

Pokročilé architektury počítačů

Petr Jedlička, Cyril Klimeš, Tomáš Pitner

Předmluva

HTML · Slides · PDF | Přehled předmětu

Cíle

- Poznáme, co jsou hlavní parametry výkonnosti počítače
- Jaké jsou možnosti zvýšení ať už na úrovni jádra tak více jader a celého systému
- Zcela nové koncepty výpočetní techniky: kvantové počítače, neuronové počítače

🔑 Moderní architektury počítačů pro **vysoké výkony**, **umělou inteligenci** atd. Tato kapitola navazuje na kapitolu **Procesory**. Soustředí se hlavně na **pokročilejší mikroarchitekturu**, **paralelismus** a **současné směry vývoje**.

Základní měřítka výkonnosti počítače

- Co a jak ovlivňuje výkon programu?

Doba odezvy/běhu
(execution/response time)

- jak dlouho počítač vykonává konkrétní úlohu
- zajímavé pro konkrétního uživatele počítače

Propustnost
(throughput)

- kolik práce počítač vykoná za jednotku času
- zajímavé pro správce serveru nebo datového centra

Jak se změní odezva a propustnost…

- pokud vyměníme procesor za **rychlejší**?
- pokud přidáme do systému **další procesory**?

Výkonnost určená dobou odezvy

- `Výkon_počítače = 1/Odezva_počítače`
- maximalizace výkonu minimalizace odezvy
- $N = \text{výkon}_A / \text{výkon}_B$
- Kolikrát je `A` rychlejší (výkonnější) než `B`…
- pokud `A` provede program za 10 sekund a `B` stejný program za 15 sekund?

Jak měřit výkonnost z pohledu uživatele?

Celkový čas běhu úlohy

- zahrnuje vše, včetně přístupu na disk a režie OS
- co když je počítač sdílen více uživateli?
- odráží **výkonnost systému** jako celku

Procesorový čas

- čas, po který procesor program skutečně vykonával
- nezahrnuje čekání na V/V zařízení a čas kdy neběžel
- zahrnuje čas strávený uvnitř OS — režii OS
- **uživatelský** a **systémový** procesorový čas
- odráží **výkonnost procesoru**

Jak měřit výkonnost z pohledu návrháře?

Rychlost provádění základních operací

- všechny události řízeny hodinovým signálem
- frekvence hodin (*clock rate*), např. 3 GHz
- taktovací frekvence, frekvence hodinového signálu, …
- délka hodinového cyklu (*clock cycle*), např. 250 ps
- doba periody hodinového signálu

Jak měřit výkonnost z pohledu návrháře? (2)

- Jak může návrhář zlepšit výkon?
- *CPU execution time for a program = CPU clock cycles for a program × Clock cycle time*
- *CPU execution time for a program = CPU clock cycles for a program / Clock rate*

Příklad: zlepšování výkonnosti

- Jakou taktovací frekvenci potřebujeme, pokud …
- program běží 10s na počítači A s hod. frekvencí 2 GHz
- Cílem je navrhnout počítač B, kde program poběží 6s
- technologie umožňuje výrazně zvýšit taktovací frekvenci…
- … ale pro vykonání programu je potřeba o 20% více cyklů

Příklad: zlepšování výkonnosti

- *CPU time A = CPU clock cycles A / Clock rate A*
- *10 = CPU clock cycles A / 2×10 [s]*
- *CPU clock cycles A = 10×2×10=20×10*
- *CPU time B= 1.2×CPU clock cycles A / Clock rate B*
- *6 = 1.2×20×10 / Clock rate B [s]*
- *Clock rate B = 1.2×20×10 /6 =0.2×20×10=4×10=4GHz*

Měření výkonnosti: SPEC CPU Benchmark

Workload

- typická pracovní zátěž — sada spouštěných programů
- porovnání počítačů porovnání doby běhu programů
- časově náročné, špatně se automatizuje

Benchmark

- program specificky určený k měření výkonnosti
- sada benchmarků ≈ typická pracovní zátěž procesoru
- doufáme, že výsledky předpoví chování při aktuální zátěži

Měření výkonnosti: SPEC CPU Benchmark (2)

SPEC

Standard Performance Evaluation Corporation

- snaha výrobců o možnost jednoduše porovnat počítače
- původně hlavně CPU, dnes i GPU, Java, mail, …
- nejnovější **SPEC CPU 2026**, předchozí **SPEC CPU 2017**

Co měříme?

Samotné jedno číslo nestačí:

- důležité jsou **jednovláknový výkon**,
- **vícevláknový výkon**,
- **paměťová propustnost**, **latence** a
- **výkon na watt**.

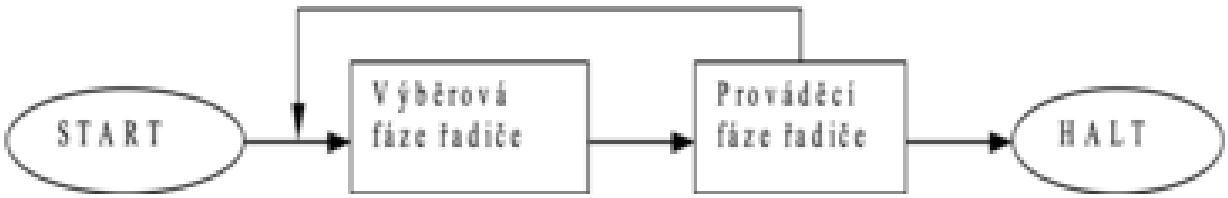


Prakticky se často používají i veřejné přehledy typu CPU Mark, ale je nutné vědět, co přesně měří.

Základní instrukční cyklus

Jde jen o stručné připomenutí. Podrobnější výklad je už v kapitole 5. Procesory.

Instrukční cyklus řadiče



Instrukční cyklus řadiče (2)

Výběrová fáze

- 1. počáteční nastavení čítače instrukcí
- 2. načtení instrukce do RI
- ČI → adresa RAP
- Načtení instrukce do RDP
- RDP → RI

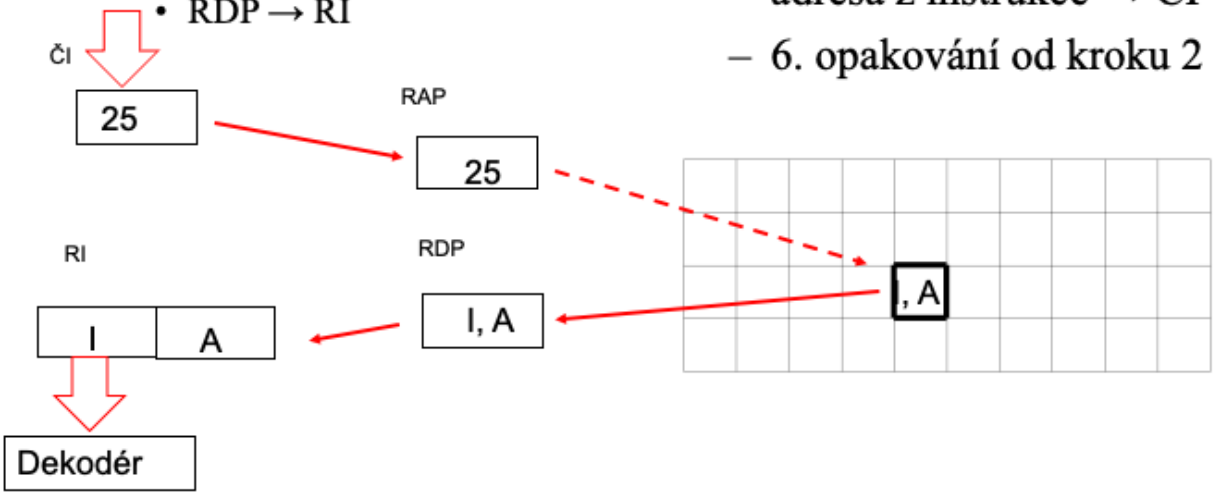
Prováděcí fáze

- 3. dekódování operačního kódu (OK) instrukce
- 4. provedení operace
- 5. ČI + 1 → ČI nebo adresa z instrukce → ČI
- 6. opakování od kroku 2

Instrukční cyklus řadiče (3)

