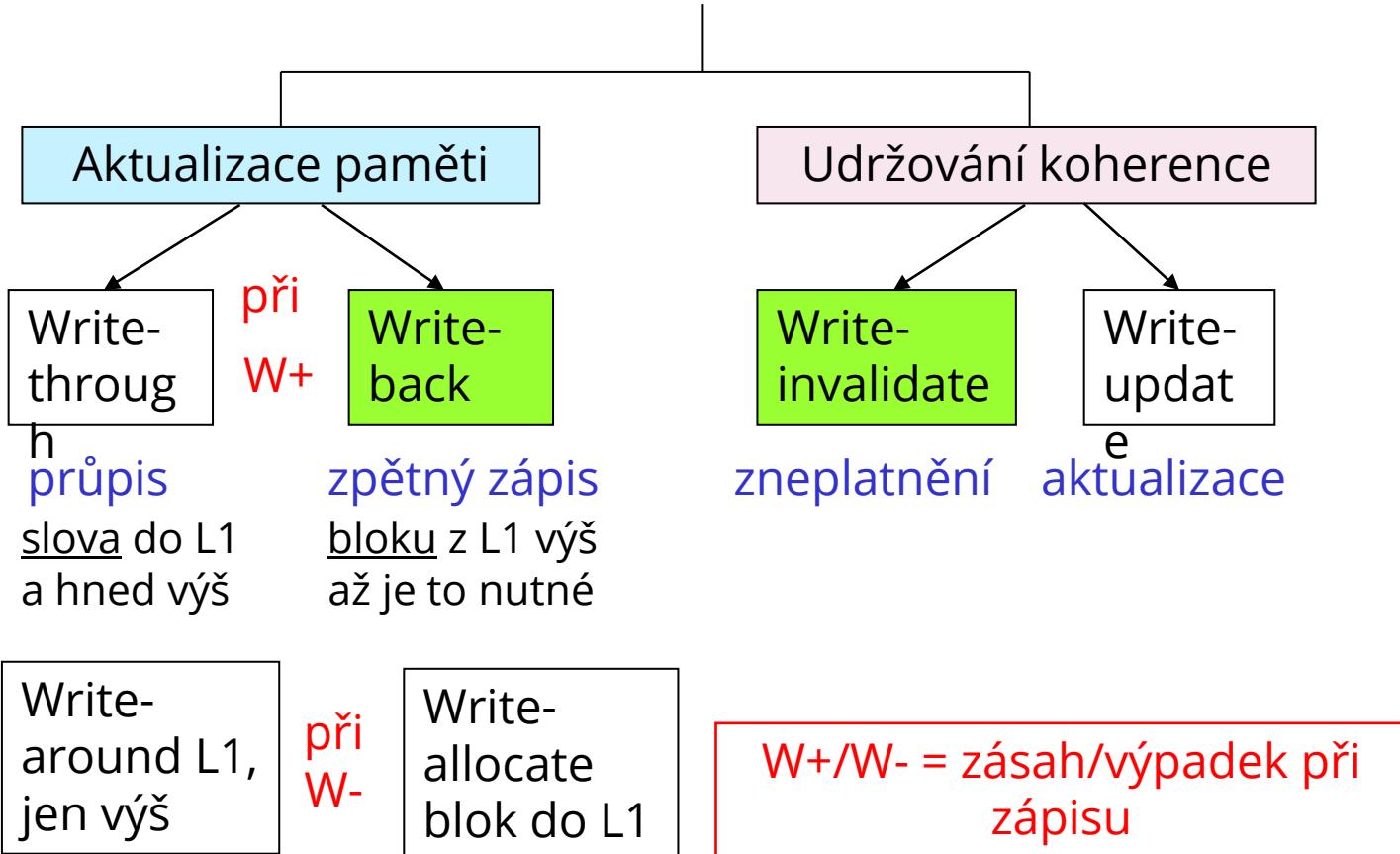
## Protokoly CC využívají:

- **Monitorování komunikace** (snooping, naslouchání).  
Stav bloku je připojen ke všem jeho kopiím v místních cache. Žádost s adresou bloku jde od žadatele **všem (broadcast)**, reagují a odpovídají jen ti, kterých se to týká.
- **Distribuované adresáře**  
Stavy bloku paměti v jednotlivých cache jsou uloženy ještě odděleně v některém z adresářů. Žádost jde od žadatele přes domovský adresář bloku **jen majitelům kopií**, jen ti také odpovídají.

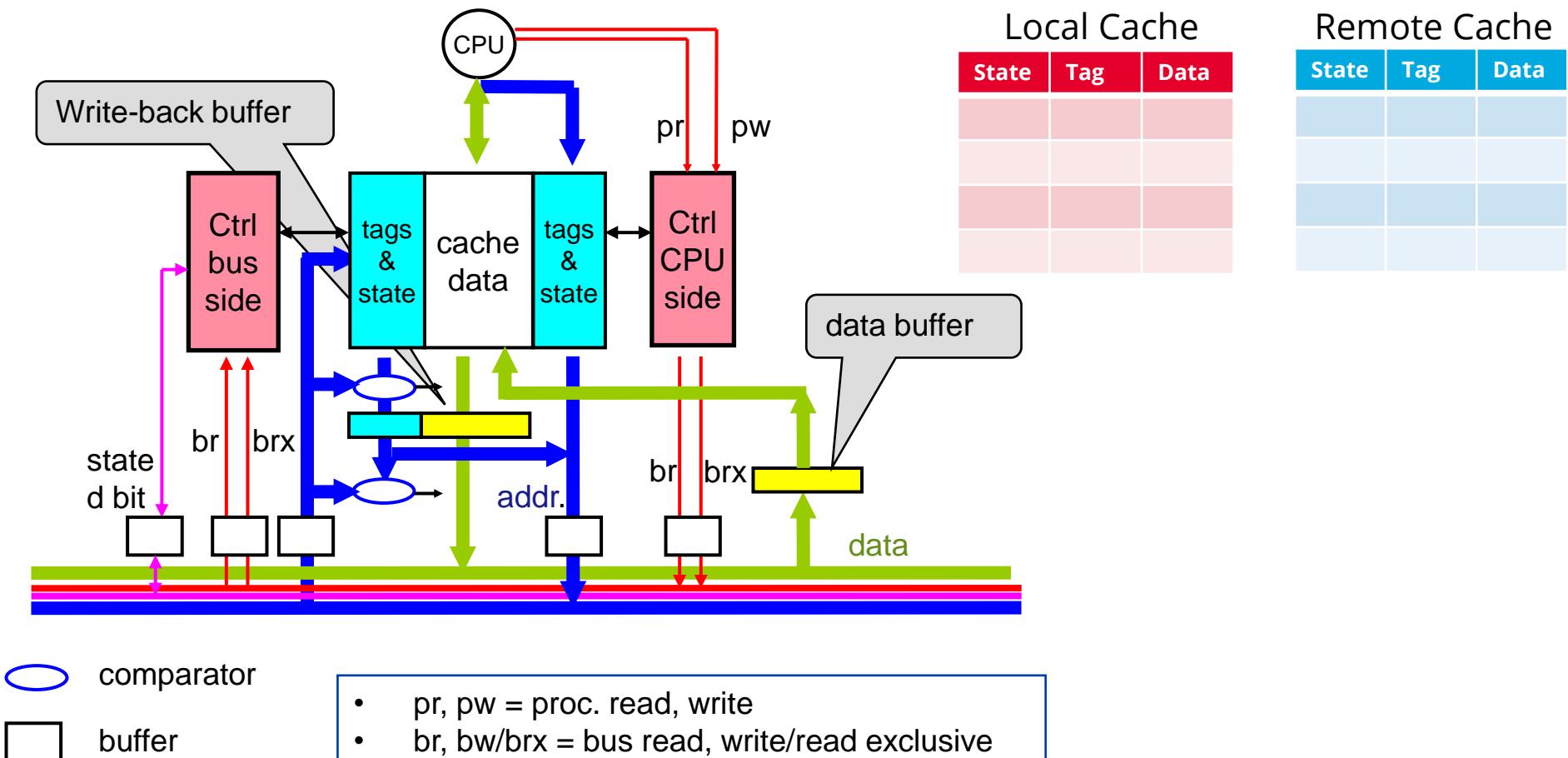
- Pro sběrnicové systémy jen monitorování komunikace.
- **NUMA systémy** používají většinou distribuované adresáře; pro případ naslouchání je adresová a datová síť oddělená.
- Mnoho implementačních záležitostí:
  - více úrovní cache (L1, L2, L3, ...),
  - sdílená paměť (SM) na úrovni L2 nebo vyšší,
  - oddělené I-cache a D-cache,
  - velikost bloku (řádku) cache,
  - způsob aktualizace dat v hlavní paměti,
  - atd., atd.

CC protokol je *distribuovaný algoritmus implementovaný souborem kooperujících řadičů cache (CacheCtrl) – konečných automatů*.

CC Je popsán diagramem stavových přechodů bloků cache.



# Řadič (Ctrl) paměti cache s nasloucháním

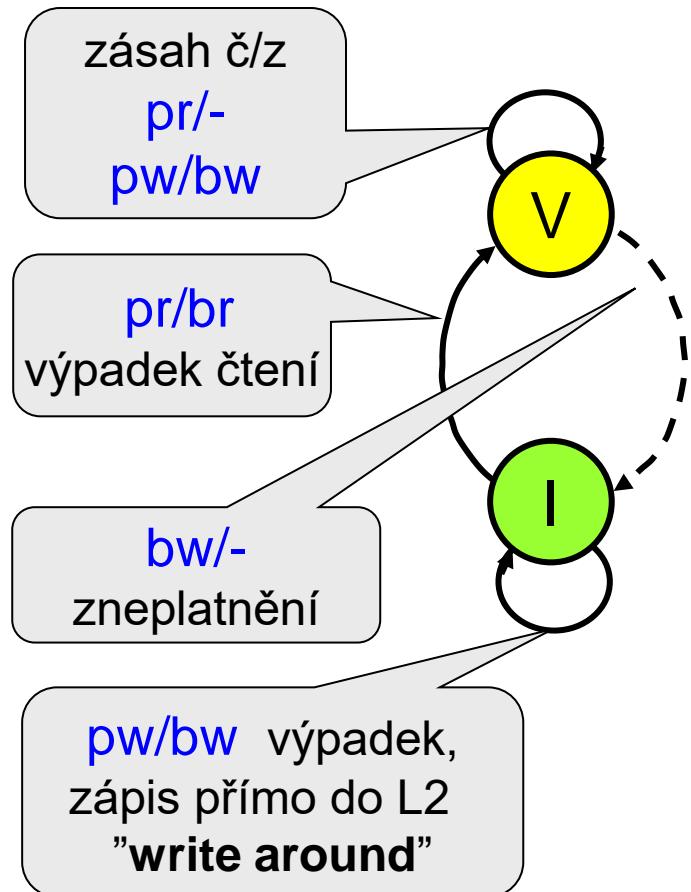


## Aktualizace paměti

## Udržování koherence

$$\left\{ \begin{array}{l} \text{write-through} \\ \text{write-back} \end{array} \right\} \times \left\{ \begin{array}{l} \text{write-update} \\ \text{write-invalidate} \end{array} \right\}$$

- **write-through, write-invalidate:** 2 stavy
- **write-back, write-invalidate:**
  - MSI (3 stavy),
  - MESI (4 stav) – Intel Pentium D
  - MOESI (5 stavů) – AMD Opteron
  - MESIF (5 stavů) – Intel Nehalem a vyšší
- **write-back, write update:** Dragon (4 stavy)



Aktualizace  
„write-through“,  
zapisuje slovo,

Koherence  
„write-invalidate“  
zneplatní blok

### Legenda:

(vstup/výstup) řadiče cache

- pr, pw = proc. read, write
- br, bw = bus read, write
- blok v cache označen jedním bitem "in/valid"

**použití:** CC mezi privátními cache L1 a sdílenou L2 ve vícejádrových procesorech.  
**Nevýhoda:** velké množství výpadků

Aby se nemusely opakované zápisy do bloku cache neustále propisovat výš do SM, je možné špinavou kopii v L1 cache označit dalším stavem M a nechat ji v L1 co nejdéle (dokud nebude blok vybrán k výměně za jiný blok).

### Modifikovaný M:

jen 1 cache má platnou špinavou kopii, kopie ve sdílené paměti SM je zastaralá  
dirty bit = 1

### Sdílený S:

1 nebo více pamětí cache má kopii shodnou se sdílenou pamětí SM (čistá kopie)

### Neplatný I:

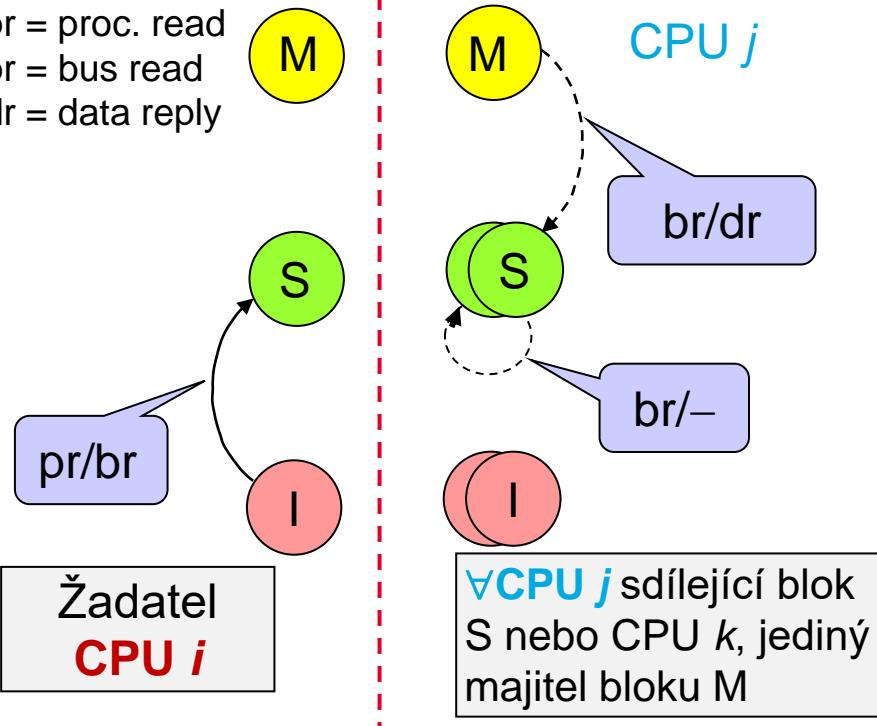
v této cache je pouze neplatná kopie (zastaralá)

**Blok S lze jen čist.** Před zápisem musí CPU změnit jeho stav na M a ostatní kopie S musí být zneplatněny; CPU je pak jediný **vlastník bloku M**, má k němu exkluzivní přístup a **může** do něj **zapisovat**.

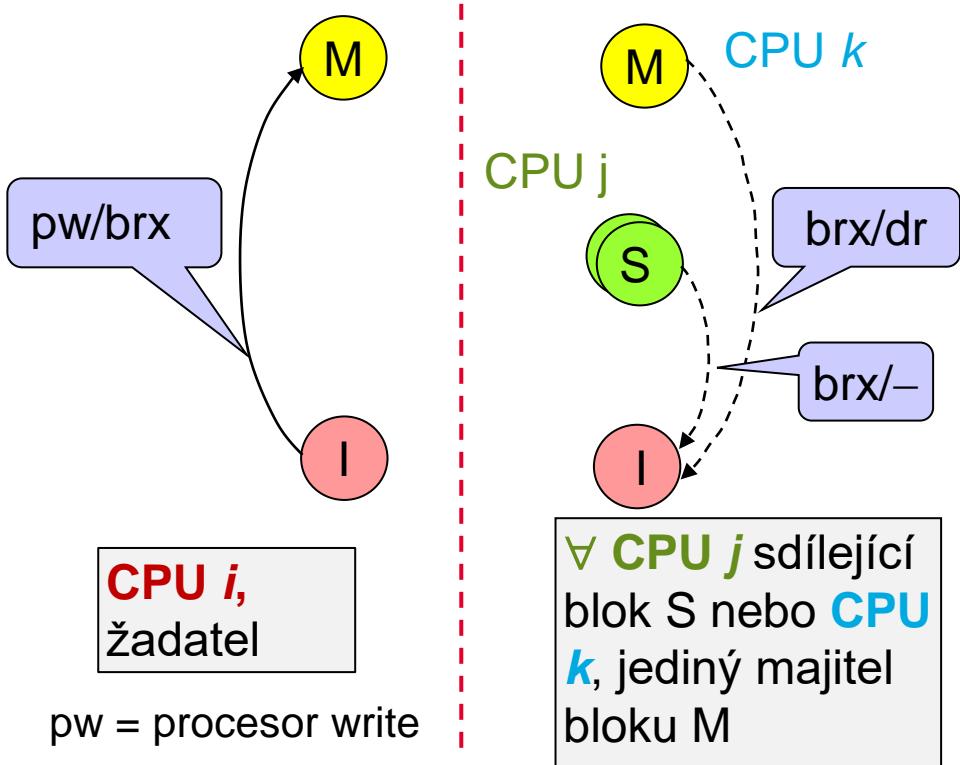
- Stav bloku dat se může lišit v různých cache. Změny stavů bloků cache (2 byty valid a dirty) provádějí řadiče cache (CacheCtrl), které reagují na lokální nebo vzdálené výpadky/zásahy v cache.
- CC protokol je *distribuovaný algoritmus implementovaný souborem kooperujících řadičů cache (CacheCtrl) – konečných automatů*.  
Je popsán diagramem stavových přechodů bloků cache.  
Značení: ([vstup/výstup](#)) řadiče cache.
- **Dále předpokládáme:**
  1. pouze 1 úroveň cache L1 a sdílená paměť RAM;
  2. CC protokol: write-back, write-invalidate;
  3. atomickou sběrnici, tj. pouze 1 rozpracovanou transakci.

1. **Žádost o sběrnici, arbitráž, přidělení sběrnice.** Arbitr sběrnice vybere jednu ze žádostí a potvrdí ji.
2. **Žádost s adresou** bloku dá vybraný žadatel na sběrnici. Ostatní CPU naslouchají sběrnici, řadiče cache vyhodnocují situaci a reagují na ni změnou nebo hlášením stavu bloku, případně odesláním dat (majitel bloku **M**).
3. **CPU, která má dostat data**, má žádaný blok buď ve stavu I nebo blok není v cache přítomen vůbec. Pak musí vybrat **blok pro výměnu (victim = oběť)** a pokud je to blok **M**, musí provést jeho **zpětný zápis (write back, wb)** do sdílené paměti. Dokud CPU není připravena, nastaví **wired-OR wait** signál.
4. **Odesílatel hlásí signálem dr (data reply / response)** přítomnost dat na sběrnici. Příjemce dat jejich dodání nepotvrzuje, zapíše blok do cache s aktualizovaným stavem.

pr = proc. read  
br = bus read  
dr = data reply



**Závěr:** data dá na sběrnici paměť (S) nebo majitel (M). Pokud byl blok ve stavu (M), putují data i do paměti.

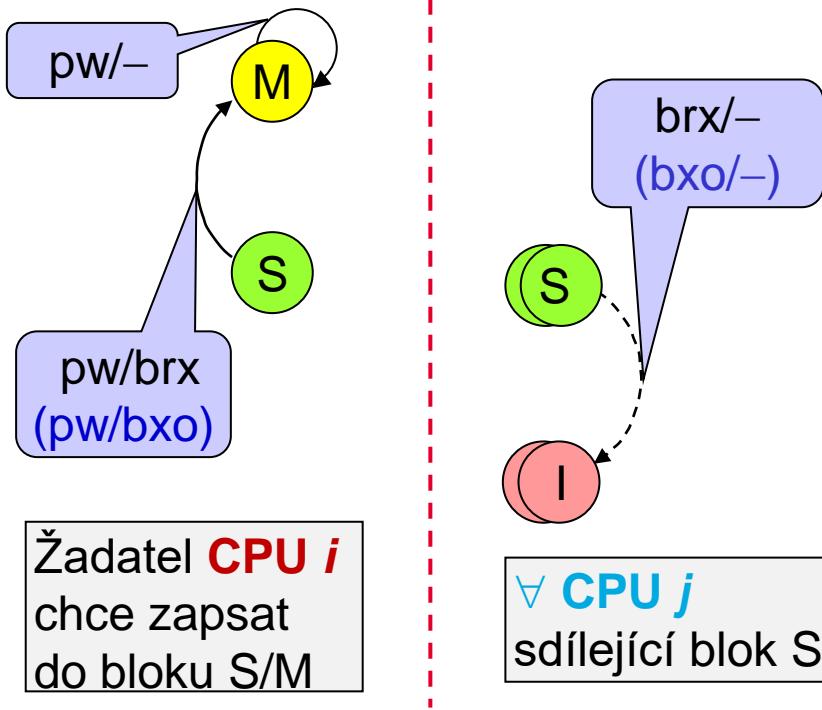


- pw → CacheCtrl i → W- :**
1. žádost o sběrnici;
  2. když přidělena: **brx (bus read exclusive)** na sběrnici;
  3. když victim dirty: **wb**;
  4. když data přijata: **I → M**;

- CacheCtrl j, k:**
1. **brx**: když stav = M: dr M;
  2. změna stavu S, nebo M → I;

**MemCtrl:**  
**brx**: když dirty bit = 0:  
dr S, S → I; kopie neplatná

**Závěr:** před zápisem se musí data zneplatnit u předchozího vlastníka (M) nebo všech vlastníků (S). Data dá na sběrnici paměť (S) nebo vlastník (M).



pw → CacheCtrl *i* → W+:

1. když stav = M: **exit**;
2. žádost o sběrnici;
3. když přidělena: **brx**  
("bus read exclusive");
4. stav S → M;

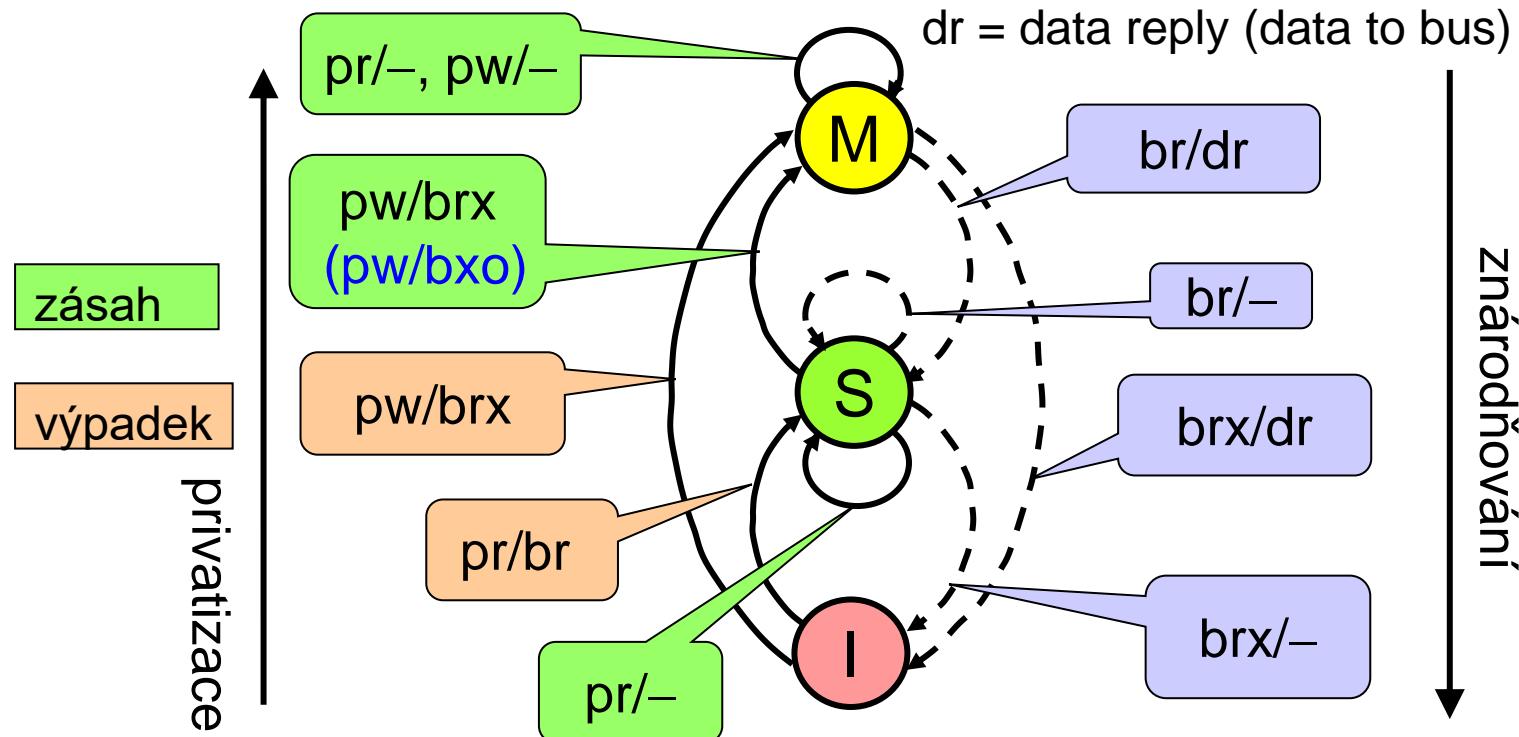
CacheCtrl *j*:

brx: stav S → I

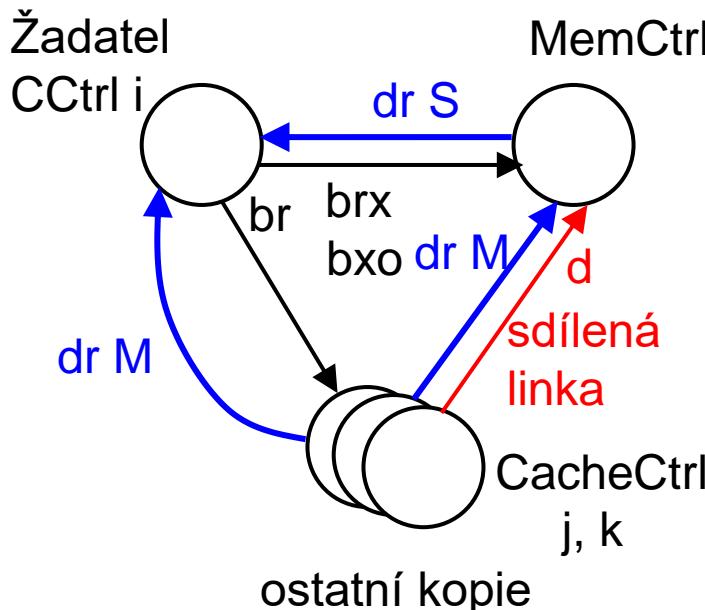
**Závěr:** před zápisem se musí data zneplatnit u všech vlastníků (S). Příkaz **b xo** místo **brx** vyžádá jen invalidaci bloku (S) u ostatních cache a blokuje načítání z paměti.

# I Celkový stavový diagram protokolu MSI

Tlusté přechody: žadatel, čárkované přechody: ostatní cache



$bxo$  = exclusive owner, variantní transakce bez přenosu dat



- Na sběrnici existují **sdílené linky**, na kterých může CPU nastavit signál nebo podmínu viditelný/nou všem CPU a též řadiči MemCtrl.

- Sdílený wired-or signál **dirty bit d** na sběrnici je interpretován paměťovým řadičem takto:

br/brx a **d = 1**: probíhá přenos bloku **dr M** z cache do cache  
a při br i do paměti, kde bude označen S

br/brx a **d = 0**: přenos bloku **dr S** z paměti do cache žadatele

b xo a **d = 0**: neproběhne žádný přenos dat.

# Příklad 1. CC protokol MSI

3 procesorový systém se společnou sběrnicí, sdílenou pamětí a koherentními pamětmi cache L1 (protokol MSI) vykonává operace se sdílenou proměnnou u dle tabulky. Vyplňte tabulku pomocí symbolů M, S, I, br, brx, SM, C1, C2, C3, - :

|     | Akce procesoru | Signály na sběrnici | Stav bloku v C1 | Stav bloku v C2 | Stav bloku v C3 | Data dodá | Data přijme |
|-----|----------------|---------------------|-----------------|-----------------|-----------------|-----------|-------------|
| 0.  | -              | -                   | M               | I               | I               | -         | -           |
| 1.  | P3 čte u       |                     |                 |                 |                 |           |             |
| 2.  | P3 píše do u   |                     |                 |                 |                 |           |             |
| 3.  | P1 píše do u   |                     |                 |                 |                 |           |             |
| 4.  | P2 čte u       |                     |                 |                 |                 |           |             |
| 5.  | P3 píše do u   |                     |                 |                 |                 |           |             |
| 6a. | P3 čte u       |                     |                 |                 |                 |           |             |
| 6b  | P3 čte u       |                     |                 |                 |                 |           |             |

# **POKROČILÉ PROTOKOLY CC S NASLOUCHÁNÍM**

- Další stav **E** bloku cache značí, že existuje čistá kopie pouze v jediné cache. Stačí 2 bitový kód jako pro MSI.
- **Výhody:**
  - Ve stavu **E** (i **M**) je možný zápis ( $E \rightarrow M$ ,  $M \rightarrow M$ ) **bez oznamování** brx nebo bxo na **sběrnici**
  - Blok **E** dodá cache dalšímu žadateli ( $E \rightarrow S$ ) rychleji místo Mem!
- Na sběrnici jsou ale teď místo signálu **dirty bit d** nově dva wired-OR signály:
  - owner bit **e** (exkluzivní vlastník, signalizuje **M** i **E**) a
  - shared bit **f** (kopie jen v paměti a v žádné cache:  $f = \text{false}$ ; existuje platná kopie alespoň v jedné cache:  $f = \text{true}$ ).

- Kolektivní cíl všech cache je minimalizovat přístupy do sdílené hlavní paměti mimo čip, využít data na čipu.
- Nový stav **F**, **read-only forwarding**, umožňuje přenos čisté kopie z jedné cache do druhé. **Je to rychlejší (např. u vícejádrových procesorů) než přenos z paměti.**
- Ve sdílených kopiích je vždy **jen** jedna ve stavu F a ta se kopíruje. Ostatní kopie jsou ve stavu S.
- Když je blok ve stavu F kopírován, stav F migruje do nové kopie zatímco zdrojová kopie přejde do stavu S.
- Blok M se musí před jeho sdílením nebo výměnou (kvůli místu) nahrázit zpět do paměti.

- MOESI má jinou optimalizaci:
  1. **eliminuje kopírování bloku M do paměti před jeho sdílením;** původní blok M přejde do stavu O (owned),
  2. **další sdílené špinavé kopie S** vznikají kopírováním dat z bloku O. Kopie v paměti je zastaralá.
- Stavy E a O a M jsou unikátní (blok jen v jedné cache)
- Sdílené kopie jsou buď **špinavé**  $M \rightarrow [O, S, S, S, \dots]$  nebo čisté  $E \rightarrow [S, S, S, \dots]$
- Stavy M, O, E dovolují zápis (O a E přejdou do M) a nahrazují paměť jako zdroj dat (přenos cache-to-cache).

# I Shrnutí vlastností bloků cache

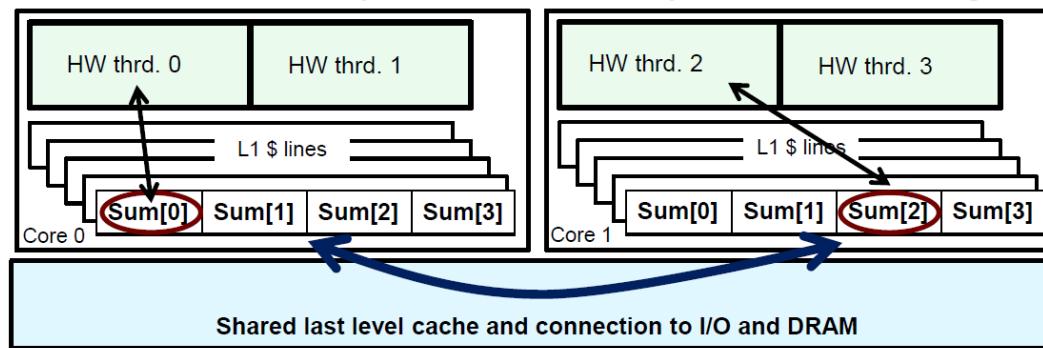
|          | čistý/<br>špinavý | unikátní | ostatní<br>cache | povolení | zdroj dat |
|----------|-------------------|----------|------------------|----------|-----------|
| <b>S</b> | čistý             | ne       | S nebo I         | R        | ne        |
| <b>M</b> | špinavý           | ano      | I                | R/W      | ano       |
| <b>E</b> | čistý             | ano      | I                | R/W      | ano       |
| <b>F</b> | čistý             | ano      | S nebo I         | R        | ano       |
| <b>O</b> | špinavý           | ano      | S nebo I         | R/W      | ano       |
| <b>I</b> | —                 | —        | —                | žádné    | ne        |

# FALEŠNÉ SDÍLENÍ

- Různá data (prvky vektoru) modifikovaná opakovaně dvěma vlákny by neměla být v jednom bloku cache!

```
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < size(); i++)
 a[i]++;
```

- Při zápisu jedním jádrem bude totiž pokaždé kopie bloku druhému jádru zneplatněna.
- Náprava: **zvýšením dimenze pole** (1 prvek na celý blok, *padding*) nebo **privatizace**.



```

float a[m][n], s[m]; // s = vektor řádkových součtů
void rowsum1(float* a[], float* s, int m, int n)
{
 int i, j;
 #pragma omp parallel for private(i, j) shared(s, a)
 for (i = 0; i < m; i++)
 {
 s[i] = 0.0f;
 for (j = 0; j < n; j++)
 s[i] += a[i][j];
 }
}

```

**P = 4:** 15.06 s  
**P = 2:** 6.08 s  
**P = 1:** 3.77 s

Dvě vlákna mohou aktualizovat prvky s[i] nacházející se v 1 bloku cache.  
V bloku 64 B bude 16 prvků s[i] s offsetem 0, 4, 8, ..., 60.

```
float a[m][n], s[m][C]; // C * sizeof (float) = velikost bloku cache [byte]
void rowsum2(float* a[], float* s[], int m, int n)
{
 #pragma omp parallel for shared(s, a)
 for (int i = 0; i < m; i++)
 {
 s[i][0] = 0.0f;
 for (int j = 0; j < n; j++)
 s[i][0] = s[i][0] + a[i][j];
 }
}
```

P = 4: 1.03 s  
P = 2: 2.04 s  
P = 1: 3.76 s

Např. pro blok cache 64 B je C = 16 →  
float s[i][0] bude vždy na začátku bloku  
cache, zbytek bloku (15 float) nebude využit.

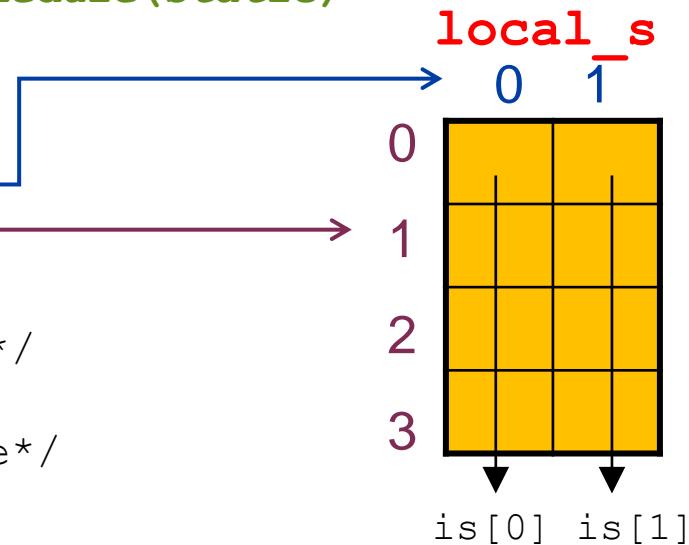
```

constexpr int MAX_NUM_THREADS = 4;
int local_s[MAX_NUM_THREADS][2], is[2];

#pragma omp parallel shared(local_s, is)
{
 int my_id = omp_get_thread_num();
 #pragma omp for private(i, index) schedule(static)
 for (i = 0; i < N; i++) {
 index = ia[i] % 2; // zbytek
 local_s[my_id][index]++;
 }
}

#pragma omp atomic
is[0] += local_s[my_id][0]; /*sude*/
#pragma omp atomic
is[1] += local_s[my_id][1]; /*liche*/
}

```



```
constexpr int MAX_NUM_THREADS = 4;
int local_s[2], is[2];
#pragma omp parallel private (local_s) shared (is)
{
 local_s[0] = 0;
 local_s[1] = 0;

#pragma omp for private(index) schedule(static)
for (i = 0; i < N; i++) {
 index = ia[i] % 2;
 local_s[index]++;
}

#pragma omp atomic
 is[0] += local_s[0];
#pragma omp atomic
 is[1] += local_s[1];
}
```

modifikace lokálních kopií  
dat v privátním zásobníku

redukce lokálních dat na  
sdílené pole is se 2 prvky

# Pokračování příště

# NUMA systémy se sdílenou pamětí

## AVS – Architektury výpočetních systémů

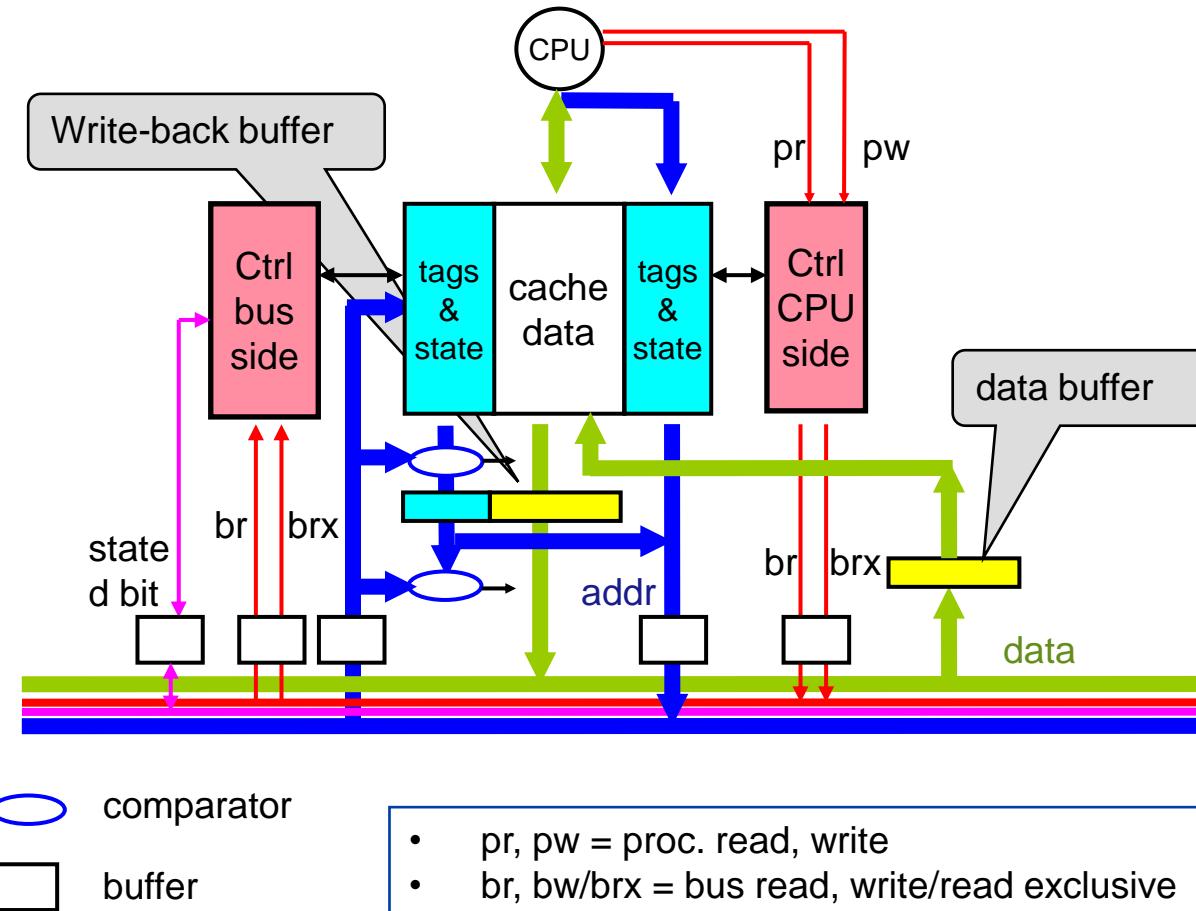
### Týden 11, 2024/2025

**Jirka Jaroš**

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

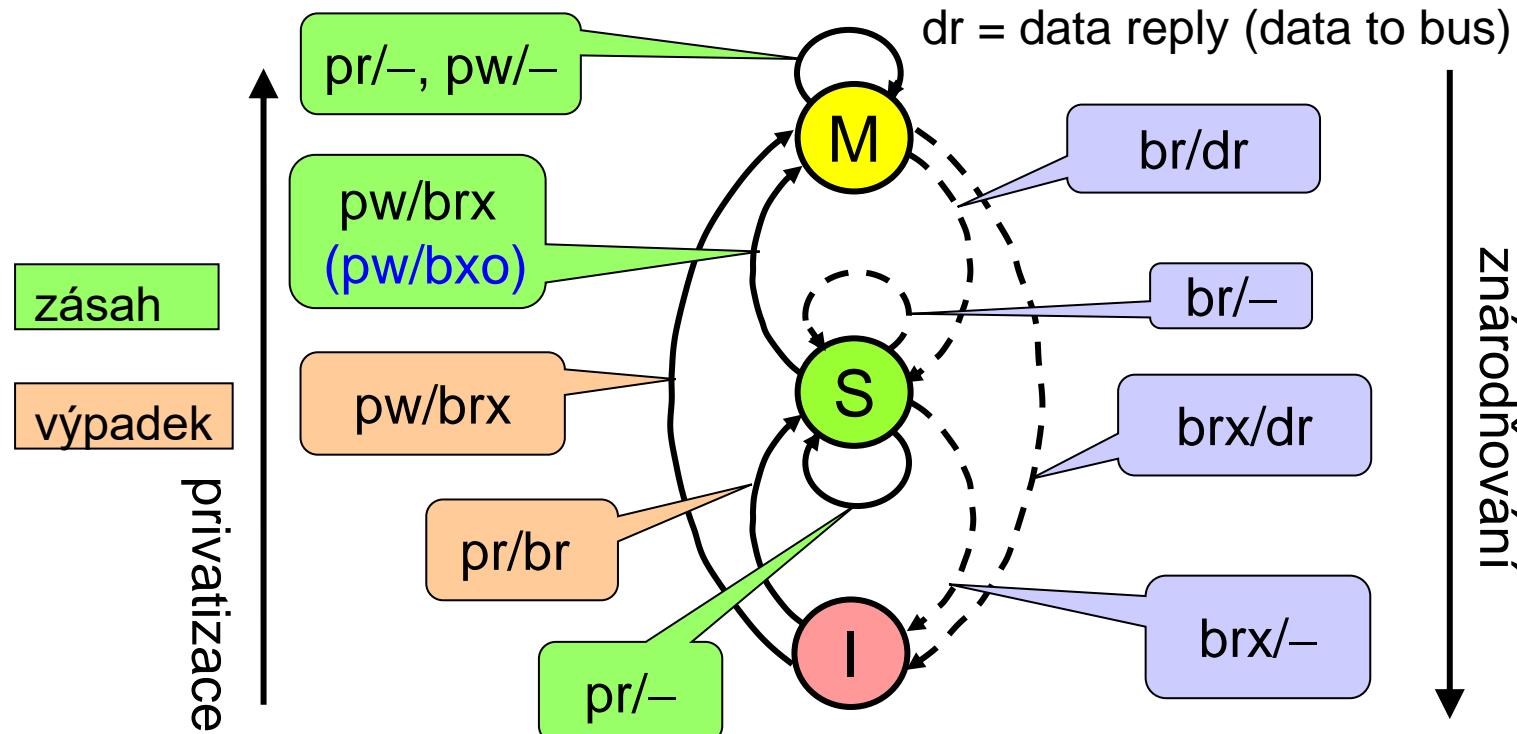


# Řadič (Ctrl) paměti cache s nasloucháním



# Celkový stavový diagram protokolu MSI

Tlusté přechody: žadatel, čárkované přechody: ostatní cache



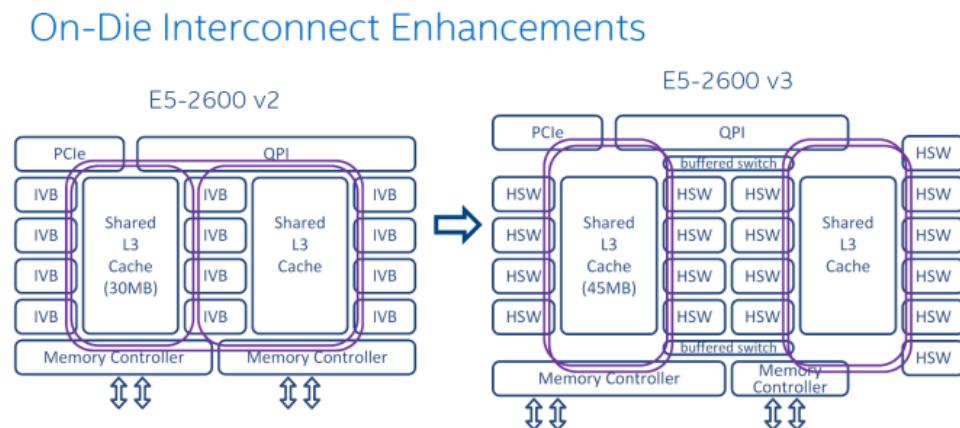
# Příklad 1. CC protokol MSI

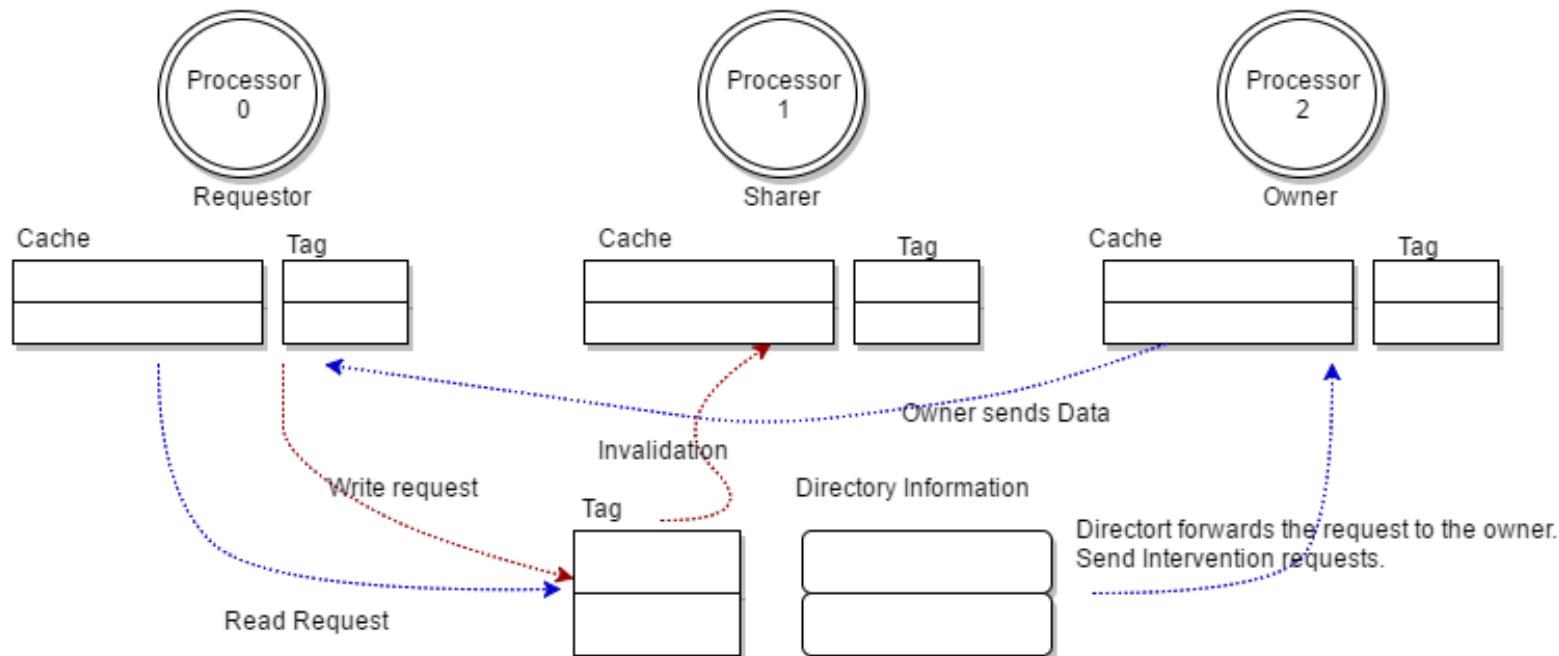
3 procesorový systém se společnou sběrnicí, sdílenou pamětí a koherentními pamětmi cache L1 (protokol MSI) vykonává operace se sdílenou proměnnou u dle tabulky. Vyplňte tabulku pomocí symbolů M, S, I, br, brx, SM, C1, C2, C3, - :

|     | Akce procesoru | Signály na sběrnici | Stav bloku v C1 | Stav bloku v C2 | Stav bloku v C3 | Data dodá | Data přijme |
|-----|----------------|---------------------|-----------------|-----------------|-----------------|-----------|-------------|
| 0.  | -              | -                   | M               | I               | I               | -         | -           |
| 1.  | P3 čte u       |                     |                 |                 |                 |           |             |
| 2.  | P3 píše do u   |                     |                 |                 |                 |           |             |
| 3.  | P1 píše do u   |                     |                 |                 |                 |           |             |
| 4.  | P2 čte u       |                     |                 |                 |                 |           |             |
| 5.  | P3 píše do u   |                     |                 |                 |                 |           |             |
| 6a. | P3 čte u       |                     |                 |                 |                 |           |             |
| 6b  | P3 čte u       |                     |                 |                 |                 |           |             |