Instrukční cyklus řadiče

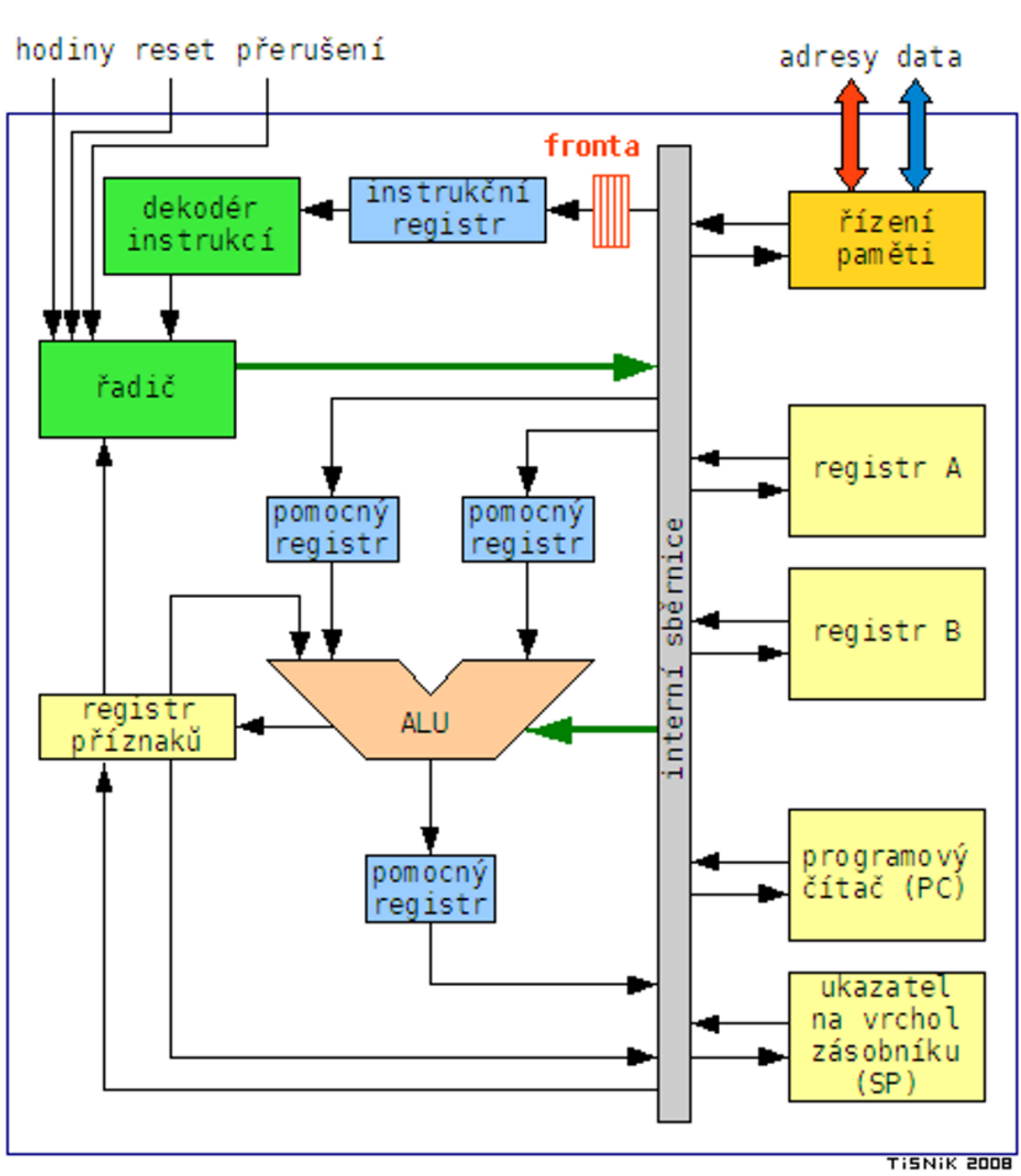
- Výběrová fáze
  - 1. počáteční nastavení čítače instrukcí
  - 2. načtení instrukce do RI
    - ČI → adresa RAP
    - Načtení instrukce do RDP
    - RDP → RI
- Prováděcí fáze
  - 3. dekódování operačního kódu (OK) instrukce
  - 4. provedeni operace
  - 5. ČI + 1 → ČI nebo adresa z instrukce → ČI
  - 6. opakování od kroku 2



Techniky zvýšení výpočetního výkonu procesorů

- Zvyšování integrace, zmenšení rozměrů a tím zvyšování taktovací frekvence
- Snižování napájecího napětí, tím snížení vyzařovaného tepla a tím další zvyšování taktovací frekvence
- Rozšíření bitové šířky zpracovaných dat
- Zvýšení počtu pracovních registrů
- Hierarchické uspořádání paměti — na vrcholu Level 1 cache, pak Level 2 cache
- Fronty instrukcí — mezi instrukčním registrem a pamětí

Umístění fronty instrukcí



Další techniky zvýšení výpočetního výkonu procesorů

- Zřetězené (proudové) zpracování instrukcí (*pipelining*) včetně vyřešení predikce skoků
- Počítače s redukovaným souborem instrukcí - *RISC*
- Explicitní *paralelní* zpracování instrukcí (VLIW)
- *Superskalární* architektura
- Vykonávání mimo pořadí (*out-of-order execution*)
- Přejmenování registrů (*register renaming*)

- *Vícevláknové* zpracování v jednom jádru (*SMT*)
- *Víceprocesorové* systémy
- Masivně paralelní architektury
- Počítače s netradičním řízením

Podstata metody proudového zpracování instrukcí

Postupné zpracování

- Předpoklad: *n* funkčních bloků
- Každý z bloků je během *n-1* časových úseků nevyužit a čeká na příchod dalšího operandu

Proudové zpracování

- Využití všech bloků současně
- V určitém okamžiku je v každém bloku zpracován jiný operand

4 fáze provádění instrukce

VI **výběr** instrukce

DI **dekódování** instrukcí

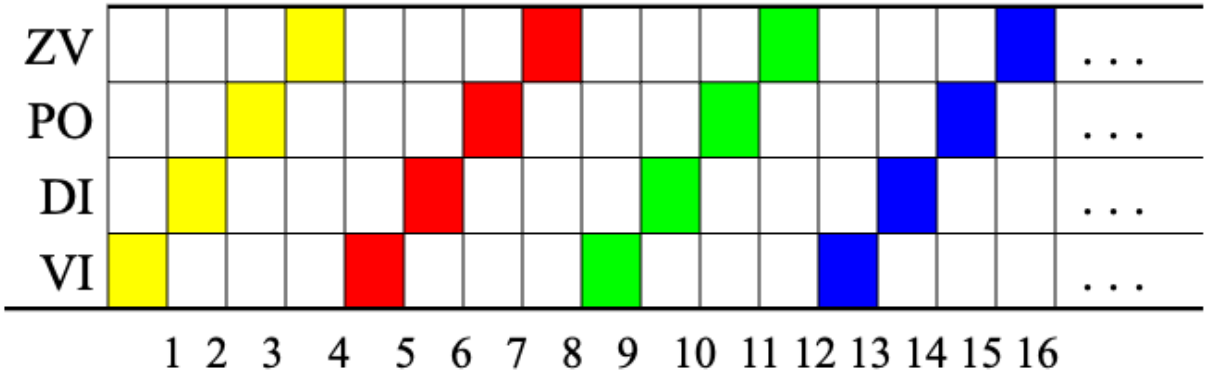
PO provedení **operace**

ZV zápis výsledku **do paměti**

Neproudové (postupné) zpracování

- Jednoduchý, leč neefektivní model bez "pipeline".
- Bez pipeline se každá instrukce zpracuje *až po dokončení* předchozí.
- Jednotlivé funkční bloky proto velkou část času *čekají*.

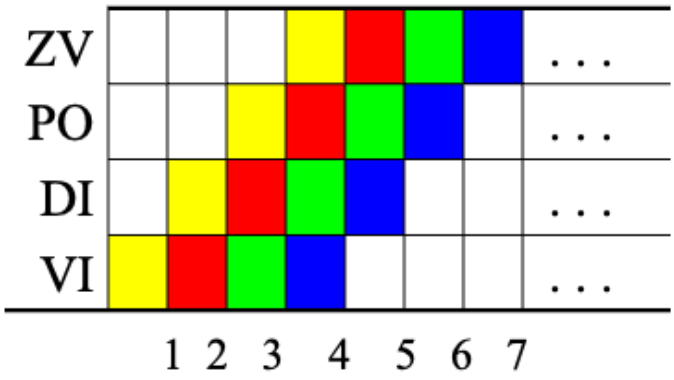
Neproudové zpracování: příklad



Proudové zpracování

- Pipeline rozděluje vykonání instrukce na více fází.
- V jednom okamžiku tak mohou být různé instrukce **v různých fázích** zpracování.
- Přínosem je hlavně vyšší **propustnost**, ne nutně kratší doba jedné konkrétní instrukce.

Proudové zpracování: příklad



Propustnost pipeline

- Pro ideální pipeline s `n` sekcemi a `k` úlohami je doba zpracování přibližně `n + k - 1` taktů.
- Bez pipeline by při stejném zjednodušení bylo potřeba přibližně `n * k` taktů.
- Pro `k >> n` se výsledná propustnost blíží jedné dokončené úloze za takt.



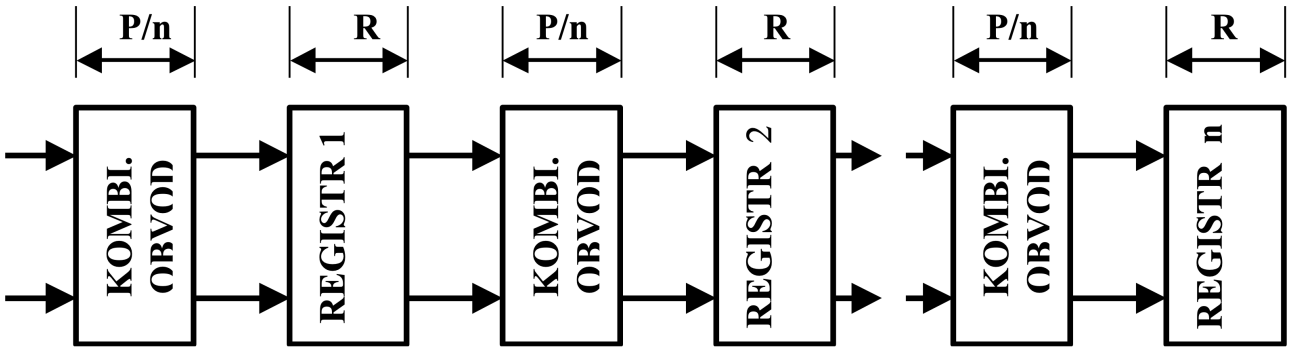
Skutečný přínos snižují **strukturální**, **datové** a **řídicí** konflikty.