# PROTOKOLY CC ZALOŽENÉ NA ADRESÁŘÍCH

- U CMP se sběrnicí je snadný broadcast signálů *br*, *brx*, *bxo* a komunikace bitů *d*, *e*, *f*. Arbitr sběrnice také žádosti seřazuje.
- U multiprocesorů **NUMA** s větším počtem CMP by byl **broadcast složitý**. Proto žadatel komunikuje **přes prostředníka** (zástupce). Tímto prostředníkem jsou **distribuované adresáře** (directories) a jejich řadiče (**DirCtrl**) u jednotlivých modulů DSM.
- **Domovský adresář** (Home Directory, *H*) pro určitý modul SM udržuje informace o blocích (cache line) v modulu (RAM):
  - jestli je blok platný, čistý nebo špinavý,
  - kde se nachází, ve které (**sdílené last level**) cache (LLC).
- Řadič domovského adresáře **DirCtrl**:
  - Seřazuje příchozí žádosti a reaguje na ně.
  - Příkazy (zprávy) pro koherenci, ale místo broadcastu zasílá **pouze do relevantních uzlů** (multicast).
  - **To u velkých systémů enormně redukuje komunikaci.**





- **Záznam v adresáři o bloku na adrese X je  $N+1$  bitový vektor:**

- nultý clean/dirty bit  $V_0$
- $N$  prezenčních bitů  $V_i$ ,  $i = 1, 2, \dots, N$  (bit-mapa):

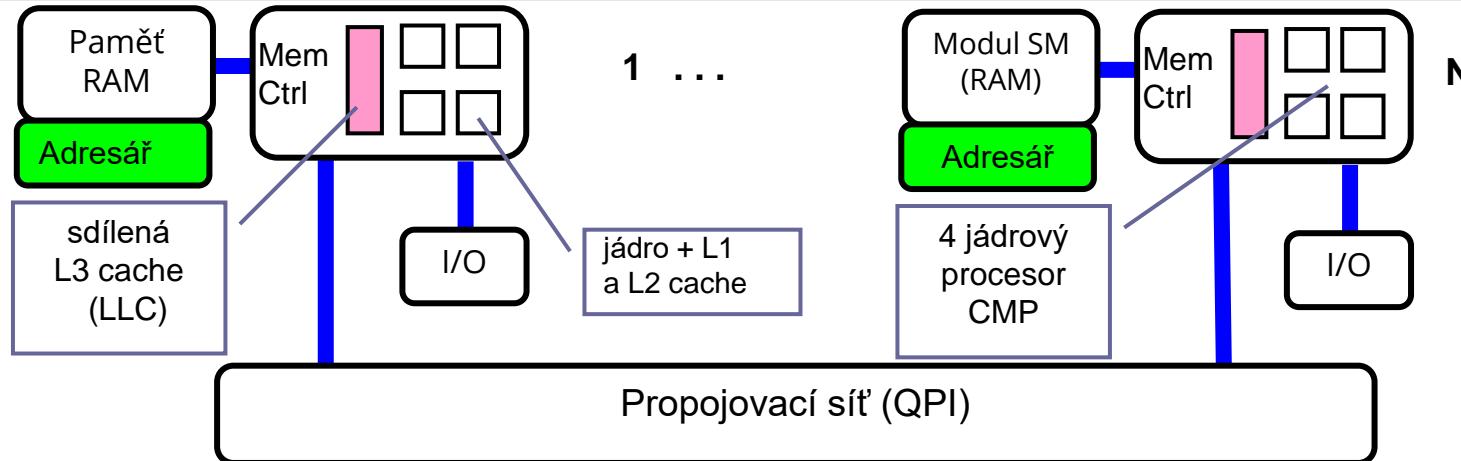
bit 0 1 2 3 4 5 ...  $N$

X [ 0 0 1 1 0 1 ... 0 1 ] **sdílený blok S** je ve více cache

X [ 0 0 0 0 0 ... 0 0 ] samé nuly: blok je jen v RAM

X [ 1 0 0 1 0 0 ... 0 0 ] **špinavý blok M** je v cache 3

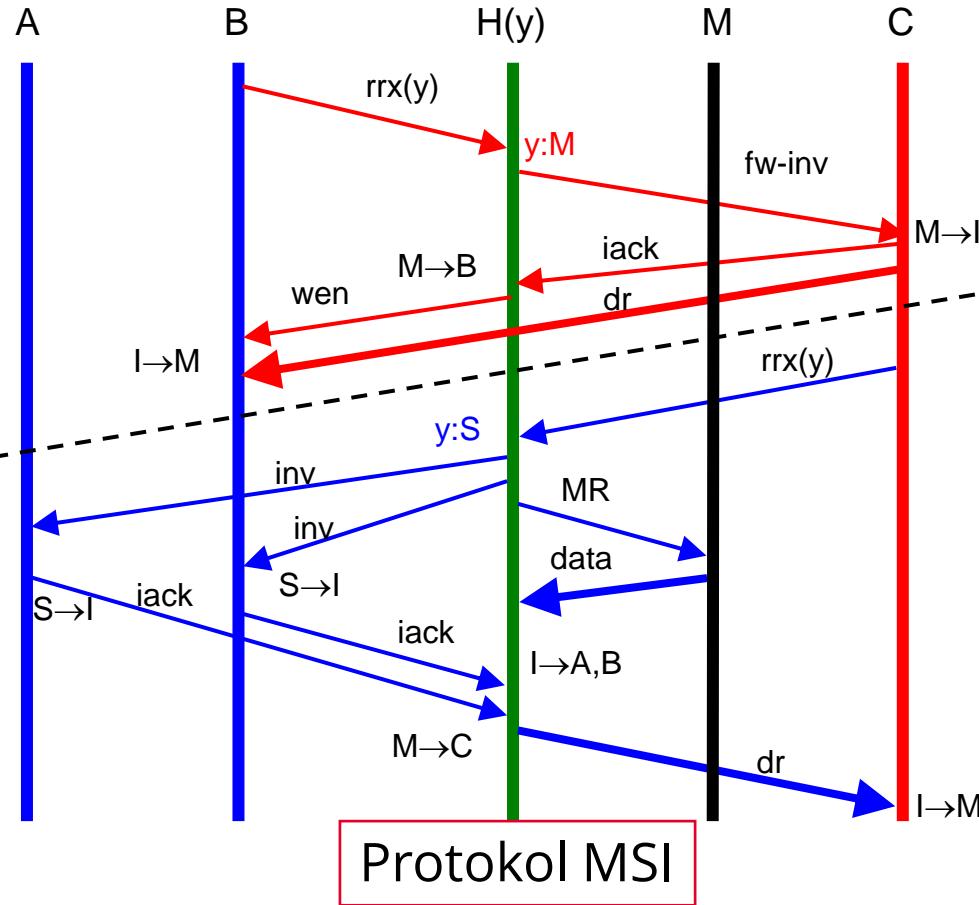
- **Bloky v cache** jsou označeny duplicitně dirty/clean bitem (kromě valid bitu).
- Stavové bity jsou čteny a modifikovány dvěma řadiči: **CacheCtrl** a **DirCtrl**.  
(V názvosloví Intel: „Cache agent“ a „Home agent“ nebo jen „Home“).  
Zpracovávají zprávy CC a datové odpovědi.



Sekvence komunikací u protokolu CC **s adresáři**:

1. žádost R/W → Home
  2. Home kontaktuje jen relevantní agenty CacheCtrl
  3. CacheCtrl: odpověď (potvrzení) → Home, datová odpověď žadateli, případně i do Home
  4. Home: povolení R/W → žadateli
- Řadiče (agenti) DirCtrl a CacheCtrl modifikují podle potřeby stavové byty bloků.
  - Všechna data procházejí přes cache L1. Mohou být při nedostatku místa vyhozeny do vyšší úrovně cache nebo až do SM RAM.

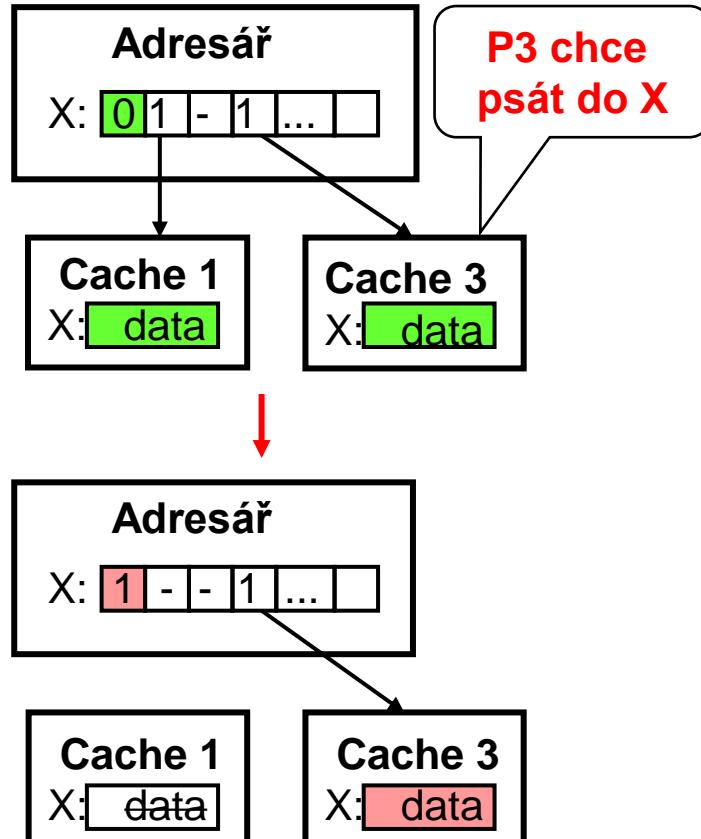
# I Ukázka: Výpadek při zápisu do bloku y (M a S)



## Legenda:

|        |                           |
|--------|---------------------------|
| rrx    | = rr exclusive            |
| fw-inv | = forward-invalidate      |
| dr     | = data reply              |
| wen    | = write enable            |
| iack   | = invalidate acknowledge  |
| inv    | = invalidate              |
| fw     | = forward and make shared |
| wb     | = write back do paměti    |

A, B = cache agenti (řadiče LLC procesorů)  
H(x) = home agent bloku x (DirCtrl)  
M = řadič paměti MemCtrl  
MR = memory read



Adresář s úplnou bitovou mapou: ke každému bloku v paměti je přiřazen

- bit-vektor  $N$  bitů prezence
- 1 clean/dirty bit.

Režie na 1 blok 64 byte, který má kopie až na  $N$  procesorech (v LLC) :

- $N = 8$ :  $9$  bitů /  $(64 * 8) = 2\%$
- $N = 64$ :  $65$  bitů /  $(64 * 8) = 13\%$
- $N = 256$ :  $257$  bitů /  $(64 * 8) = 50\%$ .

**Není škálovatelné!** Celková paměťová **režie** =  $(N + 1)$  bitů \*  $N * M_{local} \approx O(N^2)$ , kde  $M_{local}$  je počet bloků v 1 modulu DSM. Používá se do  $N = 64$ .

## Omezené adresáře:

- Sdílení bloku je **omezeno** na max.  $Q < N$  lokací.
- Je-li třeba víc než  $Q$  kopií, bude při dosažení max. počtu  $Q$  kopii třeba někomu kopii zneplatnit a jinému přidělit.

```
#pragma omp parallel for shared(N, a)
for (int i = 0; i < N; i++) a[i]++
```

## Vázané adresáře:

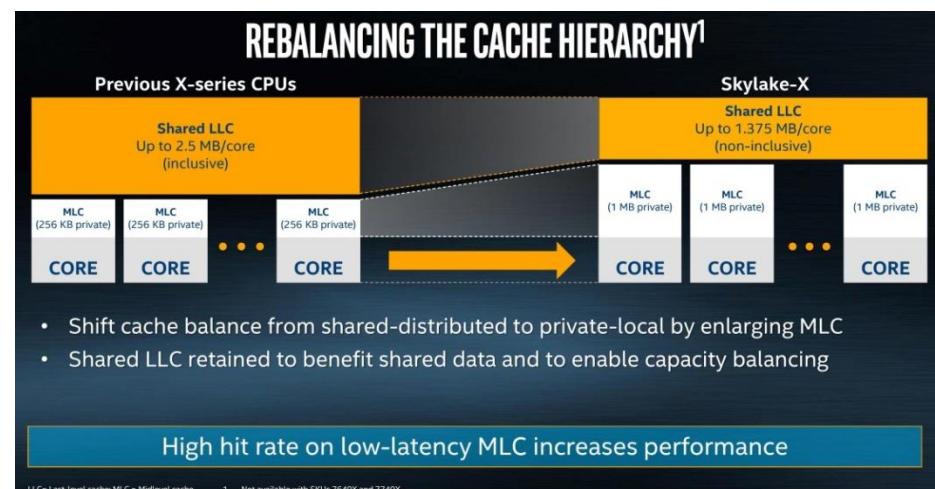
- Adresáře jsou zabudovány do samotných pamětí cache ve formě obousměrného **vázaného seznamu** kvůli snadnému vkládání a odebírání položek.
- **Režie:** 2 ukazovátka  $\log_2 N$  bitů na jeden blok v každé cache.
- **Škálovatelné** (standard SCI = Scalable Coherent Interface).
- **Nevýhoda:** invalidace vyžaduje průchod celým seznamem.

- OS udržuje tabulku stránek PT ve sdílené paměti. Položky PT jsou pro rychlý překlad VA → PA v malé cache TLB na čipu.
  - položky vkládá do TLB HW nebo OS.
  - Stránka (z disku) se načte do modulu sdílené paměti toho procesoru, jehož jádro se jí dotkne první (**first-touch**) tam bude její „Home“.
  - Pokud chce jiné jádro přistupovat do stejné stránky, bude generovat výpadek TLB.
  - Po aktualizaci TLB z PT načte data z modulu SM Home do své cache.
  - Stejná položka se tak může vyskytovat v TLB více jader. Musí se proto řešit koherence všech TLB, např. v případě výměny stránky nebo změny přístupových práv.
- **Řešení:**
  - použití části D-cache pro TLB
  - invalidace položek TLB přes OS pomocí přerušení
  - HW instrukce TLB Invalidate Entry (PowerPC)

- V dnešních vícejádrových procesorech bývá sdílená Last Level Cache LLC = L3 (několik MiB až desítek MiB); nově i sdílená embedded eDRAM L3 nebo L4 (96–128 MiB).
- Koherence mezi několika vícejádrovými čipy (CMP) je zajišťována přenosem spec. paketů (součást protokolu QPI). Intel QPI implementuje 5 stavový protokol MESIF.
- Koherenci na **nižších úrovních** (změny stavů bloků, wb, datové odpovědi) vyřídí u inkluzivních **cache LLC** její řadič (Cache agent).
- Pokud jsou nižší cache exkluzivní, musí se prohledávat stav všech nižších cache u všech jader! Bloky se navíc musí vyměňovat.

# Inkluzivní cache (preferuje Intel do Skylake-X)

- Je-li čistý blok přítomen v L1, musí být též v L2.
- Je-li v **exkluzivním** vlastnictví L1 **špinavý** blok, musí být alokován a zneplatněn i v L2.
- Nejsou-li data v **inkluzivní sdílené L3**, nejsou ani v L2 i L1 cache žádného jádra, takže se rovnou generuje požadavek do RAM (nemusí se hledat po jádrech). Najdou-li se data v L3, přečtou se přímo odtud.
- **U bloků v L3 je indikace, ve kterých járech je blok přítomen**
  - (1 inclusion bit na 1 jádro, malá bit-mapa)  
– takový malý adresář.
  - To dovoluje filtrovat žádosti *invalidace* přicházející do LLC = L3 a nepropustit ty, co se v L1 a L2 nemohou uplatnit.
  - Pak lze vystačit pouze s **jedním řadičem koherence pro mateřskou cache (L3)** a dceřiné cache (L2, L1).
- Udržování inkluze je za určitých podmínek samočinné



# | Exkluzivní cache (preferuje AMD)

- Data jsou v L1 nebo v L2, ale nikdy ne v obou
  - sdílená LLC = L3 není ani čistě exkluzivní ani inkluživní – dnes stejné jako Intel.
- Nejsou-li data v L1 až v L3, musí se prohledávat stav cache ostatních jader. Teprve při neúspěchu se jde do RAM.
- Když je v **L1 výpadek a v L2 zásah**, musí se blok mezi **L1 a L2 vyměnit**, což je více práce než jen kopírovat blok z L2 do L1 u inkluživní cache.
  - Navíc bloky L1 a L2 musí mít pro výměnu stejnou velikost.
- Systém exkluzivních cache uchovává více dat než inkluživní. To je znát hlavně když L2 a L3 mají podobnou velikost.

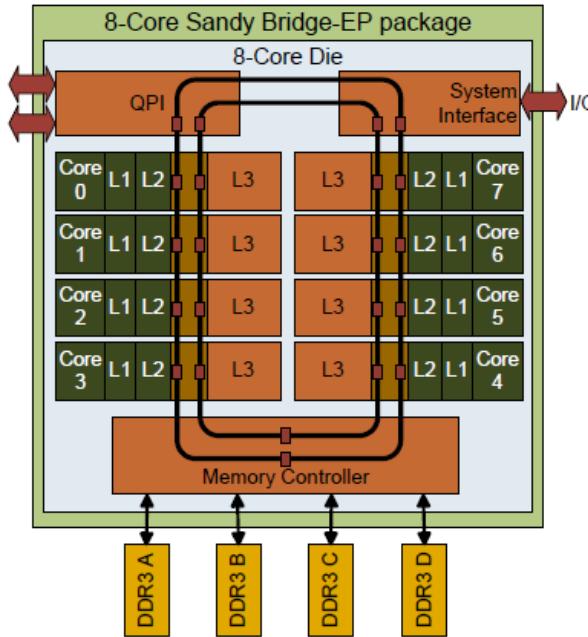


| Cache Hierarchy | AMD EPYC 7742 DDR4-3200                            | AMD EPYC 7601 DDR4-2400       | Intel Xeon 8280 DDR-2666                     |
|-----------------|----------------------------------------------------|-------------------------------|----------------------------------------------|
| <b>L1 Cache</b> | 32KB<br>4 cycles                                   | 32KB<br>4 cycles              | 32KB<br>4 cycles                             |
| <b>L2 Cache</b> | 512KB<br>13 cycles                                 | 512KB<br>12 cycles            | 1024KB<br>14 cycles                          |
| <b>L3 Cache</b> | 16MB / CCX (4C)<br>256MB Total<br>~34 cycles (avg) | 16MB / CCX (4C)<br>64MB Total | 38.5MB / (28C)<br>Shared<br>~46 cycles (avg) |

- **NCC-NUMA** – Cache bez HW podpory koherence,;
  - Pouze instrukce a privátní data mohou být v cache.
  - Sdílená data jsou komplátorem označena jako neschopná modifikace v cache (CRAY T3D, Intel SCC – Single Chip Cloud Computer).
- **Bez pamětí cache**
  - Latence paměťových přístupů skryta multi-threadingem (Cray MTA, Multi-Threaded Architecture, GPGPU = GPU pro univerzální výpočty).
- **SVM** – Shared Virtual Memory
  - Používá mechanismus virtuální paměti s podporou OS.
  - místo bloku stránka, OS označuje stránky RW, RO, INV; místo výpadků bloků výpadky stránek. Cílem není HPC.
- **Transakční paměť** (Transactional Memory)
  - Programátor specifikuje začátek a konec transakce ve zdrojovém kódu a TM systém provádí transakce paralelně a spekulativně – optimisticky předpokládá, že budou provedeny atomicky.
- **Pokud se z SM pouze čte nebo do ní pouze zapisuje, není třeba koherence**
  - Máme tedy zdrojové a cílové pole, a před přehozením pointerů zavolám FLUSH

# PROTOKOLY CC SOUČASNÝCH PROCESORŮ

# I Příklad koherence na kružnici: Intel Sandy Bridge



- Jádra, grafika, sdílená L3 cache (20 MiB), 4 kanálový MemCtrl a 2 linky QPI jsou propojeny 4 **kružnicovým propojením** s propustností 256 bit/takt (šířka AVX registru).
- L3 cache je inkluzivní pro cache L1 a L2 všech jader, rozdělená do řezů. Fyzické adresy rozděleny na řezy jednou hash funkcí. Každý řez má CacheCtrl a inkluzivní bity.
- Zprávy CC mezi čipy: QPI na 8 GT/s.

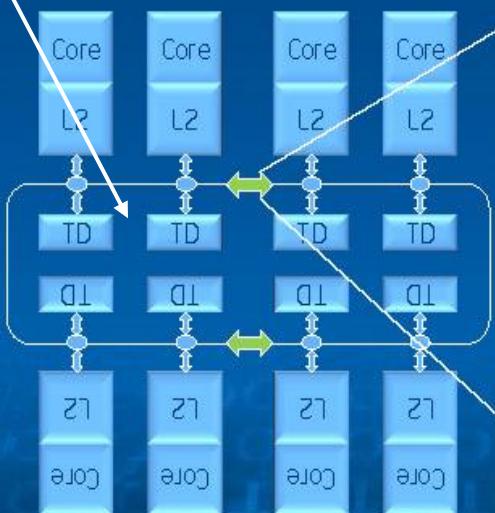
Přístup na prstenec podléhá arbitráži. Vysílání do prstence může být překryto s příjmem.

**Pokud vysílá víc než 1 uzel, zprávy nejsou totálně uspořádané.**

- Žádost **rr** nebo **rrx** putuje nejdříve od žadatele do **seřazovacího bodu** (MemCtrl) a tam je teprve aktivována (1 bitem v hlavičce).
  - Pak proběhne celá rotace (sbírání odezv uzelů – data od majitele, iack)
  - Seřazovací bod pak vyjme zprávu a pošle povolení zápisu (wen) žadateli. Průměrně  $P/2 + P + P/2 = 2P$  hopů.
- 
- **Interface na DRAM:**  
MemCtrl na žádost rr nebo rrx načte a dodá blok dat z paměti jen je-li to nutné, jinak blok ve stavu M, E, O (F) dodá rychleji přímo vlastník.
  - MemCtrl může též spekulativně načítat blok vždy a po vyhodnocení odezv jej předat žadateli na jeho explicitní žádost.

MemCtrls jsou rovnoměrně rozloženy po kruhu

## Interconnect

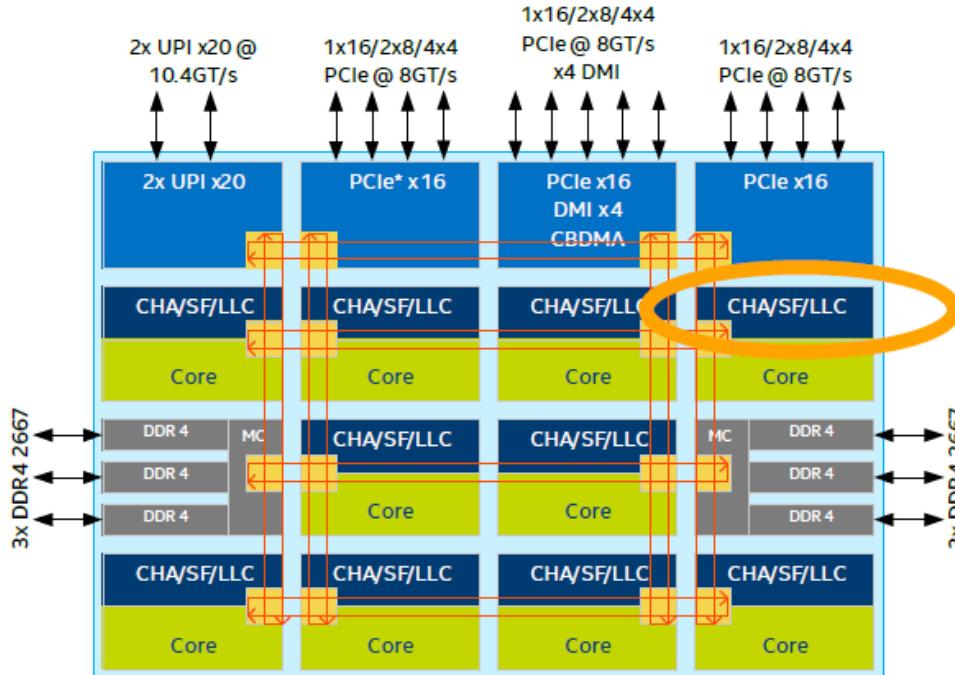


TD = tag directory



# Distribuované cache a home agent na Skylake-X

- Eliminates large tracker structures at memory controllers, allowing more requests in flight and processes them concurrently
- Reduces traffic on mesh by eliminating home agent to LLC interaction
- Reduces latency by launching snoops earlier and obviates need for different snoop modes.



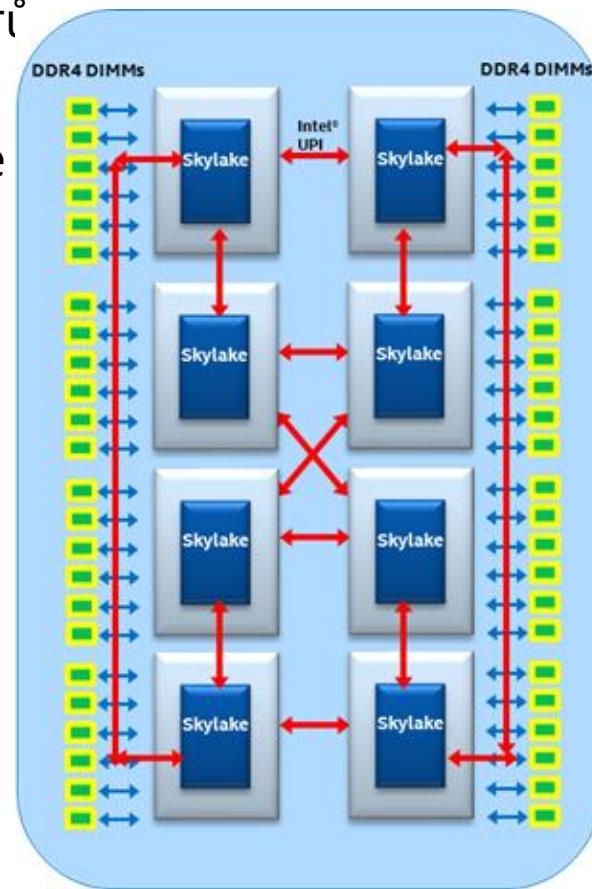
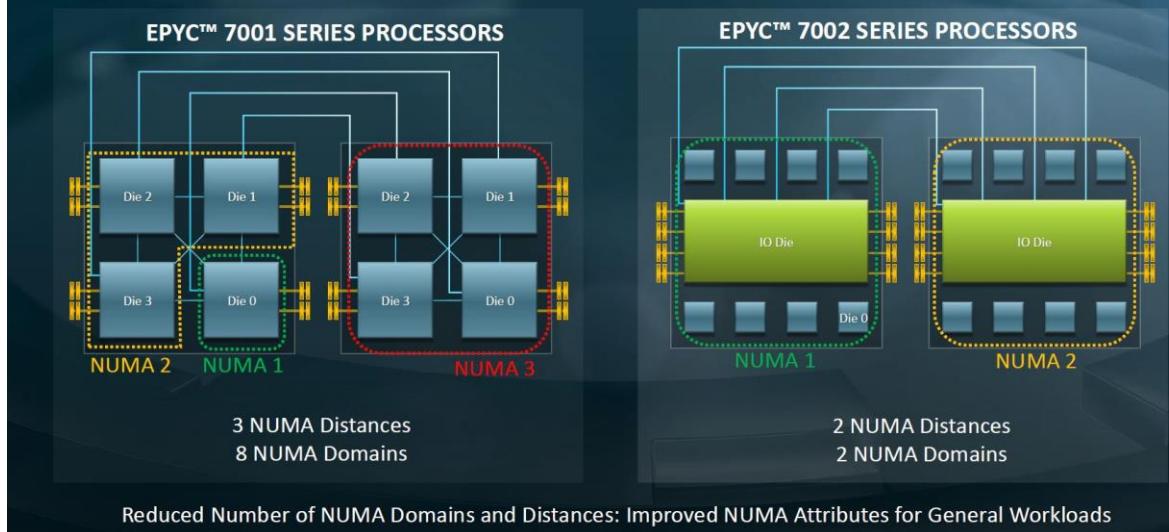
# **ARCHITEKTURY S DISTRIBUOVANOU SDÍLENOU PAMĚTÍ (NUMA DSM)**

- Paměti cache s plnou **hardwareovou podporou koherence**. Jeden protokol CC zadrátován do systému.
- Paměť **fyzicky distribuovaná** na uzly, fyzický adresový prostor je **logicky sdílený** – síť putuje žádost s fyzickou adresou PA.
- **Propojovací síť** místo sběrnice obecně znamená NUMA (NonUniform local and remote Memory Access) ⇒ rozhlašování (multicast) je obtížnější (kdy je hotovo?)
- Paměťové **transakce rozdělené**, na každou žádost je nutná odpověď / potvrzení (ACK)
- **Citlivost na distribuci dat**, optimálně data alokovat v uzlu, kde se s nimi pracuje. (U centralizované SM je alokace dat libovolná.)
- **Škálovatelnost** do stovek až 2 tisíců jader.

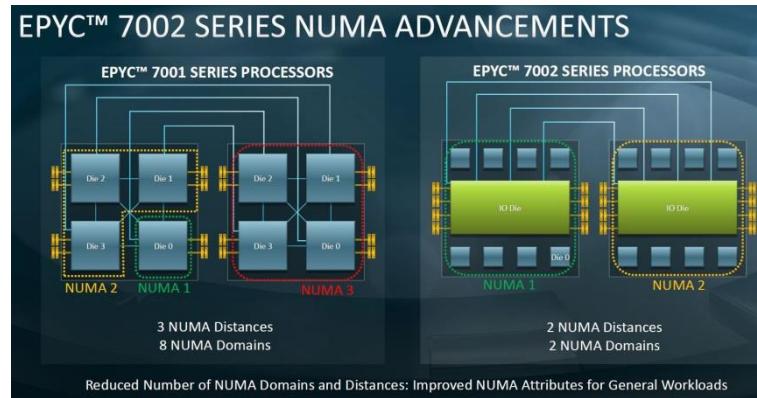
# I | NUMA systémy na jedné základní desce

- Intel UPI má 2 nebo 3 linky pro propojení procesorů
- AMD Infinity fabric pro propojení procesorů.
- AMD snižuje počet NUMA regionů vyjmutím řadiče paměti z procesoru, zvyšuje latenci.

## EPYC™ 7002 SERIES NUMA ADVANCEMENTS



# Latence a propustnosti paměti a cache

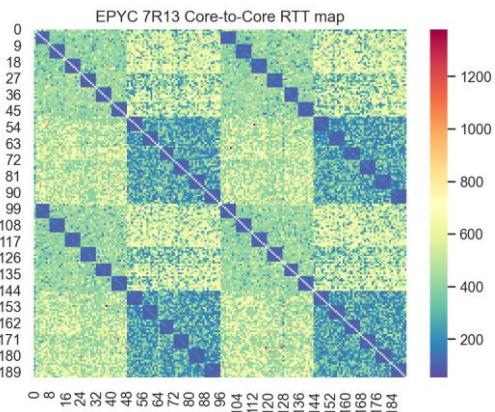
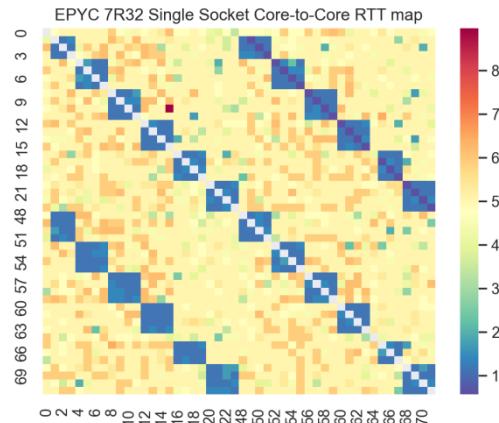


|   | 0           | 1           | 2           | 3           | 4           | 5           | 6           | 7           |
|---|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| 0 | 36346       | 19101       | 19910       | 19913       | 8143        | 8080        | <b>8143</b> | 8159        |
| 1 | 19103       | 36435       | 19913       | 19911       | 8080        | 8080        | 8084        | <b>8066</b> |
| 2 | 19905       | 19916       | 36280       | 19065       | <b>8070</b> | 8049        | 8088        | 8050        |
| 3 | 20095       | 19908       | 19093       | 36357       | 8055        | <b>8045</b> | 8055        | 8055        |
| 4 | 8072        | 8069        | <b>8062</b> | 8084        | 36409       | 19126       | 19931       | 19932       |
| 5 | 8083        | 8060        | 8086        | <b>7996</b> | 19125       | 36389       | 19927       | 19935       |
| 6 | <b>8170</b> | 8124        | 8110        | 8194        | 19933       | 19929       | 36479       | 19120       |
| 7 | 8123        | <b>8094</b> | 8169        | 8088        | 19938       | 19941       | 19101       | 36409       |

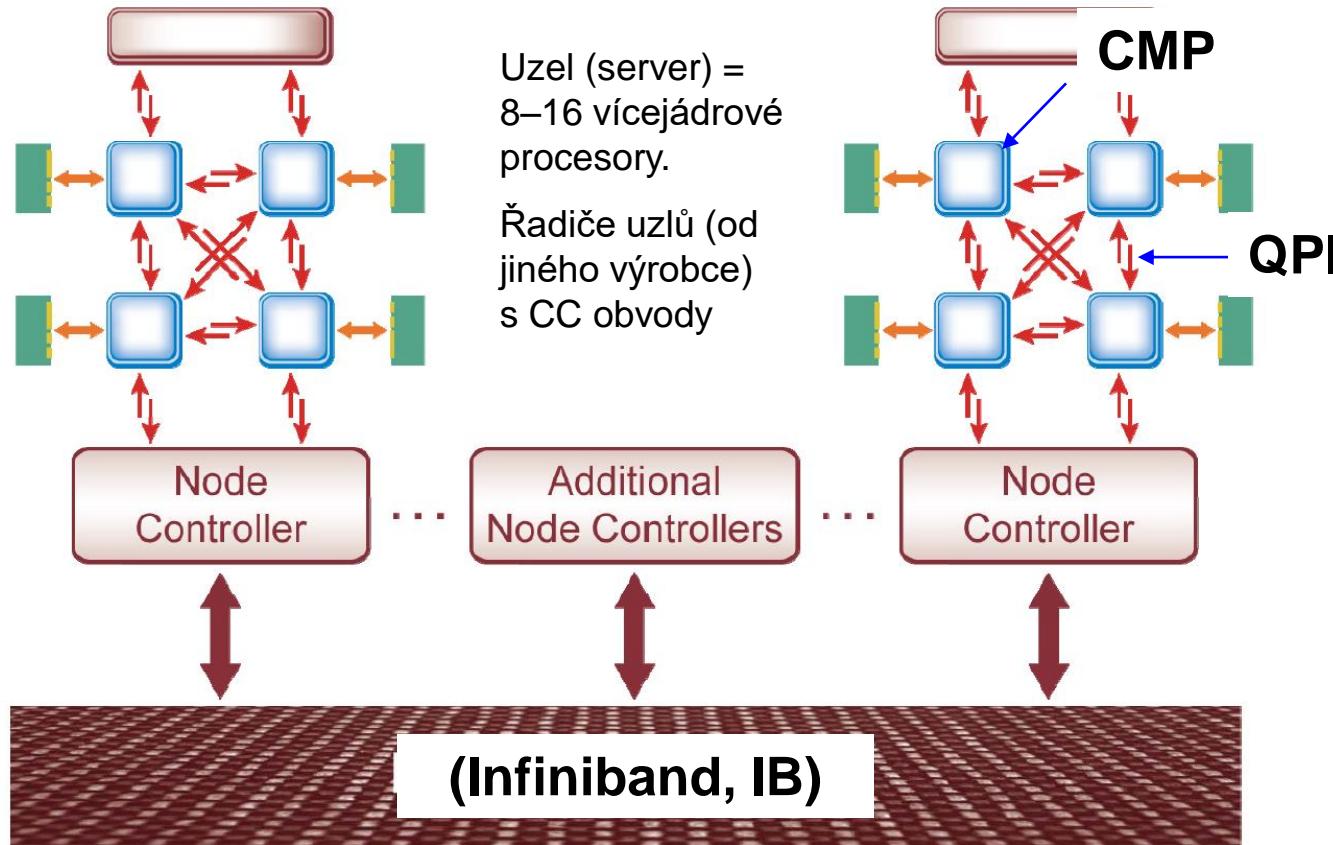
© 2017 ServeTheHome.com

|   | 0          | 1          | 2          | 3          | 4          | 5          | 6          | 7          |
|---|------------|------------|------------|------------|------------|------------|------------|------------|
| 0 | 81         | 138        | 133        | 133        | 242        | 241        | <b>200</b> | 235        |
| 1 | 137        | 85         | 132        | 133        | 241        | 241        | 235        | <b>199</b> |
| 2 | 133        | 132        | 84         | 137        | <b>199</b> | 246        | 234        | 234        |
| 3 | 133        | 133        | 137        | 84         | 245        | <b>198</b> | 234        | 234        |
| 4 | 241        | 241        | <b>199</b> | 235        | 85         | 137        | 133        | 133        |
| 5 | 241        | 241        | 237        | <b>198</b> | 137        | 85         | 133        | 133        |
| 6 | <b>199</b> | 246        | 234        | 234        | 133        | 133        | 85         | 137        |
| 7 | 245        | <b>199</b> | 234        | 234        | 133        | 133        | 137        | 84         |

© 2017 ServeTheHome.com



# Větší systémy Intel CC-NUMA propojené sítí



- Domovský adresář řadí příchozí žádosti. Nemusí vědět, kdy dokončí všechny transakce ve všech uzlech.
- Ve frontě žádostí v domovském adresáři jsou použity přídavné stavy „busy“ nebo „rozpracovaný“ (pending)
- Při indikaci že operace je v běhu, musí zpozdit další operace na stejně adrese; může se zatím zpracovat transakce pro jiný blok. Provedení:
  - bufer rozpracovaných žádostí v domovském uzlu
  - bufer žádostí u žadatele
  - (bez buferování žádostí): NACK a zkusit znova
- CC protokoly:
  - s distribuovanými adresáři
  - s nasloucháním – oddělené sítě pro adresování (podpora broadcastu) a data (např. X-bar, kružnice)

| coherence state | latency in ns |     |         |      |    |              |      |    |  |
|-----------------|---------------|-----|---------|------|----|--------------|------|----|--|
|                 | local         |     | on-chip |      |    | other socket |      |    |  |
|                 | L1            | L2  | L1      | L2   | L3 | L1           | L2   | L3 |  |
| modified        |               |     | 40.4    | 38.1 | 15 | 123          | 140  |    |  |
| exclusive       | 1.5           | 4.6 |         | 33.8 |    |              |      |    |  |
| forward         |               |     |         | 15   |    |              | 87.3 |    |  |
| shared          |               |     |         |      |    |              |      |    |  |

Latencies for accesses to various memory locations on the dual socket Intel Sandy Bridge system

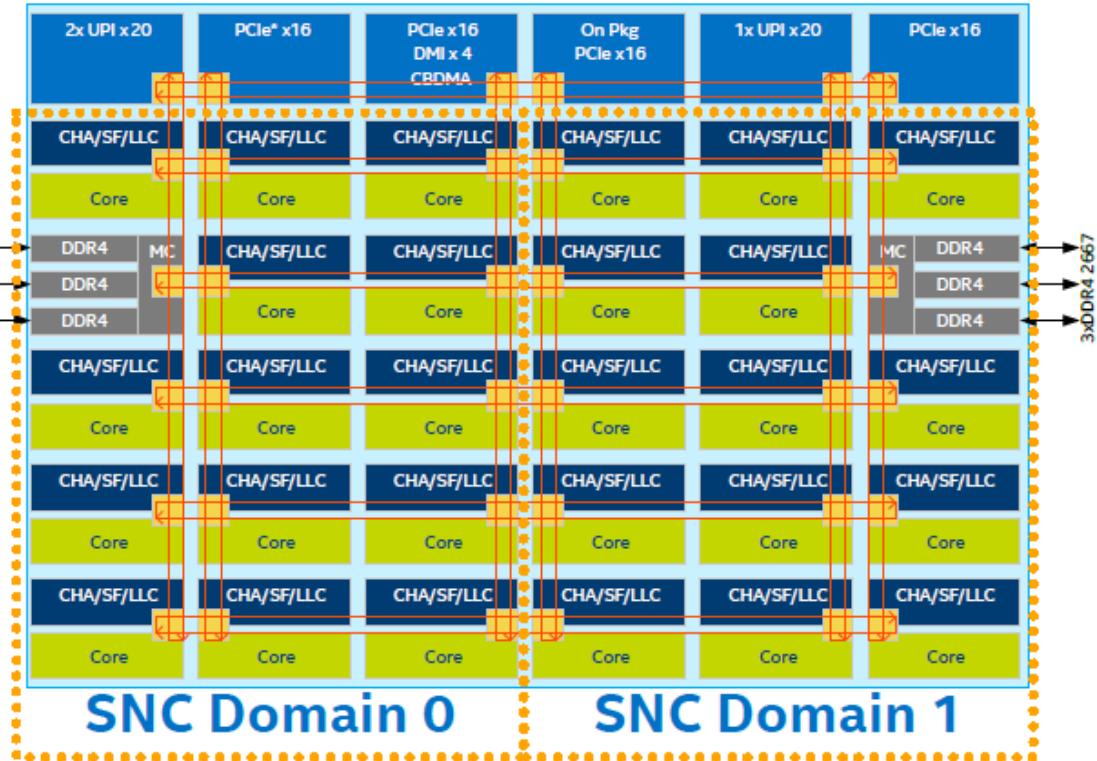
**DRAM**  
**local:** 81 ns  
**remote:** 133 ns

| mem-bind | hops | node 1 idle |        | node 1 active |        |
|----------|------|-------------|--------|---------------|--------|
|          |      | 1 thrd      | 8 thrd | 1 thrd        | 8 thrd |
| node0    | 0    | 11.7        | 39.7   | 12.9          | 42.2   |
| node1    | 1    | 7.7         | 18.7   | 8.6           | 23.6   |

**DRAM**  
**local:** 11,7 GB/s  
**remote:** 7,8 GB/s

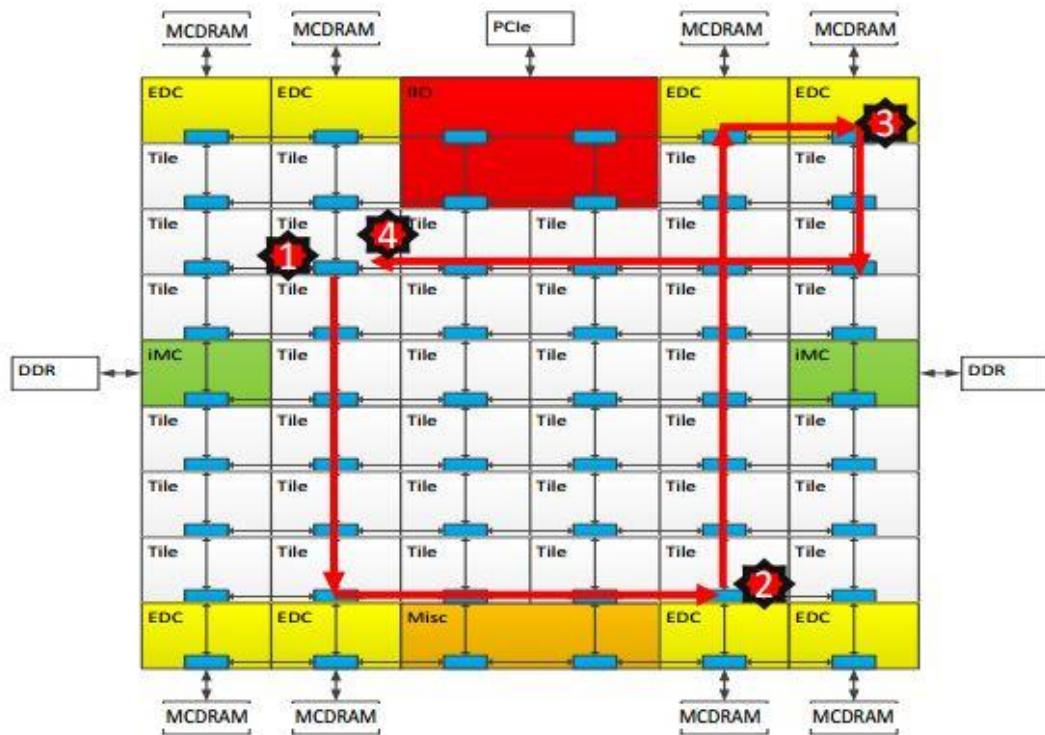
Read bandwidths in GB/s for one or eight cores running on node0 reading memory from different nodes on the two socket Intel Sandy Bridge system

# Sub-NUMA Cluster (SNC)



- Sunny Cove umožnuje rozdělit procesor na dvě NUMA domény
  - V obou případech stačí jeden UPI agent pro komunikaci s dalšími procesory
  - Latence pro přístup do paměti je nižší, pokud programátor ví co dělá.
  - Kapacita LLC je využita lépe (méně kopií téhož bloku).

# Cluster Mode: All-to-All



Address uniformly hashed across all distributed directories

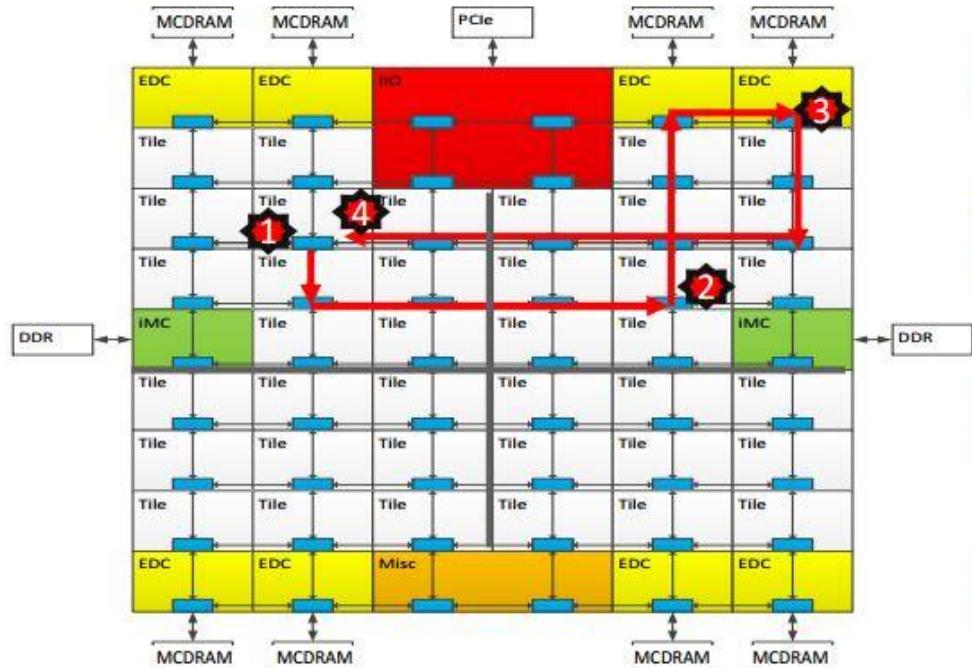
No affinity between Tile, Directory and Memory

Most general mode. Lower performance than other modes.

## Typical Read L2 miss

1. L2 miss encountered
2. Send request to the distributed directory
3. Miss in the directory. Forward to memory
4. Memory sends the data to the requestor

## Cluster Mode: Quadrant



Chip divided into four virtual Quadrants

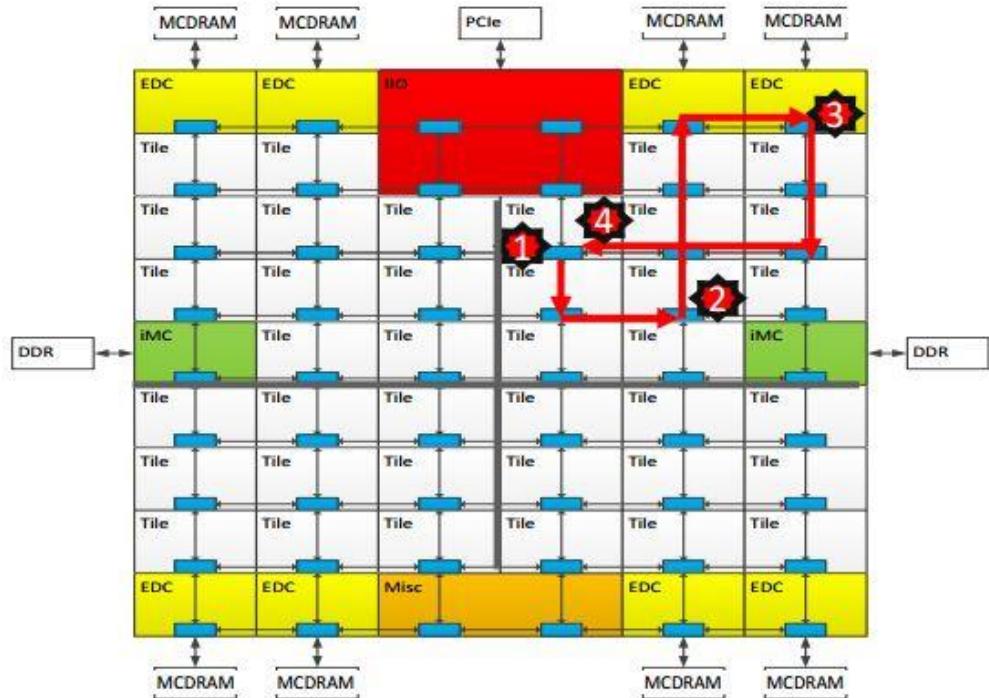
Address hashed to a Directory in the same quadrant as the Memory

Affinity between the Directory and Memory

Lower latency and higher BW than all-to-all. SW Transparent.

- 1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return

## Cluster Mode: Sub-NUMA Clustering (SNC)



Each Quadrant (Cluster) exposed as a separate NUMA domain to OS.

Looks analogous to 4-Socket Xeon

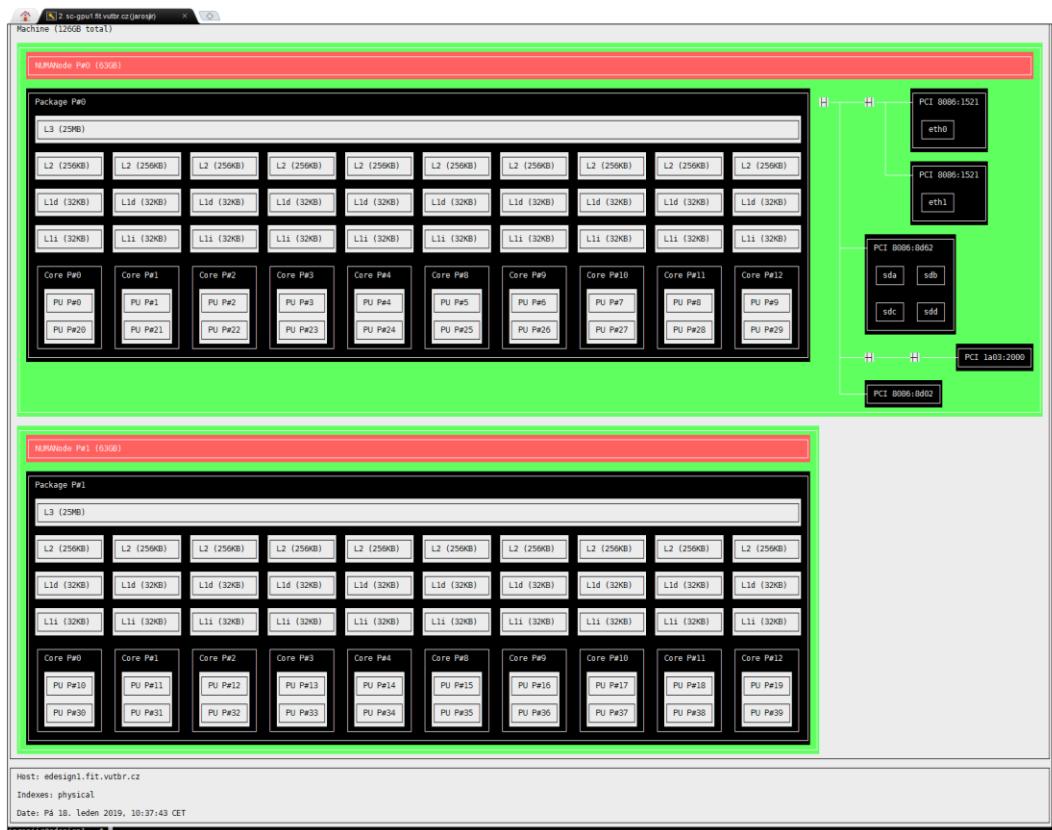
Affinity between Tile, Directory and Memory

Local communication. Lowest latency of all modes.

SW needs to NUMA optimize to get benefit.

- 1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return

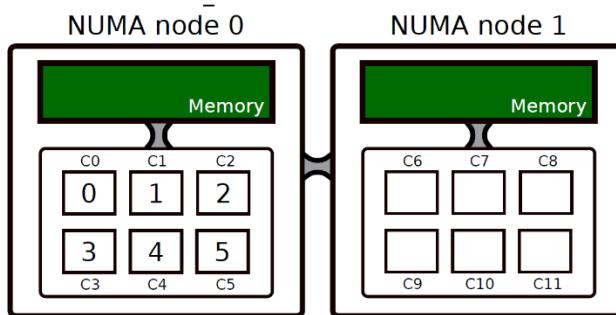
- Jak zjistit topologii
  - \$ hwloc-ls --output-format txt
- Bindování procesů na NUMA domény:  
numactl
  - # only use cores from node 0
  - # allocate all memory on node 1
  - \$ **numactl** --cpunodebind 0 -- membind 1 ./my-program
- Programově: libnuma
  - int\* memory = ...;
  - int node = 1;
  - **move\_pages(0, 1, &memory, &node, &status, 0);**
- Pomocí OpenMP proměnných prostředí a NUMA funkcí



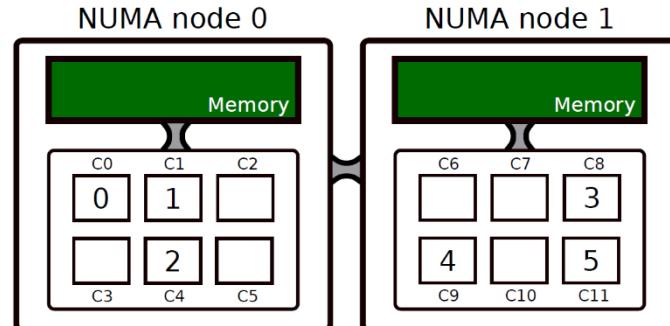
- **\$OMP\_PROC\_BIND** - binds threads to a specific *place*
  - false do not bind
  - true pin threads to a single core
  - master place threads at the same *place* as master thread
  - close place threads close to master
  - spread spread threads evenly
  
- **\$OMP\_PLACES** - defines what is a *place*
  - threads *place* is a hardware thread
  - cores *place* is a core
  - sockets *place* is a socket (~NUMA node)

# I FIT Ukázka OpenMP bindování

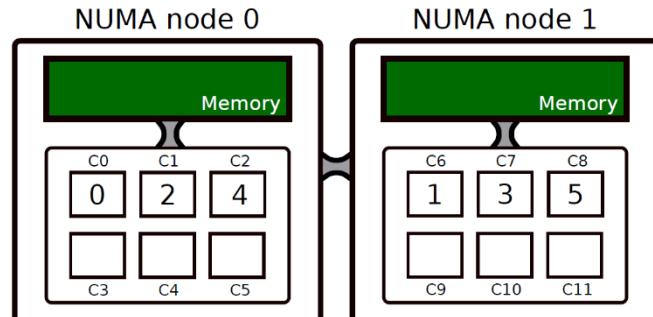
- OMP\_NUM\_THREADS=6
- OMP\_PROC\_BIND=close
- OMP\_PLACES=cores



OMP\_NUM\_THREADS=6  
OMP\_PROC\_BIND=close  
OMP\_PLACES={0,1,4,8,9,11}



- OMP\_NUM\_THREADS=6
- OMP\_PROC\_BIND=spread
- OMP\_PLACES=cores



- **NUMA FIRST TOUCH** – První přístup do stránky rozhodne o umístění do paměti, která je nejblíže jádru které způsobilo výpadek

```
const int SIZE = 1024 * 1024;
float* array = malloc(SIZE * sizeof(int));
```

Alokace

```
#pragma omp parallel for schedule(static)
for (auto i = 0; i < SIZE; i++)
 array[i] = 0.0f;
```

First touch  
mapování do  
RAM

```
#pragma omp parallel for schedule(static)
for (auto i = 0; i < SIZE; i++)
 processElement(array, i);
```

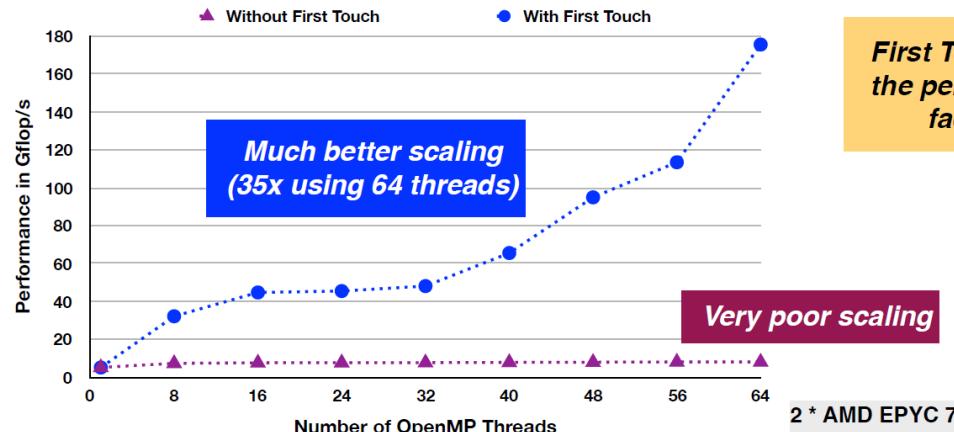
Použití

```
free(array);
```

- **Paměť na NUMA doméně 0 bývá obsazena IO buffery**
  - Uživatelská data mohou přetéct do jiné NUMA domény
  - Diskové a síťové operace by měly jít z domény 0
  - Připojená GPU by měla být obsluhována z jader, která mají přímý přístup k danému PCI
- **Přístup k datům musí vykazovat stejný vzor jako při First touch (statické plánování)**
  - Distribuce dat probíhá na úrovni stránek (4 KB, ale 1 GB).
  - Dynamické plánování může poškodit afinititu vláken k datům.
  - Tasky mohou poškodit datovou lokalitu (untied).
  - Různé přístupové vzory v různých částech algoritmu (po řádcích, po sloupcích, atd.).

# Ukázka – AMD EPYC – 2x32 jader, 8x NUMA

## Performance of the matrix-vector algorithm (4096x4096)



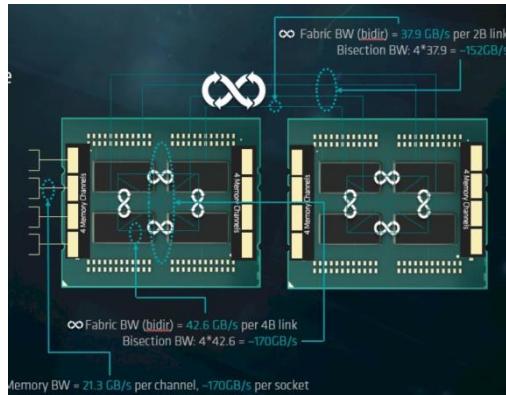
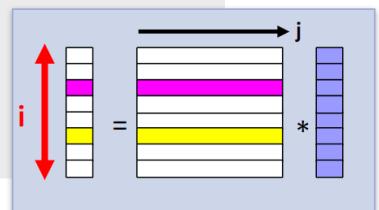
**First Touch improves the performance by a factor of 22x**

2 \* AMD EPYC 7551 32 Core Processor  
Oracle Linux 4.14.35-1821.el7uek.x86\_64

```
#pragma omp parallel for default(none) \
 shared(m,n,a,b,c)
for (int i=0; i<m; i++)
{
 double sum = 0.0;
 for (int j=0; j<n; j++)
 sum += b[i][j]*c[j];
 a[i] = sum;
}
```

```
$ OMP_PLACES={0}:2:1,{8}:2:1,{16}:2:1,{24}:2:1
$ OMP_PLACES+=,{32}:2:1,{40}:2:1,{48}:2:1,{56}:2:1
$ export $OMP_PLACES

$ export OMP_PROC_BIND=close
```



# Pokračování příště

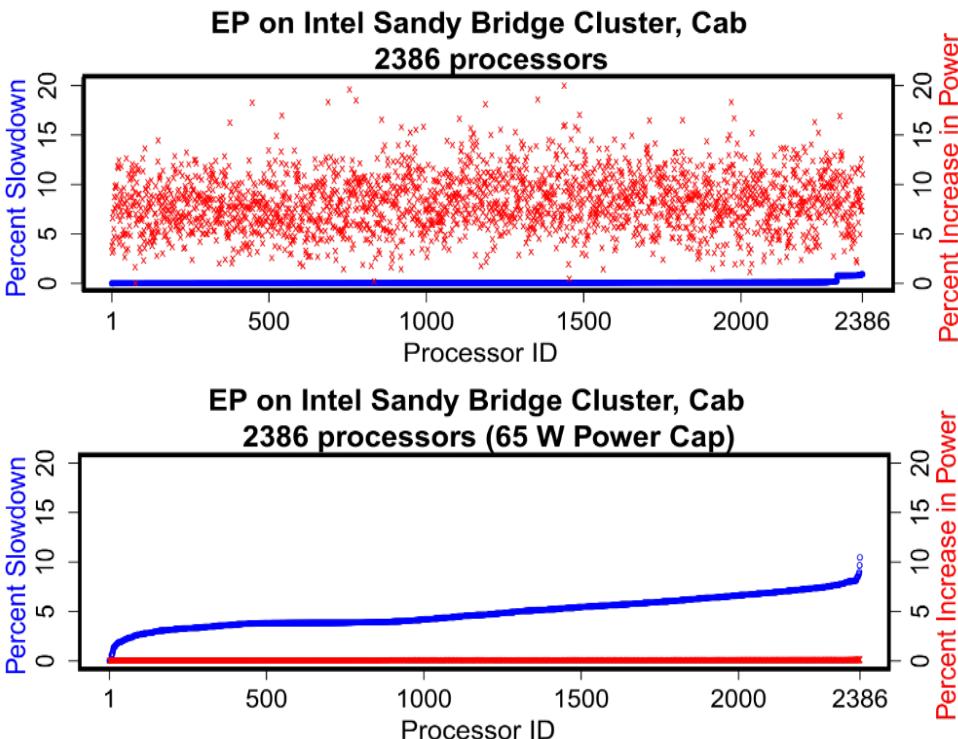
# Řízení spotřeby procesoru, Historie procesorů Intel AVS – Architektury výpočetních systémů Týden 12, 2024/2025

**Jirka Jaroš**

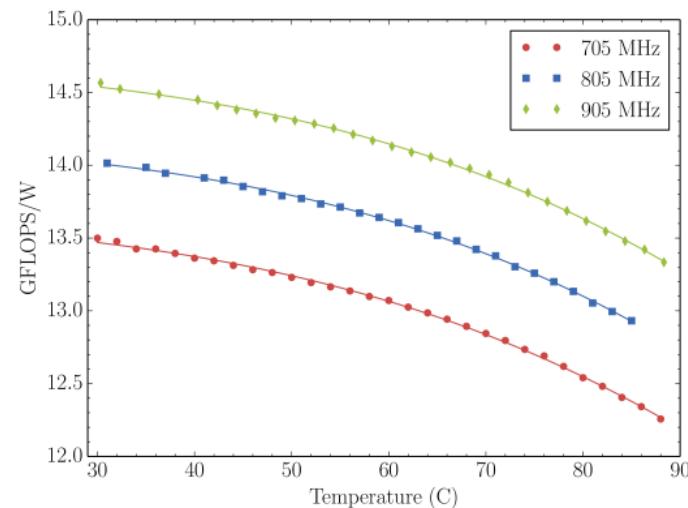
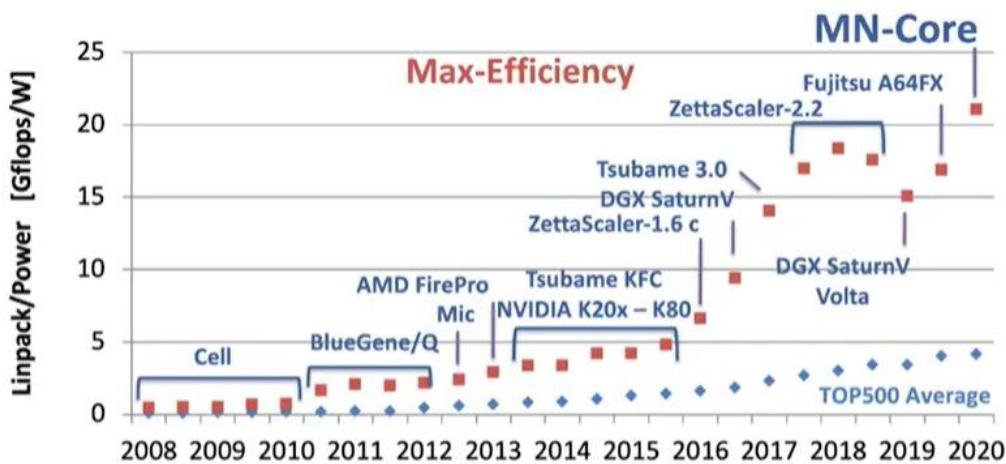
Vysoké učení technické v Brně, Fakulta informačních technologií  
Božetěchova 1/2, 612 66 Brno - Královo Pole  
[jarosjir@fit.vutbr.cz](mailto:jarosjir@fit.vutbr.cz)

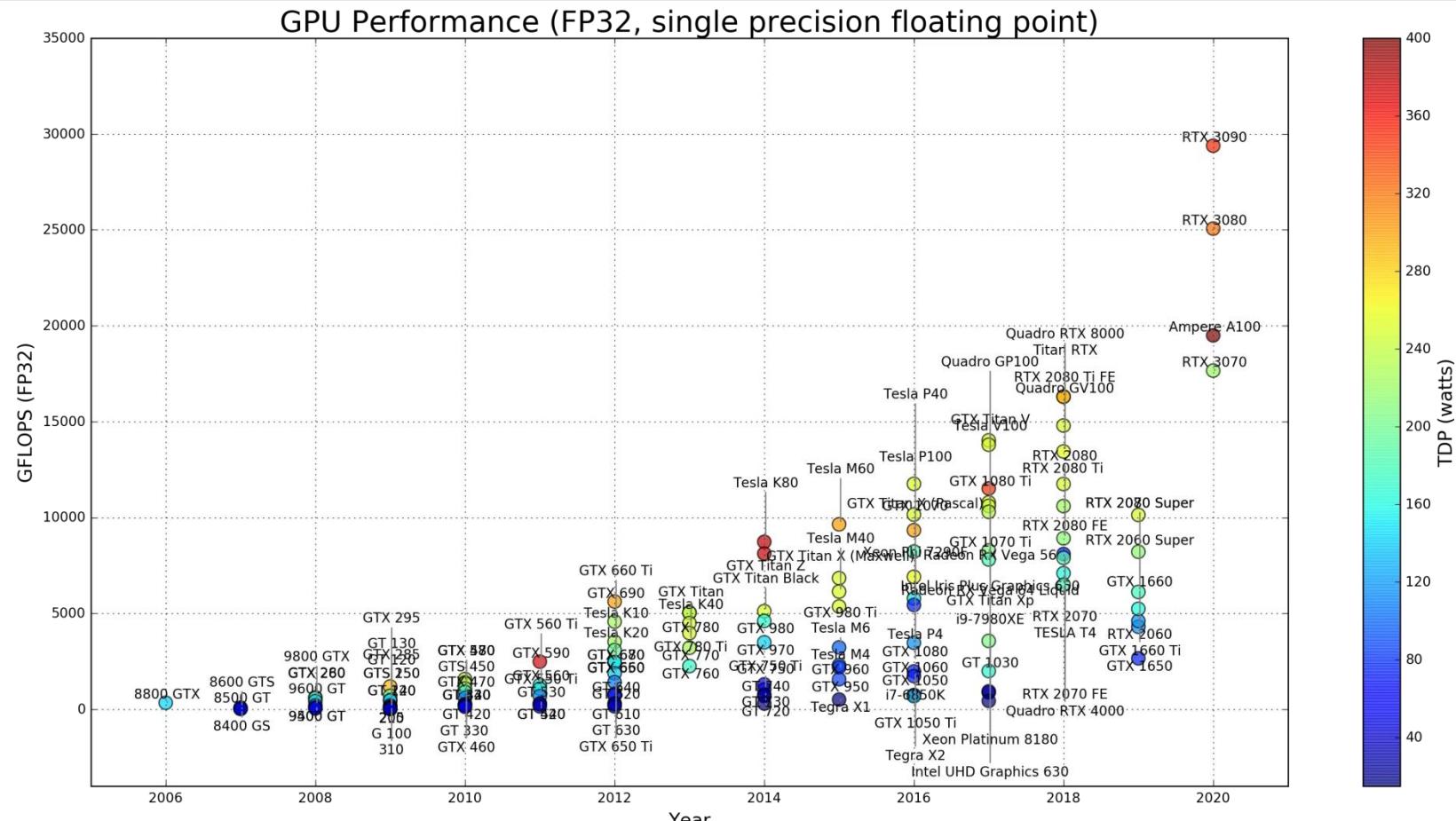


- **Limity chladicího systému**
  - Vzduchové, kapalinové, dusík, ...
  - S teplotou roste příkon
- **Rozvod napájení po čipu**
  - Širší dráty, vyšší vodivost, ...
- **Nestabilita výkonu při fixním příkonu**
  - Díky výrobním tolerancím má každý čip jinou spotřebu při dané frekvenci
- **Omezená kapacita baterií**
  - Pro mobilní zařízení je důležitá výdrž na baterii
  - Výkon bývá na druhém místě
- **Náklady na energii mohou snadno přesáhnout pořizovací cenu HW**
  - Intel Core i9-10900K  $\approx$  125/250W
  - Spotřeba za rok: 1,1 MWh  $\approx$  13.200,- Kč
  - Maximální spotřeba pro ExaScale stroj stanovena na 20 MW, reálně bude 50 MW



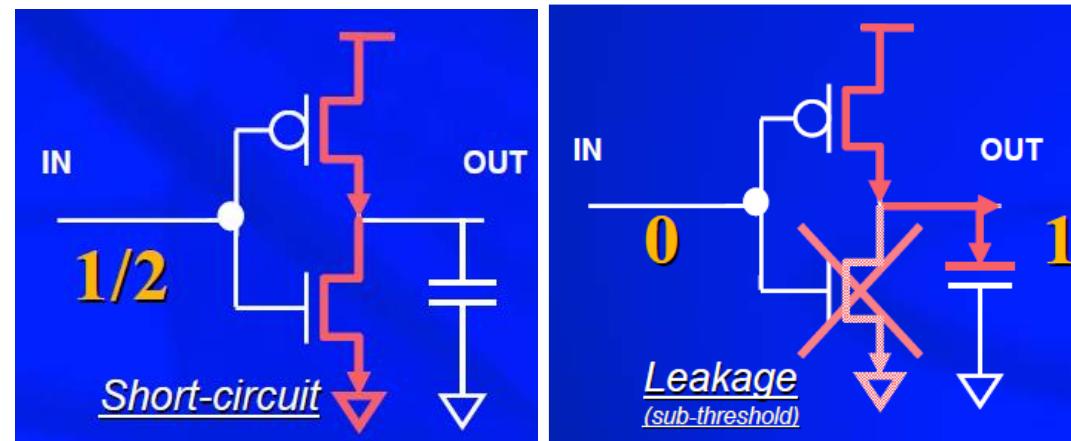
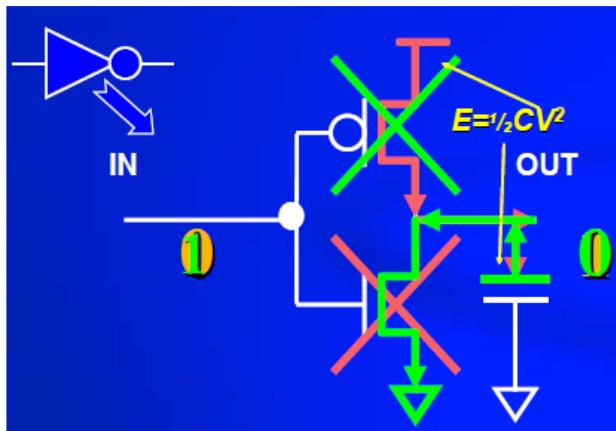
- Dynamické řízení příkonu DPM (Dynamic Power Management)
- Dynamické odstupňování napětí DVS (Dynamic Voltage Scaling)
- Spouštění instrukcí z mnoha vláken
  - V jednom cyklu SMT (simultaneous multithreading)
  - Na více jádrech procesoru typu CMP
- Základní metrika MIPS/W (GFlops/W)





- Návrh na úrovni tranzistorů
  - Redukce energie pro přepnutí ON-OFF, klidový/zkratový proud
- Návrh na úrovni obvodu
  - Snaha o asynchronní řízení, různé frekvence různých bloků
- Návrh na úrovni bloků
  - Vypínání nevytížených bloků (jádra, cache, FX, FP, sběrnice)
- Návrh na úrovni systému
  - Alokaci architektury, mapování aplikací, plánování procesů...
- Na úrovni kompilátoru
  - Optimalizace strojového kódu pro optimální využití zdrojů
  - Poskytnutí informací pro CPU o náročích procesu

- Na tranzistorové úrovni lze formulovat celkový ztrátový výkon jako součet tří hlavních složek
  - Přepínací ztráty
  - Ztráty zkratovým proudem
  - Ztráty klidovým proudem



## Přepínací ztráty

$$P_{device} = \frac{1}{2} C \cdot V_{DD} V_{swing} a \cdot f$$

## Ztráty klidovým proudem

$$I_{leakage} V_{DD}$$

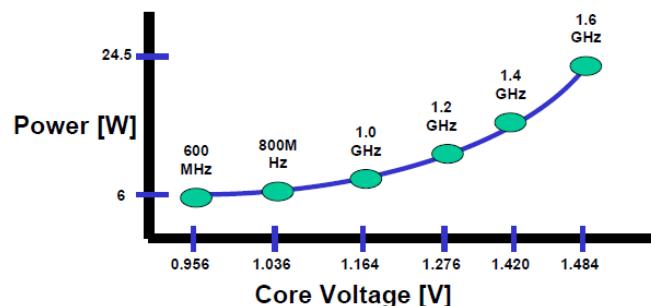
## Ztráty zkratovým proudem

$$I_{SC} V_{DD}$$

- $C$  je kapacita na výstupu tranzistoru
- $V_{DD}$  je napájecí napětí
- $f$  je hodinový kmitočet čipu
- $a$  je faktor aktivity ( $0 < a < 1$ )
- $V_{swing}$  je napěťový rozkmit na výstupní kapacitě
- $I_{leakage}$  je klidový proud
- $I_{SC}$  je zkratový proud

**Nejdůležitější metodou redukce příkonu je tedy**

- Snižování napájecího napětí
- Snižování frekvence



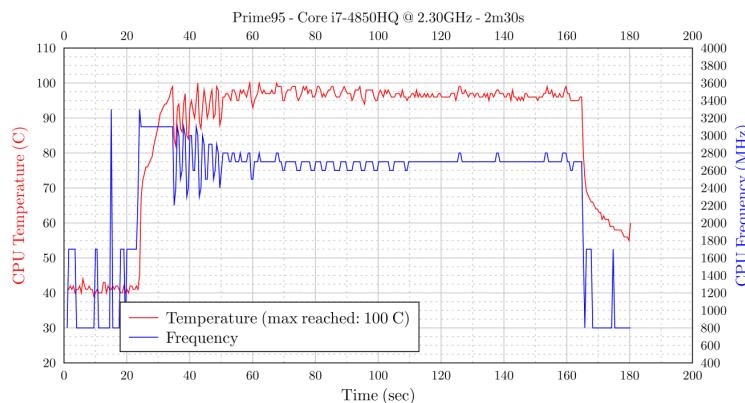
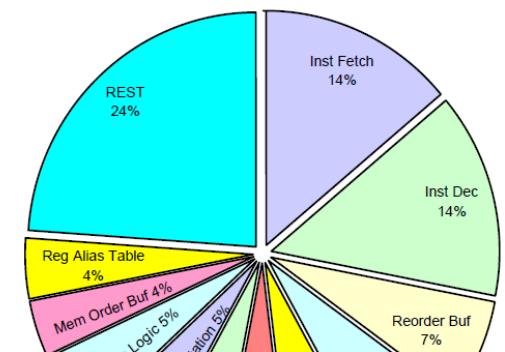
- **Paměti RAM, cache, registrové soubory, tabulky predikce skoků, atd...**

- Paměťové vodiče jsou dlouhé, představují určitou kapacitu a tím pádem spotřebu.
- Cache s vypínáním paralelních cest, tedy se snižováním stupně asociativity.
- Fronty instrukcí a tabulky přeložených adres

- **Filtrovací cache**

- Vychází se z předpokladu že většina přístupů do cache může být obslužena malou filtrovací cache.
- Omezení příkonu za cenu snížení výkonnosti způsobené zdržením přístupu do hlavní cache
- Využívají se např. cache s frontou požadavků s proměnnou délkou, řízenou dynamickou pracovní zátěží.
- Náklady na přídavné tranzistory jsou pod 2 %

- Hradlování umožní pozastavit některé stupně pipeline v případě, že procesor provádí instrukce z nesprávně předpovězené větve skoku.
  - Hradlování pipeline řídí stupeň spekulace superskalárních procesorů, u kterých se používá predikce skoků.
  - Rozhodnutí o pozastavení pipeline se provádí pomocí obvodu **odhadu konfidence**.
- **Zatlumení pipeline (throttling)**
  - Jestliže příkon překročí určitou mez a teplota procesoru se zvýší k nebezpečné hranici, dojde k zatlumení pipeline.
    - Provádí se buď snížením frekvence (pozastavením hodin).
    - Nebo vkládáním prázdných operací.

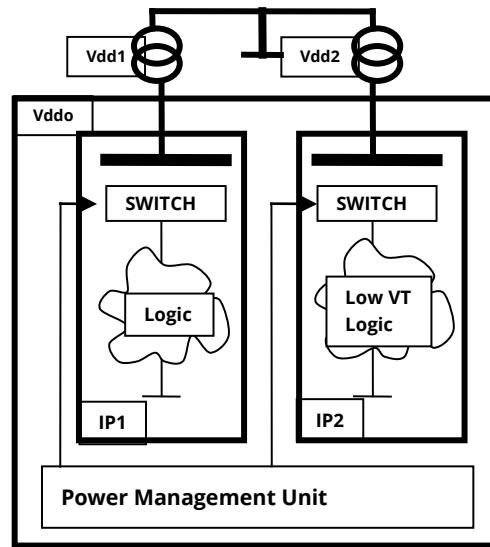


- **Na úrovni bloků**

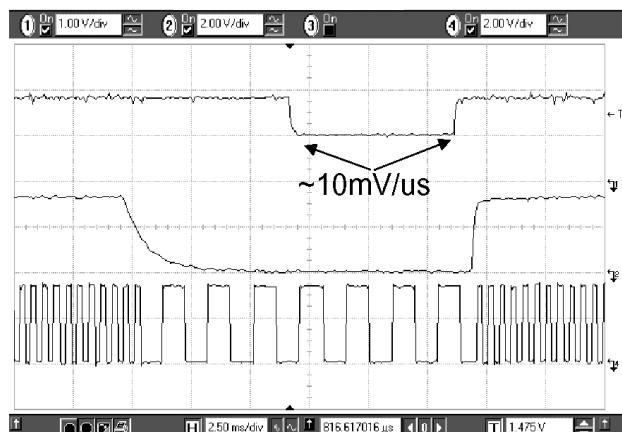
- Odpojování nepotřebných částí (power gating)
- Snižování frekvence nepotřených jednotek (clock gating, frequency stepping)
- Pokročilé techniky (detekce cyklů, fúzování mikroinstrukcí)

- **Na úrovni systému**

- Zpracování instrukcí z více vláken



*Logic VDD*  
*Freq change request*  
*"Dhrystone loop complete" indicator*



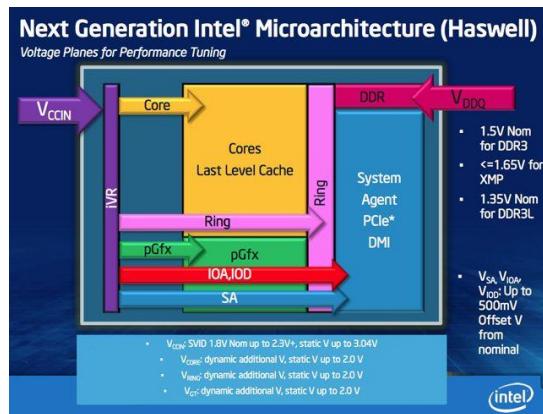
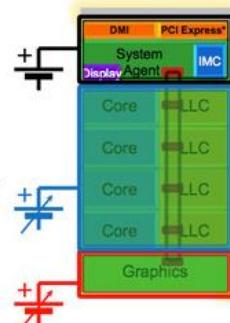
- Jedná se především o optimalizace mezikódu
  - Snaha o zabránění vykonávání zbytečných instrukcí
    - Rozbalování smyček
    - Propagace konstant
    - Odstranění mrtvého kódu
  - Automatická detekce paralelizmu
    - Pokud nelze některé bloky vypnout jednotlivě (např. pouze celá jádra) je vhodné využít všechny jednotky
  - Snižování počtu přístupů do paměti
  - Velice důležitá alokace registrů a plánování instrukcí

# PŘÍKLADY TECHNIK POWER MANAGEMENTU

# I Sandy Bridge a Haswell – System Agent

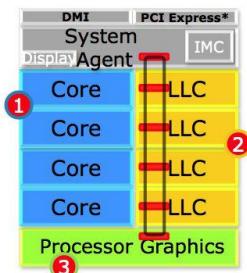
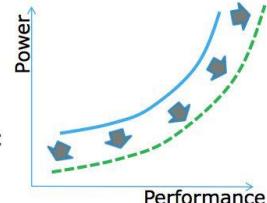
## Lean and Mean System Agent

- Contains PCI Express\*, DMI, Memory Controller, Display Engine...
- Contains **Power Control Unit**
  - Programmable uController, handles all power management and reset functions in the chip
- Smart integration** with the ring
  - Provides cores/Graphics /Media with high BW, low latency to DRAM/IO for best performance
  - Handles IO-to-cache coherency
- Separate **voltage and frequency** from ring/cores, **Display integration** for better battery life
- Extensive **power and thermal management** for PCI Express\* and DDR



## Maximizing Power-Limited Performance

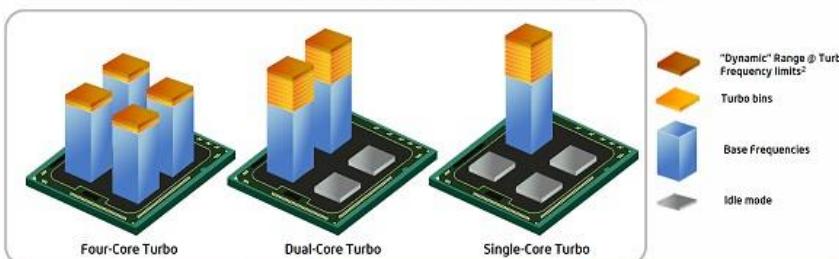
- Extended operating range
  - Power efficient features: better than voltage / frequency scaling
  - Continued focus on gating unused logic and low-power modes
  - Optimized manufacturing and circuits
- Independent frequency domains
  - Cores separated from LLC+Ring for fine-grained control
  - Power Control Unit dynamically allocates budget when power-limited
  - Prioritization based on run-time characteristics selects domain with the highest performance return



- Do čipu jdou pouze 2 různá napětí.
- Procesor již sám vytváří všechna další
- V<sub>DDQ</sub> napájí paměťový řadič
- V<sub>CCIN</sub> napájí zbytek procesoru

- Technologie umožňující vyšší výkon pro sekvenční aplikace ( jádro Nehalem+ )
  - Aktivuje se pokud některé jádro pracuje na plný výkon, ale nejsou vytížena všechna jádra
  - Zvyšování frekvence je závislé na
    - Počtu aktivních jader
    - Odhadovaném proudovém zatížení
    - Odhadované spotřebě příkonu
    - Teplotě procesoru

## Intel® Turbo Boost Technology 2.0



### Efficient.

Adapts by varying turbo frequency to conserve energy depending upon the type of instructions

### Dynamic.

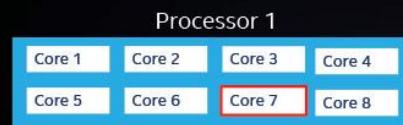
Boosts power level to achieve performance gains for high intensity "dynamic" workloads

### Intelligent.

Power averaging algorithm manages power and thermal headroom to optimize performance

Intel® Turbo Boost Technology 2.0 delivers intelligent and energy efficient performance on demand

## INTEL® TURBO BOOST MAX TECHNOLOGY 3.0



VS



Best Core

- In-Die Variation naturally produces parts with some cores that are faster than others (higher performance/ lower voltage)
- Intel® Turbo Boost Max Technology 3.0
  - Identifies the best performing core to provide increased single threaded performance
  - Requires OS awareness or Intel's core affinization driver to get the performance benefits.
- Processor continues to operate within specifications/warranty, this is not "overclocking"

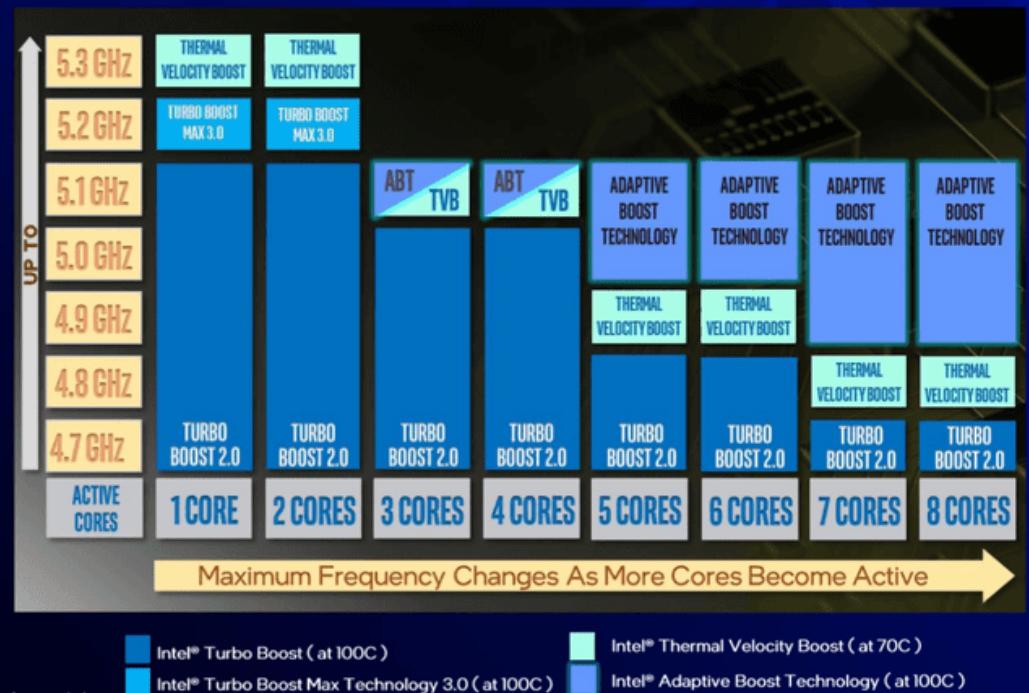
**Intel® Turbo Boost Max Technology 3.0 improves single thread performance more than 15% vs. previous gen<sup>1</sup>**

## Intel® Adaptive Boost Technology Unleashing Multi-Core Turbo Performance

Intel Adaptive Boost Technology improves the 11th Gen Intel® Core™ i9 K and KF desktop processors performance by opportunistically allowing higher multi-core turbo frequencies.