Optimalizace struktury proudově pracující jednotky

- Čím je použit **větší počet sekcí**, tím více úloh může jednotka zpracovávat současně a tím může růst propustnost.

- Každá sekce musí mít **registr** pro zaznamenání současně zpracovaných úloh — tím roste cena funkční jednotky a roste zpoždění, způsobené průchodem jednotlivými registry.

Struktura vhodná pro proudové zpracování



Předpoklad: stejná doba zpracování ve všech sekcích

- Zpoždění v jedné sekci bude `T = P / n + R`
- Výkon = frekvence, s jakou je možno přivádět operandy `V = 1 / T`
- Cena `N=nL+C` kde `C` je cena kombinačního obvodu a `L` je cena jednoho registru
- Součinitel kvality `Q = V / N`
- `Q = 1/(PL + CR + nLR + PC/n)`

Plnění fronty instrukcí

- Při lineárním programu postupné plnění fronty
  - *Nepodmíněný* skok
  - *Podmíněný* skok — metody řešení

Konflikty u řetězeného zpracování

Strukturální

obvodová struktura neumožňuje současné provedení určitých akcí, např. současné čtení dvou hodnot z paměti nebo současné provedení dvou sčítání, pokud má procesor jednu ALU

Datové

když jsou zapotřebí data z předcházející instrukce, která není dokončena.

Řídicí

když skoková instrukce mění obsah PC, nebo jiné.

5 fází provádění instrukce

VI **Výběr** instrukce z paměti

DI **Dekódování** instrukce

VA Výpočet **reálné adresy**

PO Provedení **operace**

ZV Zápis výsledku **do paměti**

Provádění instrukcí bez překrytí

1. instrukce	VI	DI	VA	PO	ZV										
2. instrukce						VI	DI	VA	PO	ZV					
3. instrukce											VI	DI	VA	PO	ZV

Proudové zpracování: Strukturální konflikt

1. instrukce	VI	DI	VA	PO	ZV										
2. instrukce		VI	DI	VA	PO	ZV									
3. instrukce			VI	DI	VA	PO	ZV								
4. instrukce				VI	DI	VA	PO	ZV							
5. instrukce					VI	DI	VA	PO	ZV						

Strukturální konflikt

- požadavek na současné *čtení 5. instrukce* a *zápis operandu v 1. instrukci*
- *rozdělit paměť* na paměť instrukcí a paměť dat, obecné řešení:
- *přidání výpočetních jednotek*

Datové konflikty

1. **Sečti do R1, hodnoty R2, R3** (R1 ← R2 + R3)
2. **Sečti do R4, hodnoty R5, R1** (R4 ← R5 + R1)
   - Nemohu začít sečítání R5 + **R1** před vypočtením **R1** ← R2 + R3.

Řídicí konflikty

```
Instr 1 VI DI VA PO ZV
Instr 2    VI .  .  .  . DI VA PO ZV
Instr 3                  VI DI VA PO ZV
Instr 4                     VI DI VA PO ZV
```

- Nevím, kam skočit! Adresa bude známa až po fázi `PO` na řádku `BC`.

Řešení

1. Zpožděný skok, tedy vložit jiné, užitečné instrukce, což znamená *přeskládat instrukce programu* (**out-of-order execution**). Nelze-li takové užitečné instrukce najít, vložit *NOP* instrukce.
2. *Predikce* skoku, rozpracovat pouze predikovanou možnost (**branch prediction**).
3. Rozpracovat *obě možnosti* pokračování skokové instrukce (**speculative execution**).

Předávání dat

Mezi instrukcemi, které za sebou těsně následují — *rozdílná délka* provádění instrukcí:

- metoda **vkládání cyklů** až se dokončí předcházející instrukce
- metoda **vzájemného blokování** zpoždění druhé instrukce, než je operand k dispozici

Metoda vkládání cyklů

	VI	DI	VA	PO	ZV			
1.vložená instrukce		VI	DI	VA	PO	ZV		
2. vložená instrukce			VI	DI	VA	PO	ZV	
				VI	DI	VA	PO	ZV

Metoda vzájemného blokování instrukcí

Procesor musí respektovat datové závislosti mezi instrukcemi.

```
ldr x0, [x1]        // načtení hodnoty z paměti
add x2, x0, x3      // čeká na výsledek LDR
add x5, x6, x7      // nezávislá instrukce
```

Instrukce `add x2, x0, x3` nemůže být provedena, dokud není známa hodnota registru `x0`.

Moderní procesor s *out-of-order execution* však může mezitím provést nezávislou instrukci `add x5, x6, x7`.



Blokována není nutně celá pipeline. Procesor pozastaví především instrukce závislé na dosud nedostupných operandech a pokračuje ve vykonávání nezávislých instrukcí.

Technika určení cíle skoku

- Procesor se snaží určit cíl skoku ještě dříve, než je instrukce plně dekódována a vyhodnocena.
- Aby mohl předvybrat instrukci.

```
loop:
```

```
    ...
    cbnz x0, loop
```

U podmíněného skoku musí procesor predikovat dvě věci:

uskutečnění skoku zda bude skok proveden (*taken / not taken*)

cíl skoku kam se má pokračovat, pokud bude skok proveden

Predikce cíle skoku: Branch Target Buffer

- tabulka s cílovými adresami skoků uchovává přibližně vztah:

```
adresa instrukce skoku  ->  predikovaná cílová adresa
0x10001020              ->  0x10000FE0
```

- Díky tomu může *fetch* začít načítat instrukce z predikovaného cíle ještě před dokončením dekódování skoku.

Předvídání cíle u nepřímých skoků

- U nepřímých skoků je predikce cíle ještě důležitější:

```
br x8
```



Cílová adresa zde není přímo zakódována v instrukci a závisí na aktuální hodnotě registru x8.

Když umíme předvídat skok a cíl

- *Řešení je pokračovat* ve zpracování sekvenčním způsobem a pokud se *skok neprovede*, tak se rozpracovaná fronta instrukcí *použije*.
- V opačném případě se rozpracované instrukce *ignorují* a fronta se začne plnit znovu.
- U vysoce výkonných počítačů se implementují *obě paralelní fronty*.
- Druhá fronta je obvykle kratší (podle počtu sekcí např. 4).
- Prováděcí jednotka pak může začít vybírat alternativně připravené instrukce z druhé fronty.
- O splnění podmínky je možné spekulovat - jde o tzv. **predikci skoků**.
- Vychází se z předpokladu, že program nemůže modifikovat sám sebe a dále, že podmíněné *skoky jsou vždy na pevně danou adresu*.

Jak předvídat skok?

Procesor se snaží předpovědět, zda bude podmíněný skok proveden:

Taken skok bude proveden

Not taken pokračuje se následující instrukcí

Predikce může být:

statická rozhodnutí nezávisí na historii běhu programu,

dynamická procesor využívá informace získané během předchozího vykonávání programu.

Dynamická predikce

- Moderní výkonné procesory používají především **dynamickou predikci**.
- Prediktor může využívat **historii předchozích výsledků skoků**:

```
T T N T T N ...
```

- kde `T` znamená *taken* a `N` znamená *not taken*.

Historii lze reprezentovat několika bity a použít ji k predikci dalšího výsledku.

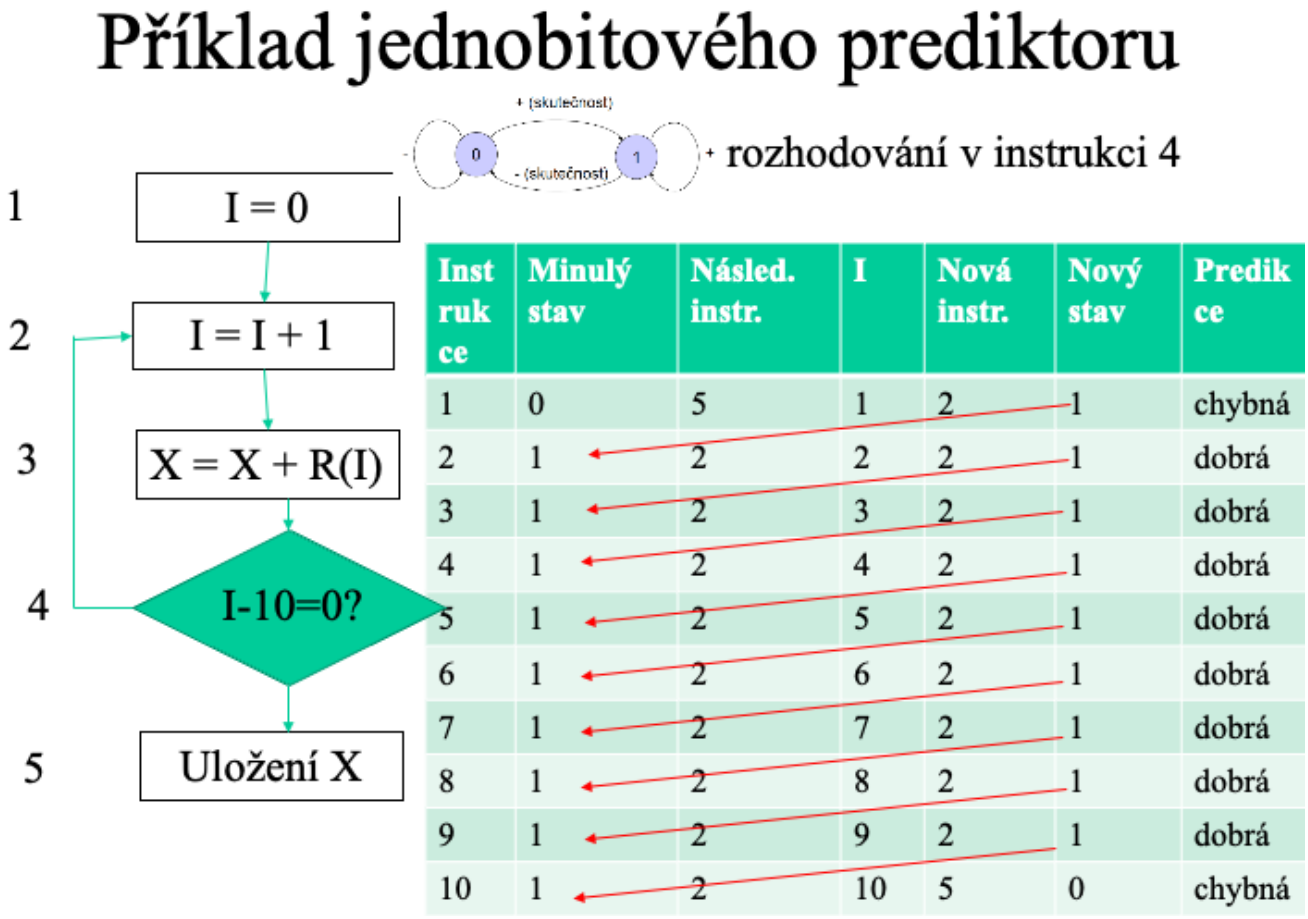
```
historie:  1 1 0 1
           T T N T
                 |
                 +-- poslední výsledek
```