In systems equipped with enhanced power delivery and cooling solutions, Intel Adaptive Boost Technology allows additional multi-core turbo frequency while still within the spec's current and temperature limits.

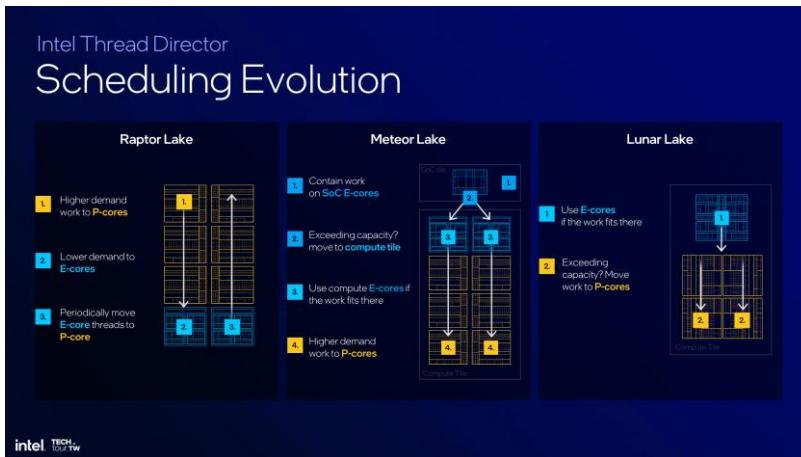
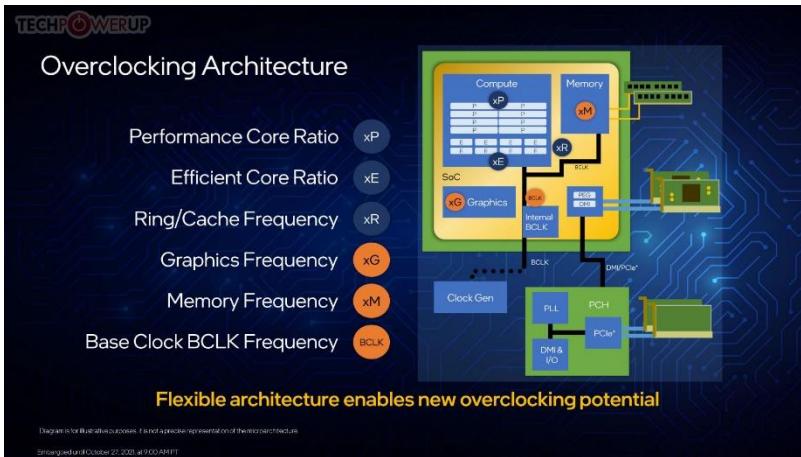
Like past Intel turbo technologies, Intel® Adaptive Boost Technology will be within spec operation and is not considered overclocking.



Intel® Adaptive Boost Technology Disclaimer: When enabled, Intel® Adaptive Boost Technology (Intel® ABT) is a feature that opportunistically allows additional multi-core Intel® Turbo Boost Technology frequencies, while operating within system power and temperature specifications. While current, power and thermal specification headroom exists. The frequency gain and duration is dependent on the workload, capabilities of the processor and the processor cooling solution.

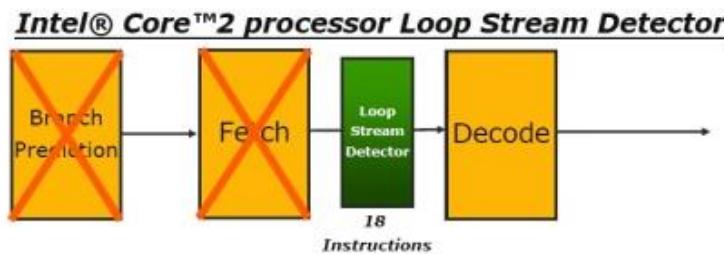
Performance varies by use, configuration and other factors. Learn more at [www.intel.com/PerformanceIndex](http://www.intel.com/PerformanceIndex).

# Overclocking Architecture and Thread Scheduling



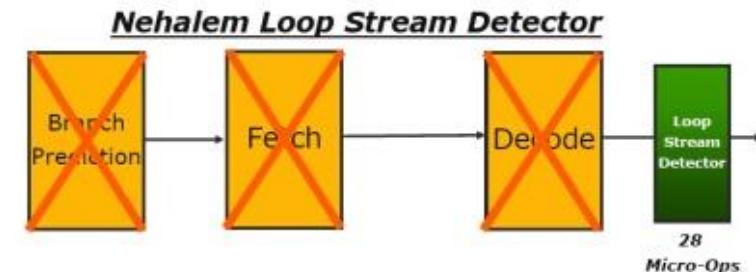
## Loop Stream Detector Reminder

- Loops are very common in most software
- Take advantage of knowledge of loops in HW
  - Decoding the same instructions over and over
  - Making the same branch predictions over and over
- Loop Stream Detector identifies software loops
  - Stream from Loop Stream Detector instead of normal path
  - Disable unneeded blocks of logic for **power savings**
  - **Higher performance** by removing instruction fetch limitations



## Nehalem Loop Stream Detector

- Same concept as in prior implementations
- **Higher performance:** Expand the size of the loops detected
- **Improved power efficiency:** Disable even more logic



- U starších čipů (Core 2, Nehalem) jsou data součástí mikroinstrukce a cestují po OOO enginu společně s instrukcí (RoB).
- Fyzický registrový soubor drží data na jednom místě
- V instrukci je tak pouze pointer na fyzický registr.

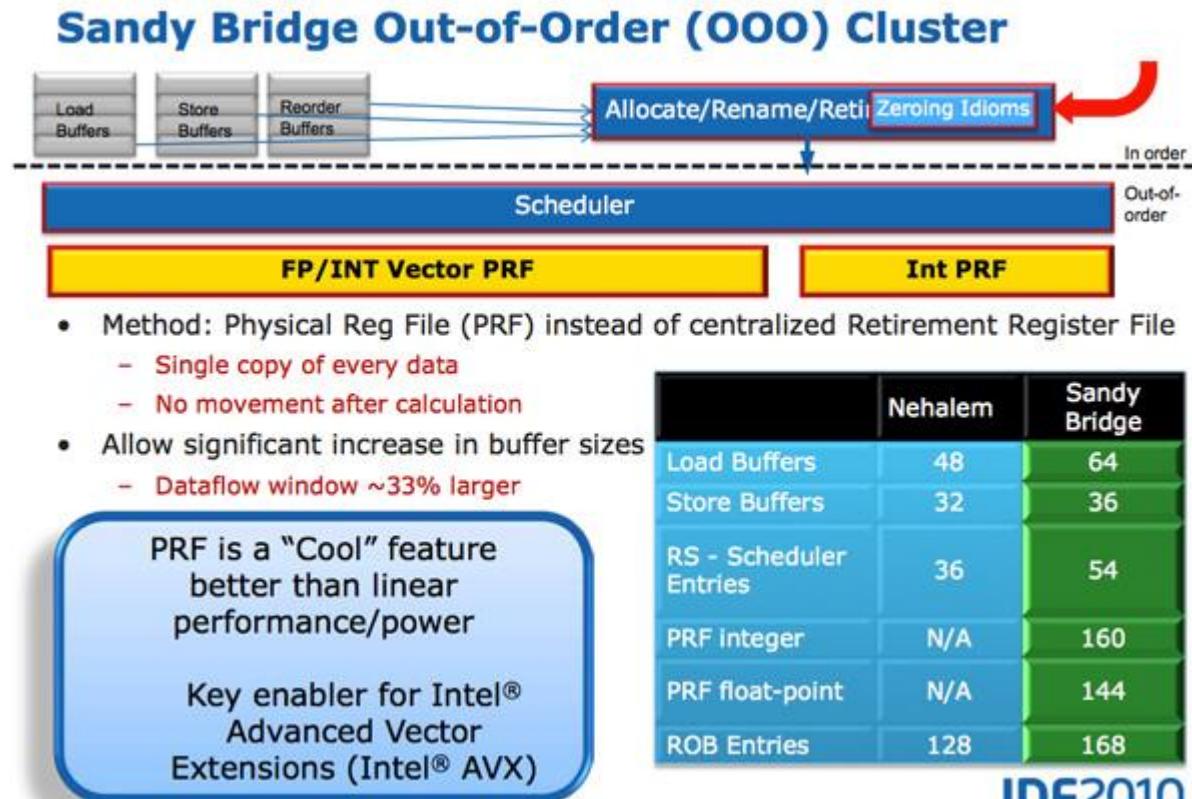
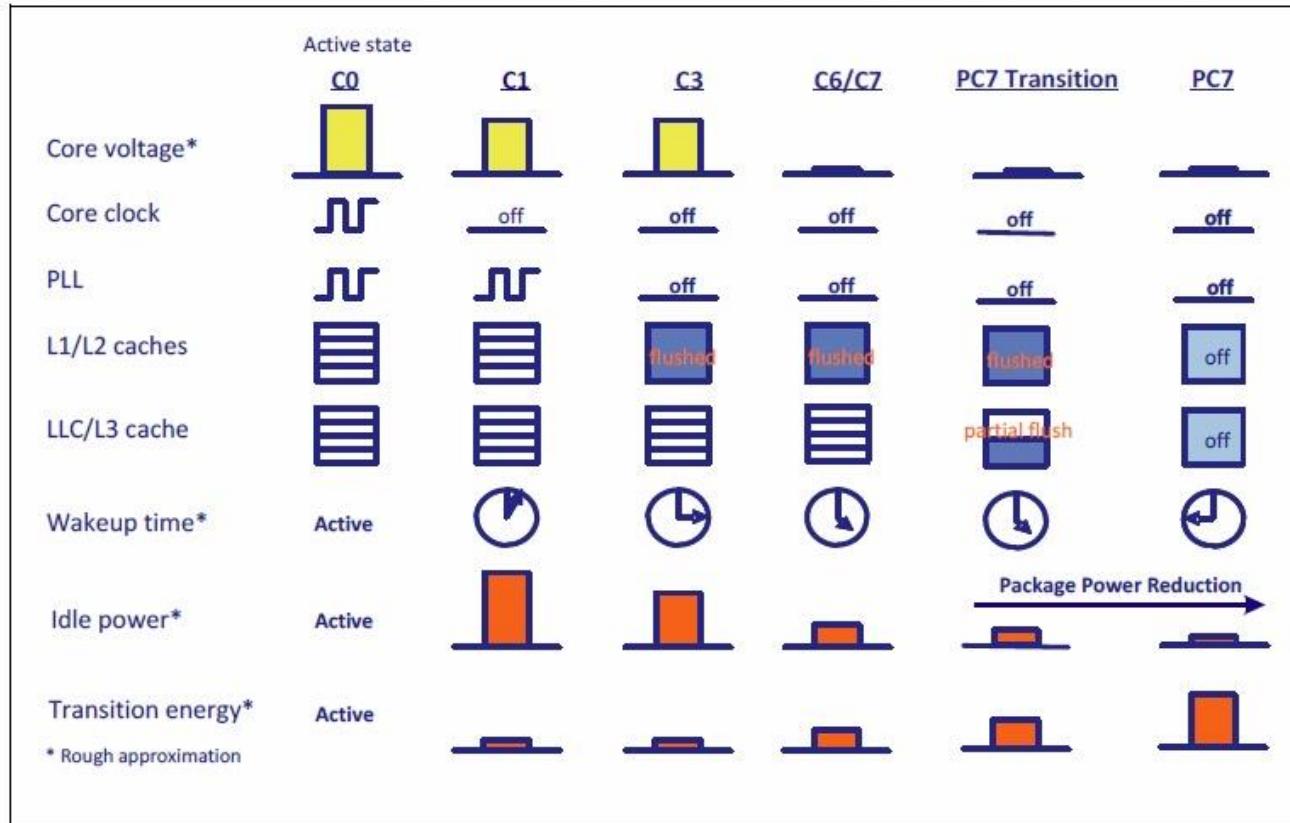
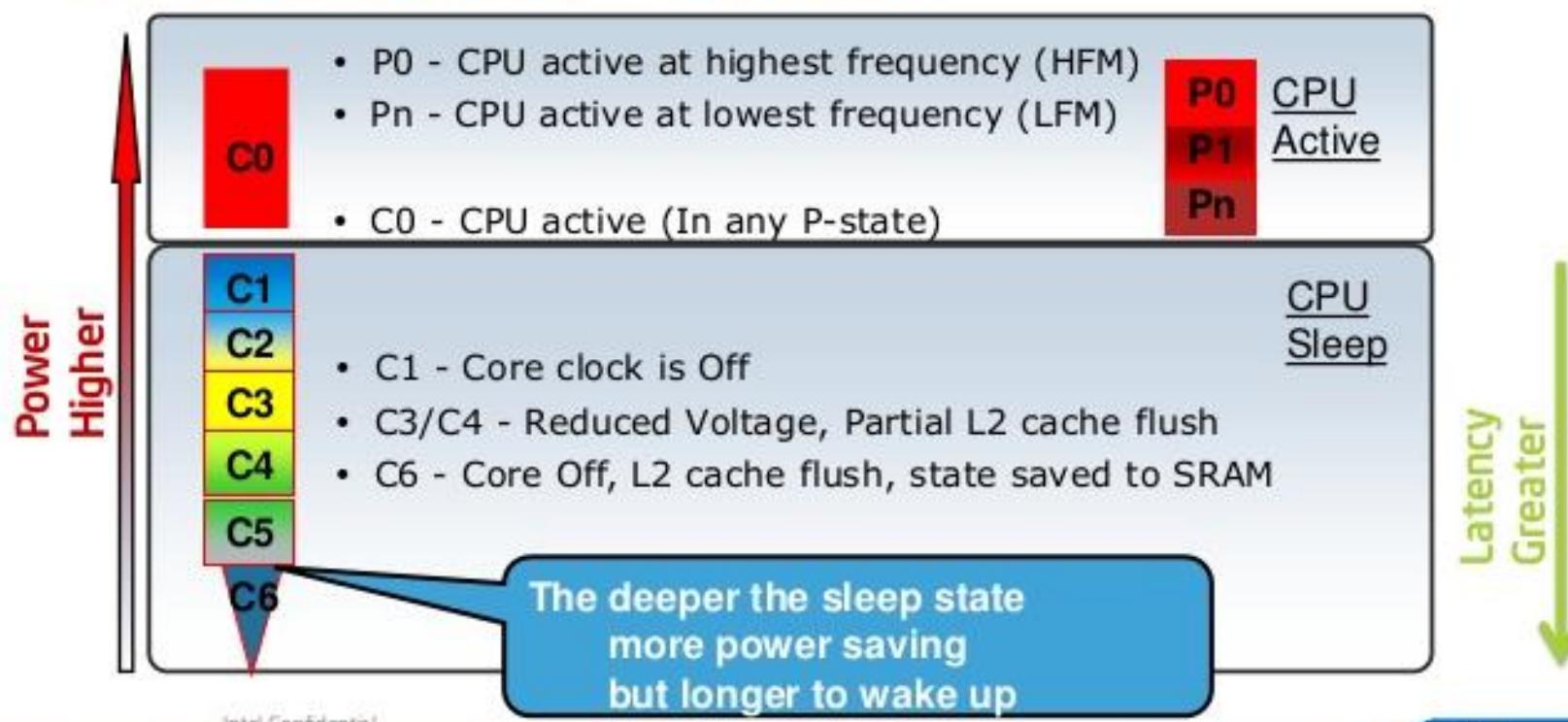


Figure 4: Flexible C-states to select Idle Power Level vs. Responsiveness



## CPU C-States / P-States

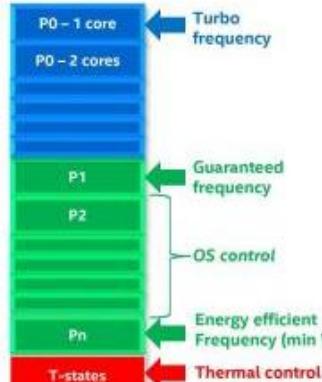


Intel Confidential

# Intel Speed Shift (Skylake)

## P-state

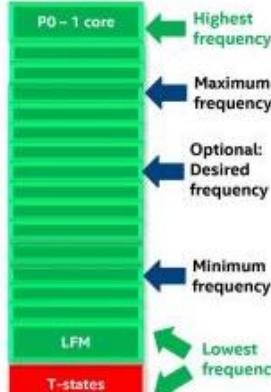
(All CPUs, plus Skylake)



Fixed, Limited  
Discrete P-states

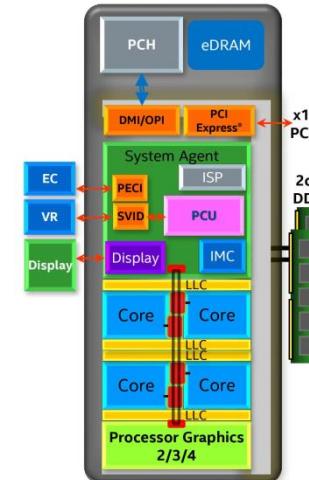
## Speed Shift

(Skylake Only)

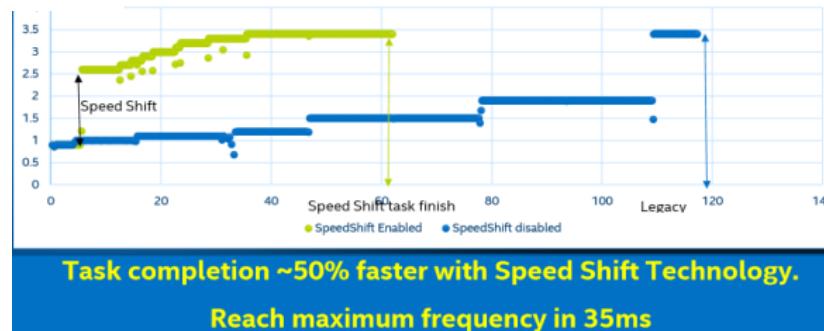


Full Multiplier Range

- P-state defined in BIOS, Managed by OS
- Speed Shift gives full or partial frequency control back to CPU
  - Can give full range or narrow window
- Speed Shift requires OS support
  - Windows 10 via new update
  - Others perhaps later
- Speed Shift means quicker response to performance burst requests
  - Javascript, Office Tools, Web Browsing
- Performance increase in regular tasks
- Slight overall power reduction
- Requires any Intel 6<sup>th</sup> Gen Skylake CPU
- Collaboration between Intel and Microsoft specifically for W10 + Skylake



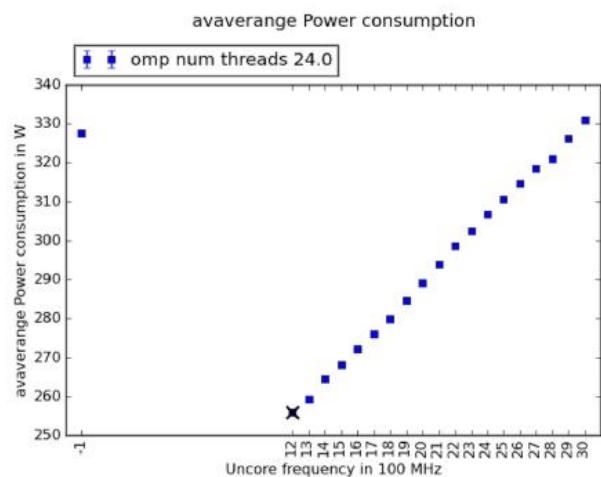
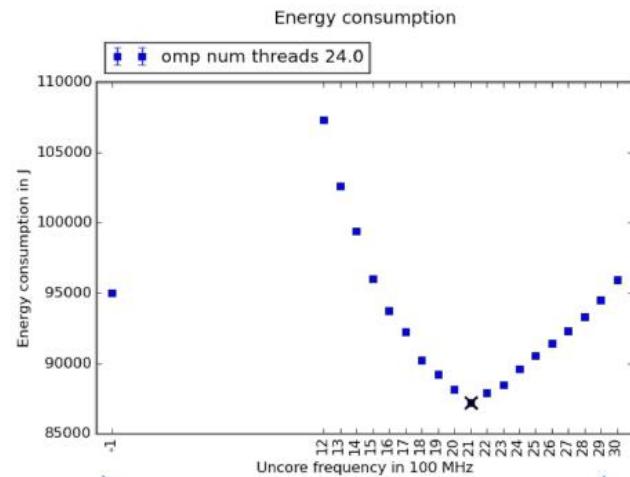
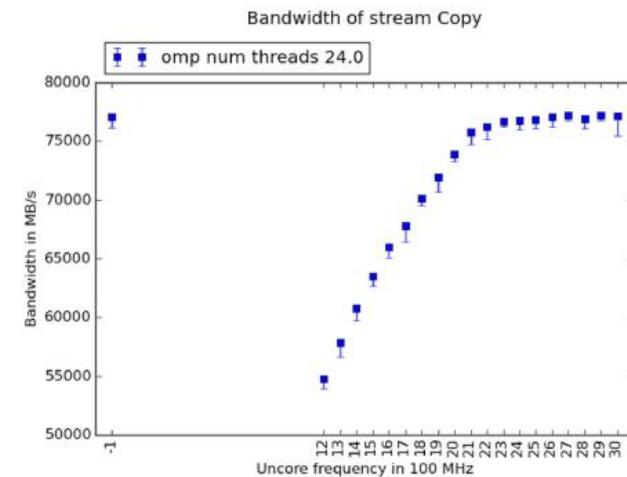
Speed Shift Frequency transitions  
Frequency (GHz) vs Time (ms)  
Representative task with and without Speed Shift Technology



- **Taktování pod Linuxem**
  - Zjištění aktuálního nastavení
    - `cpupower frequency-info`
  - Nastavení řídicího governoru
    - `sudo cpupower frequency-set --governor performance`
  - Nastavení rozsahu frekvence
    - `sudo cpupower frequency-set --min 1600MHz --max 2000MHz`
- **Měření spotřeby – nástroj *Intel Performance Counter Monitor***
  - `sudo modprobe msr`
  - `sudo ./pcm-power.x -- ./my_app my_arguments`
- Alternativně s pomocí knihovny PAPI RAPL

- Vliv CPU uncore frekvence (Cache, propojení jader, řadič paměti) na propustnost stream benchmarku a spotřebu energie
  - Optimální frekvence má minimální dopad na výkon, ale uspoří energii

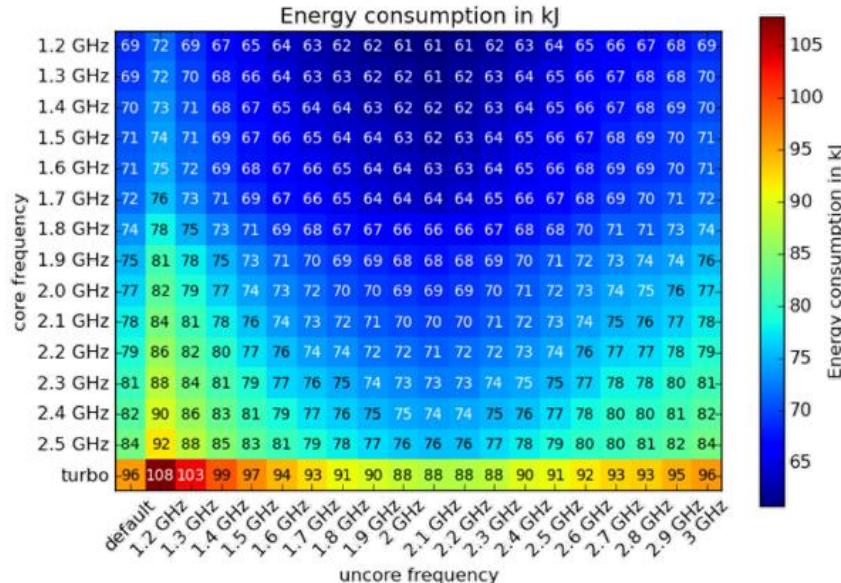
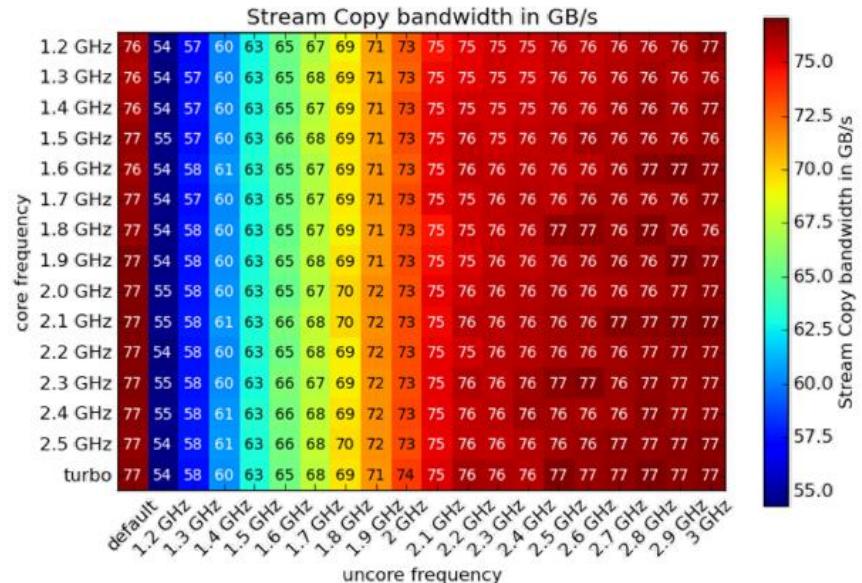
Evaluation using STREAM Copy benchmark



# | Praktické ukázky – Benchmark stream (memcpy) | FIT

- Vliv změny core frekvence (jádra) a uncore frekvence (paměť) pro memory bound problémy (nízká aritmetická intenzita)

Evaluation using STREAM Copy benchmark

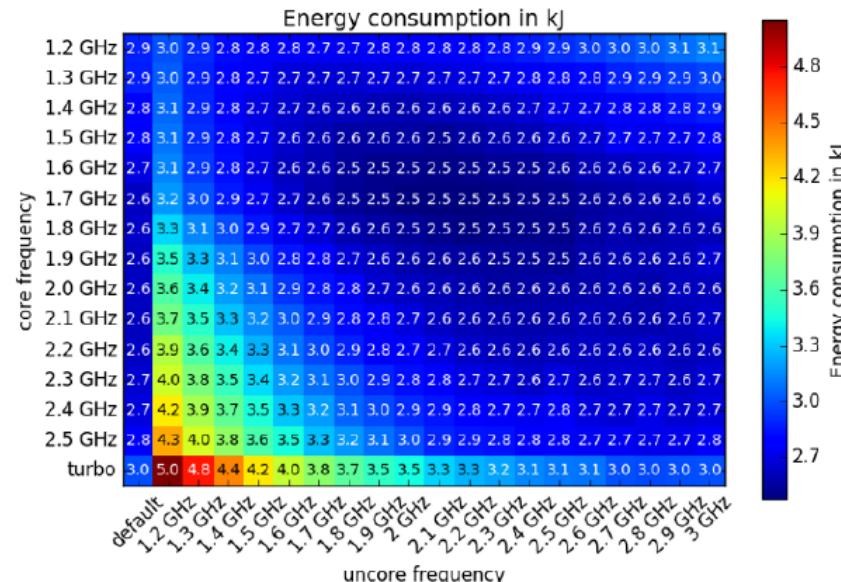
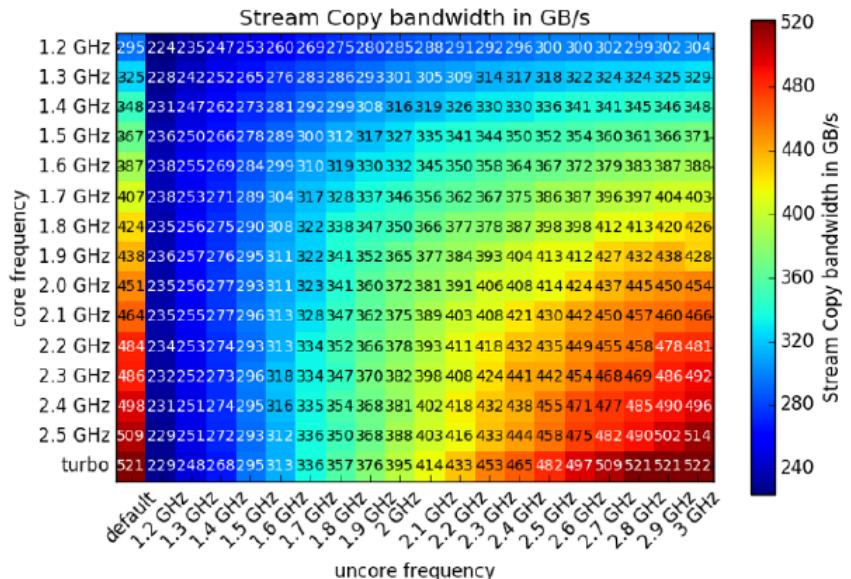


Heatmap of the energy consumption of a stream benchmark for different core and uncore frequencies.  
The data array does not fit in the processor's L3 processor cache

- Efektivita práce s L3 cache při různých core a uncore frekvencích**

- Data se vlezou do cache

Evaluation using STREAM Copy benchmark



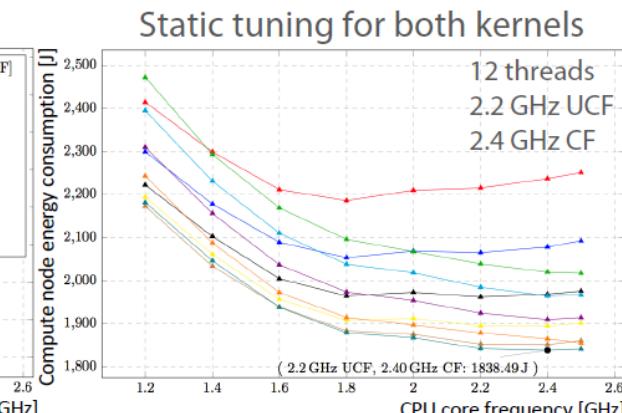
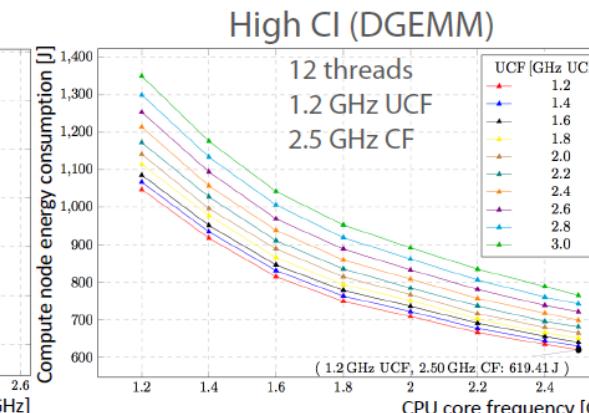
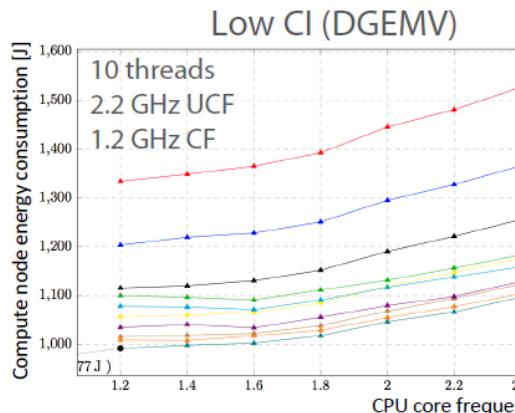
Heatmap of the energy consumption of a stream benchmark for different core and uncore frequencies.  
The data array does fit in the processor's L3 processor cache

# I Hledání optimálních parametrů procesoru

Behavior of the simple application with two kernels

- Low computational intensity – DGEMV
- High computational intensity – DGEMM
- Tuning of three parameters
  - Core frequency
  - Uncore frequency
  - Number of OpenMP threads
- Visualized by RADAR

| Two kernels with 1:1 workload ratio | Energy consumption | Energy savings |
|-------------------------------------|--------------------|----------------|
| Default settings                    | 2017J              | - -            |
| Static optimal                      | 1833J              | 179J 9%        |
| Dynamic optimal                     | 1612J              | 221J 12%       |
| Total savings                       | -                  | 400J 20%       |



Note: runtime of both kernels was equal for default settings

# HISTORIE A PŘÍKLADY SUPERSKALARNÍCH ARCHITEKTUR

- **P5 (1993):**

- první superskalární IA-32 mikroarchitektura – 1993:
  - In Order, dvojitá integer pipeline (**U** a **V**) 5 stupňů.
  - Dokončují až 2 instrukce/takt. Kompilátor plánoval dvojice staticky.

- **P6 (1995):**

- **OOO**, zavedeno **super-řetězení** (14 stupňů).
- **Procesory**: Pentium Pro, Pentium II, III; MMX a SSE.
- **Modernizovaná P6**: Pentium M, Core Solo, Core Duo.

- **NetBurst (2000):**

- Trace cache, 31 stupňů, SSE2, SSE3, hyper-threading HT, EM64T.
- **Procesory**: Pentium 4, Pentium D, Xeon

- **Core (2006):**

- příkon ↓, 14 stupňů pipeline, 65 nm, multi-core, SSE3, Intel 64
- Procesory: Pentium dual core, Celeron, Xeon, Core 2

- **Nehalem (2008): řady: i3, i5, i7**

- 45 nm, HT, L3C, Quick Path, integrované MemCtrl, bufer μops.
- 32 nm Nehalem = Westmere: IGP (Integrated GPU).

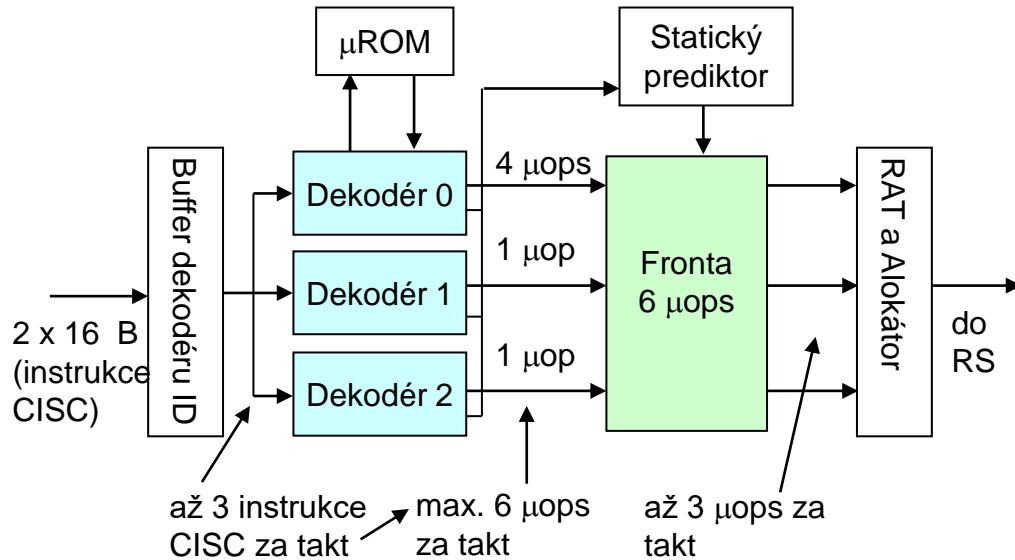
- **Sandy Bridge 2010:**

- 32 nm, AVX 256 bitů, μop-cache, HT.
- 22 nm Sandy Bridge = Ivy Bridge: 3D-tranzistor.

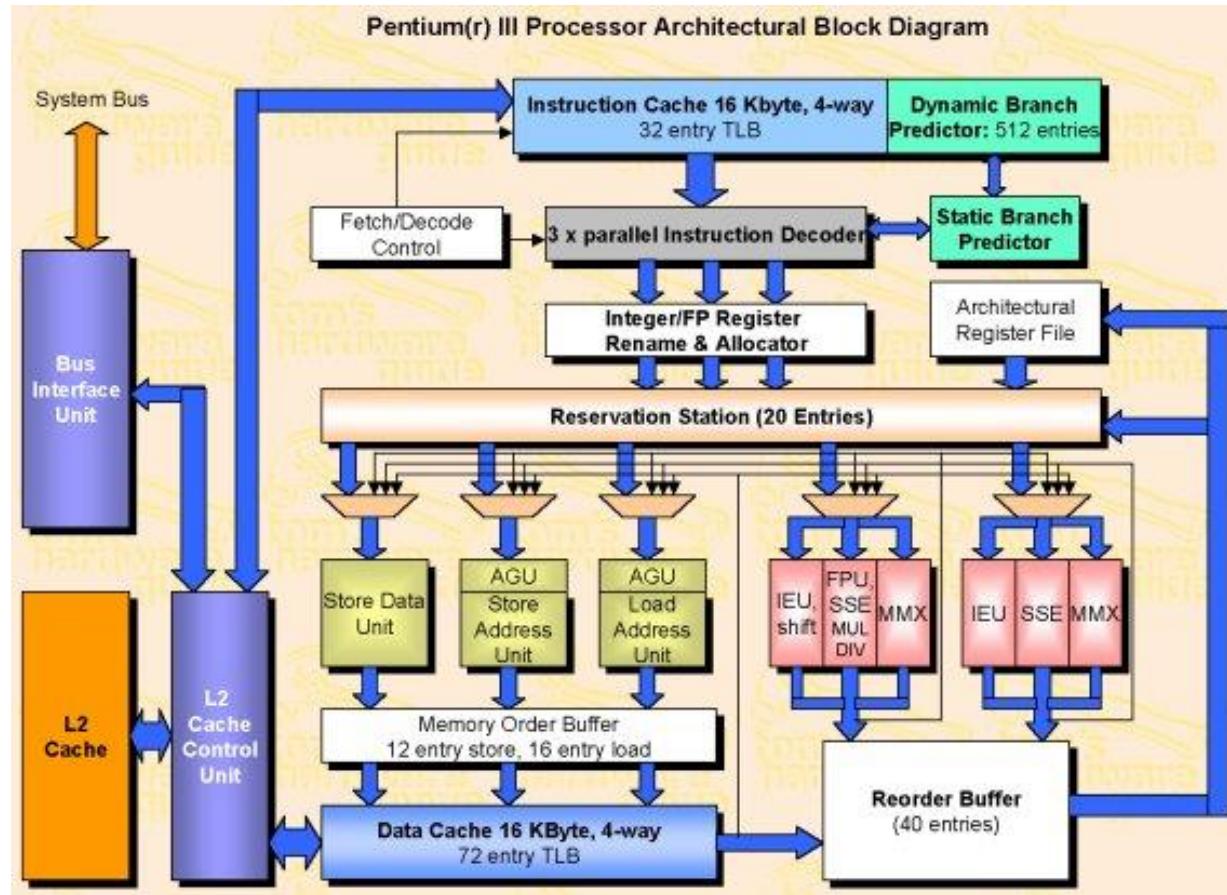
- Haswell 2013:
  - 22 nm, 4 ALU, 3 AGU, 2 jednotky predikce skoků, AVX2, FIVR (Fully Integrated Voltage Regulator)
  - 35–40 MB LLC. Server procesory až 20 jader, možnost rozdělit jádra do 2 uzlů NUMA (COD, cluster on die)
  - 4 verze integrované GPU (až 40 EU), TDP 35–140 W.
  - 14 nm Sandy Bridge = Broadwell
- Skylake 2015:
  - 14 nm, 4 typy Y, U, H a S (TDP 4–95W) integrovaná L4 eDRAM cache (64/128 MB), podpora DDR3/4
  - Změna struktury cache – výrazný nárůst L2 na úkor L3
- Sunny Cove 2015:
  - Zvětšena trace cache, 384 položek ROB, Rozšířeny L/S jednotky a L/S bufery
- Alder Lake 2021:
  - Heterogenní procesor – silná (Golden Cove) a slabá jádra (Gracemont)
  - Výrazný nárůst zdrojů na silných jádrech (512 položek ROB, 12 exekučních portů, 6-wide dekodér)
  - AVX-512 vypnuto (slabá jádra totiž nepodporují)

|                                   |                           |       |
|-----------------------------------|---------------------------|-------|
| P5 (Pentium)                      | superskalární, „in-order“ | 5     |
| P6 (Pentium Pro)                  |                           | 14    |
| P6 (Pentium III)                  |                           | 10    |
| NetBurst Pentium 4 (180 a 130 nm) |                           | 20    |
| NetBurst Pentium 4 (90 a 45 nm)   |                           | 31    |
| Core                              | superskalární             | 14    |
| Nehalem                           | „out-of-order“            | 16    |
| Sandy Bridge                      |                           | 14–19 |
| Ivy Bridge                        |                           | 14–19 |
| Haswell                           |                           | 14–19 |
| Bonnell (Atom)                    |                           | 16    |
| Quark                             | skalární                  | 5     |

- Řetězené zpracování CISC-ových instrukcí x86 se řeší transformací (dekódováním) na RISC-ové mikrooperace délky 72 bitů.
- Délka instrukcí x86: 1–15 B, **dekodér délky instrukcí** posílá až 3 instrukce x86 na 3 dekodéry:
  - D0 zpracovává 1. instrukci, která generuje až 4 µop/takt.
  - D1 a D2 zpracovávají jednodušší 2. a 3. instrukci, které nejsou delší než 8 B a generují jen 1 µop.
  - 2. a 3. instrukce musí čekat na D0, pokud to nesplňují.
  - Pro dekódování instrukcí, které generují více než 4 µop je použita paměť mikrokódu a generování trvá 2 nebo více taktů.



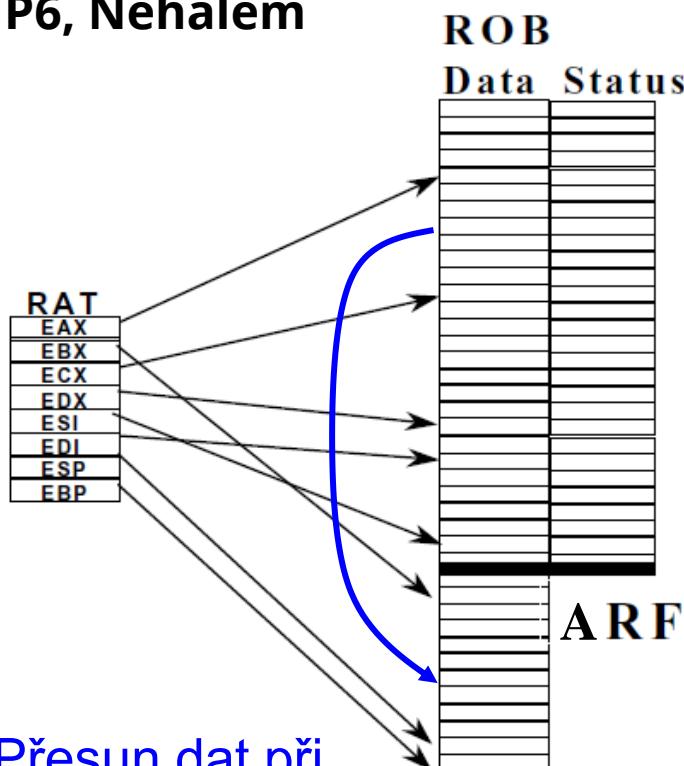
- Prediktor podmínky je 2 úrovňový adaptivní (autoři Yeh a Patt) s  $k = 4$ -bitovým lokálním BHSR
  - perfektně predikuje libovolné periodické sekvence až  $k + 1 = 5$  bitů
  - na 1 skok je třeba 36 bitů (= 16 dvoubitových prediktorů + 4 bity BHSR).
- BTB je organizován jako skupinově asociativní cache (128 skupin, 4 cesty, tj. 512 položek)
- Položka obsahuje adresu skokové instrukce  $b$ , cílovou adresu skoku  $t$  a 4 bitový lokální BHSR. Jeden index do PHT je část adresy skoku  $b$ , jako druhý index se použije obsah BHSR.
- **Pokuta za špatnou predikci je 10–20 taktů.**
- Není-li skok v BTB, použije se statická predikce (skok v kódu dopředu -, skok dozadu +)



- r. 2000, IA-32 procesor (adresa 32 bitů, instrukce x86)
- SSE2 (Streaming SIMD Extension 2)
- Trace Cache (kapacita 12k  $\mu$ ops, cca 64 bitů /  $\mu$ op
  - TC může rozeslat do RS 3  $\mu$ ops/takt, vydat do FJ se může až 6  $\mu$ ops/takt a propustit opět 3  $\mu$ ops/takt.
- Přejmenování mapuje 8 standardních registrů x86 na 128 vnitřních fyzických registrů PRF, 2 tabulky RAT (front-end a propouštěcí) → není nutno kopírovat registry při propouštění.
- ROB: až 126  $\mu$ ops v pořadí bez hodnot dst operandů
- **Co bylo špatné:** chyběla L3 cache na čipu, malá L1 D-cache (8 KiB), výkonnost  $\approx$  Pentium III, velký příkon

# Přejmenování registrů v P6 a NetBurst

## P6, Nehalem



Přesun dat při propuštění instrukce

## Sandy Bridge

## NetBurst

přepis při špatné predikci skoku

Frontend RAT

|     |
|-----|
| EAX |
| EBX |
| ECX |
| EDX |
| ESI |
| EDI |
| ESP |
| EBP |

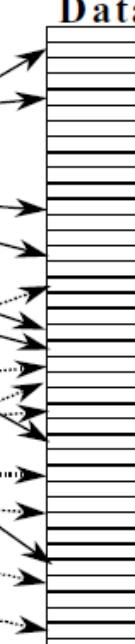
Retirement RAT

|     |
|-----|
| EAX |
| EBX |
| ECX |
| EDX |
| ESI |
| EDI |
| ESP |
| EBP |

## PRF

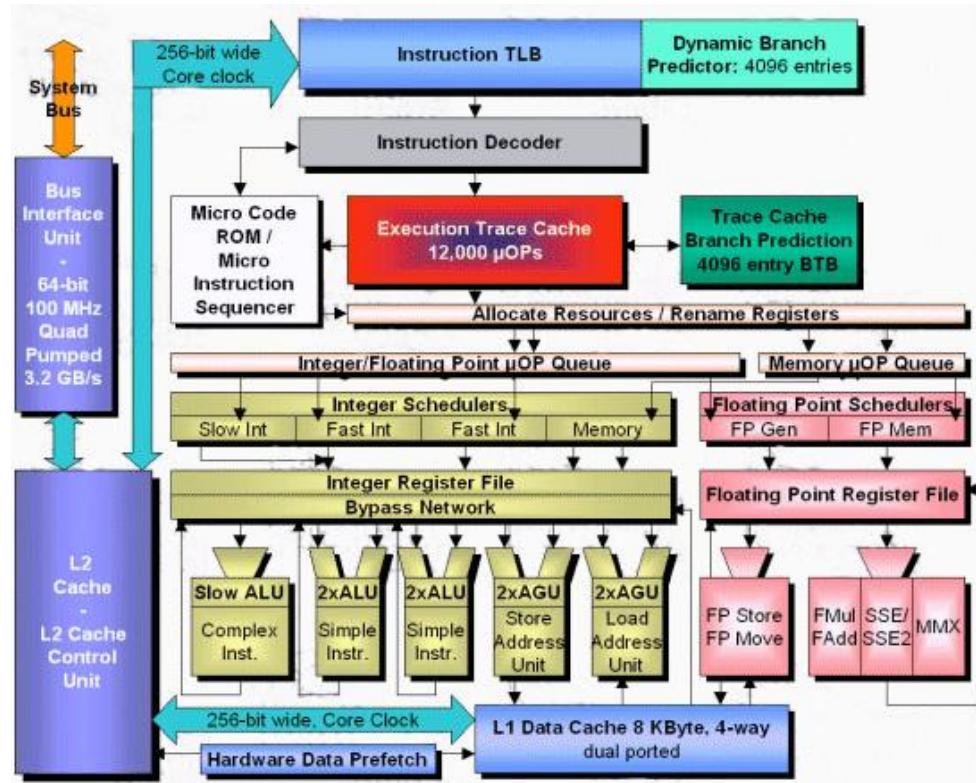
## ROB

Status



Při propuštění instrukce se sem vloží mapování jejího dst registru

# Pentium 4 – architektura NetBurst

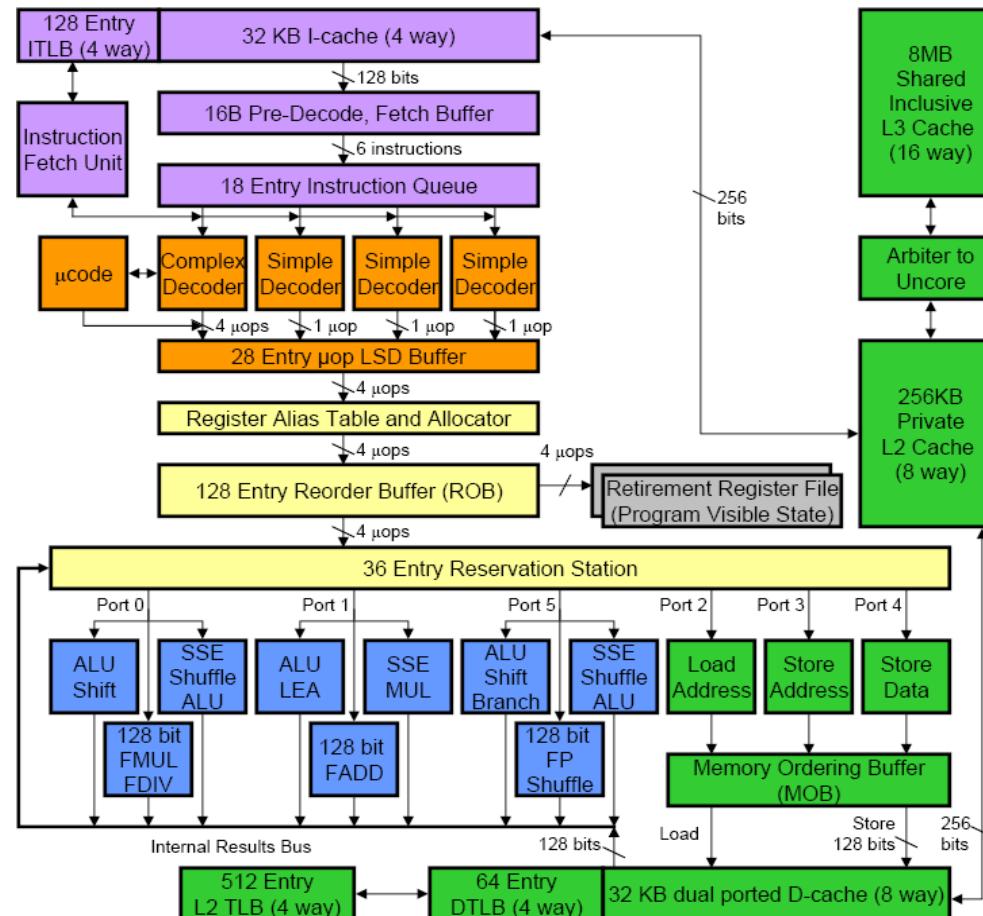


|           |          |             |   |        |     |     |     |     |      |      |    |    |    |      |       |       |    |    |    |
|-----------|----------|-------------|---|--------|-----|-----|-----|-----|------|------|----|----|----|------|-------|-------|----|----|----|
| 1         | 2        | 3           | 4 | 5      | 6   | 7   | 8   | 9   | 10   | 11   | 12 | 13 | 14 | 15   | 16    | 17    | 18 | 19 | 20 |
| TC Nxt IP | TC Fetch | Drive Alloc |   | Rename | Que | Sch | Sch | Sch | Disp | Disp | RF | RF | Ex | Flgs | Br Ck | Drive |    |    |    |

- Použita u vícejádrových procesorů se sdílenou pamětí cache L2. Snížení příkonu a zvýšení výkonnosti činí kolem 40 %. **Vychází z P6.**
- **Široká instrukční linka.**
  - Dekóduje a propouští až **4 instrukce za takt**, rozesílat a provádět může až **5 µops**.
  - Umí sdružovat instrukce x86 (*macrofusion*) a také sdružovat µops vzniklé z jedné x86 (*microfusion*), čímž lze dosáhnout až 6 µops za takt.
  - Ukazovatel zásobníku je modifikován speciálním HW. To dovoluje **načítání dat ze zásobníku již na začátku linky** (25 % všech načítání je ze zásobníku).
  - Tyto inovace zachovány i v novějších mikroarchitekturách
- **Pokročilá práce s multimédii.** Instrukce **MMX, SSE, SSE2, SSE3** se **128 bity** provedené **za 1 takt** znamenají výkonnost až 24 GFLOP/s (1 jádro na 3 GHz, SP).
- **Inteligentní napájení.** Dynamické odpojování subsystémů dle potřeb nebo přepojování do úsporného režimu neovlivňuje responzivitu.
- **Pokročilá chytrá cache.** Sdílená sjednocená cache úrovně 2 může být celá k dispozici jen 1 jádru, když druhé není aktivní. Špičková přenosová rychlosť je 96 GB/sec @ 3 GHz.
- **Chytrý přístup do paměti.** Je zavedena podpora pro **RPW** i pro případ dosud neznámé adresy zápisu (dynamické rozlišování adres, *memory disambiguation*)
- **Přednačítání dat** do L1/L2 D-cache pomocí tabulky historie čtení

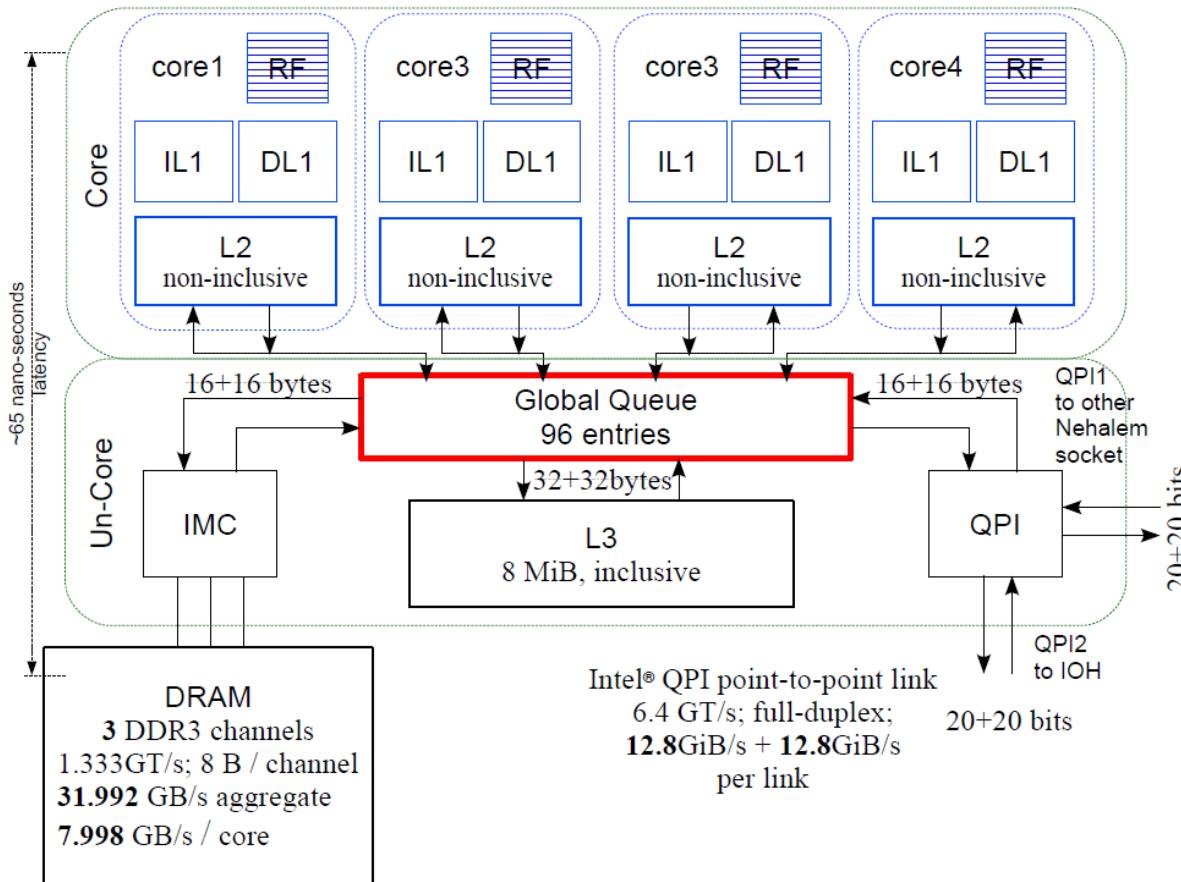
- Druhá generace architektury Core, orientace na výkonnost: 2, 4, 6 nebo 8 jader (45 nm, 4 jádra: 731 M tranzistorů)
- 16 stupňová linka, 6 FJ (3 paměťové, 3 výpočetní) a **HyperThreading (HT)**
- Větší cache a vyšší propustnost pamětí 32 KiB L1 I-cache, 32 KiB L1 D-cache, L2C: 256 KB.
- **Register Alias Table (RAT)** může přejmenovat až 4 μop za takt a každé přidělit dst. registr v ROB více rozpracovaných mikrooperací.
- **Dvouúrovňový prediktor skoků i dvouúrovňový TLB**.
- **Loop Stream Detector LSD**: ve frontě 18 před-dekódovaných instrukcí detekuje každé tělo smyčky, uloží je dekódované do LSD buferu (až 28 μop) a pak opakovaně používá (bez načítání a dekódování) až do špatné předpovědi skoku (malá náhrada Trace cache).
- **Turbo režim**: kmitočet hodin se zvyšuje, pokud není překročena teplotní mez.

# Mikroarchitektura Nehalem

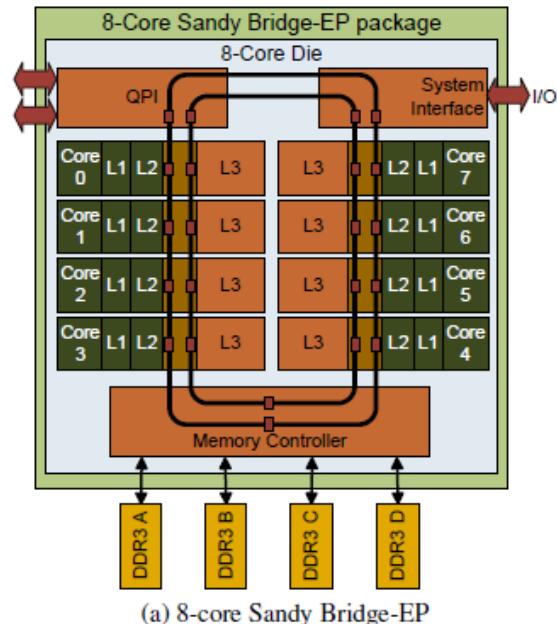


- **Inovace:** nové propojení soketů: front-side bus FSB nahrazen linkami (QPI, Quick Path Interconnect).
- **Inovace:** integrovaný řadič paměti, podporuje 3 paměťové kanály DDR3 SDRAM nebo 4 FB-DIMM, severní most eliminován.
  - Firma AMD zavedla linky HyperTransport (HT) a integrované řadiče paměti již v roce 2003, o 5 let dříve.
- **Sdílená L3C:** 4–8 MB je inkluzivní, obsahuje data z L1 i L2 a info, kde jsou lokálně (menší komunikace).
- **Čip procesoru grafiky** v témže pouzdro jako CPU.
- **Řízení příkonu:** vestavěný mikrořadič a senzory teploty, proudu a příkonu, odepínání jader, možnost redukce příkonu pamětí a QPI.
- Global Queue (GQ) uchovává, spravuje a plánuje tok dat v „uncore“. Má 3 fronty požadavků:
  - WQ, žádosti zápisu z lokálních jader, 16 položek
  - LQ, žádosti čtení z lokálních jader, 32 položek
  - QQ, fronta QPI, žádosti jdoucí mimo čip, 12 položek
- Obsahuje **křížový přepínač** pro výměnu dat mezi propojenými částmi (L2, L3, IMC, QPI).
- **Funkce:**
  - lokální žádost jádra o čtení: GQ sonduje další jádra. Z více vlastníků kopí jedno jádro dodá data.
  - Když nikdo nemá kopii a L3 ano, dodá data inkluzivní L3
  - Výpadek v L3: data dodá lokální IMC (Integrated Memory Controller) za 65 ns, popřípadě vzdálený IMC přes QPI za 105 ns

# Celkový diagram procesoru Intel Nehalem



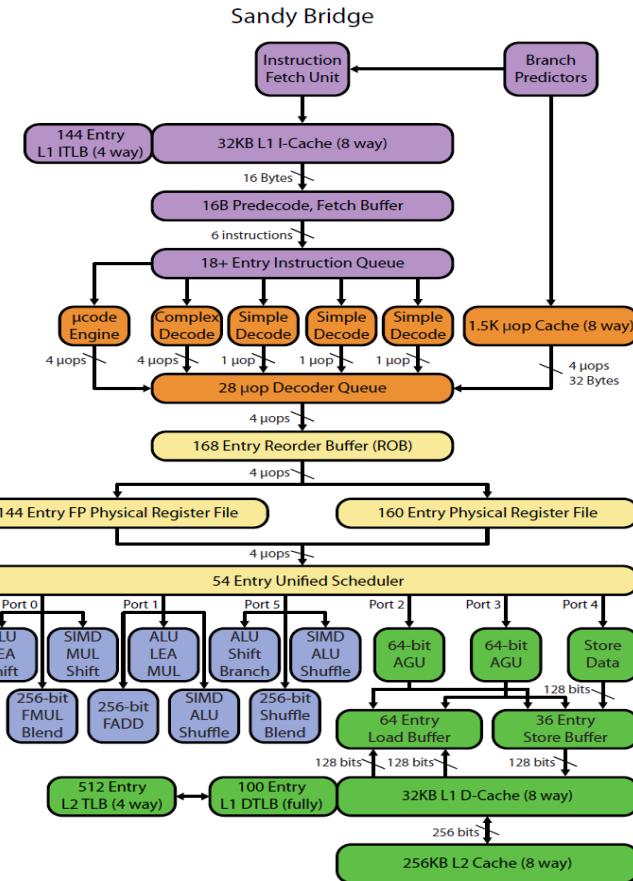
- 4, 6 a 8 jader na 3,0–3,8 GHz s podporou HyperThreadingu (HT) a s technologií Turbo Boost
- Jádra, grafika, L3 cache a systémový agent jsou propojeny **kružnicovým propojením** s propustností 256 bitů/takt.
- Podpora **Advanced Vector Extension (AVX): 256 bitů, 32 GFLOPS/jádro (8 FP/takt),**
- Každé jádro: 32 KiB L1 D-cache + 32 KiB L1 I-cache (3 takty), 256 KiB L2 cache (8 taktů).
- 8 MiB **sdílená L3 cache** (25 takty). Je též sdílena s integrovaným grafickým jádrem. Blok cache 64 byte.
- **Integrované jádro grafiky na 1–1,4 GHz, 16 ex. jednotek.**
- **Integrovaný řadič paměti** s max. propustností 25,6 GB/s, podporuje DDR3-1600 dual channel RAM.
- CPU ↔ L3 cache: průměrně jen 1,5 skoků (do lokálního bloku cache není třeba jít po kružnici). Latence sdílené L3 cache je 26 až 31 taktů.



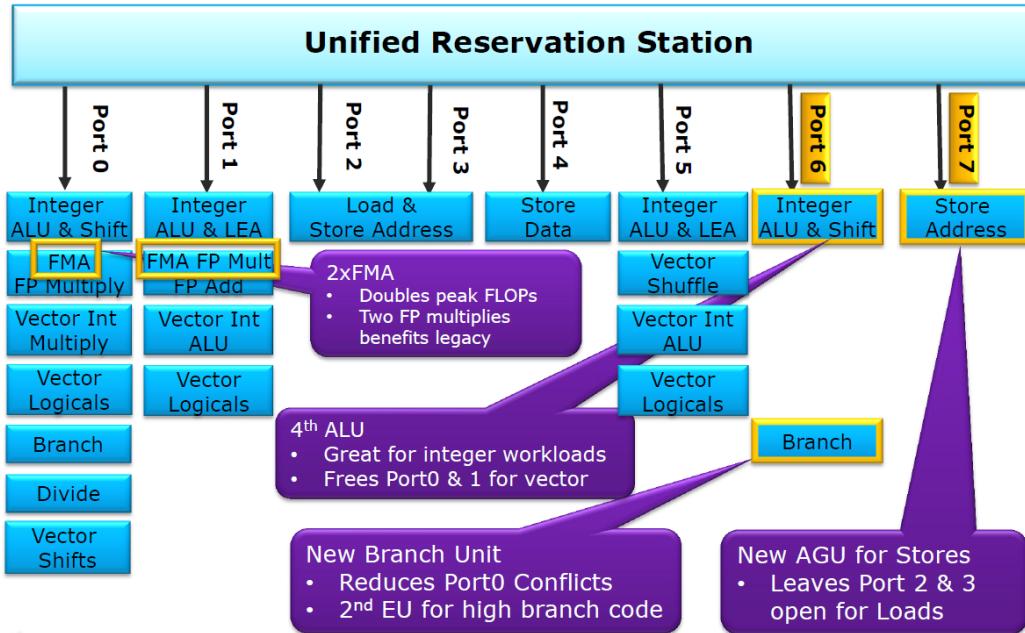
Dvě propojovací kružnice, jedna pro směr nahoru, druhá dolů. Pro každý přenos je vybrán směr kratší cesty do cíle, nejvíce 4 skoky.

# $\mu$ op-cache (dekódovaná I-cache) v Sandy Bridge

- Je částí L1 I-cache, zachovává výhody Trace cache, eliminuje složité dekódování při mnohem nižším příkonu.
- $\mu$ op-cache má kapacitu 1536  $\mu$ ops, 10 % velikosti Trace cache Pentia 4.
- Mapování instrukcí do  $\mu$ op-cache probíhá po blocích 32 B instrukcí, 1 blok může zabrat až 18  $\mu$ ops. Každý blok  $\mu$ op-cache uchovává „metadata“ včetně počtu platných  $\mu$ ops v bloku a délku odpovídajících x86 instrukcí.
- Jestliže okénko 32 B instrukcí má více než 18  $\mu$ ops, musí jít přes tradiční front-end.
- **Mikrokódované** instrukce nejsou v  $\mu$ op-cache – jsou reprezentovány ptr do ROM mikrokódu a případně několika prvními  $\mu$ ops.

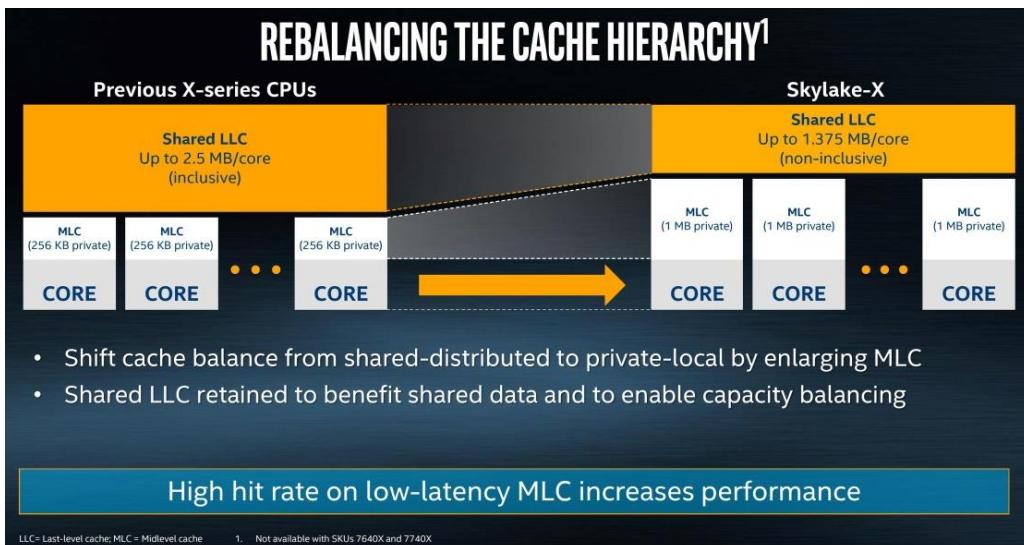


- Haswell je zaměřen na **nižší příkon pro mobilní zařízení** (hybridní laptop-tablety) ale i pro superpočítáče. Dřívejší TDP (Thermal Design Power) 35 až 45 W pro mobilní procesory je redukován na ULT: 13,5 W a 15 W TDP, ULTX: 10 W TDP.
- Superpočítáč v Ostravě obsahuje 24 192 jader Haswell-EP!
- Podpora pro AVX2 a MAD operace.**
- Haswell má výkonnější grafiku GT3e, ze 16 jednotek na 1150 MHz (GT1 u Sandy Bridge) narostla na 40 ex. jednotek a 1300 MHz.
- eDRAM (embedded DRAM) 128 MiB je na vlastním čipu, ale ve stejném pouzdru jako procesor. Pracuje jako sdílená L4 cache jak pro grafiku, tak pro jádra procesoru. Vylepšuje paměťovou propustnost.



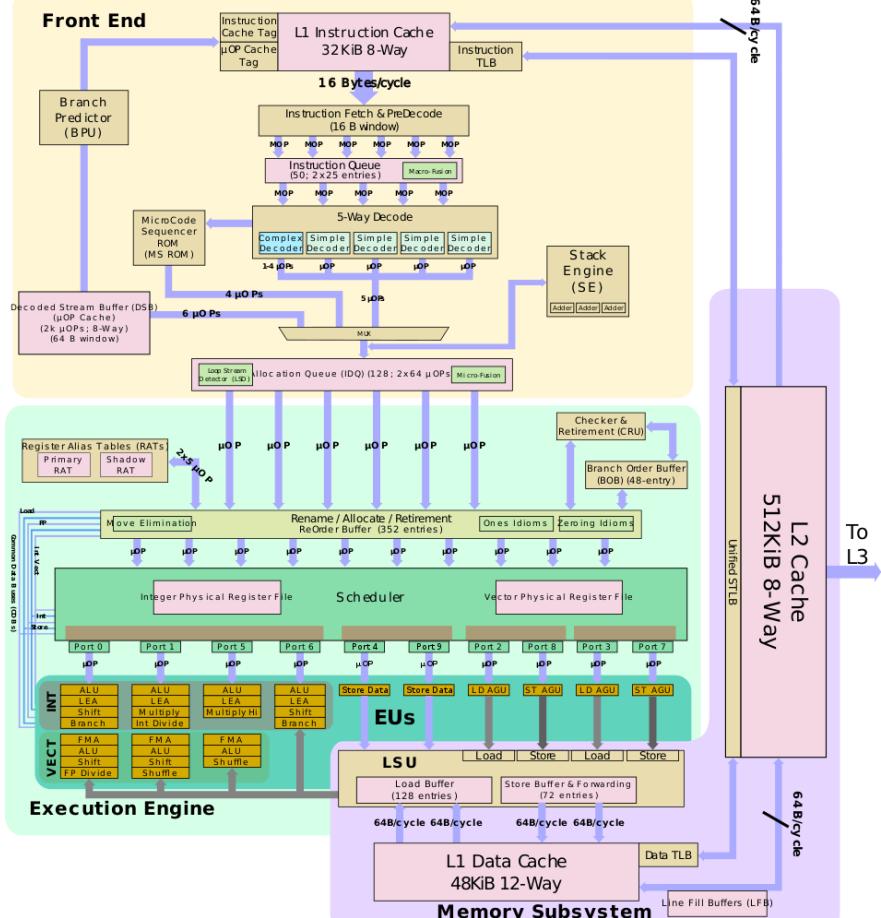
- Broadwell = Haswell předělaný na 14 nm (2014).

- Podpora pro AVX-512**
- Nová organizace cache**
  - Zvětšení L2 cache na úkor L3 cache
- Změna propojovací sítě**
  - Z hierarchických kruhů na 2D mřížku



| Comparison: Skylake-S and Skylake-SP Caches                              |          |                                                                            |
|--------------------------------------------------------------------------|----------|----------------------------------------------------------------------------|
| Skylake-S                                                                | Features | Skylake-SP                                                                 |
| 32 KB<br>8-way<br>4-cycle<br>4KB 64-entry<br>4-way TLB                   | L1-D     | 32 KB<br>8-way<br>4-cycle<br>4KB 64-entry<br>4-way TLB                     |
| 32 KB<br>8-way<br>4KB 128-entry<br>8-way TLB                             | L1-I     | 32 KB<br>8-way<br>4KB 128-entry<br>8-way TLB                               |
| 256 KB<br>4-way<br>11-cycle<br>4KB 1536-entry<br>12-way TLB<br>Inclusive | L2       | 1 MB<br>16-way<br>11-13 cycle<br>4KB 1536-entry<br>12-way TLB<br>Inclusive |
| < 2 MB/core<br>Up to 16-way<br>44-cycle<br>Inclusive                     | L3       | 1.375 MB/core<br>11-way<br>77-cycle<br>Non-inclusive                       |

# Architektura SunnyCove (2019)



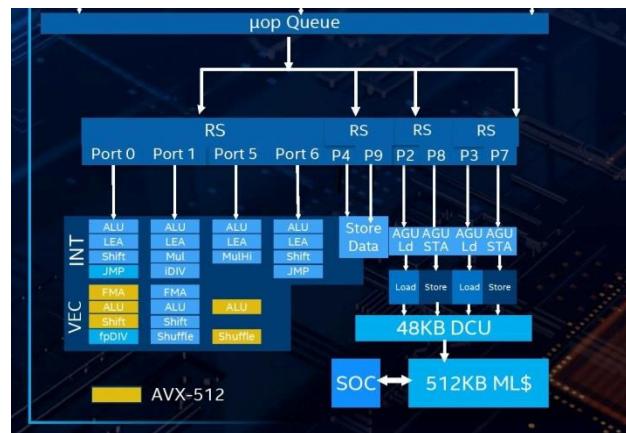
## Změna velikostí cache

- L1 z 32 KB → 48 KB
- L2 z 256 KB → 512 KB
- Zvětšena TraceCache (2,25k)

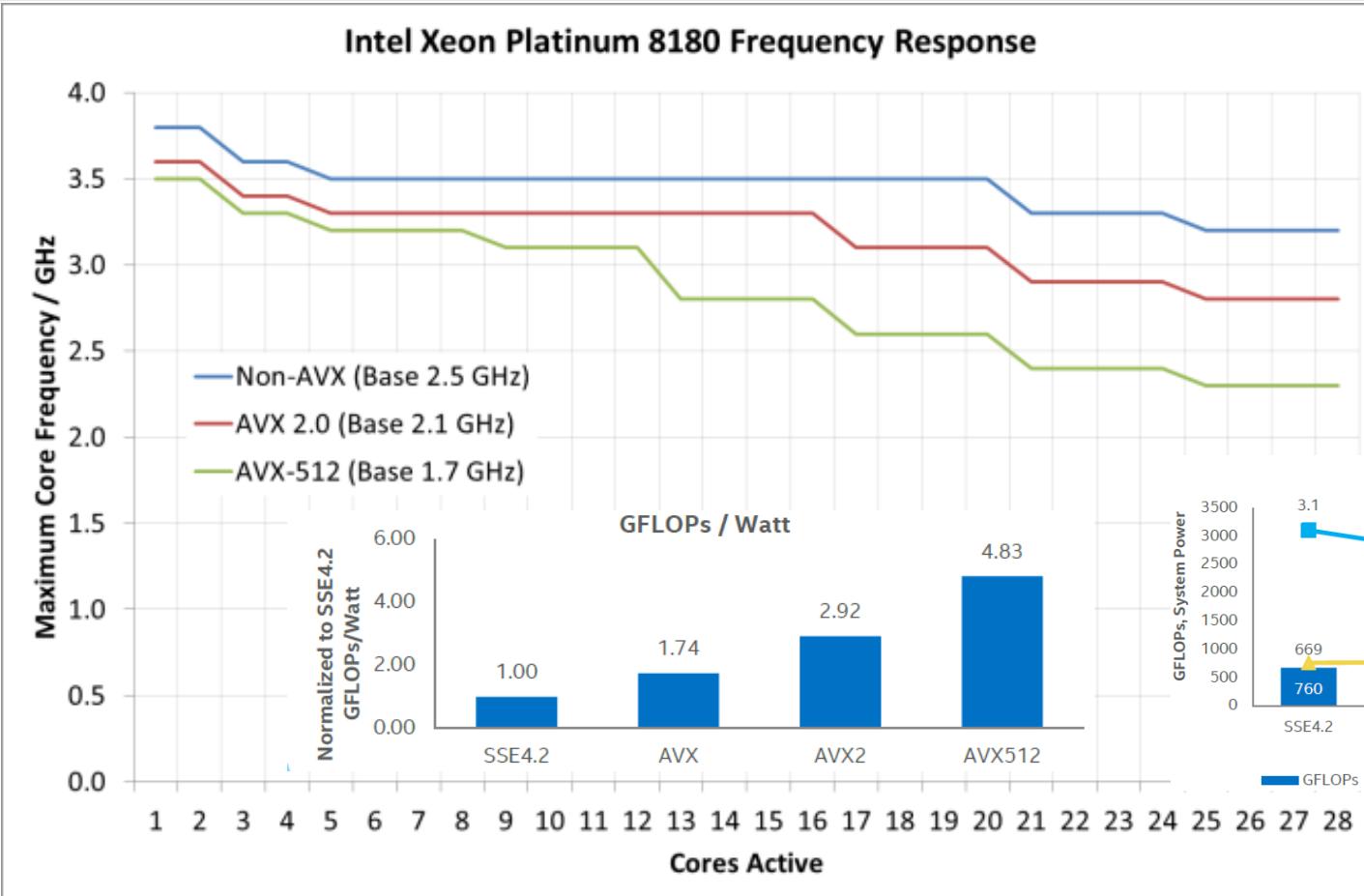


## Back-end

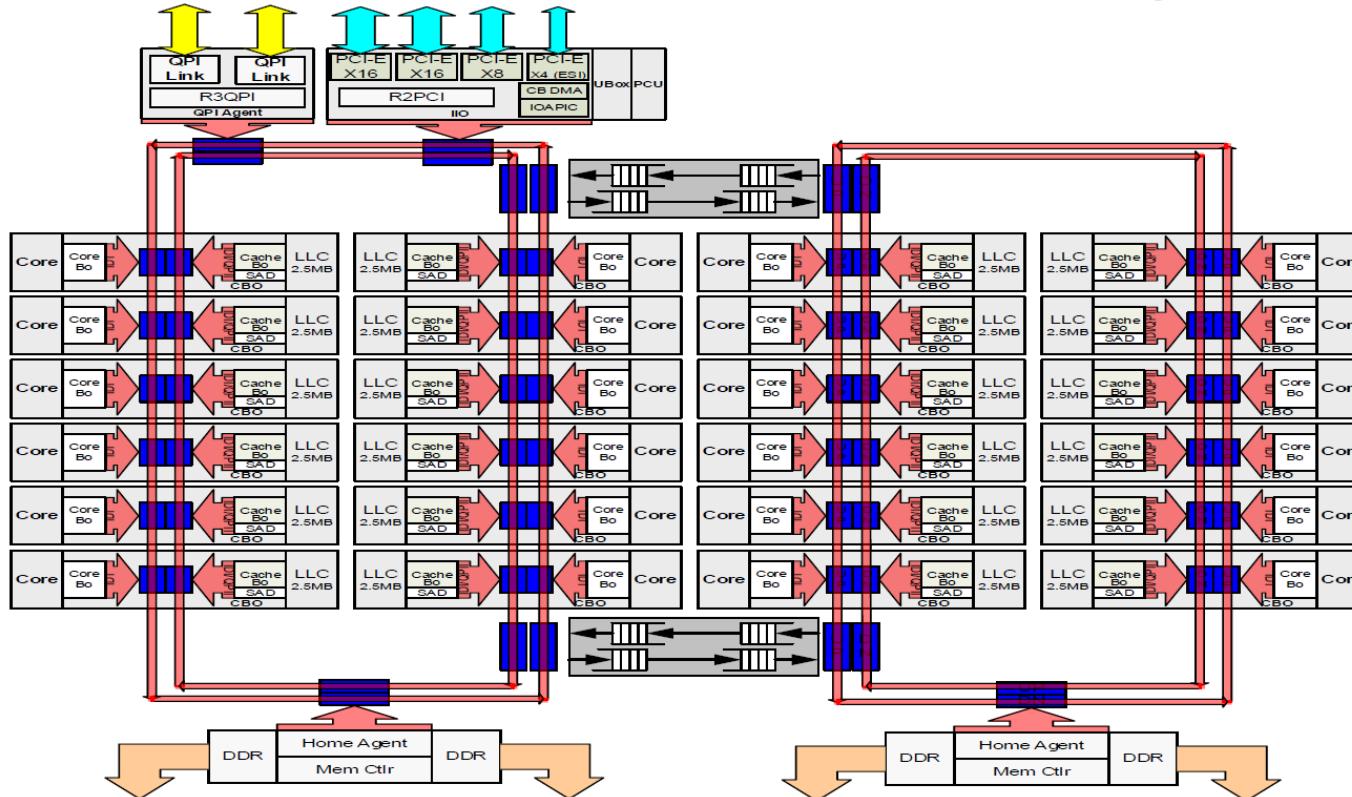
- Zvýšeno IPC o 18 %
- 8 → 10 výpočetních linek
- ROB zvětšen z 224 na 352 záznamů
- Nová AGU jednotka (4)
- Výrazně zvětšeny load/store buffery



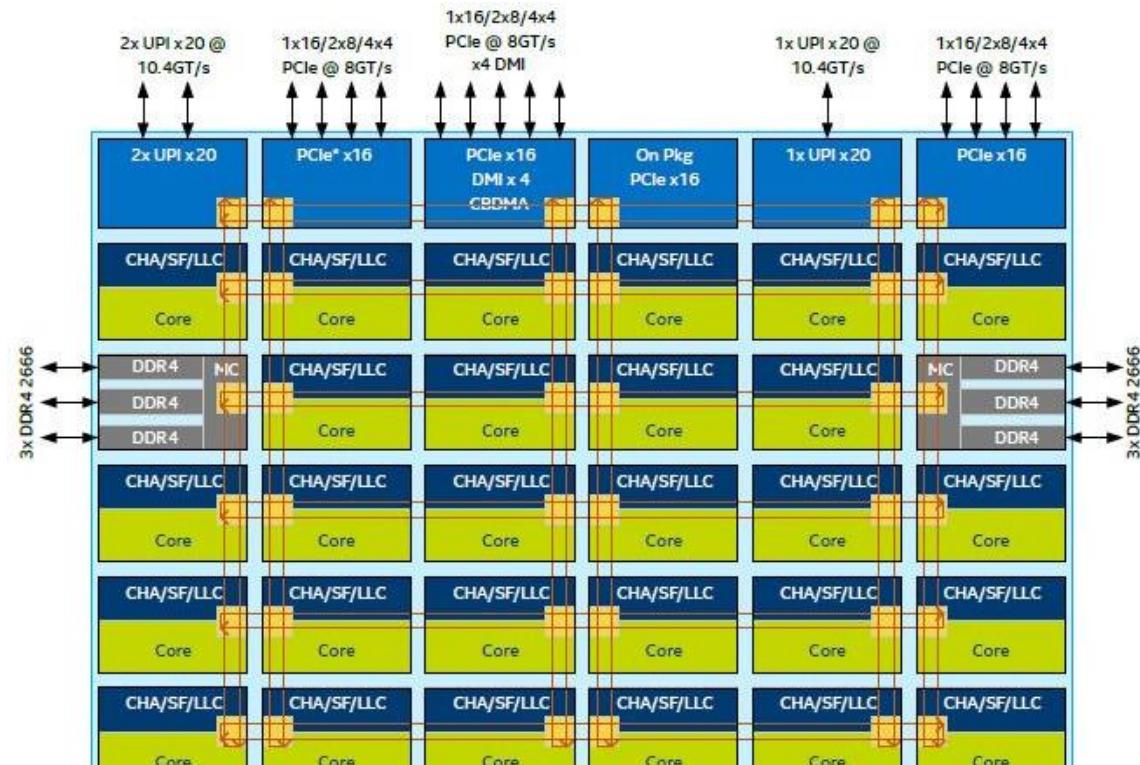
# Frequency Scaling with Instruction Types



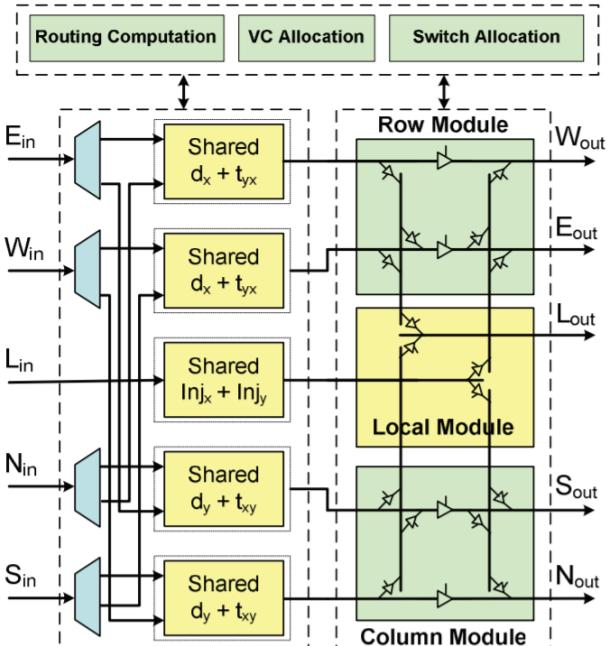
## Intel® Xeon® Processor E5 v4 Product Family HCC



# Zapojení do mřížky 6x6 (28 jader)

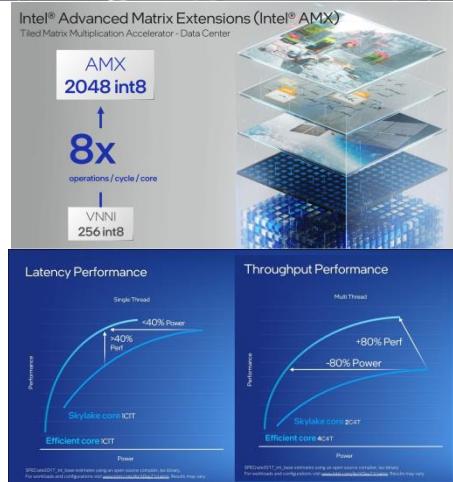
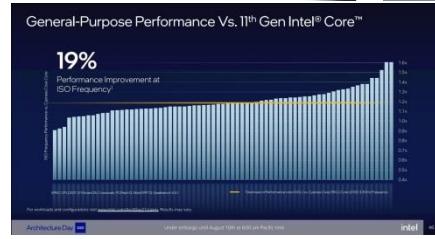
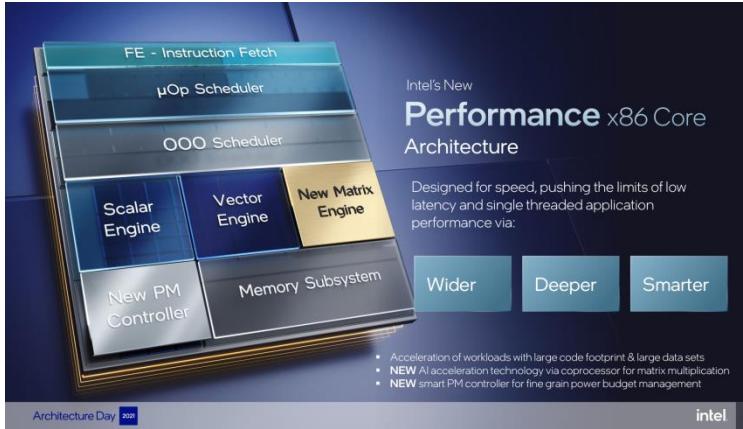
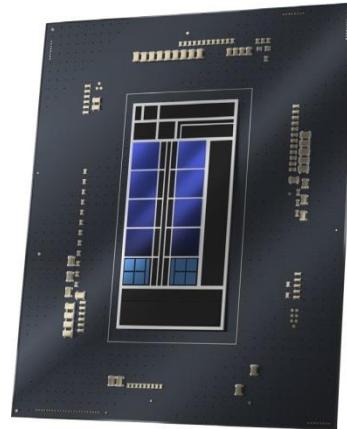


CHA – Caching and Home Agent ; SF – Snoop Filter; LLC – Last Level Cache;  
Core – Skylake-SP Core; UPI – Intel® UltraPath Interconnect



(c) 5x5 MoDe-X-Single Router Design (Single Injection)