Jednoduché prediktory lze modelovat jako **konečné automaty**, například známým 2bitovým saturujícím čítačem.

Současné prediktory skoků

- Současné procesory, včetně Apple Silicon, používají mnohem složitější prediktory kombinující více druhů historie a dalších informací.
- Přesná mikroarchitektura prediktoru Apple Silicon není veřejně dokumentována.

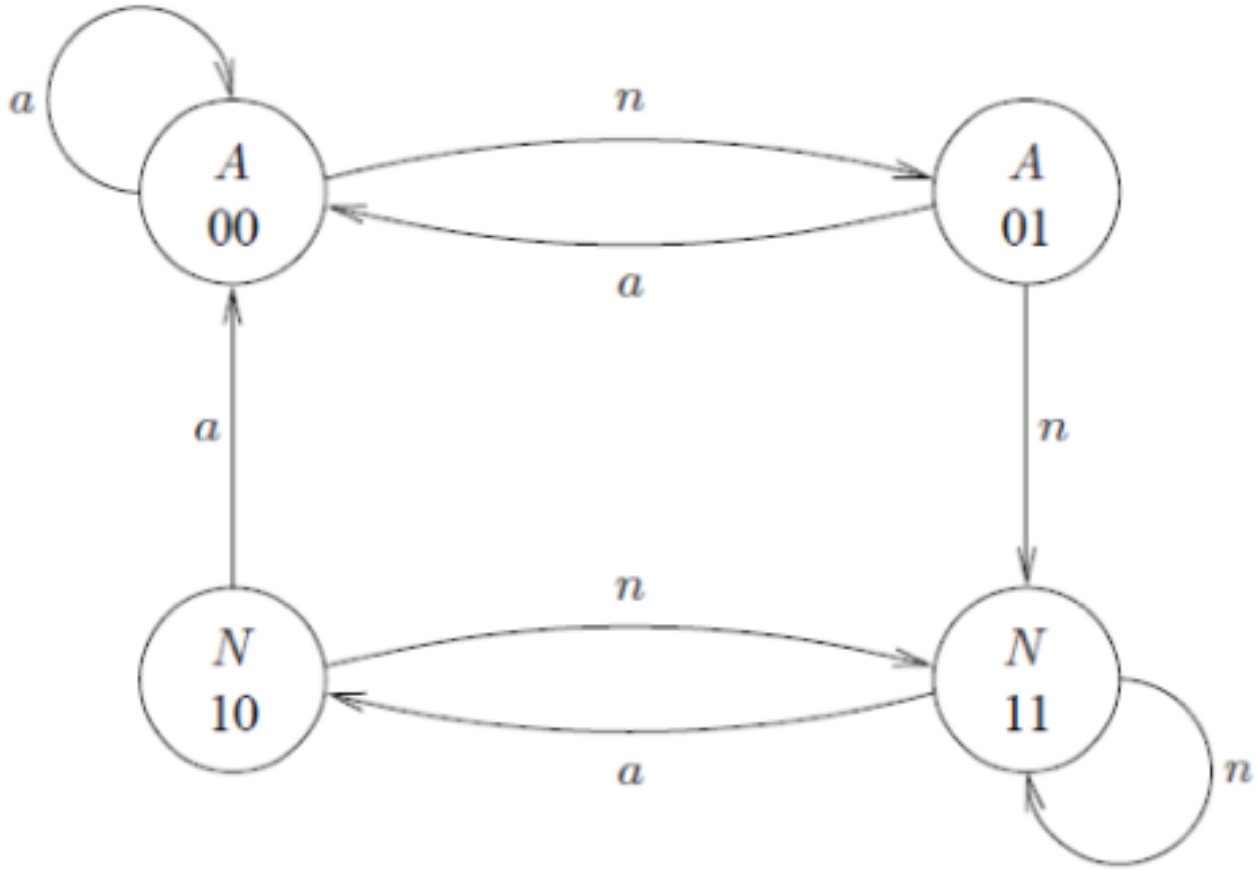
Příklad jednobitového prediktoru



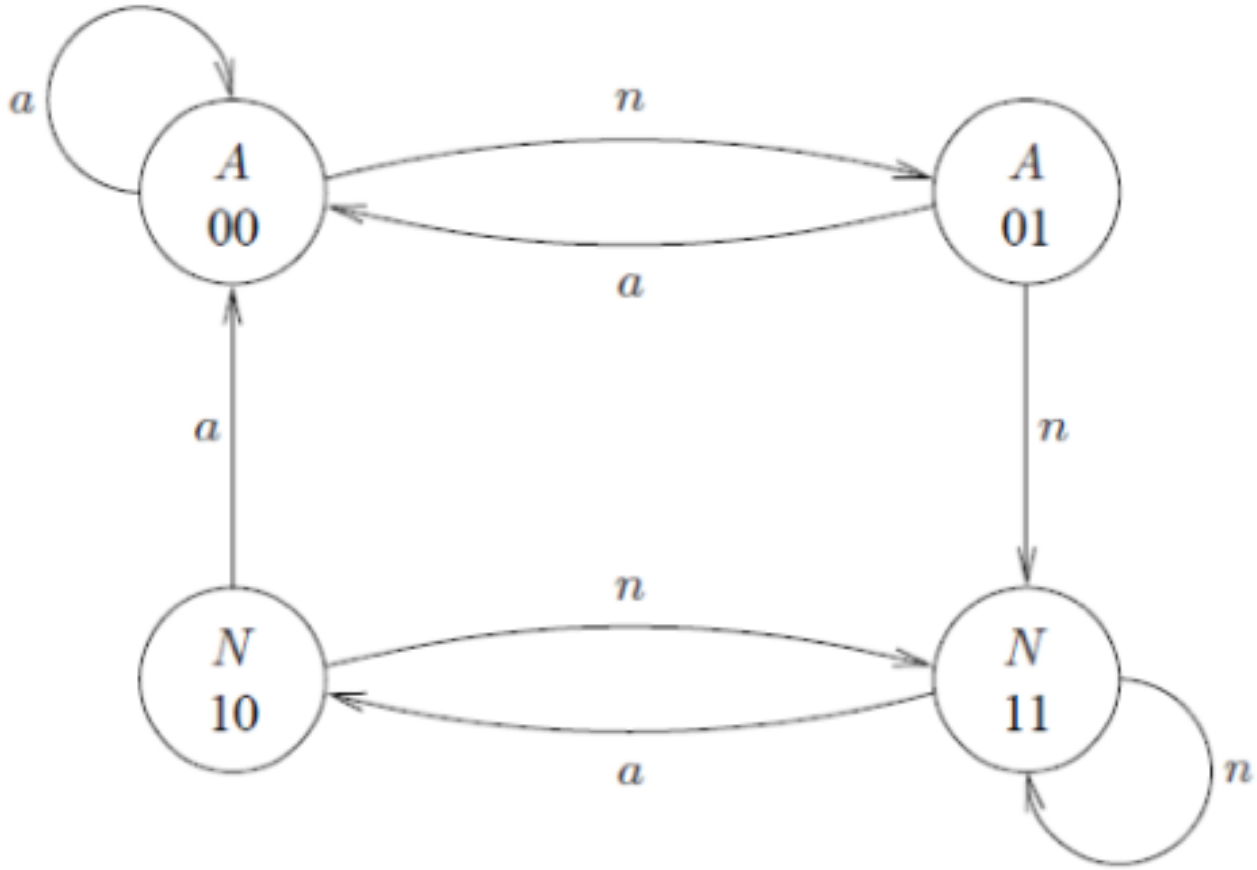
Instrukce	Minulý stav	Následující instr.	I	Nová instr.	Nový stav	Predikce
1	0	5	1	2	1	chybná
2	1	2	2	2	1	dobrá
3	1	2	3	2	1	dobrá
4	1	2	4	2	1	dobrá
5	1	2	5	2	1	dobrá
6	1	2	6	2	1	dobrá
7	1	2	7	2	1	dobrá
8	1	2	8	2	1	dobrá
9	1	2	9	2	1	dobrá
10	1	2	10	5	0	chybná

Obr. 1. Rozhodování v instrukci 4

Dvoubitový prediktor

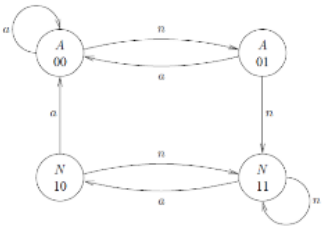


- Prediktor je ve dvou stavech pozitivní, v dalších dvou negativní.
- Ke změně predikce v saturovaném stavu (`00` nebo `11`) jsou zapotřebí alespoň dvě špatné predikce.

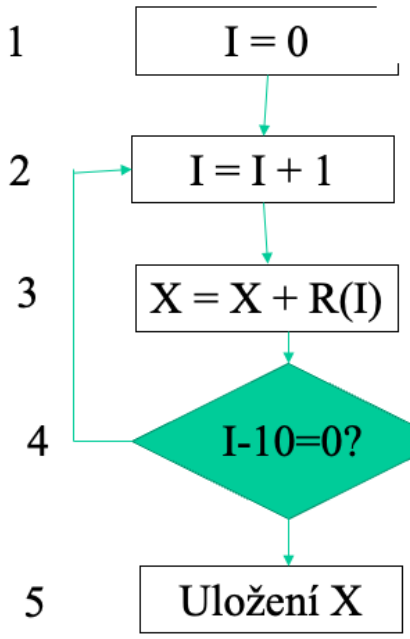


Obr. 2. Prediktor jako konečný automat (a: skok proběhl, n: neproběhl)

Příklad dvoubitového prediktoru



rozhodování v instrukci 4



Instrukce	Původní stav	Následný instr.	I	Nová instr.	Nový stav	Predikce
1	01	2	1	2	00	dobrá
2	00	2	2	2	00	dobrá
3	00	2	3	2	00	dobrá
4	00	2	4	2	00	dobrá
5	00	2	5	2	00	dobrá
6	00	2	6	2	00	dobrá
7	00	2	7	2	00	dobrá
8	00	2	8	2	00	dobrá
9	00	2	9	2	00	dobrá
10	00	2	10	5	01	chybná

Realizace predikce

Statická příslušné bity predikce se vkládají do instrukce při kompilaci

Dynamická zapisuje se při podmíněném skoku do instrukce (komplikované — kód programu je chráněn)

Paměť skoků součást procesoru — ukládají se adresy posledních provedených skoků

Počítače s redukovaným souborem instrukcí (RISC)

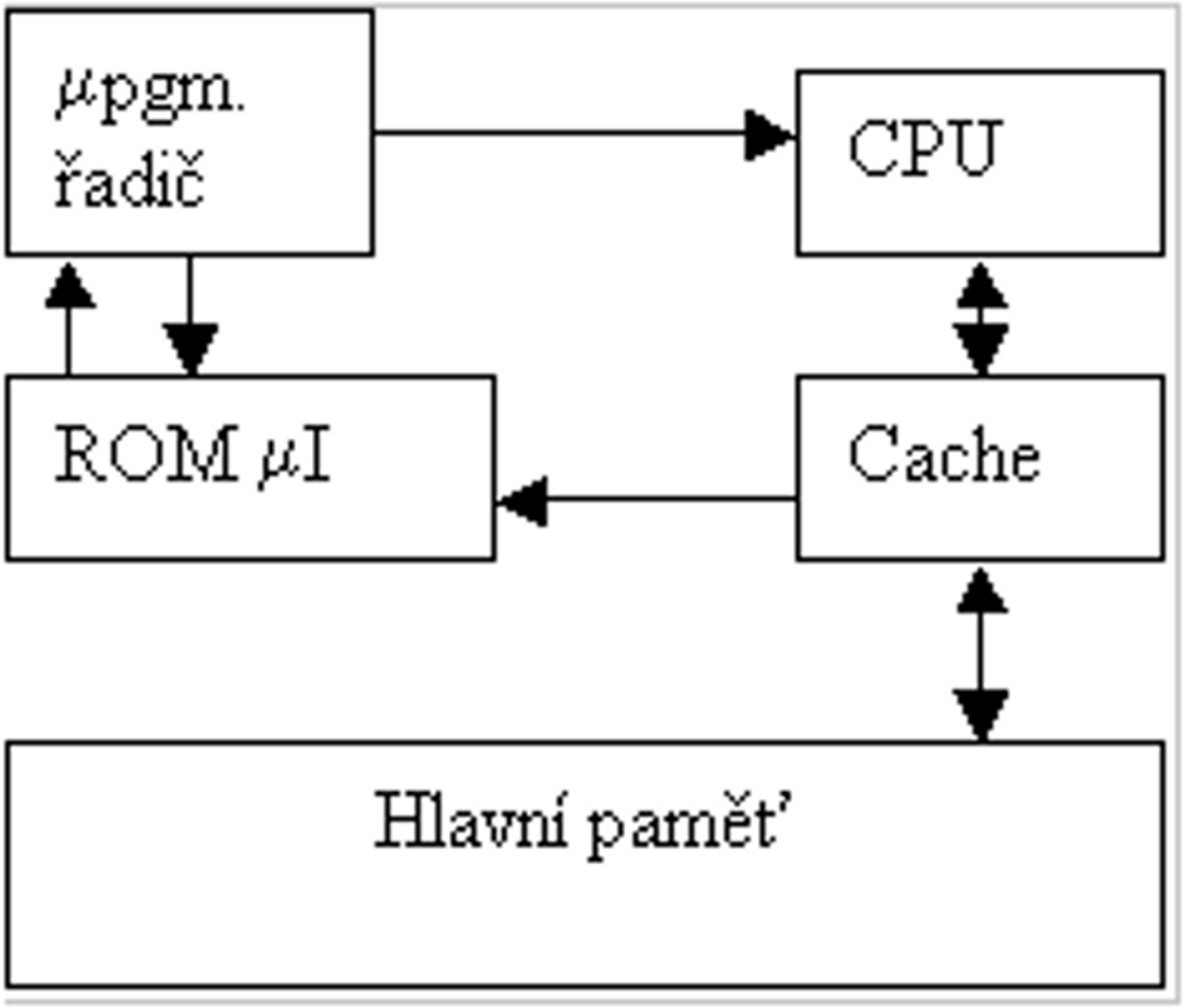


Základní rozdíl **RISC** vs. **CISC** je už v kapitole 5. Procesory. Zde jde hlavně o důsledky pro pipeline a paralelismus.

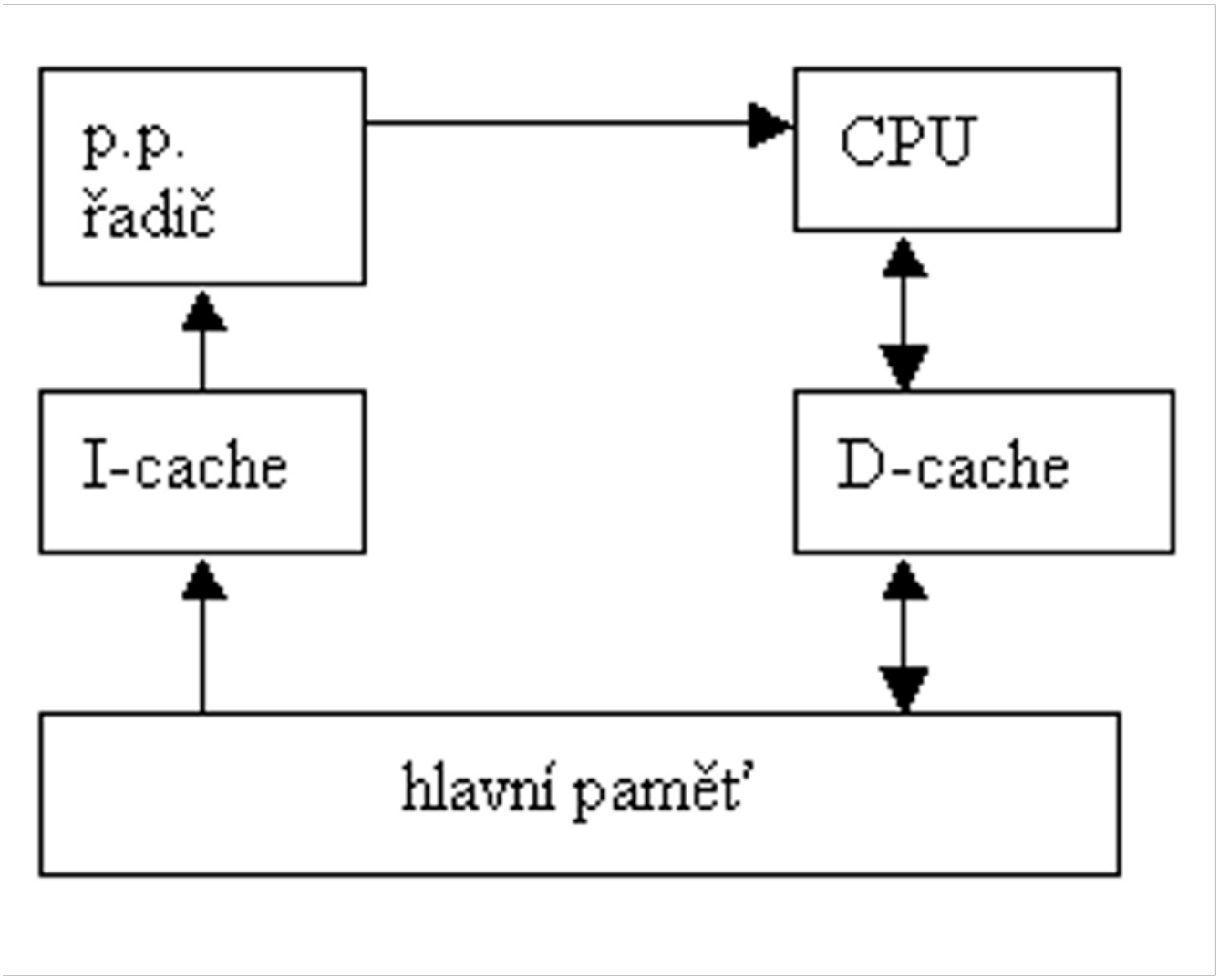
Vlastnosti RISC (Reduced Instruction Set Computer)

- Podstatná část instrukcí se provádí během **jednoho strojového cyklu**.
- Instrukce se provádějí v **obvodovém řadiči** (nepoužívá se mikroprogramování).
- Celkový počet instrukcí a způsobů adresování je **malý**, ale instrukce umí pracovat s různými registry.
- Pro spolupráci s **hlavní pamětí** jsou použity pouze typově dvě instrukce: čtení a zápis z/do paměti (load-store architektura).
- Instrukce mají **pevnou délku a pevný formát**, který přesně vymezuje funkci jednotlivých bitů nebo skupin bitů.
- V procesoru je použit **velký počet registrů**.
- Složitost se z technického vybavení a ze souboru instrukcí částečně přesouvá do optimalizujícího **kompilátoru**.