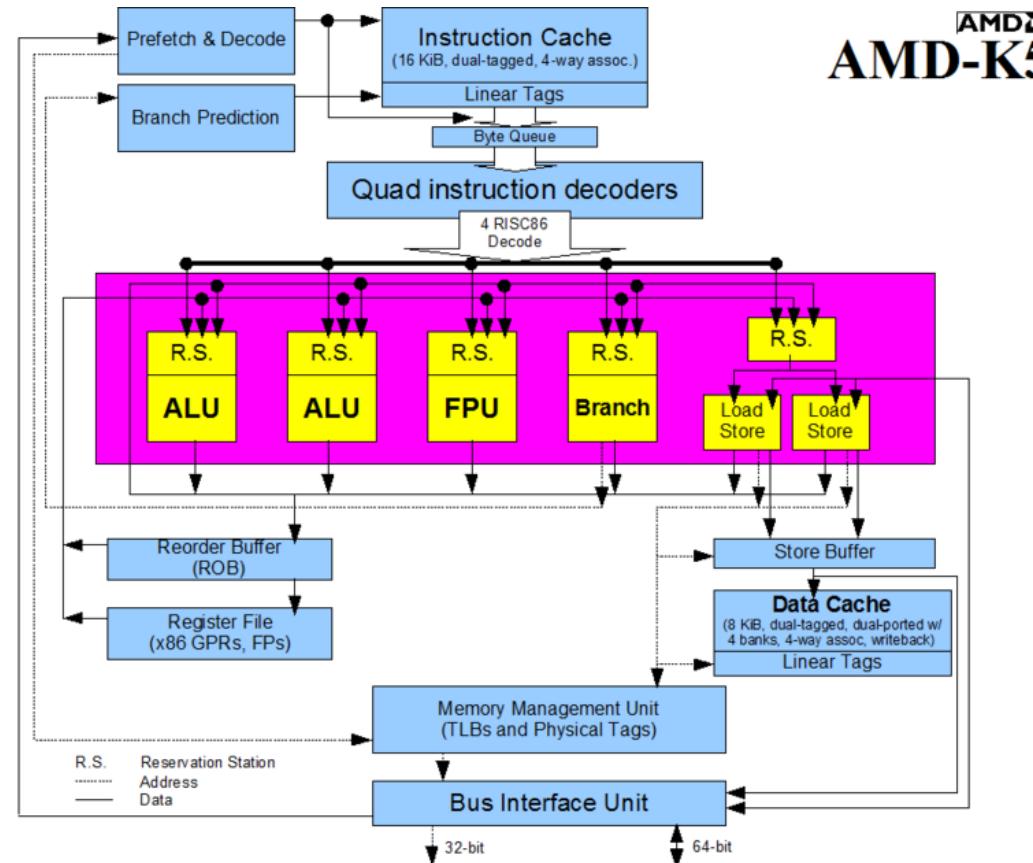
# Intel Alder lake



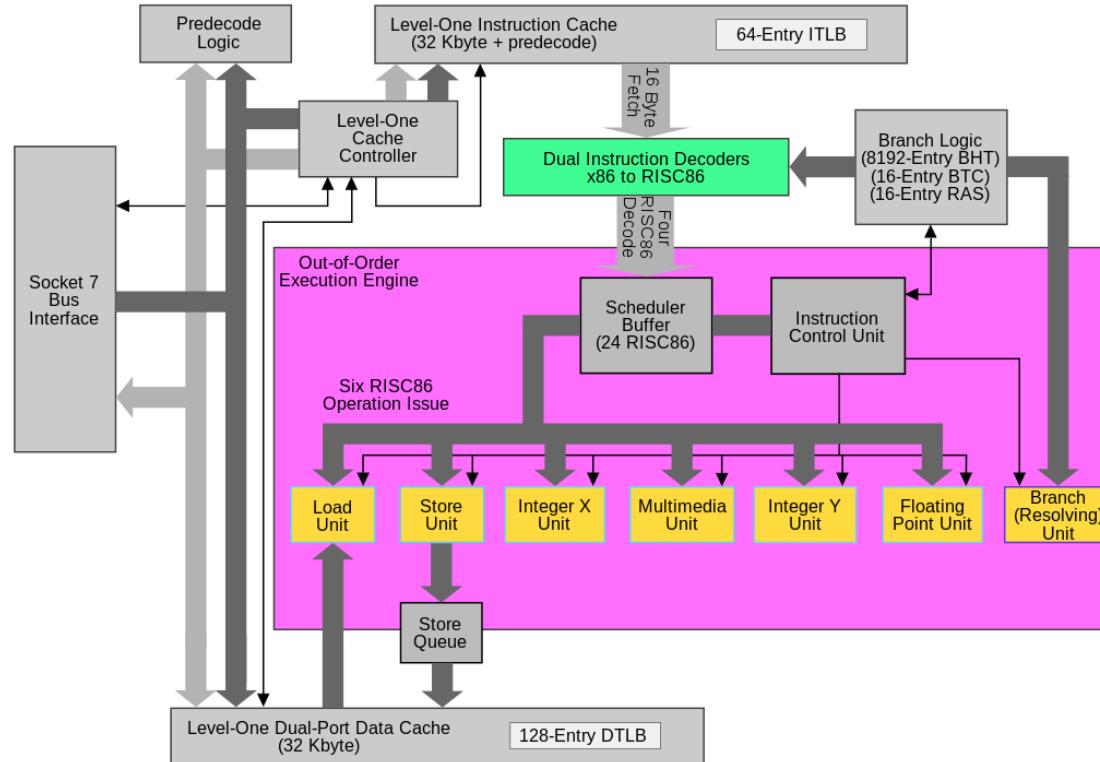
# PROCESORY AMD

- **AMD K5 (1996)**

- První vlastní OOO procesor AMD
- 6 výpočetních jednotek, 4 vydání instrukce za takt, 5 stupňů linky.
- Spekulativní provádění podél 3 predikovaných větví
- Penalta 3 takty při špatné predikci
- Přejmenování registrů
- 16 KB L1, přístup do L1 v 1 taktu!
- Podpora MESI cache coherent protokolu

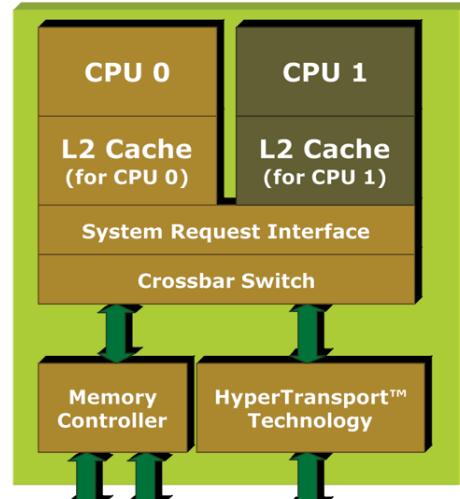
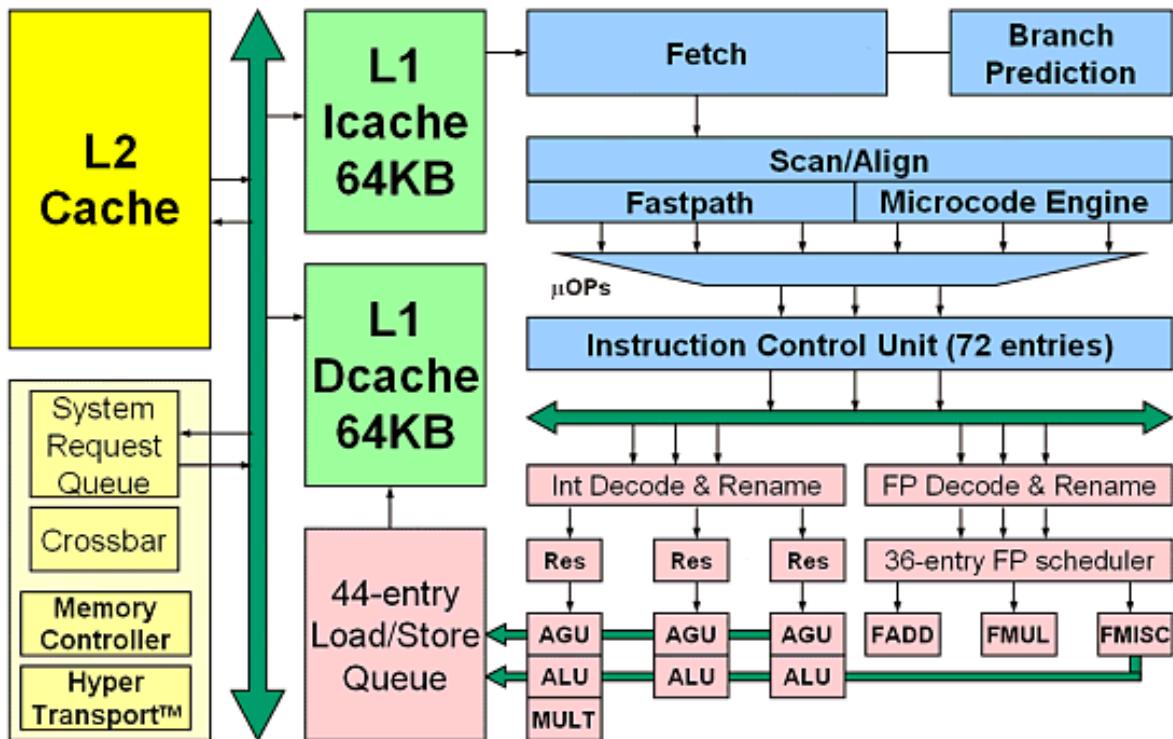


- Uvedena na trh v roce 1997
- Přináší instrukce MMX, později 3DNow

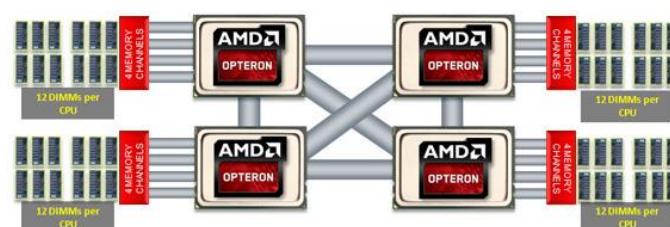


**První CPU s instrukcemi x86 na 64 bitech**, kompatibilní s Windows (2003), 32 bitové i 64 bitové aplikace, SW investice neznehodnoceny. Reakce Intelu: Extended Memory 64-bit Technology) EM64T a pak Intel® 64.

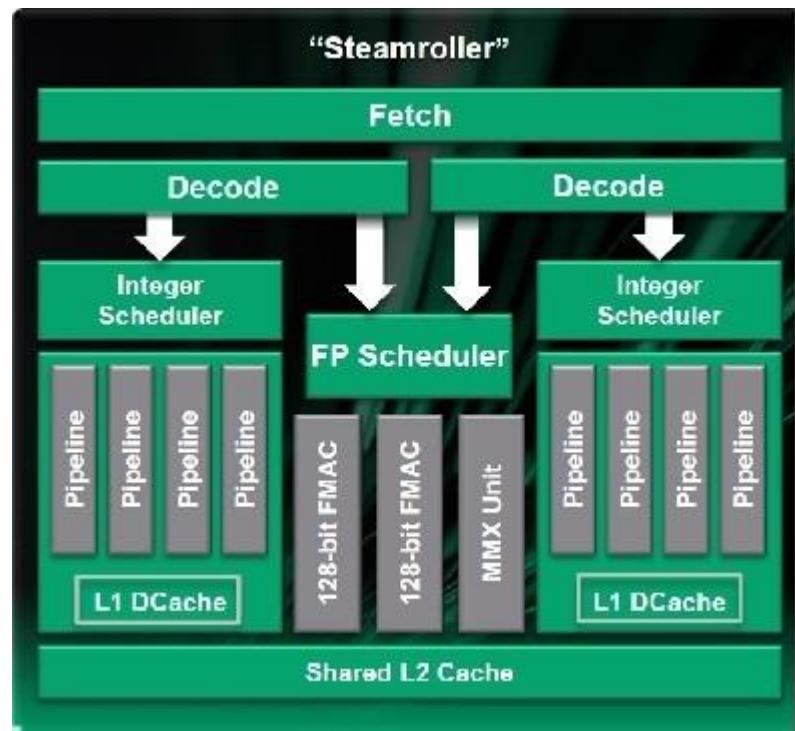
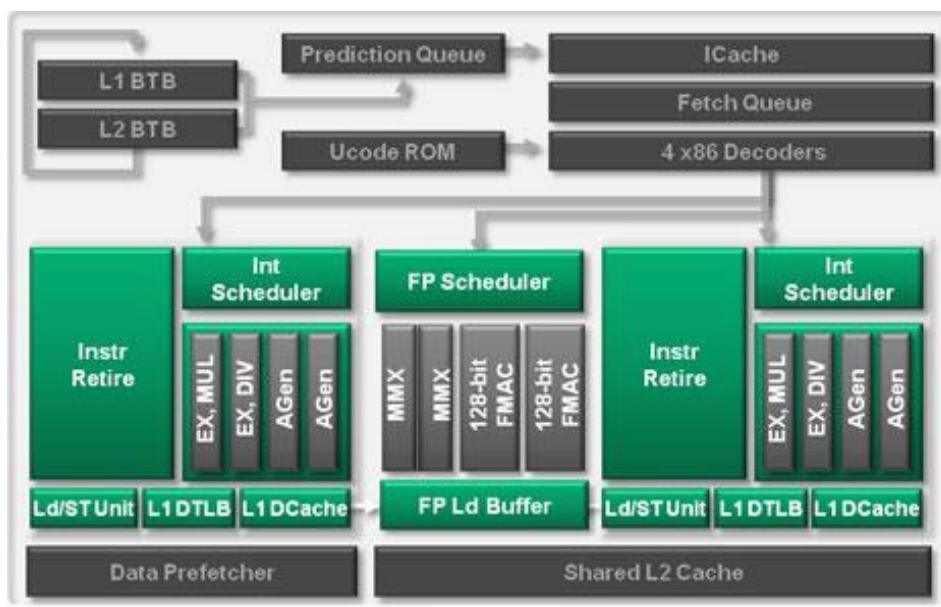
- **2, 4, 6, 8, 12, nebo 16 jader**, linka 12 stupňů, technologie SOI (Silicon on Insulator)
- **Linky Hyper Transport – 2003** (point-to-point) pro propojení s dalšími CPU (nahradily sběrnici) nebo I/O. Šířka 16 bitů, při 800 MHz to znamená 3,2 GB/s. Umožněna stavba multiprocesorů bez dalších součástek.
- **Řadič paměti DDR na čipu – 2003**, 128 bitové rozhraní na 333 MHz paměť.
- **Mikroarchitektura K10: Phenom II, 2,3,4 nebo 6 jader, 2008–12.**

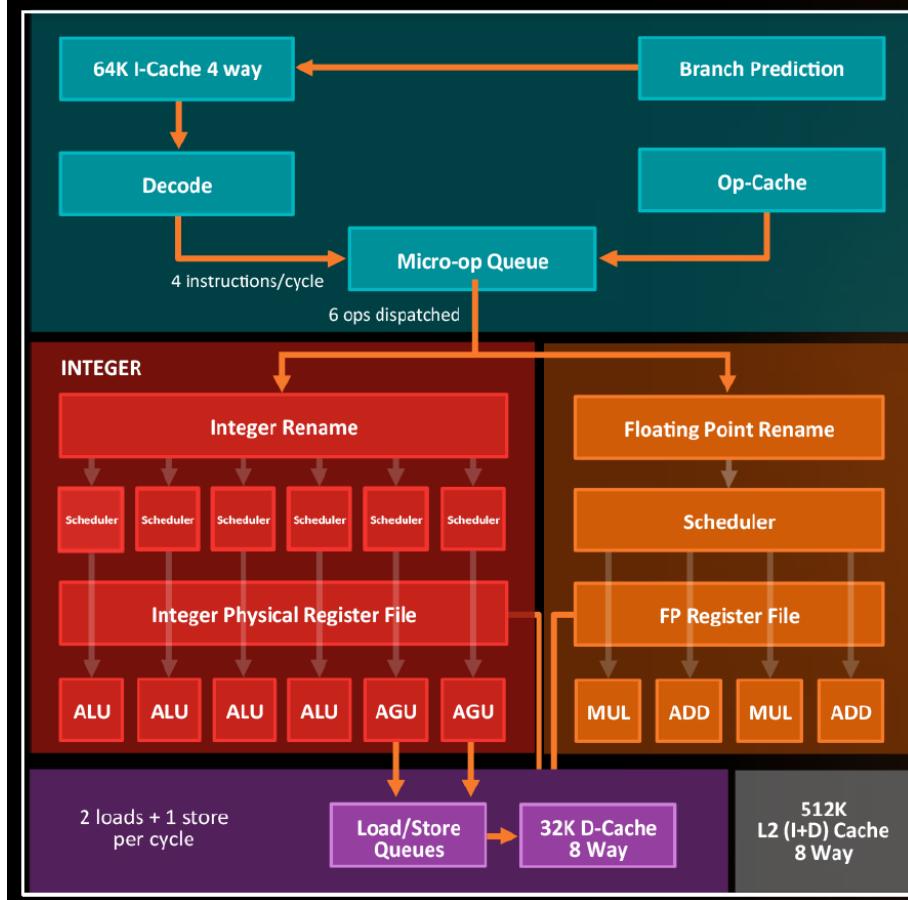


AMD Athlon™ 64 X2  
Dual-Core Processor Design



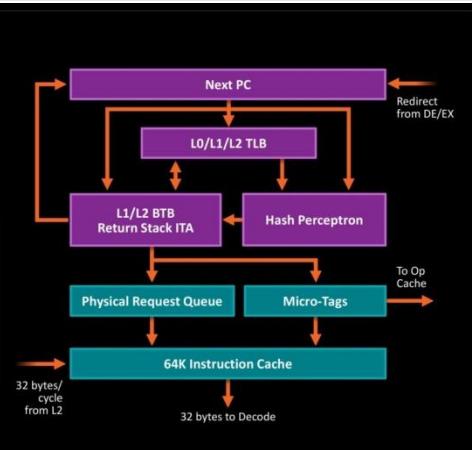
- Mikroarchitektura Bulldozer: 1 až 4 moduly se 2 jádry (tzv. CMT, Clustered MultiThreading, 1 až 2 vlákna/modul).
  - 10 až 100 W, 3,6–4 GHz, 2012.
  - Každý 2-jádrový modul sdílí L1-I cache, stupně načítání a dekódování, L2 cache, FPU.
  - Později přidány dedikované dekodéry (Steamroller)





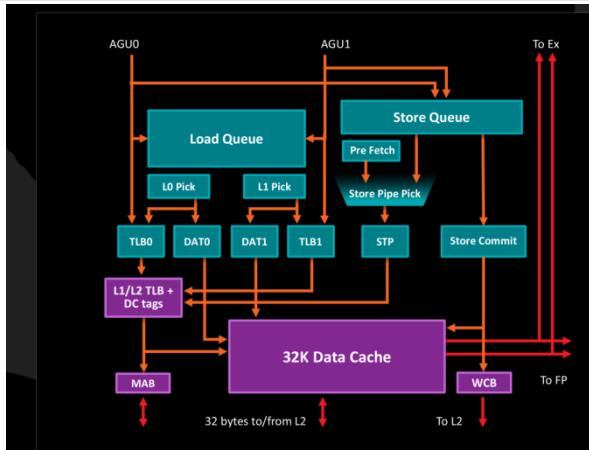
## ZEN MICROARCHITECTURE

- ▲ Fetch Four x86 instructions
- ▲ Op Cache instructions
- ▲ 4 Integer units
  - Large rename space – 168 Registers
  - 192 instructions in flight/8 wide retire
- ▲ 2 Load/Store units
  - 72 Out-of-Order Loads supported
- ▲ 2 Floating Point units x 128 FMACs
  - built as 4 pipes, 2 Fadd, 2 Fmul
- ▲ I-Cache 64K, 4-way
- ▲ D-Cache 32K, 8-way
- ▲ L2 Cache 512K, 8-way
- ▲ Large shared L3 cache
- ▲ 2 threads per core



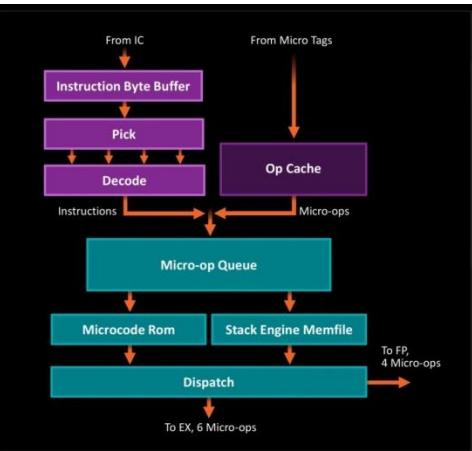
## FETCH

- ▲ Decoupled Branch Prediction
- ▲ TLB in the BP pipe
  - 8 entry L0 TLB, all page sizes
  - 64 entry L1 TLB, all page sizes
  - 512 entry L2 TLB, no 1G pages
- ▲ 2 branches per BTB entry
- ▲ Large L1 / L2 BTB
- ▲ 32 entry return stack
- ▲ Indirect Target Array (ITA)
- ▲ 64K, 4-way Instruction cache
- ▲ Micro-tags for IC & Op cache
- ▲ 32 byte fetch



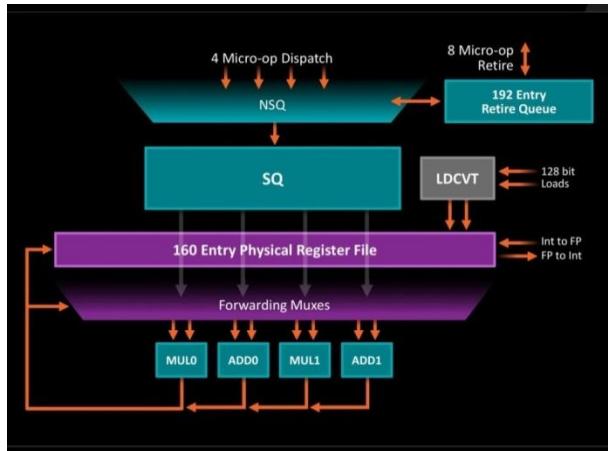
## LOAD/STORE AND L2

- ▲ 72 Out of Order Loads
- ▲ 44 entry Store Queue
- ▲ Split TLB/Data Pipe, store pipe
- ▲ 64 entry L1 TLB, all page sizes
- ▲ 1.5K entry L2 TLB, no 1G pages
- ▲ 32K, 8 way Data Cache
  - Supports two 128-bit accesses
- ▲ Optimized L1 and L2 Prefetchers
- ▲ 512K, private (2 threads), inclusive L2



## DECODE

- ▲ Inline Instruction-length Decoder
- ▲ Decode 4 x86 instructions
- ▲ Op cache
- ▲ Micro-op Queue
- ▲ Stack Engine
- ▲ Branch Fusion
- ▲ Memory File for Store to Load Forwarding

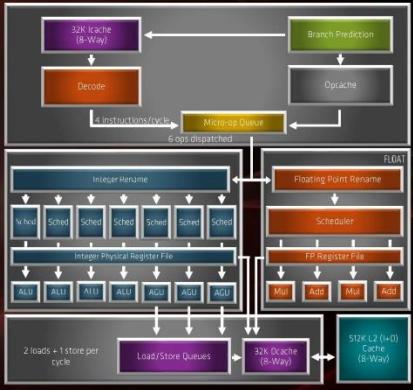


## FLOATING POINT

- ▲ 2 Level Scheduling Queue
- ▲ 160 entry Physical Register File
- ▲ 8 Wide Retire
- ▲ 1 pipe for 1x128b store
- ▲ Accelerated Recovery on Flushes
- ▲ SSE, AVX1, AVX2, AES, SHA, and legacy mmx/x87 compliant
- ▲ 2 AES units

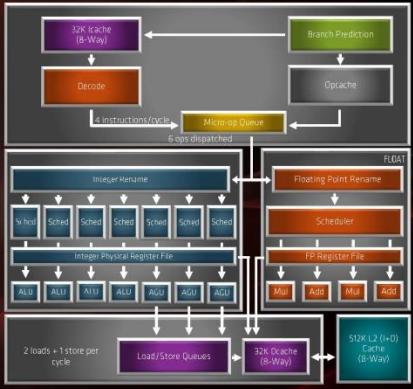
## "ZEN 2" MICROARCHITECTURE OVERVIEW

- 2 threads per core (SMT) carried forward
- New TAGE branch predictor
- Larger Micro-Op Cache, now 4K instructions
- Larger L3 cache, now 2X "Zen" and "Zen+"
- 4 integer units
  - Large rename space ~ 180 registers
  - Increased AGUs from 2 to 3
- 3 AGENs per cycle
- 2 loads and 1 store per cycle
- 2 floating point units x 256 Fmacs
  - built as 4 pipes, 2 Fadd, 2 Fmul
  - Now supports single-op AVX256



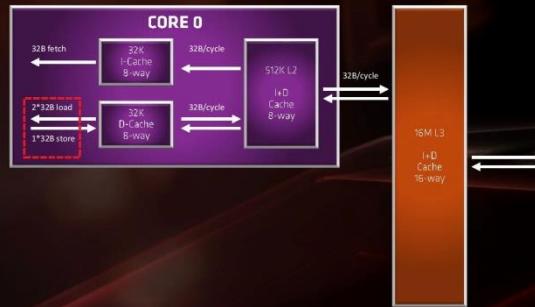
## "ZEN 2" MICROARCHITECTURE OVERVIEW

- I-cache 32k, 8-way
- D-cache 32k, 8-way
- L2 cache 512k, 8-way
- TLBs
  - L1 64 entries I & D, all page sizes
  - L2 512 I, 2K D, everything but 1G
- Faster Virtualization Based Security
  - With Guest Mode Execute Trap
- Hardware-enhanced Security mitigations



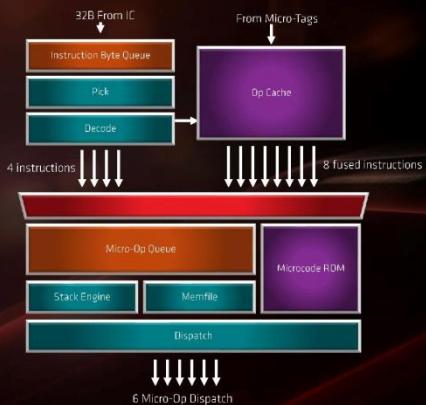
## "ZEN 2" CACHE HIERARCHY

- Doubled L1 load/store bandwidth over Zen
- Improved L1 and L2 prefetch throttling
- Fast private 512k L2 cache
- Fast shared L3 cache
- High bandwidth enables prefetch improvements
- L3 is filled from L2 victims
- Fast cache-to-cache transfers
- Large Queues for Handling L1 and L2 misses



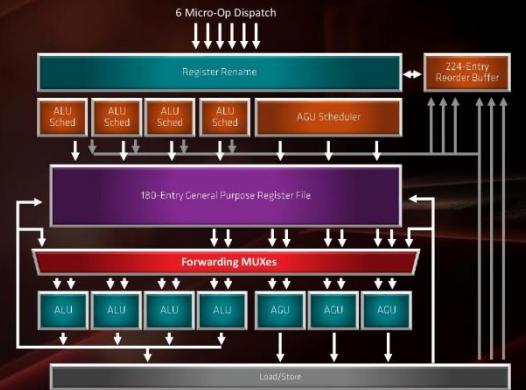
## DECODE

- Op cache improvements
- Doubled capacity to 4K fused instructions
- Better instruction fusion
- Increased effective throughput



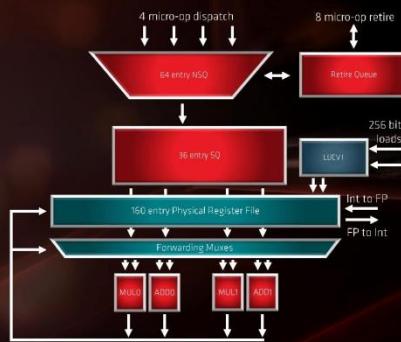
## INTEGER EXECUTION

- 92 entry integer scheduler, up from 84
- 4, 16-entry ALU queues
- 1, 28-entry AGU queue
- 180 entry physical register file (up from 168)
- 7 issue per cycle, up from 6
- 4 ALUs, 3 AGUs
- 224 entry ROB, up from 192
- Improved SMT fairness for ALU and AGU schedulers
- Watermarked ALU tokens to manage spinlocks



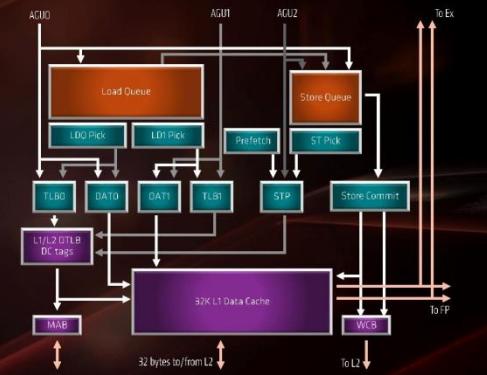
## FLOATING POINT UNIT

- Doubled Floating Point & Load Store bandwidth from 128b to 256b
- Improved performance for instructions using 256b ymm registers which are generated by AVX intrinsics or /arch:[AVX|AVX2] compiler flags
  - Faster inline memcpy & memset
  - Faster physics simulation
  - Faster audio effects processing (Microsoft™ XAudio2\_9)
- Improved mul latency from 4 to 3 cycles



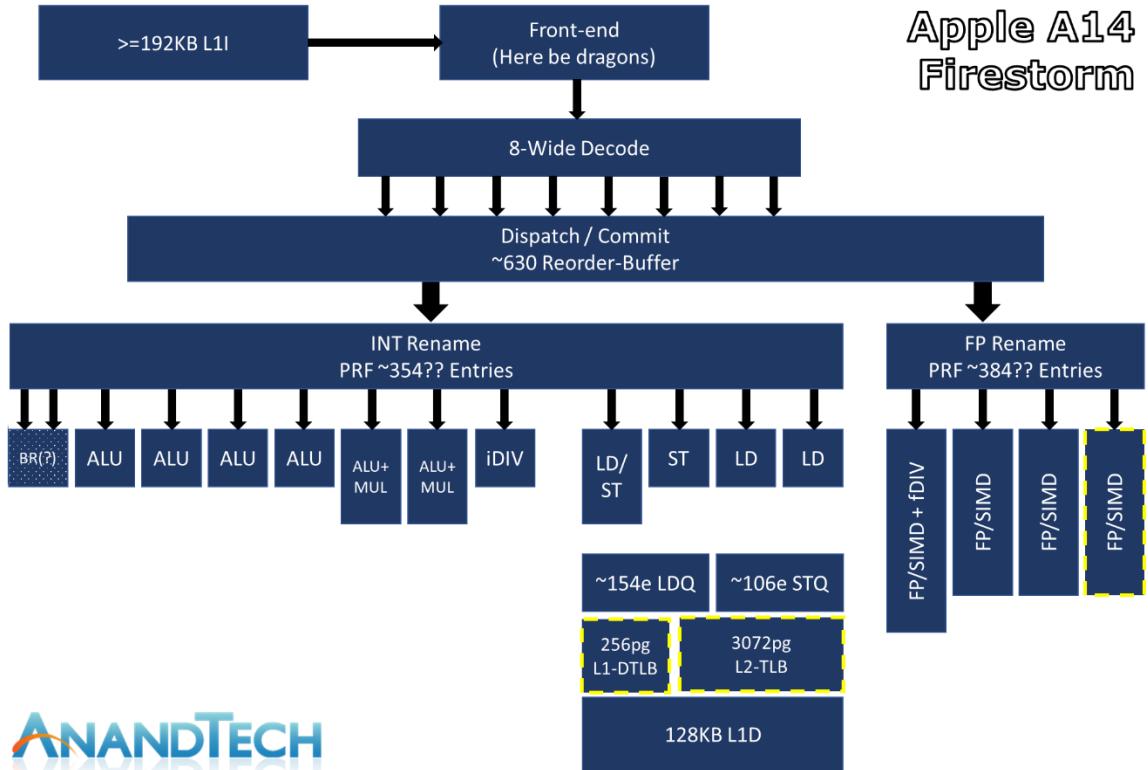
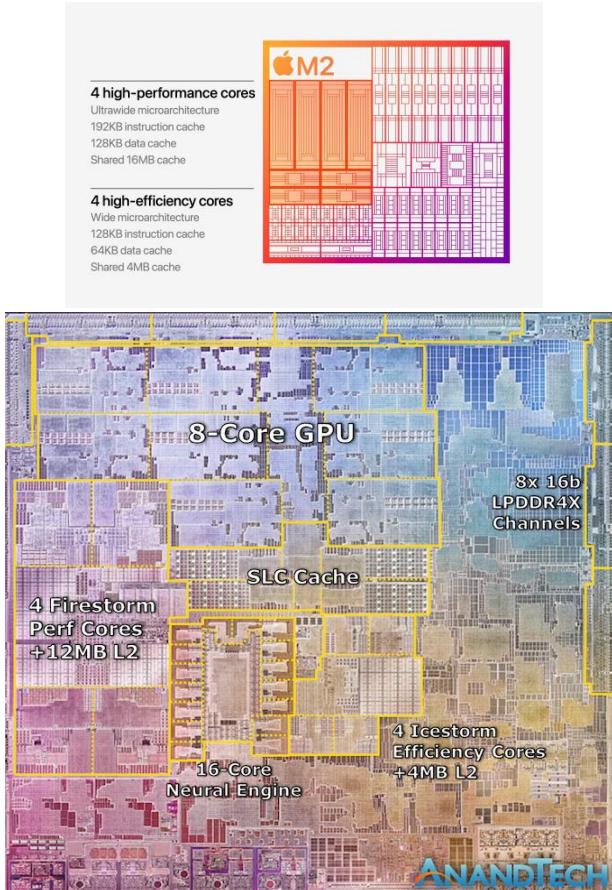
## LOAD/STORE

- 48 entry store queue, was 44
- 2K entry L2 DTLB, 1G as 2M, was 1.5K no 1G
- Improved L2 DTLB latency
- 32KB, 8-way L1 data cache
  - Two 256-bit reads
  - One 256-bit write
  - 64B load, 32B store alignment boundaries
- Increased Load/Store bandwidth to 32B/clk (up from 16B/clk)
- Faster string copy and float-point point performance
- Improved write-combining buffer performance
  - While using multiple streams, the hardware avoids closing buffers before they are completely full
- Improved prefetch throttling



# PROCESORY APPLE/ARM

# Apple M1 – ARM (RISC) procesor



# Pokračování příště

# Grafické akcelerátory pro obecné výpočty

## AVS – Architektury výpočetních systémů

### Týden 13, 2024/2025

**Jirka Jaroš**

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



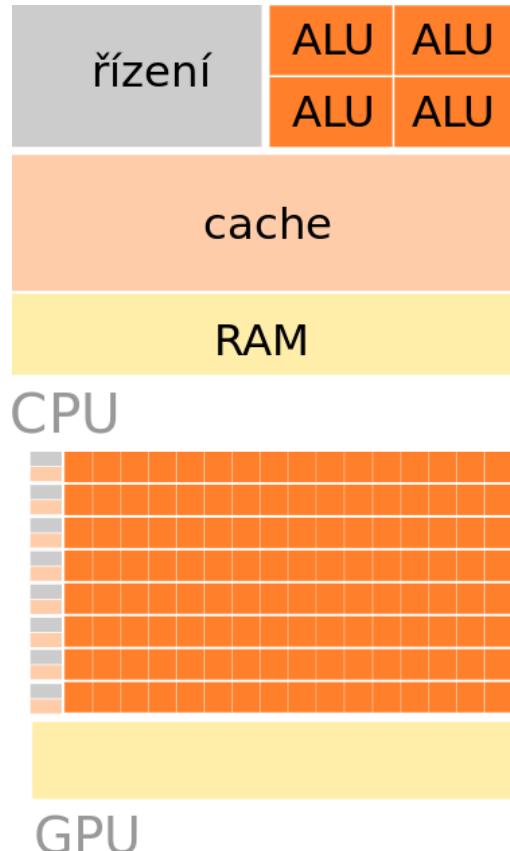
# **ARCHITEKTURA GRAFICKÝCH KARET**

- **CPU**

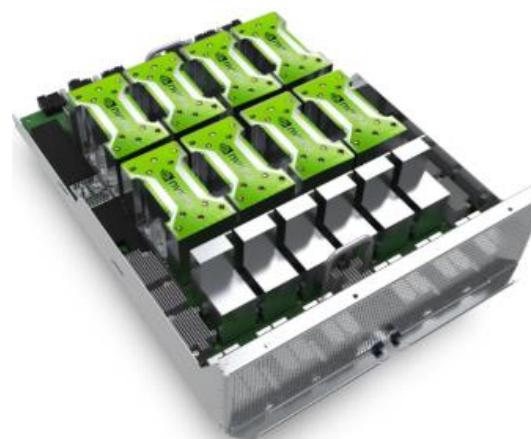
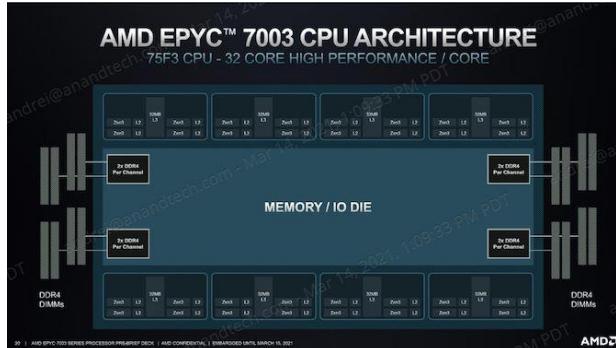
- Velmi rychlé a velké paměti cache
- Dobře zpracované techniky predikce skoků
- Vysoce výkonné zpracování sekvenčních programů
- I/O, přerušení, virtuální paměť, izolace procesů
- Vláknový a datový paralelismus

- **GPU**

- Velké množství ALU – především MAD/FMA
- Mnoho HW vláken – SMP
- Rychlé lokální paměti
- Paměti na desce s velkou propustností ale i latencí
- Zpracování technikou SIMT (varianta SIMD)
- In-order zpracování



- **64** fyzických jader doplněných o HyperThreading, 2,45 – 3,5GHz
- **SIMD** jednotky AVX2 (256b MAD), 32 op / takt v SP
- **256MB** L3 cache, 32MB L2 cache, 512 GB RAM



## Teoretický výkon na procesor

- **2,7 DP TFLOPS**
- **5,3 SP TFLOPS**

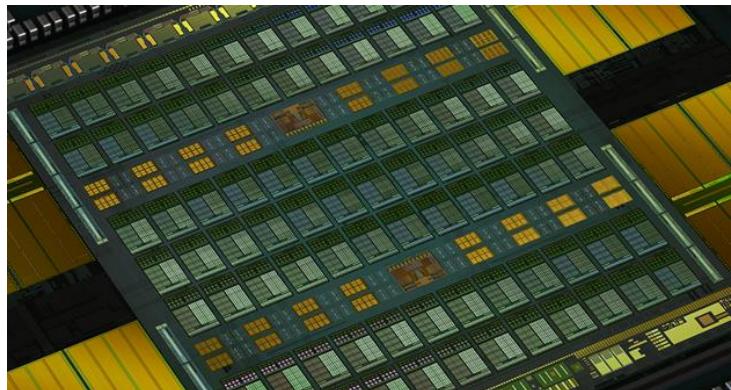
Propustnost do paměti

- **204** GB/s
- šířka sběrnice 512b

- <https://www.anandtech.com/show/16529/amd-epyc-milan-review>
- <https://docs.it4i.cz/karolina/compute-nodes/>

- **Obsahuje**

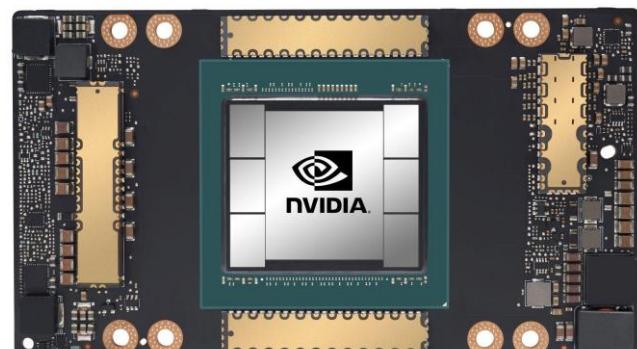
- 108 SM procesorů, 6192 CUDA jader
- 40 MB L2 cache
- 40 GB HBMA RAM
- Připojení na NV Link (600GB/s přes 4 GPU)
- Max frekvence 1.4GHz



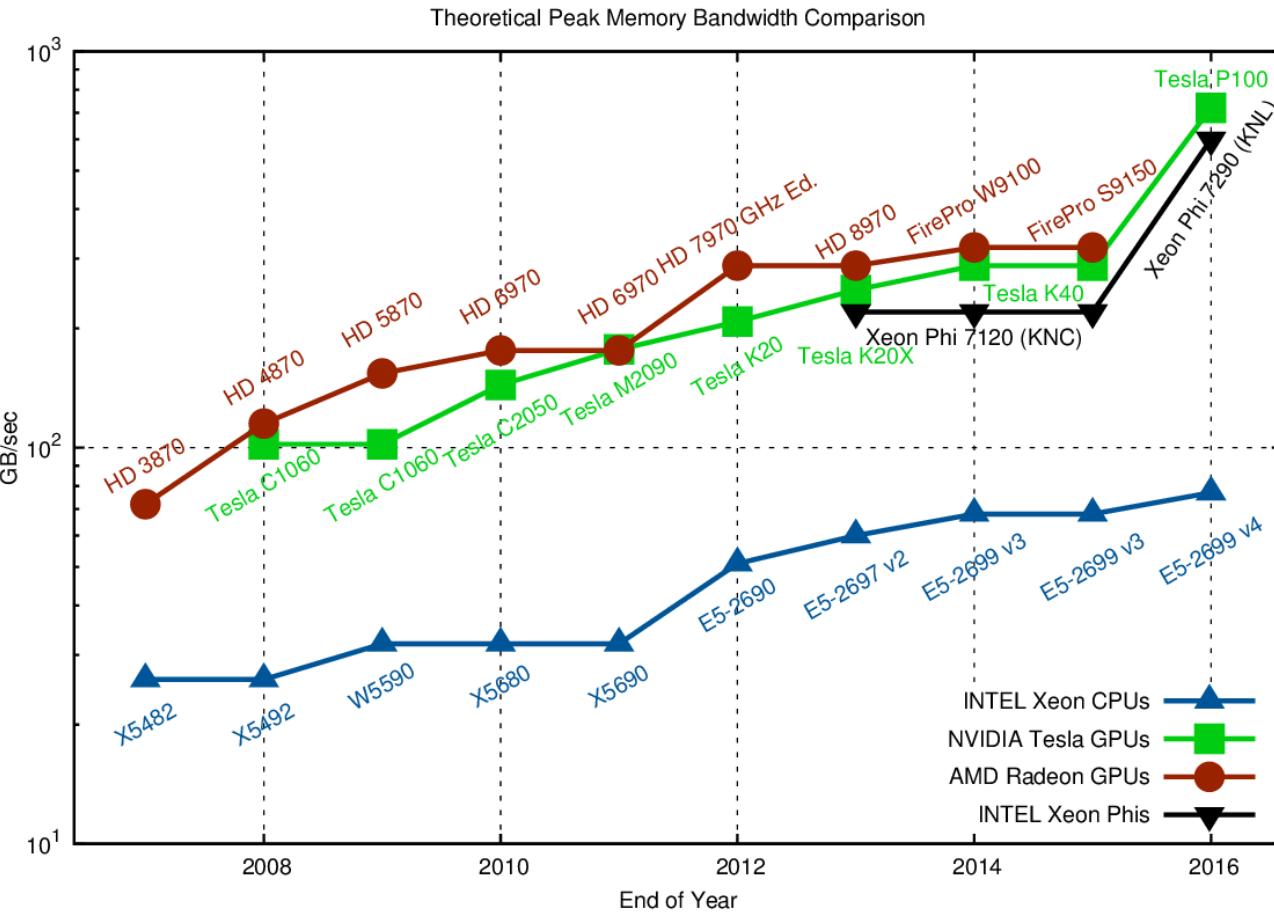
- **Teoretický výkon:**

- **9,7 DP** TFLOPS (FP 64)
- **19,5 SP** TFLOPS (FP 32)
- **312 Tensor** TFLOPS (FP 16)
- **1555 GB/s** – šířka sběrnice 5192b

- <https://www.anandtech.com/show/15801/nvidia-announces-ampere-architecture-and-a100-products>



# Porovnání teoretické propustnosti paměti



Barbora

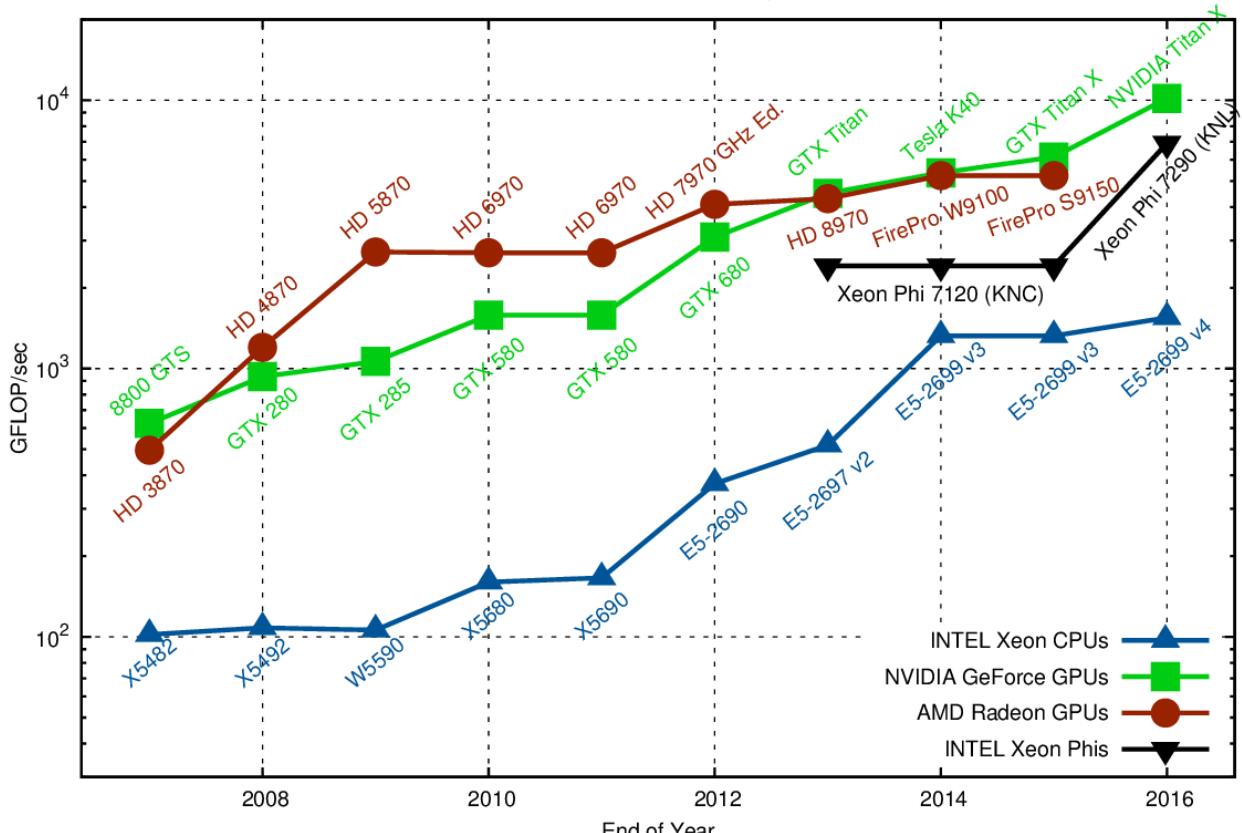
| Arch.      | GB/s         |
|------------|--------------|
| V100       | 900          |
| Intel 6240 | 141          |
| Zrychlení  | <b>6,38x</b> |

Karolina

| Arch.     | GB/s         |
|-----------|--------------|
| A100      | 1555         |
| EPYC 7763 | 204          |
| Zrychlení | <b>7,62x</b> |

# I Porovnání teoretického výpočetního výkonu

Theoretical Peak Performance, Single Precision



## Barbora

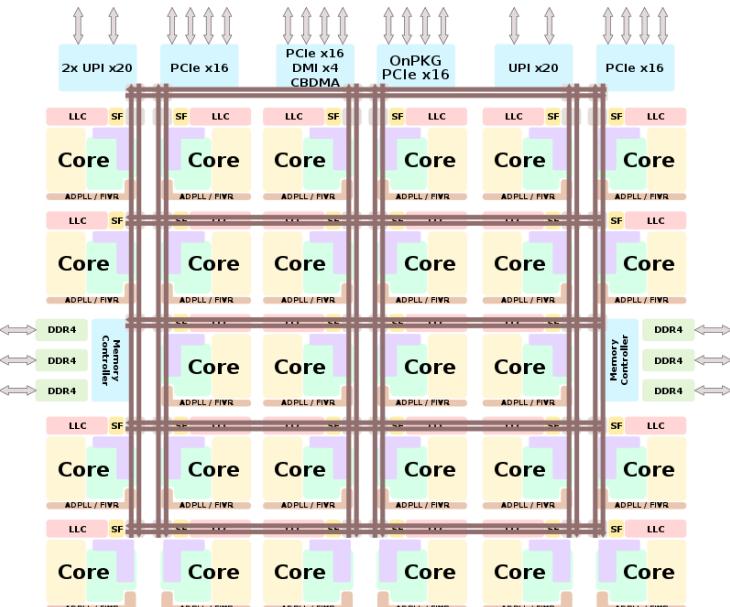
| Arch.      | GFLOPS      |
|------------|-------------|
| V100       | 15 700      |
| Intel 6240 | 3 000       |
| Zrychlení  | <b>5,1x</b> |

## Karolina

| Arch.     | GFLOPS      |
|-----------|-------------|
| A100      | 19 500      |
| EPYC 7763 | 5 300       |
| Zrychlení | <b>3,6x</b> |

<https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

- **18** fyzických jader doplněných o HyperThreading, 2,6-3,9 GHz
- **SIMD** jednotky AVX-512 (512b MAD), 64 op/takt v SP
- **24,75 MB** L3 cache, 18 MB L2 cache, 192 GB RAM



## Teoretický výkon na procesor

- **1,5 DP TFLOPS**
- **3,0 SP TFLOPS**

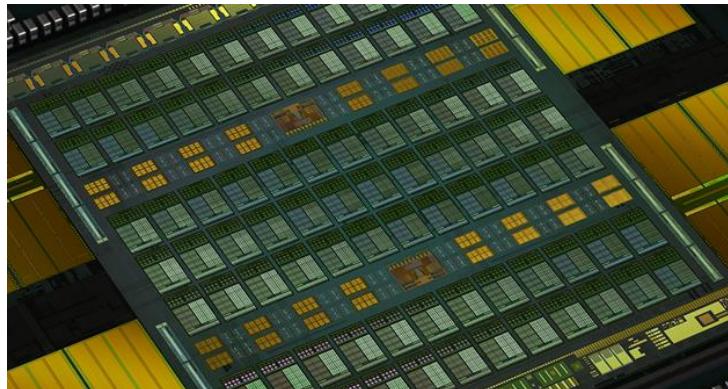
Propustnost do paměti

- **141 GB/s**
- šířka sběrnice 384 b

- <https://www.anandtech.com/show/15039/the-intel-core-i9-10980xe-review>
- <https://docs.it4i.cz/barbora/compute-nodes/>

- **Obsahuje**

- 80 SM procesorů, 5120 CUDA jader
- 6144 kB L2 cache
- 16 GB HBMA RAM
- Připojení na NVLink (300 GB/s přes 4 GPU)
- Max frekvence 1,5 GHz



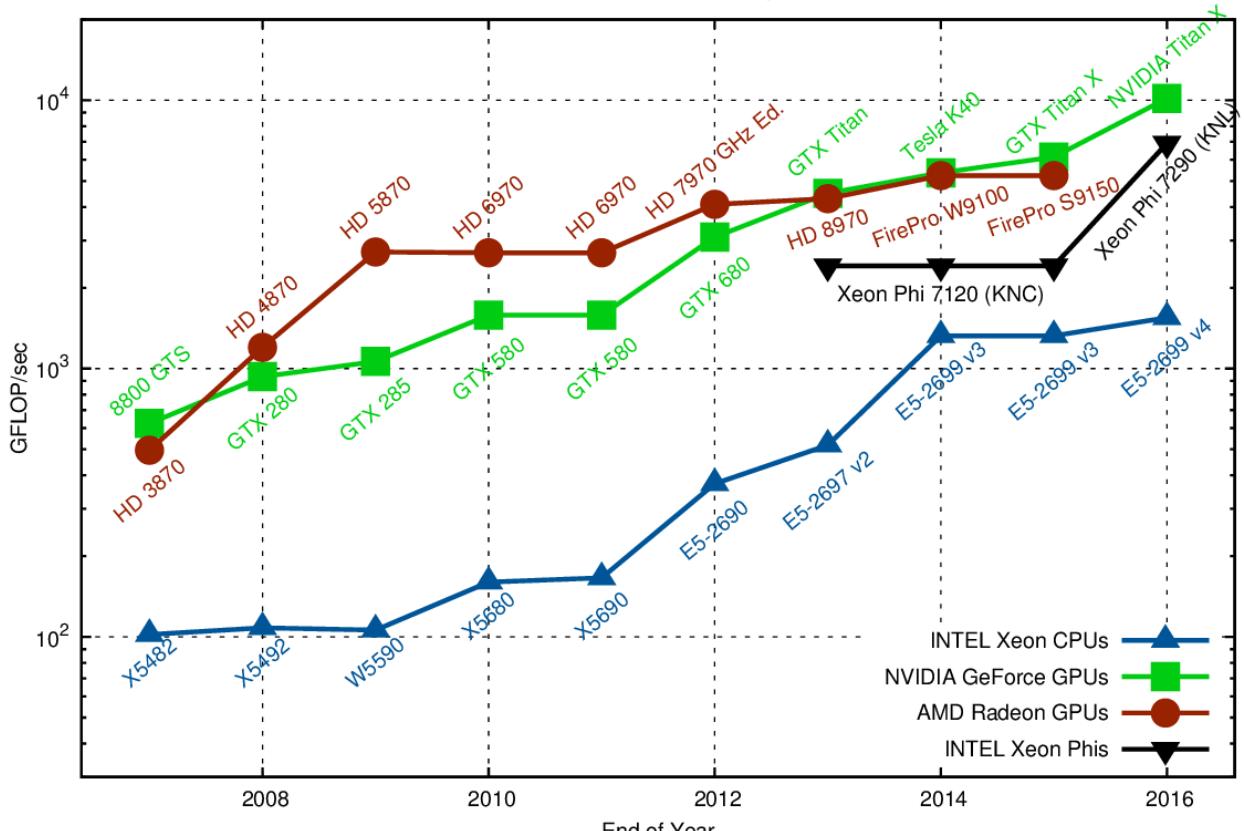
- **Teoretický výkon:**

- **7,8 DP** TFLOPS (FP 64)
- **15,7 SP** TFLOPS (FP 32)
- **125 Tensor** TFLOPS (FP 16)
- **900 GB/s** – šířka sběrnice 4096 b
- <https://www.anandtech.com/show/11367/nvidia-volta-unveiled-gv100-gpu-and-tesla-v100-accelerator-announced>



# Porovnání teoretického výpočetního výkonu

Theoretical Peak Performance, Single Precision



Barbora

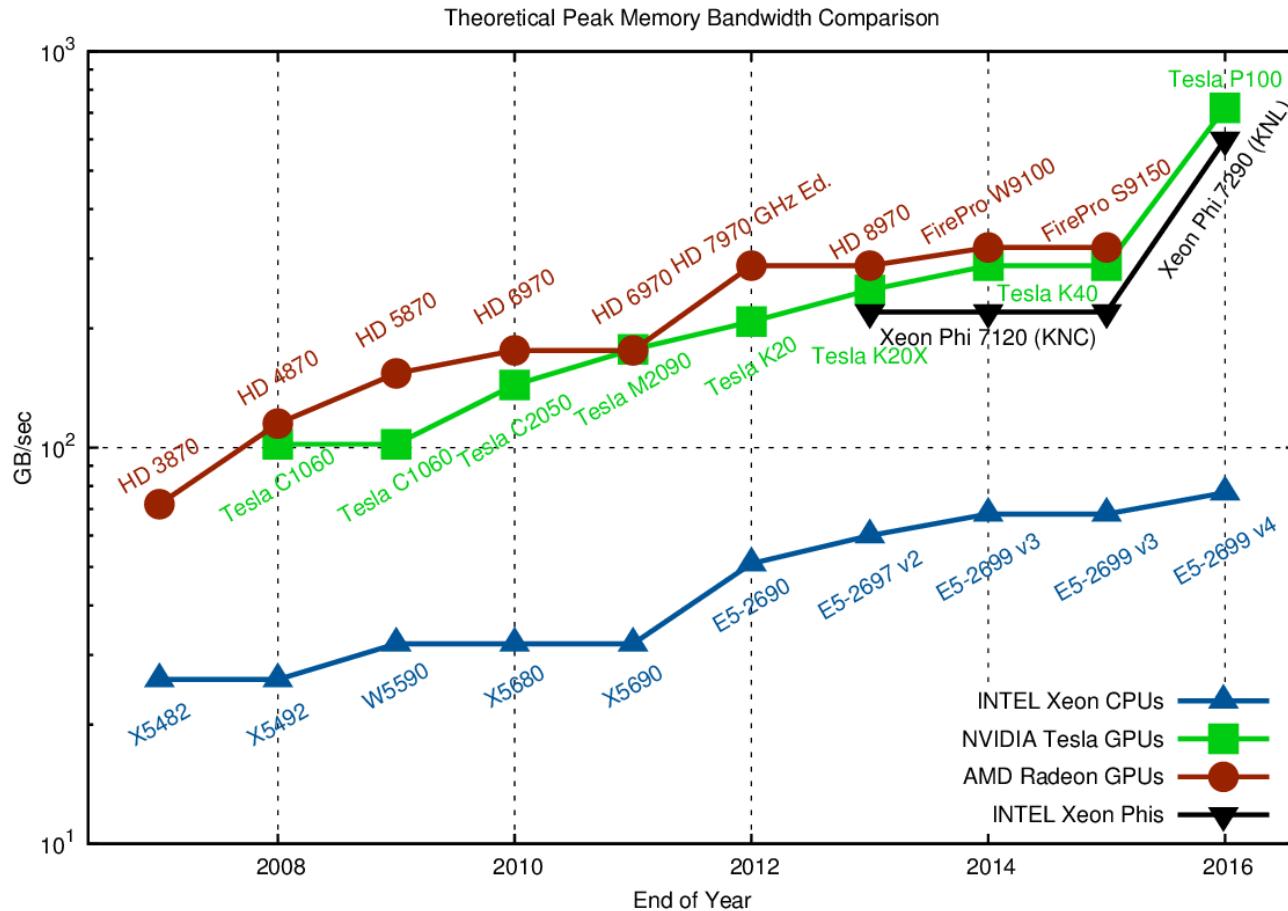
| Arch.      | GFLOPS      |
|------------|-------------|
| V100       | 15 700      |
| Intel 6240 | 3 000       |
| Zrychlení  | <b>5,1x</b> |

FIT O204

| Arch.     | GFLOPS       |
|-----------|--------------|
| GTX 970   | 3920         |
| i5-4460   | 243          |
| Zrychlení | <b>16,1x</b> |

<https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

# Porovnání teoretické propustnosti paměti

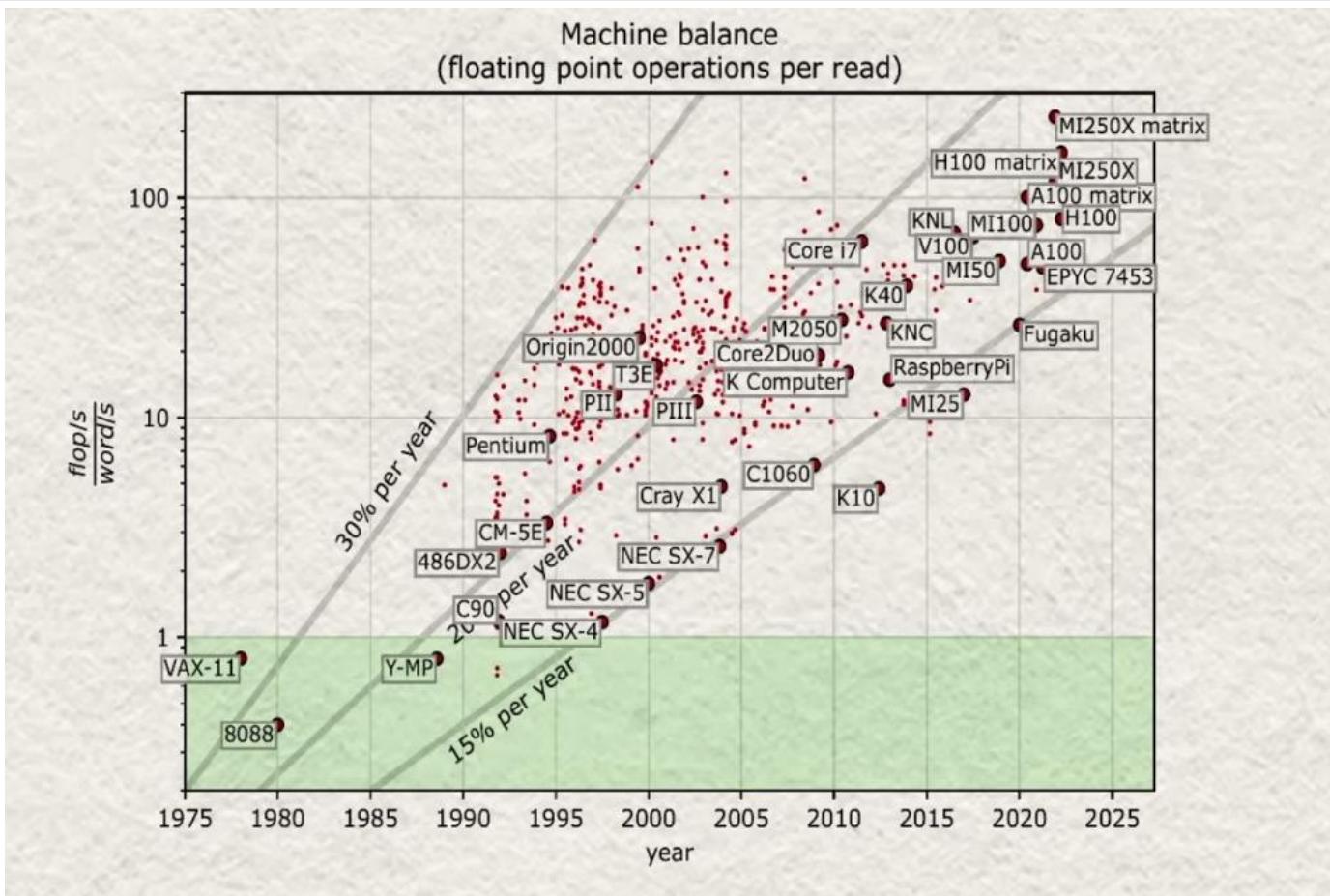


## Barbora

| Arch.      | GB/s         |
|------------|--------------|
| V100       | 900          |
| Intel 6240 | 141          |
| Zrychlení  | <b>6,38x</b> |

## FIT O204

| Arch.     | GB/s         |
|-----------|--------------|
| GTX 970   | 224          |
| i5-4460   | 25,6         |
| Zrychlení | <b>8,75x</b> |



- Výpočet rozčleněn do velkého množství vláken.
- Všechna vlákna mají stejnou strukturu.
- SIMT zpracovává **jednu instrukci** napříč **několika vlákny**
  - Balíky instrukcí se vykonávají pomocí SIMD stroje
  - Každé vlákno z balíku zpracovává stejnou instrukcí nad jinými daty.
  - Balík vláken zpracovávaný v jednom okamžiku se nazývá **WARP**. Jeho velikost bývá závislá na počtu výpočetních jednotek.
- **Divergence vláken** na podmínce způsobí sekvenční vykonávání větví programu.
- Důležitou podmínkou je **zanedbatelná latence pro přepínání warpů**.
- Pro **překrytí paměťových latencí** je nutné mít **velké množství vláken**.

## Součet dvou matic

```
for (int i = 0; i < N; i++)
 for (int j = 0; j < N; j++)
 c[i*N+j] = a[i*N+j] + b[i*N+j];
```

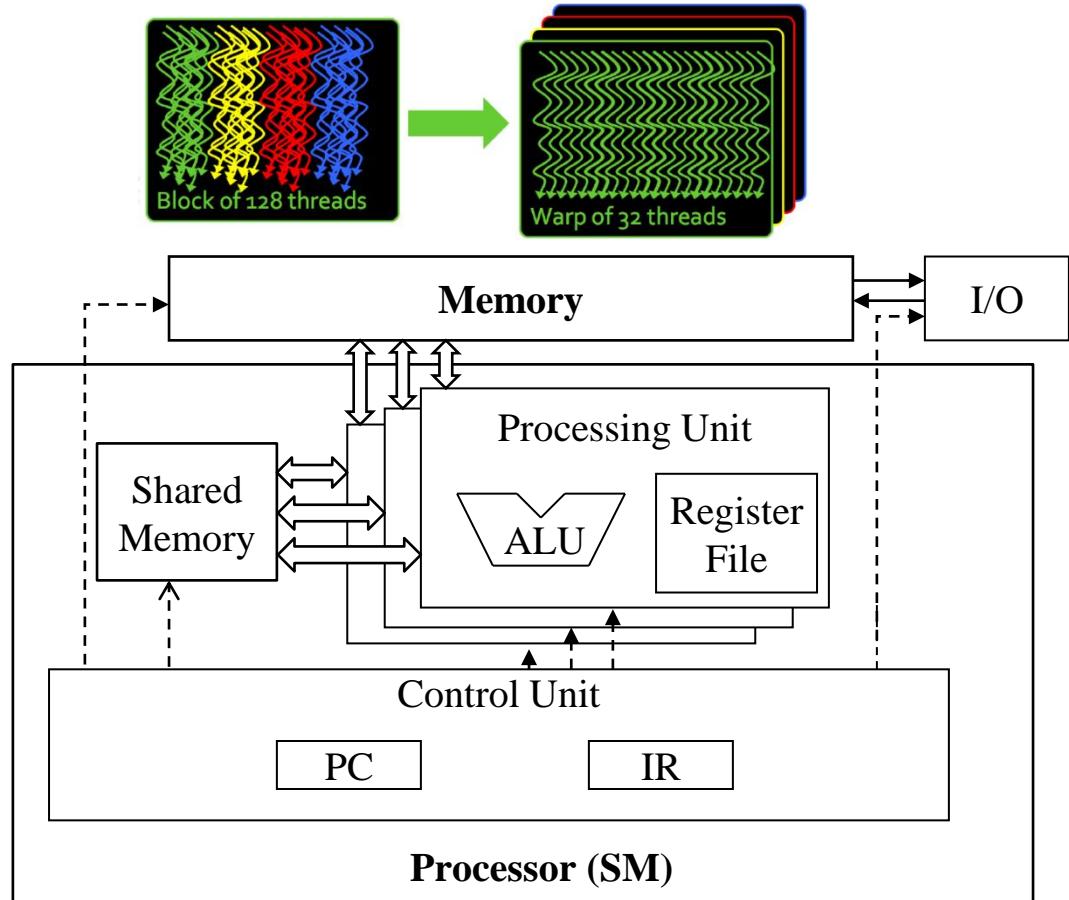
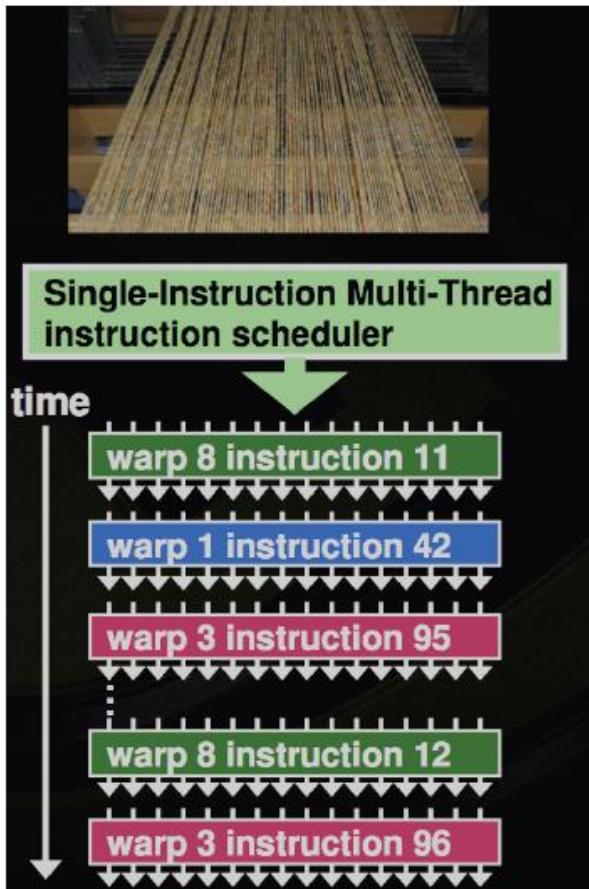
- Matici distribuuujeme po blocích na vlákna
- Výpočet každého řádku vektorizujeme pomocí AVX
  - smyčka se rozbalí
  - HW zpracovává 8/16 elementů

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
 #pragma omp simd
 for (int j = 0; j < N; j++)
 c[i*N+j] = a[i*N+j] + b[i*N+j];
```

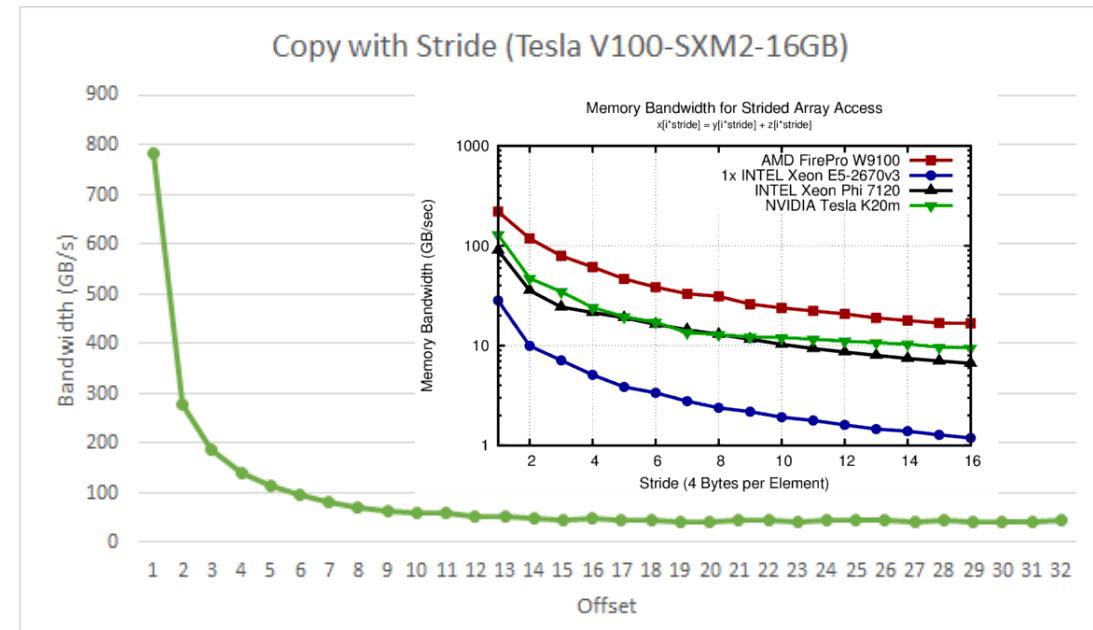
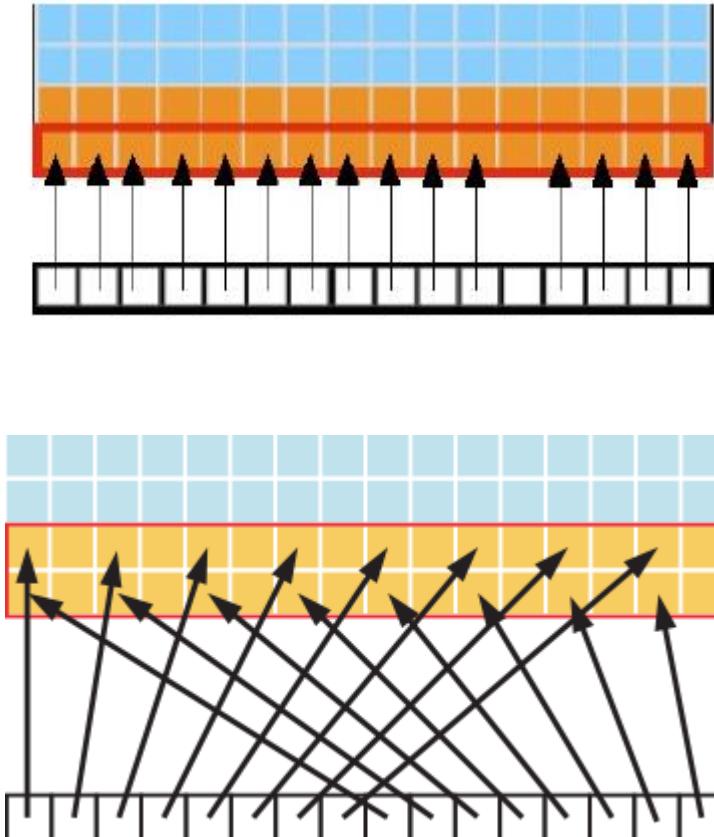
- Sestrojíme kód pro výpočet 1 prvku
  - každé vlákno umí najít svůj prvek
- Vlákna seskupíme do bloků a gridu
  - 1D/2D/3D organizace
  - o plánování vláken se stará HW