Srovnání s CISC



Obr. 3. Procesor CISC s mikroprogramovaným řízením a společnou pamětí cache



Obr. 4. Architektura RISC s pevně propojeným řadičem a oddělenými pamětmi cache pro instrukce a data

Výhody a nevýhody malého počtu instrukcí

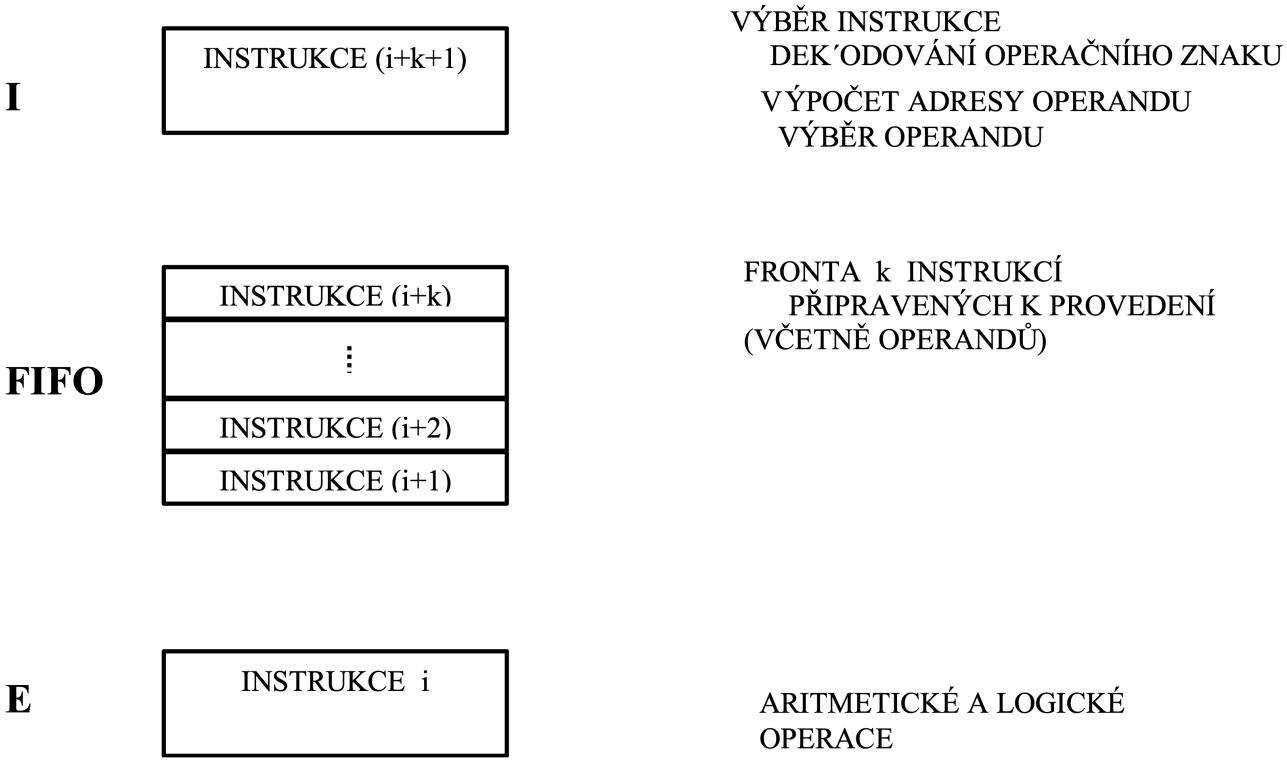
- Robustnost instrukcí
- Provádění instrukcí je stejné
- Proto možné překrývání instrukcí
- Vkládání instrukce v každém cyklu hodin
- Tím dosahování velkých rychlostí
- Delší programy
- Nutnost většího počtu registrů

RISC vs CISC dnes?

- V dnešních procesorech už nejde o čisté rozdělení na dva oddělené světy.
- Moderní **x86** procesory jsou z pohledu ISA stále CISC, ale interně používají techniky blízké RISC a rozkládají instrukce na jednodušší mikrooperace.
- Moderní **ARM** procesory (např. Apple Silicon) jsou z pohledu ISA RISC, ale některé instrukce

jsou složitější a mohou zahrnovat i poměrně komplexní operace nad registry a pamětí. == Příklad rozčlenění procesoru

- Příklad rozčlenění procesoru do sekcí I (instrukční) a E (execution unit)



Reakce jednotky I na instrukce skoku

- V případě podmíněného skoku musí jednotka I mít vypočtenou adresu, kterou počítá jednotka E
- Jednotka I zastaví svou činnost a čeká na výpočet jednotky E
- nebo pokračuje v plnění fronty, jako kdyby instrukce podmíněného skoku neexistovala, ale v případě že po vyhodnocení se skok uskuteční, je třeba zanedbat instrukce počínaje za instrukcí skoku
- Efektivnost výběru metody závisí na procentu splněných podmínek skoku (zdá se být účelnější 2. Metoda)

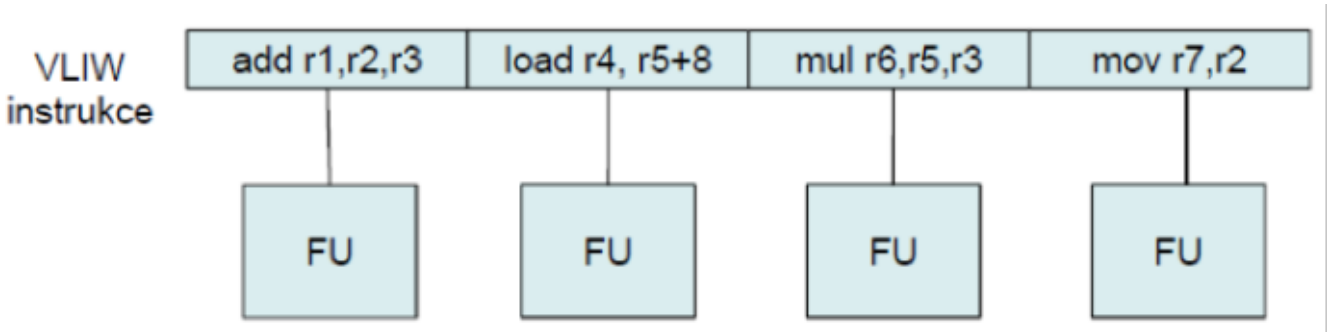
Využití dvou front instrukcí

- Do hlavní fronty zapisuje jednotka I instrukce, které odpovídají provádění programu bez větvení (samozřejmě s uvažováním nepodmíněných skoků)
- Vedlejší (kratší) fronta obsahuje instrukce, vybrané na základě splněné podmínky skoku
- Jednotka E vybírá instrukce podle toho, zda je podmínka pro skok splněna nebo ne. Při splnění podmínky je obsah hlavní fronty neplatný a jednotka I ji neprodleně začne plnit instrukcemi, které následují za obsahem vedlejší fronty

Procesory VLIW (Very Long Instruction Word)

- **VLIW architektura** umožňuje paralelní zpracování (původních několika instrukcí) jednou novou instrukcí (typicky několik instrukcí 4-16).
- Paralelně zpracovatelné instrukce jsou naplánovány *předem* při kompilaci.
- VLIW je důležité hlavně jako **specializovaný** nebo **historicky významný** směr, *ne jako převládající architektura běžných desktopových a serverových CPU*.
- Klasický VLIW nepodporuje hazard detection — předpokládá nezávislost „instrukcí“ uvnitř instrukce ⇒ jednodušší HW.
- Pevný formát instrukce obsahuje kód několika operací, které se ale mohou provést paralelně.

Formát instrukce VLIW



- HW vykonává všechny operace zakódované v instrukci nezávisle (jemnozrnný paralelizmus) paralelizmus na úrovni instrukcí (ILP)
- Kompilátor je zodpovědný za rozvržení instrukcí a extrakci ILP z programu.

Princip zpracování instrukcí VLIW

- Instrukce zpracovávají *paralelně*.
- Jsou sdruženy do *instrukčního paketu* a vznikne tím *velmi dlouhé* instrukční slovo, např. 256 bitů.
- Paralelní umožňuje provádění několika různých operací současně: např. aritmetické operace, práce s pamětí atd.
- Zároveň jednotlivé instrukce instrukčního paketu mohou být typu SIMD, takže mohou pracovat s více daty současně.
- Taková architektura umožňuje vytvořit velmi efektivní program, ale většinou je potřeba "ruční" práce při optimalizaci.
- Kompilátor ne vždy dokáže seřadit všechny instrukce efektivně do navazujících paketů.