```
int i = blockIdx.y * blockDim.y +
 threadIdx.y;
int j = blockIdx.x * blockDim.x +
 threadIdx.x;
if (i < N && j < N)
 c[i*N+j] = a[i*N+j] + b[i*N+j];
```

# Single Instruction Multiple Threads



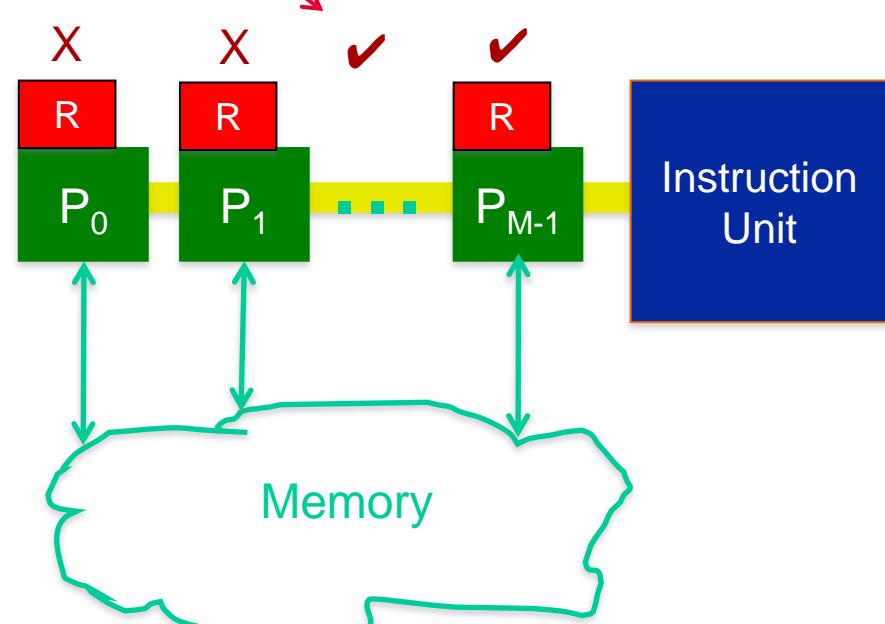
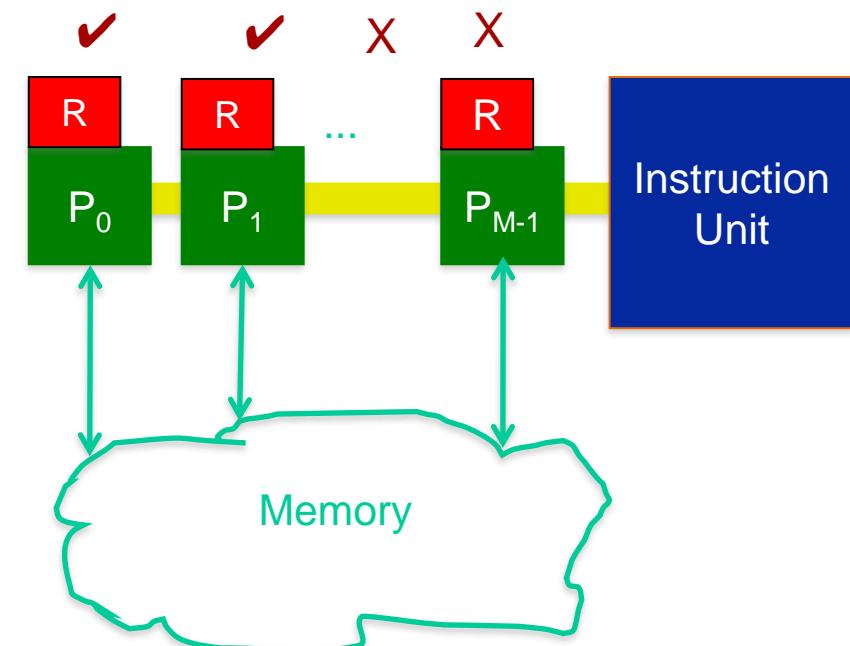
# Datová lokalita přístupu do paměti



Sousední vlákna musí číst ze sousedních lokací – jinak je problém

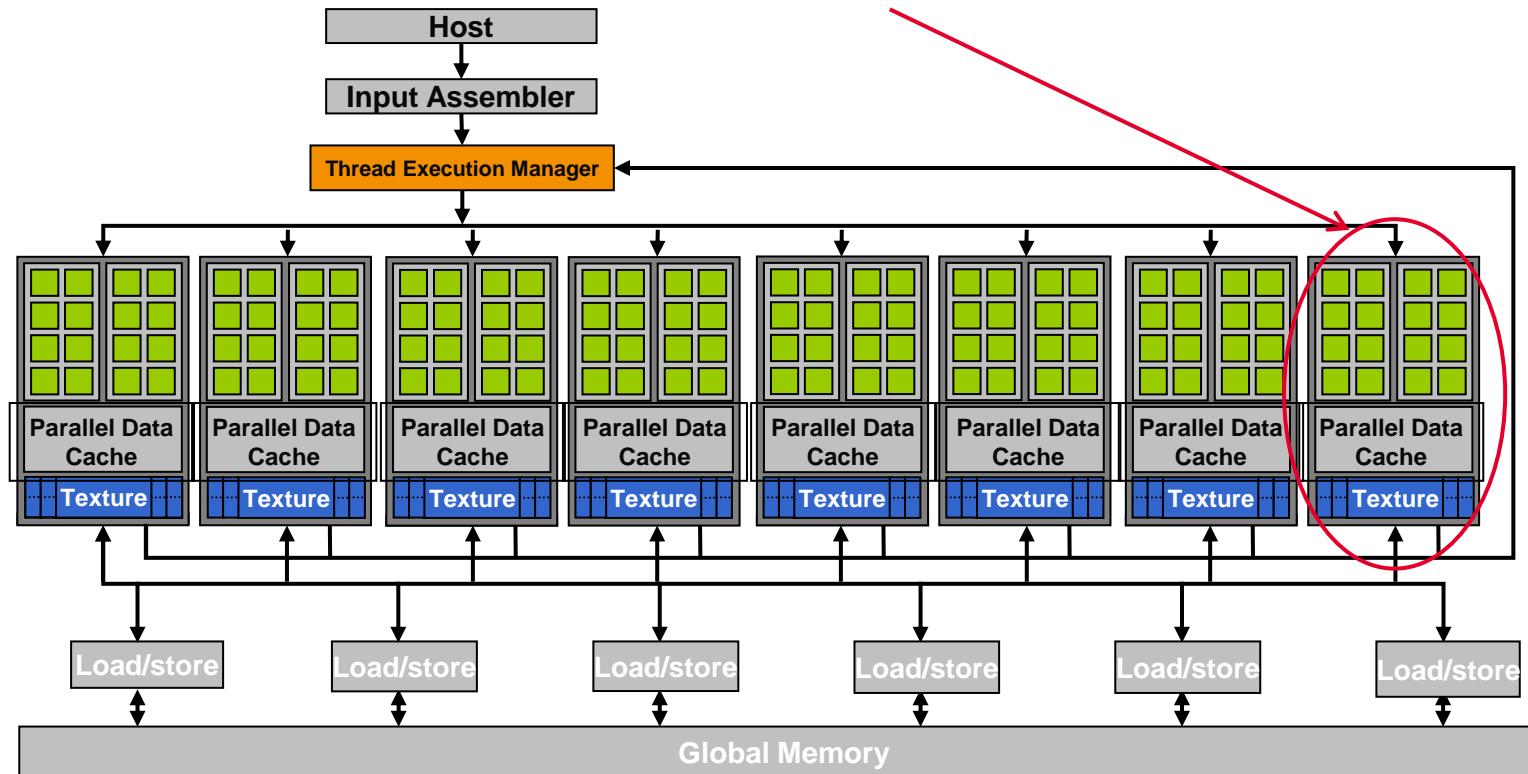
# I Divergence vláken – větvení kódu

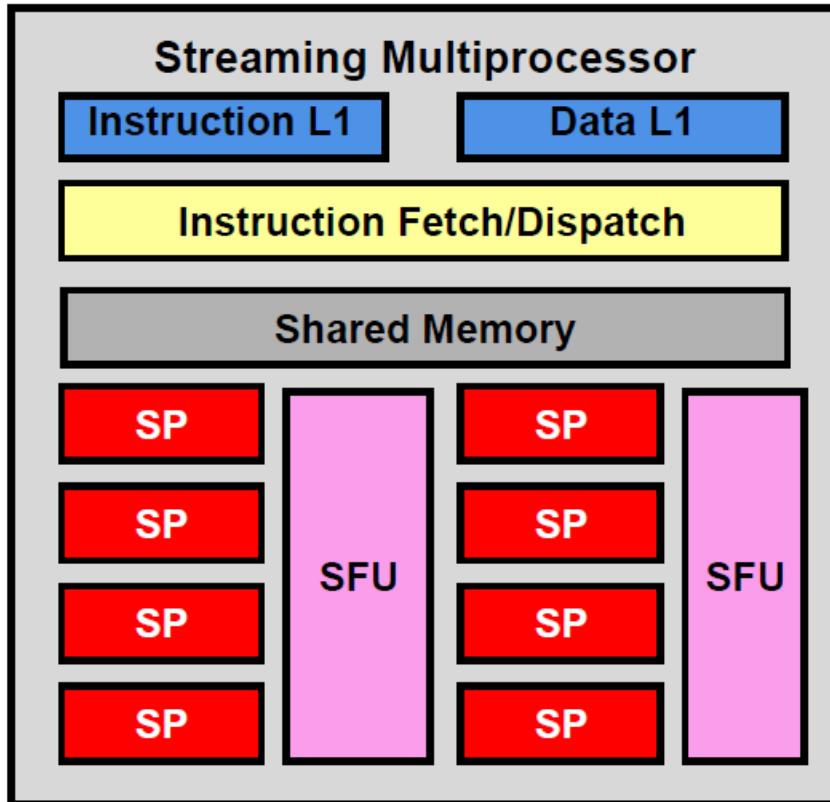
```
if (threadIdx >= 2)
 out[threadIdx] += 100;
else
 out[threadIdx] += 10;
```



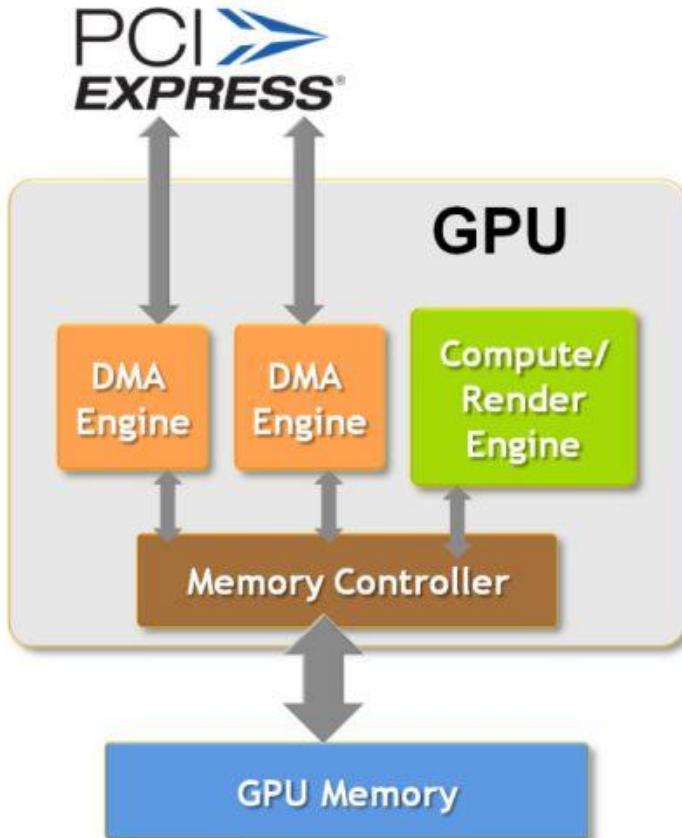
# I Unifikovaná architektura Nvidia G80

- G80 první jádro určené pro GPGPU s architekturou CUDA
- Jádro členěno do několika Streaming Multiprocesorů (SM)





- SM je složeno z
  - 8 CUDA jader (SP) – obsahuje skalární ALU (FMA)
  - 2 Super Function Units (SFU) – speciální jednotka pro počítání dělení, sin, cos, ln,...
  - 8k registrů
- Načítaní/vydávání instrukcí z více vláken
  - až 768 aktivních vláken
  - vždy se načítá instrukce pro 32 vláken (warp)
- 16 KB sdílené paměti



- GPU obsahuje DMA Engine pro přímé kopírování dat mezi pamětí počítače a GPU
- GPU grafické karty se tedy o kopírování dat nestará a ani o něm nemusí vědět
- Lze překrývat přenosy dat a výpočet
- Využitelné pro přístup do paměti jiné grafické karty v Multi-GPU systémech

# Architektura Nvidia Volta (GV100)



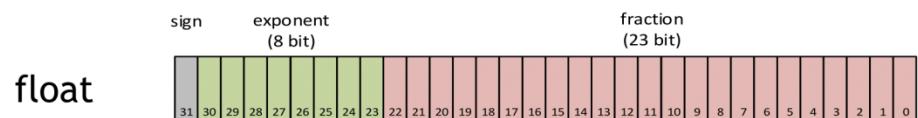
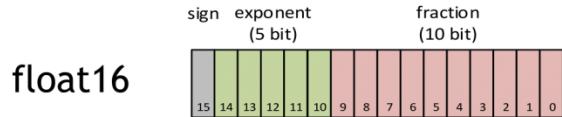
- Až 84 SM procesorů
  - 64 FP32 jednotek
  - 64 INT32 jednotek
    - Float a INT jde současně!
  - 32 FP64 jednotek
  - 8 Tenzorových jader

$$D = A \times B + C$$

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

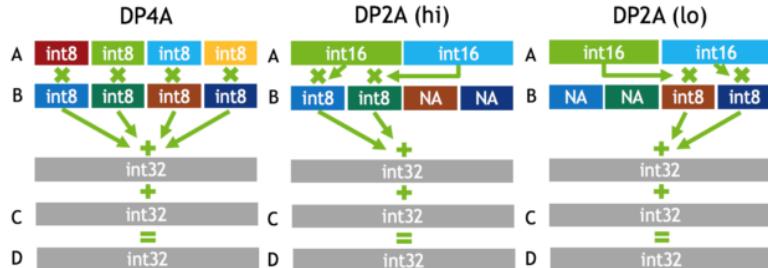
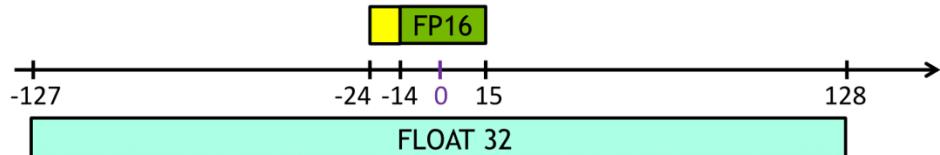
- Zvýšena efektivita L1
- Nový SIMT model
- Superskálární procesor, in-order procesor
- Nová L0 instrukční cache

## HALF-PRECISION FLOAT (FLOAT16)



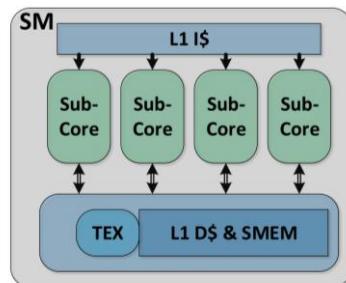
FLOAT16 has wide range ( $2^{40}$ ) ... but not as wide as FP32!

Normal range:  $[ 6 \times 10^{-5} , 65504 ]$   
 Sub-normal range:  $[ 6 \times 10^{-8} , 6 \times 10^{-5} ]$



## VOLTA GV100 SM

Redesigned for Productivity and Accessible Performance

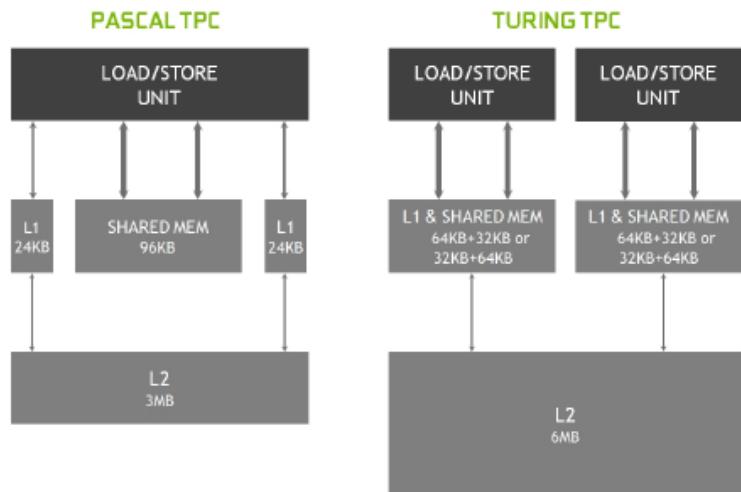


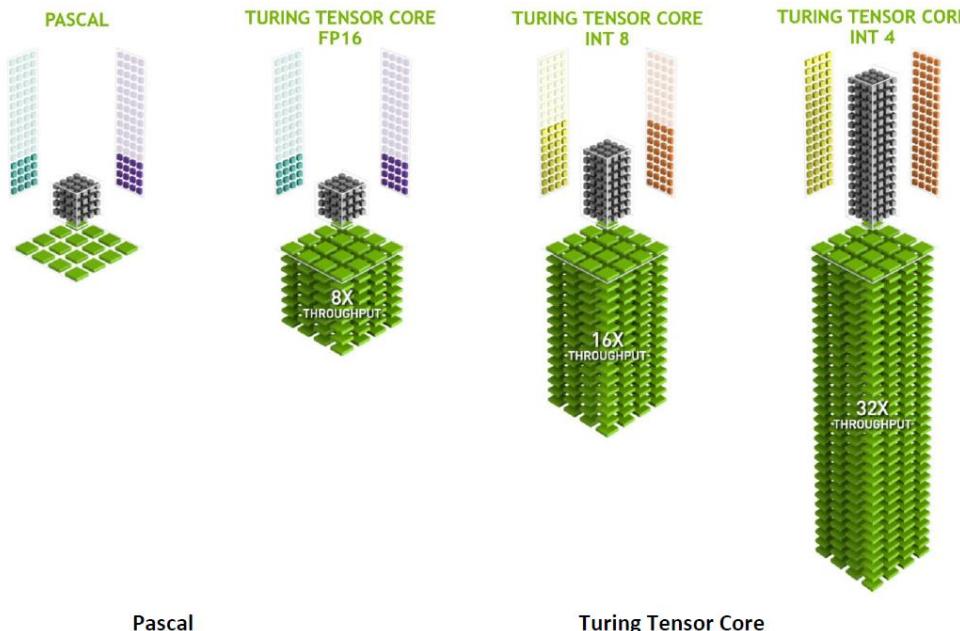
- Twice the schedulers
- Simplified Issue Logic
- Large, fast L1 cache
- Improved SIMD model
- Tensor acceleration
- +50% energy efficiency vs GP100 SM

# Architektura Nvidia Turing (TU102)



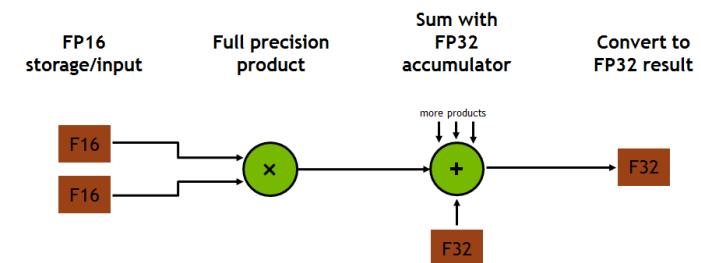
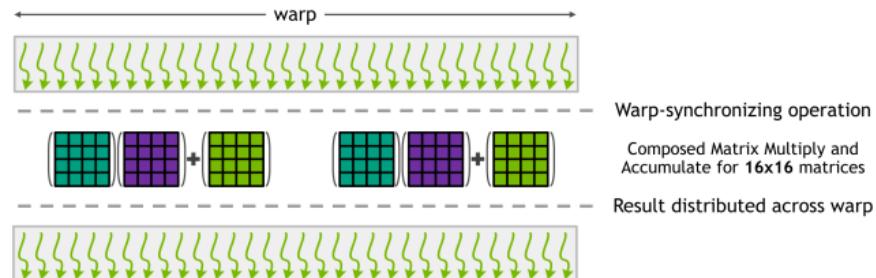
- Superskalární procesor
  - INT a FP32 současně
- Unifikovaná L1 a sdílená paměť
  - Zdvojnásobena propustnost i kapacita
- Přidána RTX jádra



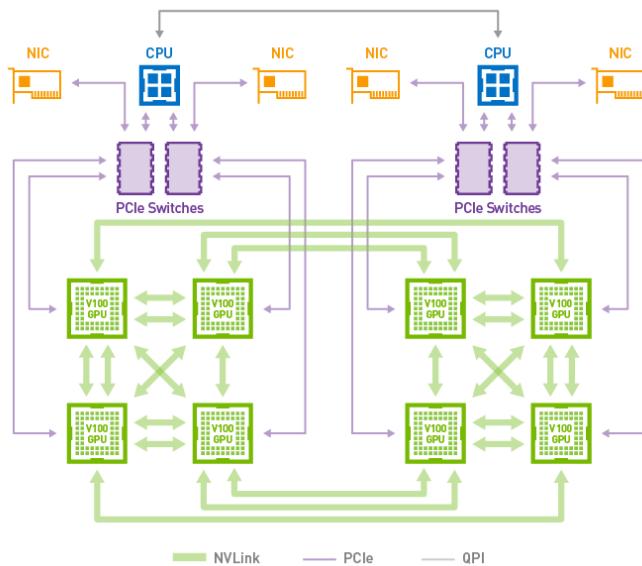


## TENSOR SYNCHRONIZATION

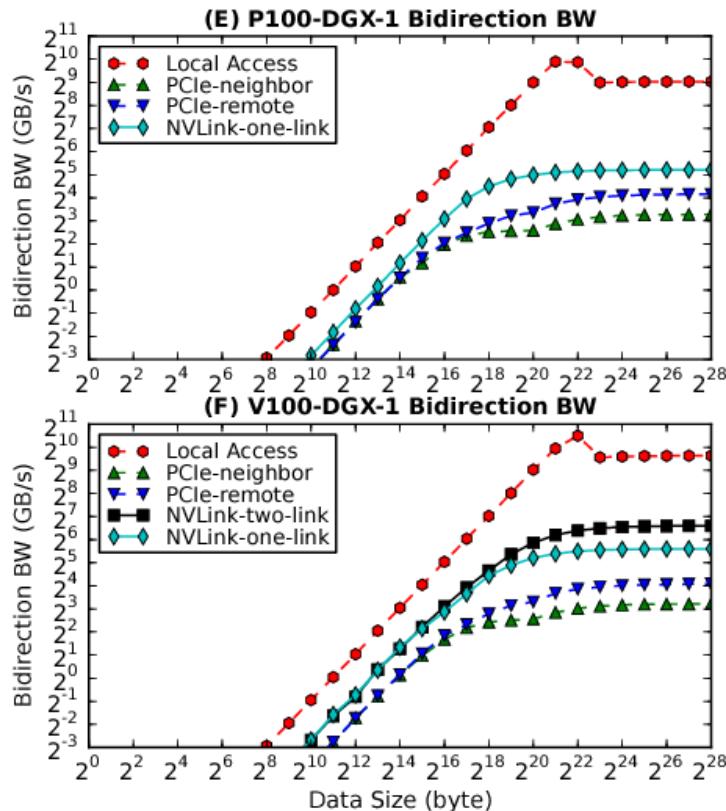
Full Warp 16x16 Matrix Math



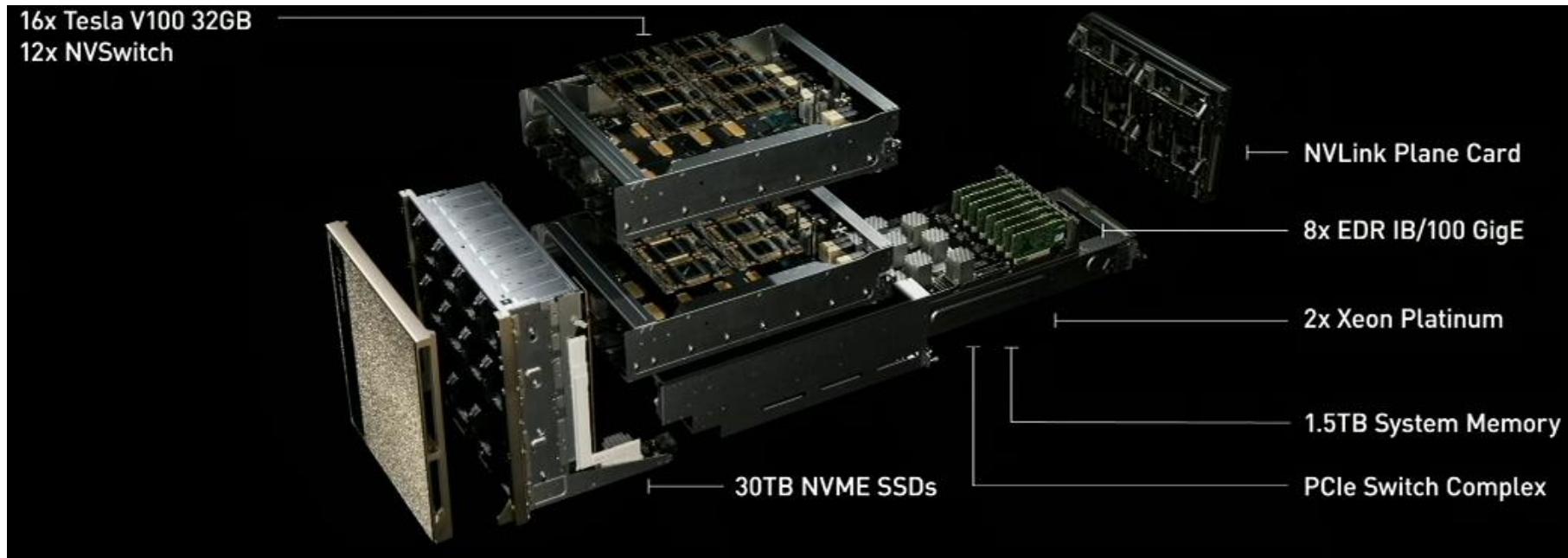
# I Nvidia NVLink propojení



- Vysokorychlostní propojení GPU
- GPU lze připojit i k CPU s podporou NVLink
- Hybrid Cube-mesh propojení
- Při 8 GPU až 160 GB propustnosti



- Ostravský nejvýkonnější GPU server
- [https://www.youtube.com/watch?v=gAByU0i6G\\_E](https://www.youtube.com/watch?v=gAByU0i6G_E)



# PROGRAMOVÁNÍ GRAFICKÝCH KARET

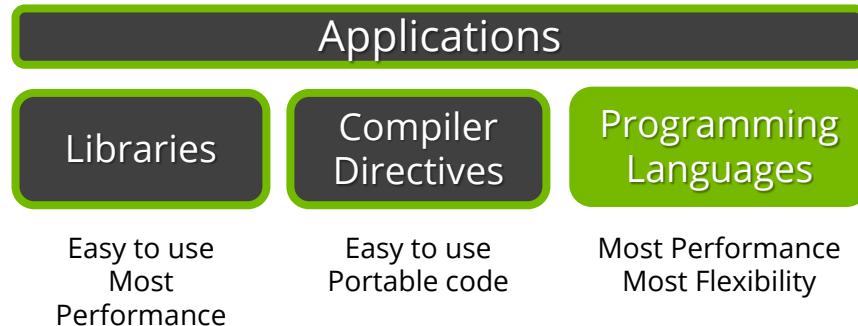
- **GPGPU – “General-purpose”** computing on graphics processing units

- **Vysokoúrovňové jazyky**

- **OpenMP** od verze 4.5 obsahuje konstrukci **target** jenž umožní vykonání kódu na akcelerátoru
- **OpenACC** – (PGI/NVIDIA) – obdoba OpenMP
- **MATLAB** – přetypováním na **gpuArray**
- **NVIDIA Thrust** – C++ interface pro GPU NVIDIA

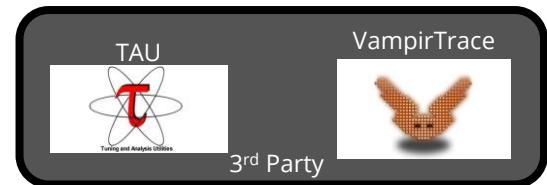
- **Nízkoúrovňové jazyky**

- **CUDA** – (NVIDIA), vysoce optimalizováno pro karty NVIDIA, obsahuje sadu důležitých knihoven FFT, BLAS, RAND ...
- **OpenCL** – (Khronos) – podpora i pro karty AMD, CPU, XeonPhi, DSP, FPGA
- **HIP** - C++ Heterogeneous-Compute Interface for Portability – AMD

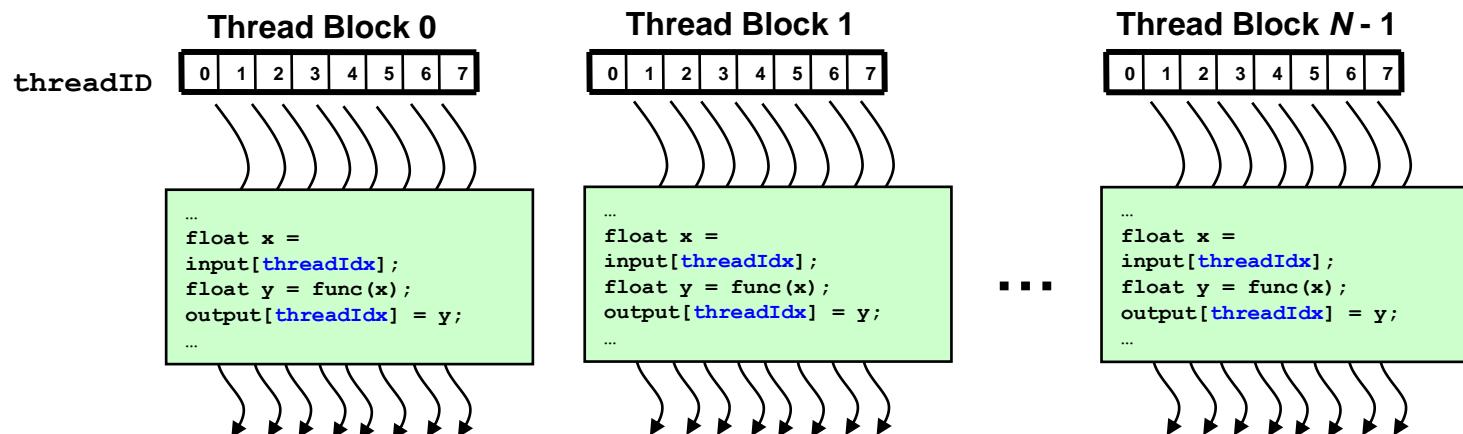


# Nvidia CUDA

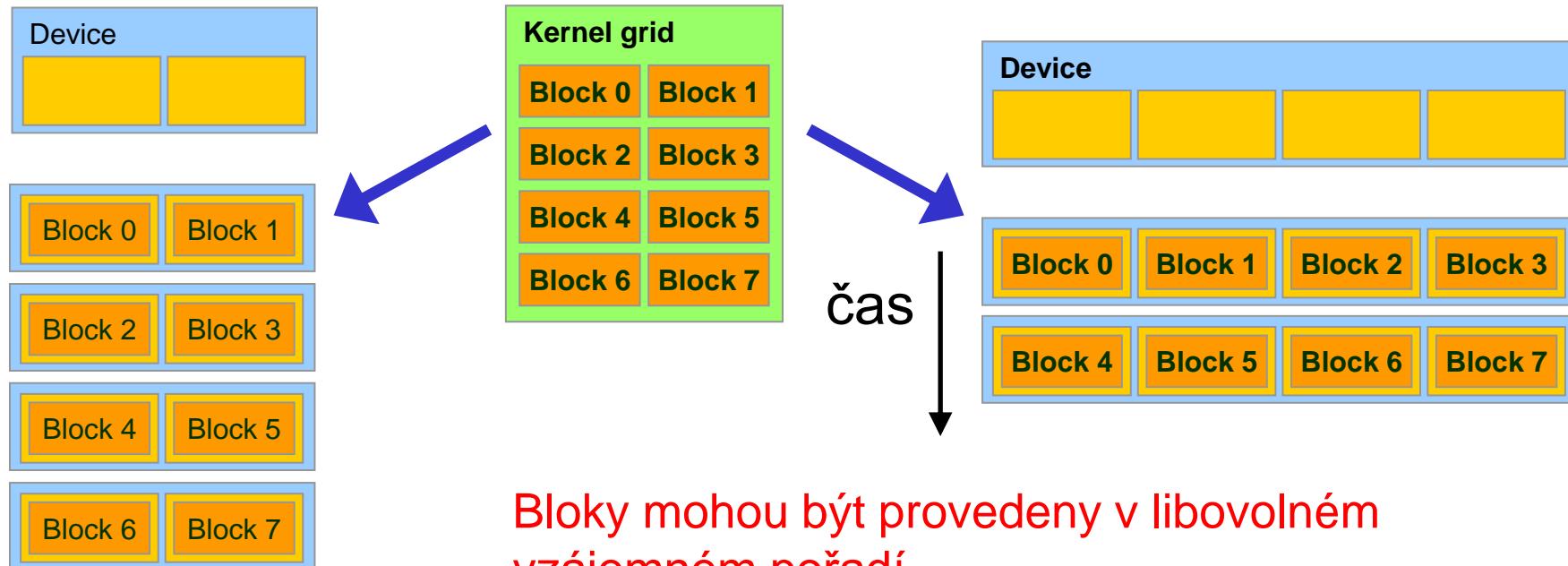
- „Compute Unified Device Architecture“
- Rozšíření jazyka C/C++ a knihovní funkce pro využití GPU jako obecné platformy
- Binding pro Fortran, Python, Java, Matlab, Ruby,...
- Obsahuje knihovny: cuBLAS, cuFFT, cuRand, Cuda Math Library, Thrust, Magma, NPP
- Debuggery a profilery
- Obsahuje funkce pro**
  - Přenos dat do paměti grafického adaptéru a zpět
  - Podpora pouze pro statická 1D pole a CUDA arrays
  - Rozčlenění výpočtu do vláken a bloků
  - Funkce pro synchronizaci vláken na úrovni bloků
  - Spuštění výpočtu na grafické kartě
  - <https://developer.nvidia.com/cuda-zone>



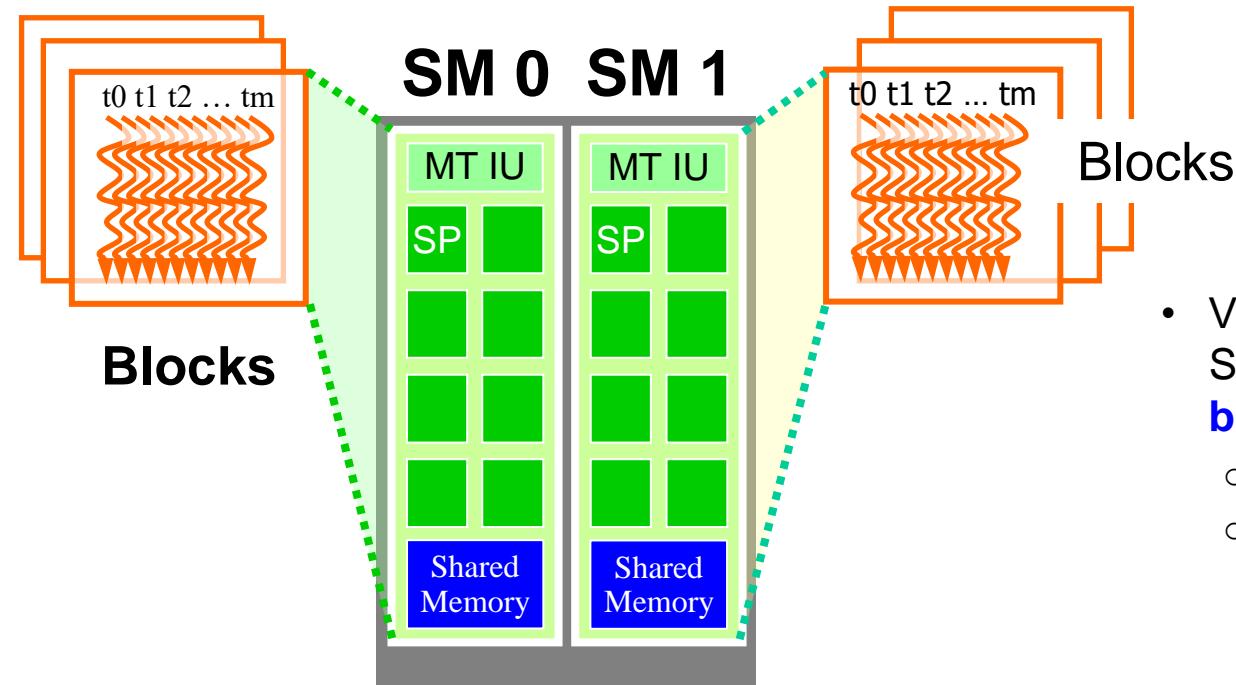
- **Funkce akcelerované pomocí GPU se nazývá kernel**
  - Kód se zapisuje z pohledu jednoho vlákna, ale vždy je nutné brát v potaz, že běží celý WARP.
- **Monolitická sada vláken je rozdělena do bloků**
  - Vlákna v rámci bloku spolupracují pomocí **sdílené paměti**, **atomických operací** a **bariérové synchronizace**
  - Vlákna v různých blocích **NEMOHOU** spolupracovat



- GPU může rozhodnout o libovolném přiřazení bloků na SM procesory
  - Spuštěný kernel (grid – sada všech bloků) je dobře škálovatelný na libovolném počtu SM procesorů.



Bloky mohou být provedeny v libovolném  
vzájemném pořadí

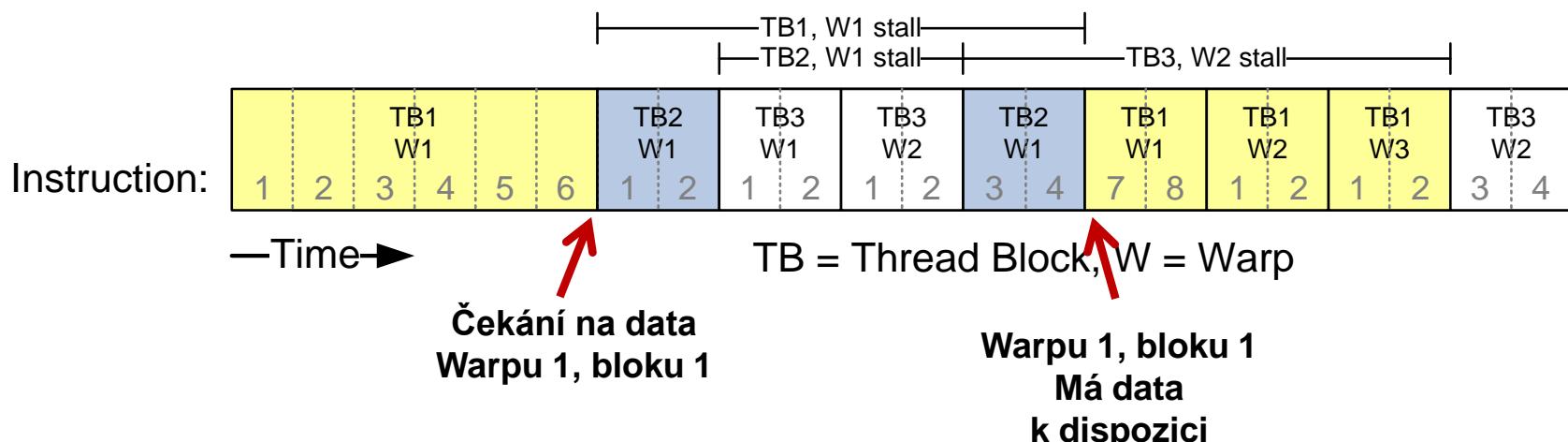


Flexibilní alokace zdrojů  
(registry, sdílená paměť)

- Vlákna jsou přiřazována na Streaming Multiprocesory **po blocích**
  - Až **8** bloků může sdílet zdroje SM
  - SM v **G80** zvládne až **768** threads
    - Např. 256 (vláken na blok) \* 3 bloky
    - Nebo 128 (vláken na blok) \* 6 bloků
    - atd.
- Vlákna běží současně
  - SM udržuje ID vláken/bloků
  - SM plánuje provedení vláken

- SM dokáže přepínat warpy s nulovou režií**

- Všechna vlákna v rámci warpu vykonávají **stejnou instrukci SIMT**
- V jeden okamžik může být vykonáván pouze jeden warp (Kepler a Maxwell dovolují více warpů)
- Pouze warpy jejichž následující instrukce má připraveny všechny operandy mohou být spuštěny (ready).
- Ready warpy jsou vybírány pro spuštění na základě priorit.
- Pokud nějaký warp zahájí čtení/zápis z/do globální paměti (až 200 taktů čekání) je odložen a nahrazen jiným, který může běžet

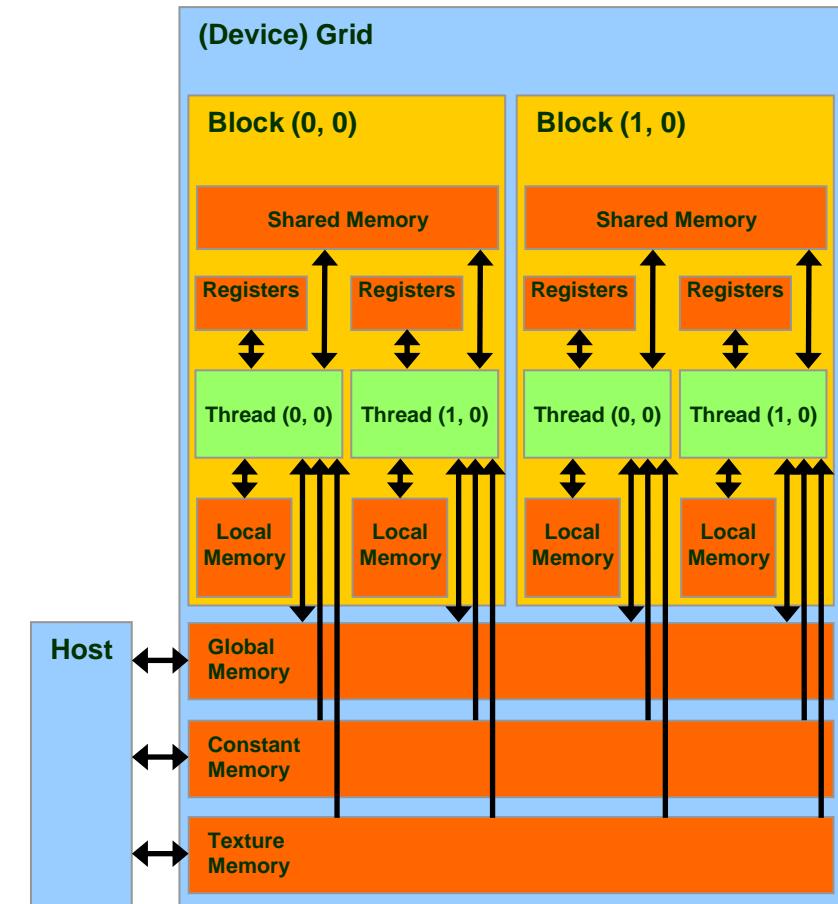


- **Vlákna CUDA pracují s**

- R/W **registry** vlákna
- R/W **lokální paměť** vlákna
- R/W **sdílená paměť** bloku
- R/W **globální paměť** gridu
- R **konstantní paměť** gridu
- R **texturní paměť** gridu

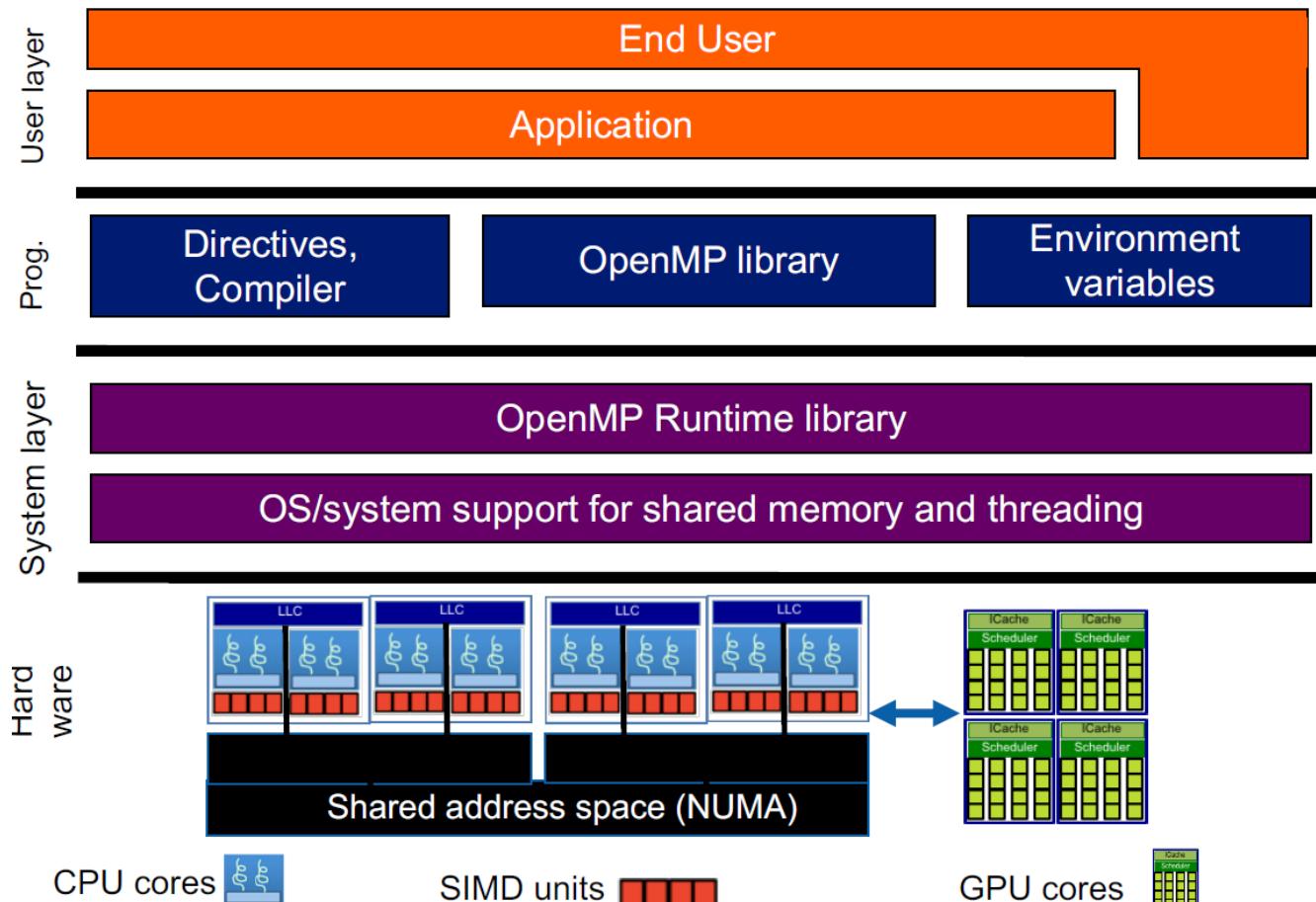
- **Host může přistupovat do**

- R/W **globální paměti**
- R/W **konstantní paměti**
- R/W **texturní pamětí**



# OPENMP FOR ACCELERATORS

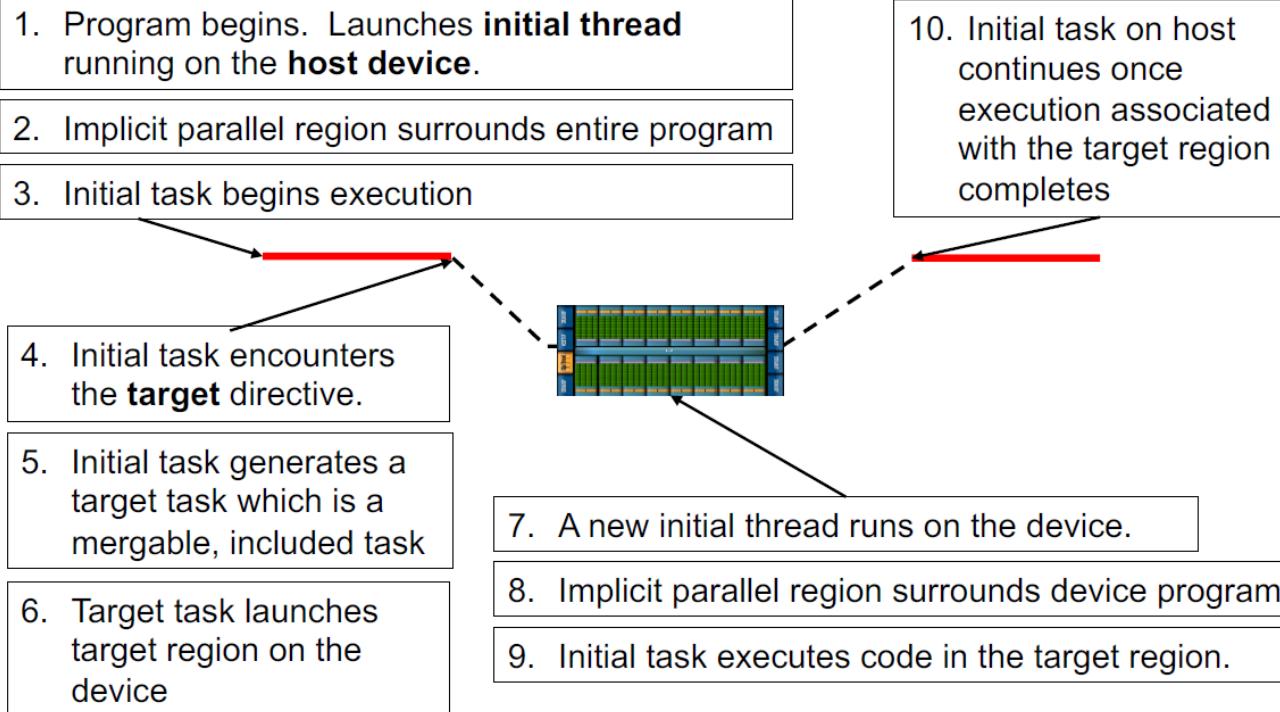
# Rozšíření OpenMP o podporu akcelerátorů



- Direktiva **target** předává (offload) výpočet na akcelerátor (**device**)

```
#pragma omp target
```

```
{....} // a structured block of code
```



- **Pamatujte: host (CPU) a device (GPU) mají oddělené paměťové prostory**
  - OpenMP používá kombinaci implicitních a explicitních data transferů.
  - Data mohou migrovat mezi host a device pouze v určitých, přesně definovaných místech
    - Typicky na začátku a na konci target regionu

```
#pragma omp target
{ // Data se kopírují z host na device
 ...
}
// Data se kopírují z device na host
```

- Implicitně se přenáší:
  - Skalární proměnné (`int N`) a to vždy jako **firstprivate**. Vždy pouze na device!
  - Pole, pokud je známá velikost (`double A[1000]`). Provádí se kopie tam i zpět!
  - **Pointery se kopírují jako firstprivate – ALE už ne data na která ukazují!!!**

# Příklad: Transfery dat mezi host a device

```
int main(void) {
 int N = 1024;
 double A[N], B[N];

 #pragma omp target
 {
 for (int ii = 0; ii < N; ++ii) {
 A[ii] = A[ii] + B[ii];
 }
 } // end of target region
}
```

1. Variables created in host memory.

2. Scalar **N** and stack arrays **A** and **B** are copied to device memory. Execution transferred to device.

3. **ii** is **private** on the device as it's declared within the target region

4. Execution on the device.

5. stack arrays **A** and **B** are copied from device memory back to the host. Host resumes execution.

# | Paralelní vykonávání kódu na host a device

```
#pragma omp target nowait
{
// code defines a target region
}
```

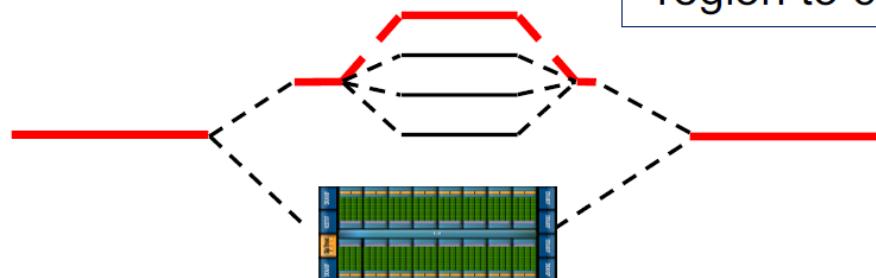
- Make the target task (running on the host) execute as a deferred task

```
#pragma omp parallel for
for(int i=0;i<N;i++) {
 big_stuff(i);
}
```

- The thread's implicit task can define other (potentially parallel) work.

```
#pragma omp taskwait
```

- The implicit task running on the host waits for the target region to complete.

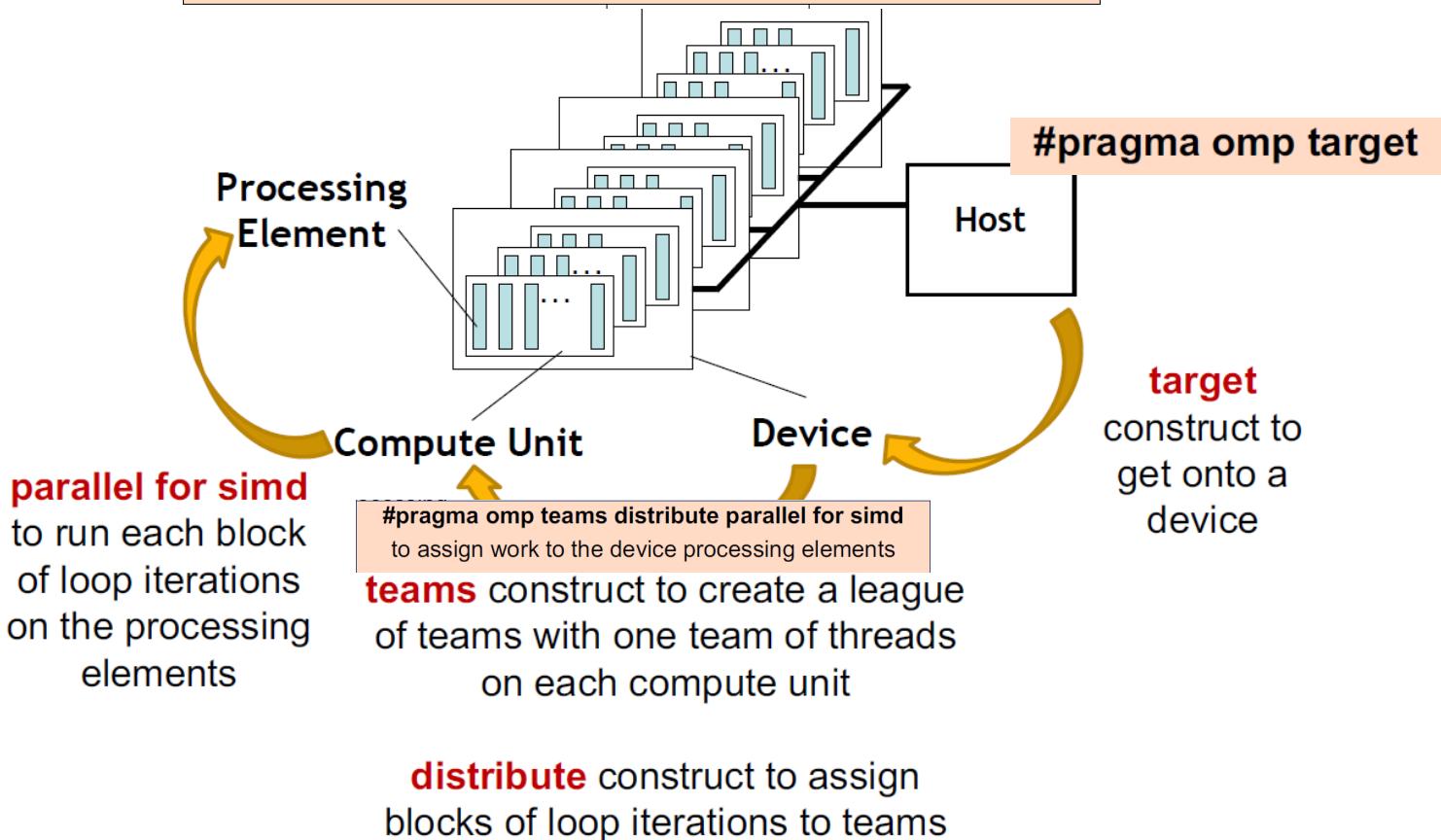


## #pragma omp target [clause[,]clause]...] structured-block

- **if(scalar-expression)**
  - Pokud se vyhodnotí jako true, vykonání proběhne na device, v opačném případě na hostu
- **device(integer-expression)**
  - ID device, které se má použít
- **private(list) firstprivate(list)**
  - Seznam privátních a first privátních proměnných
- **map(map-type: list)**
  - Kopie polí na device.
    - **to** – na začátku oblasti target kopíruj data na device. Např. `map (to:A[100:200])` – vždy od:kolik
    - **from** – na konci oblasti target kopíruj data zpět na host. Např. `map (from:B[100:200])`
    - **tofrom** – kopíruj data oběma směry
    - **alloc/delete** – nic nekopíruj, pouze alokuj a pak zlikviduj pole na GPU (typicky pomocná pole)
- **nowait**
  - Nečekej na dokončení výpočtu na device

```
int N = 1024;
int* A = malloc(sizeof(int)* N);
#pragma omp target map(A[0:N])
{
 // N, ii and A all exist here
 // The data that A points to DOES exist here!
} 0
```

Typical usage ... let the compiler do what's best for the device:



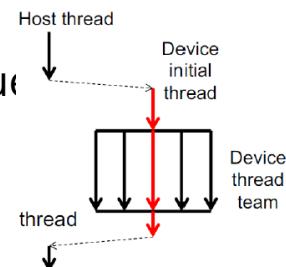
- **The teams construct**

- Similar to the parallel construct
- It starts a league of teams
- Each team in the league starts with one initial thread – i.e. a team of one thread
- Threads in different teams cannot synchronize with each other
- The construct must be “perfectly” nested in a target construct

```
#pragma omp target
#pragma omp parallel for
for (i=0;i<N;i++)
...
...
```

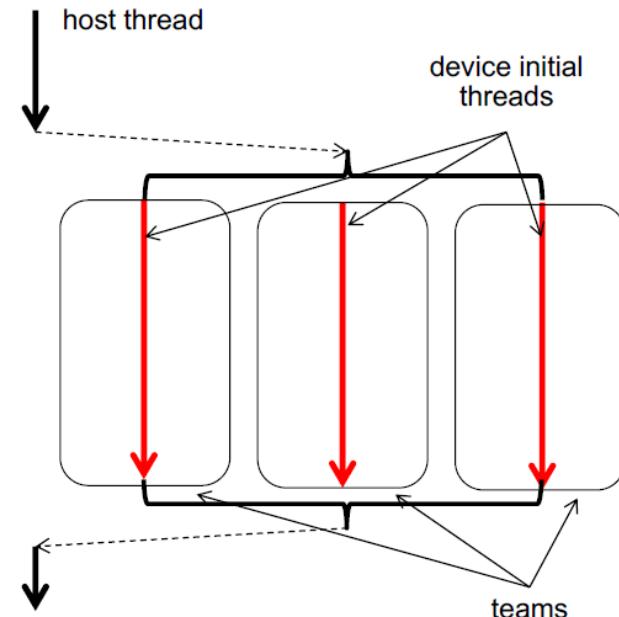
- **The distribute construct**

- Similar to the for construct
- Loop iterations are workshared across the initial threads in a league
- No implicit barrier at the end of the construct
- dist\_schedule(kind[, chunk\_size])
  - if specified, scheduling kind must be static
  - Chunks are distributed in round-robin



- teams construct
- distribute construct

```
#pragma omp target
#pragma omp teams
#pragma omp distribute
for (i=0;i<N;i++)
...
...
```

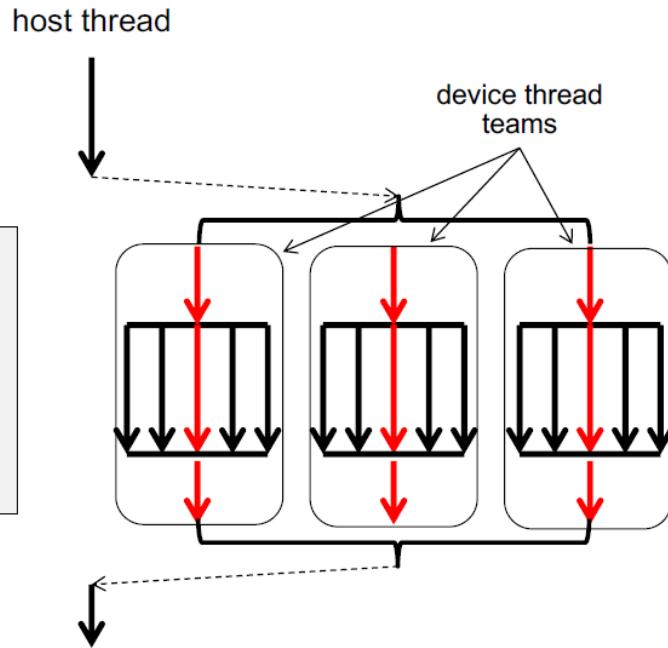


- Transfer execution control to **MULTIPLE** device initial threads
- Workshare loop iterations across the initial threads.

Note: number of teams is implementation defined, good for portable performance. Compilers can choose how they map teams and threads.

- teams distribute
- parallel for simd

```
#pragma omp target
#pragma omp teams distribute
for (i=0;i<N;i++)
#pragma omp parallel for simd
for (j=0;j<M;j++)
...
...
```



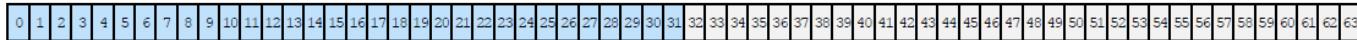
- Transfer execution control to **MULTIPLE** device initial threads (one per team)
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the master thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for simd)

# Ukázka distribuce práce

```
#pragma omp target teams distribute parallel for simd \
 num_teams(2) num_threads(4) simdlen(2)
for (i=0; i<64; i++)
...
...
```

64 iterations assigned to 2 teams;  
Each team has 4 threads;  
Each thread has 2 SIMD lanes

Distribute iterations across 2 teams



In a team, **workshare** (parallel  
for) iterations across 4 threads



In each thread use  
**SIMD** parallelism

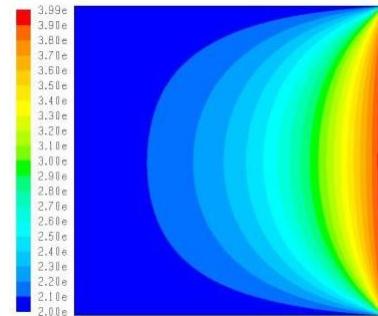


# Ukázka: Paralelní stencil operace

```
// Compute the next timestep, given the current timestep
void solve(const int n, const double alpha, const double dx, const double dt, const double * restrict u,
double * restrict u_tmp) {
 // Finite difference constant multiplier
 const double r = alpha * dt / (dx * dx);
 const double r2 = 1.0 - 4.0*r;

 // Loop over the nxn grid
#pragma omp target map(tofrom: u[0:n*n], u_tmp[0:n*n])
#pragma omp teams distribute parallel for simd collapse(2)
 for (int i = 0; i < n; ++i) {
 for (int j = 0; j < n; ++j) {

 // Update the 5-point stencil, using boundary conditions on the edges of the domain.
 // Boundaries are zero because the MMS solution is zero there.
 u_tmp[i+j*n] = r2 * u[i+j*n] +
 r * ((i < n-1) ? u[i+1+j*n] : 0.0) +
 r * ((i > 0) ? u[i-1+j*n] : 0.0) +
 r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
 r * ((j > 0) ? u[i+(j-1)*n] : 0.0);
 }
 }
}
```



Add the BUD to the loops  
Use collapse clause to increase parallelism

$$u_{i,j}^{n+1} = u_{i,j}^n + \frac{\alpha k}{h^2} (u_{i,j+1}^n + u_{i,j-1}^n + u_{i+1,j}^n + u_{i-1,j}^n - 4u_{i,j}^n)$$

| Teorie znáte, tak hurá do praxe



Veselé Vánoce a šťastný Nový rok  
přeje Jirka Jaroš a kol.