Superskalární procesory

- Podmínkou je *více stejných jednotek* schopných paralelního vykonávání instrukcí.
- Umožňuje počítači provádět *několik pokynů* najednou a nezávisle.
- Operační kódy instrukcí jsou načteny z fronty instrukcí a *teprve řadič rozhodne*, zda a jak je možné instrukce zpracovávat paralelně.

Rozdíl mezi superskalárními procesory a VLIW

- Zásadní rozdíl spočívá v tom, že zatímco u VLIW je paralelní spouštění instrukcí založeno na instrukčním slovu, ve kterém se nachází jednotlivé operační kódy současně prováděných operací (explicitní instrukční paralelismus),
- U superskalární architektury rozhoduje o paralelním provádění operací až řadič na základě toho, jak vyhodnotí obsazení pracovních registrů a strukturu programu. To převažuje dnes.

ANDES



Architecture with Non-sequential Dynamic Execution Scheduling je vhodné chápat spíše jako *historický příklad* dynamického plánování instrukcí. Pro dnešní dobu je důležitější obecný princip **out-of-order execution**, nikoli konkrétní značka `ANDES`.

- Zpomalení způsobeno čekáním na data
- Dynamický paralelismus
- Vícenásobné fronty instrukcí
- Celočíselná fronta pro celočíselné instrukce
- Adresní fronta pro operace Load/Store
- Fronta pohyblivé řádové čárky
- Nezávislá pipeline pro každou frontu

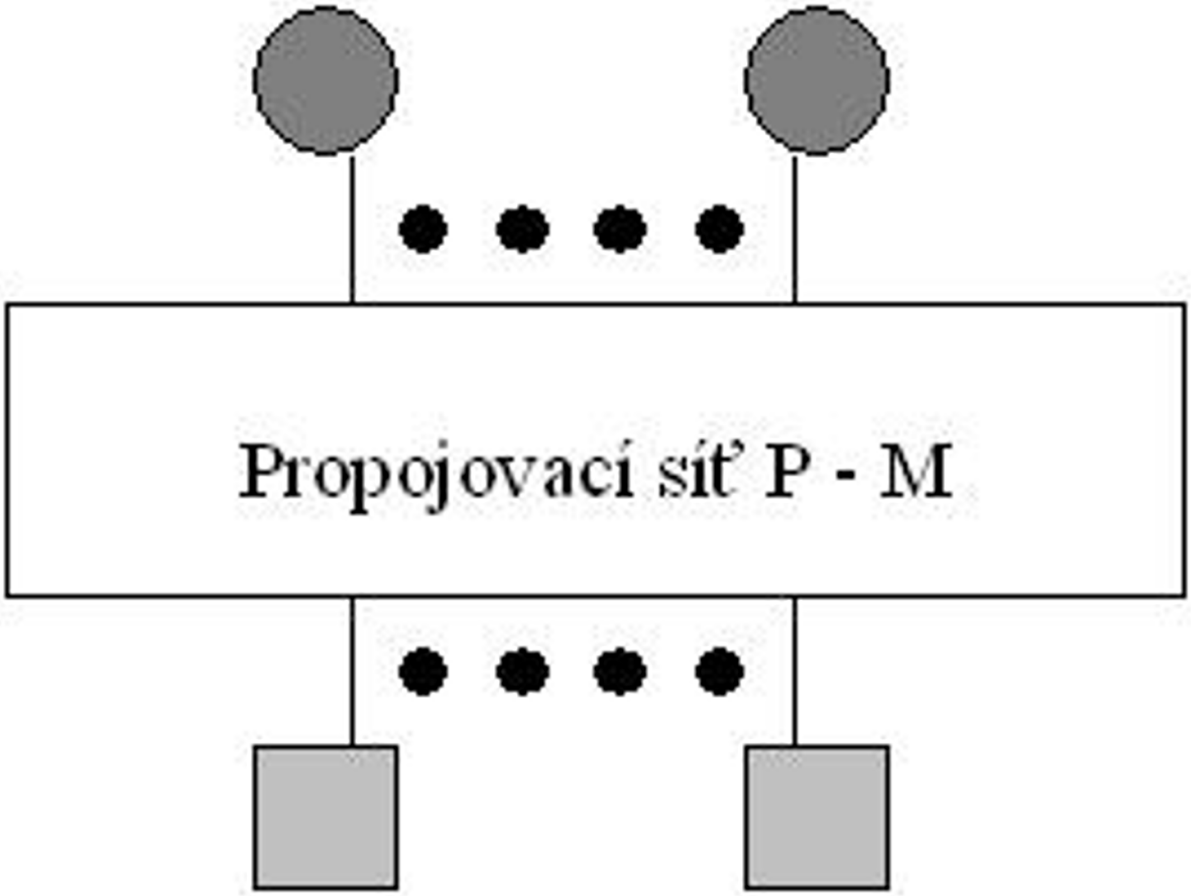
Vlastnosti ANDES

- instrukce vybrány podle **připravenosti**
- **není dodrženo pořadí** instrukcí v programu
- dokončení instrukce zajišťuje **správné uspořádání**
- Spekulativní skoky:
- výpočet pokračuje **předpovězenou větví**
- **nečeká** na výsledek instrukce
- **neblokující** Load/Store

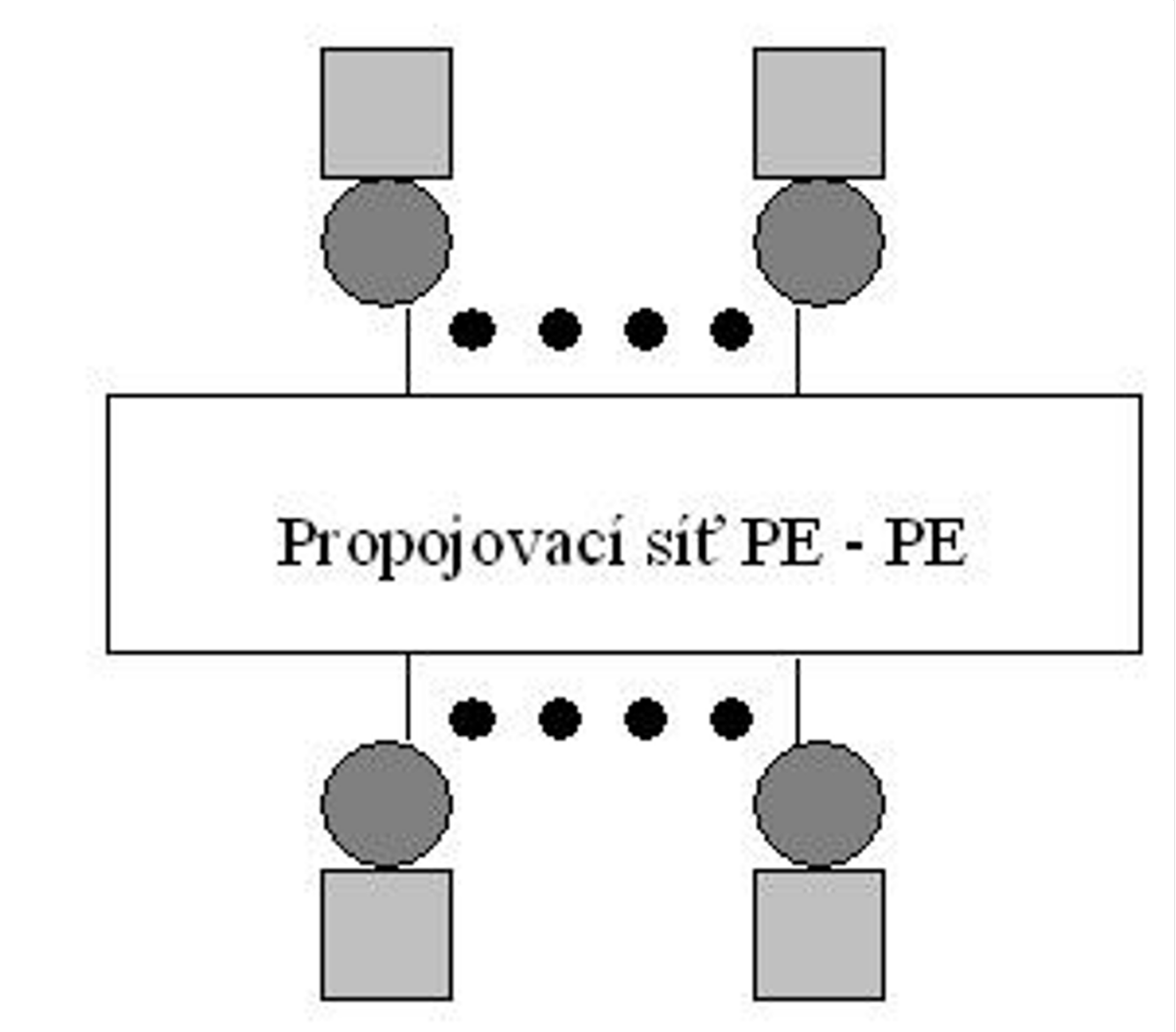
Paralelní systémy

- Paralelní počítač se **sdílenou pamětí**
- Paralelní počítač s **distribuovanou pamětí**
- Paralelní počítač s distribuovanou **sdílenou pamětí**

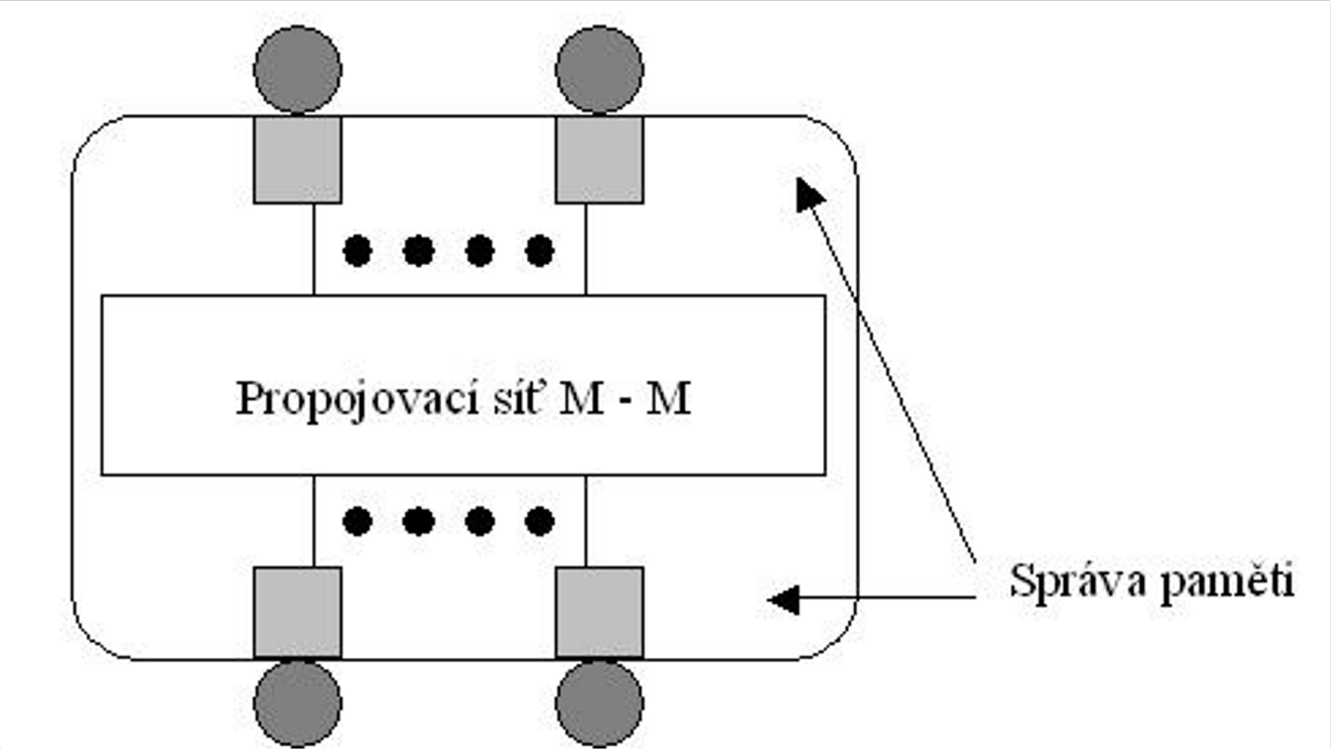
Paralelní počítač se sdílenou pamětí



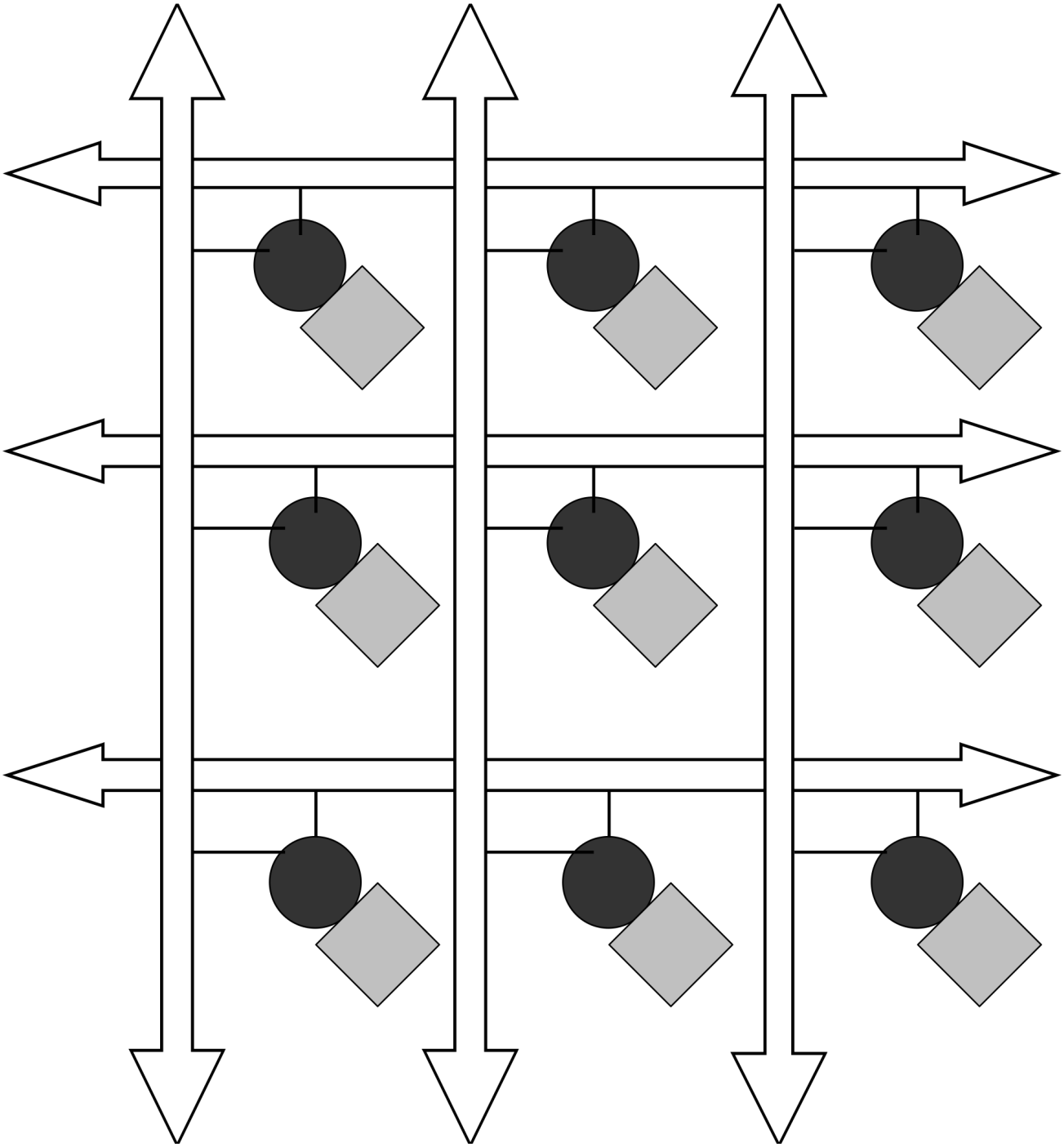
Paralelní počítač s distribuovanou pamětí



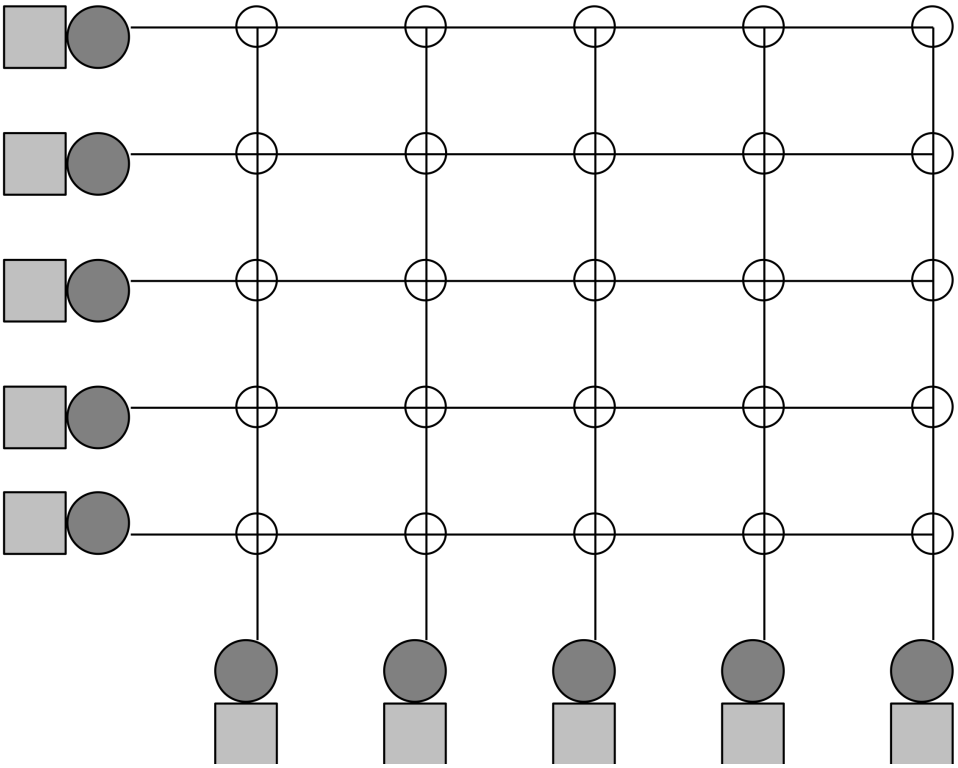
Paralelní počítač s distribuovanou sdílenou pamětí



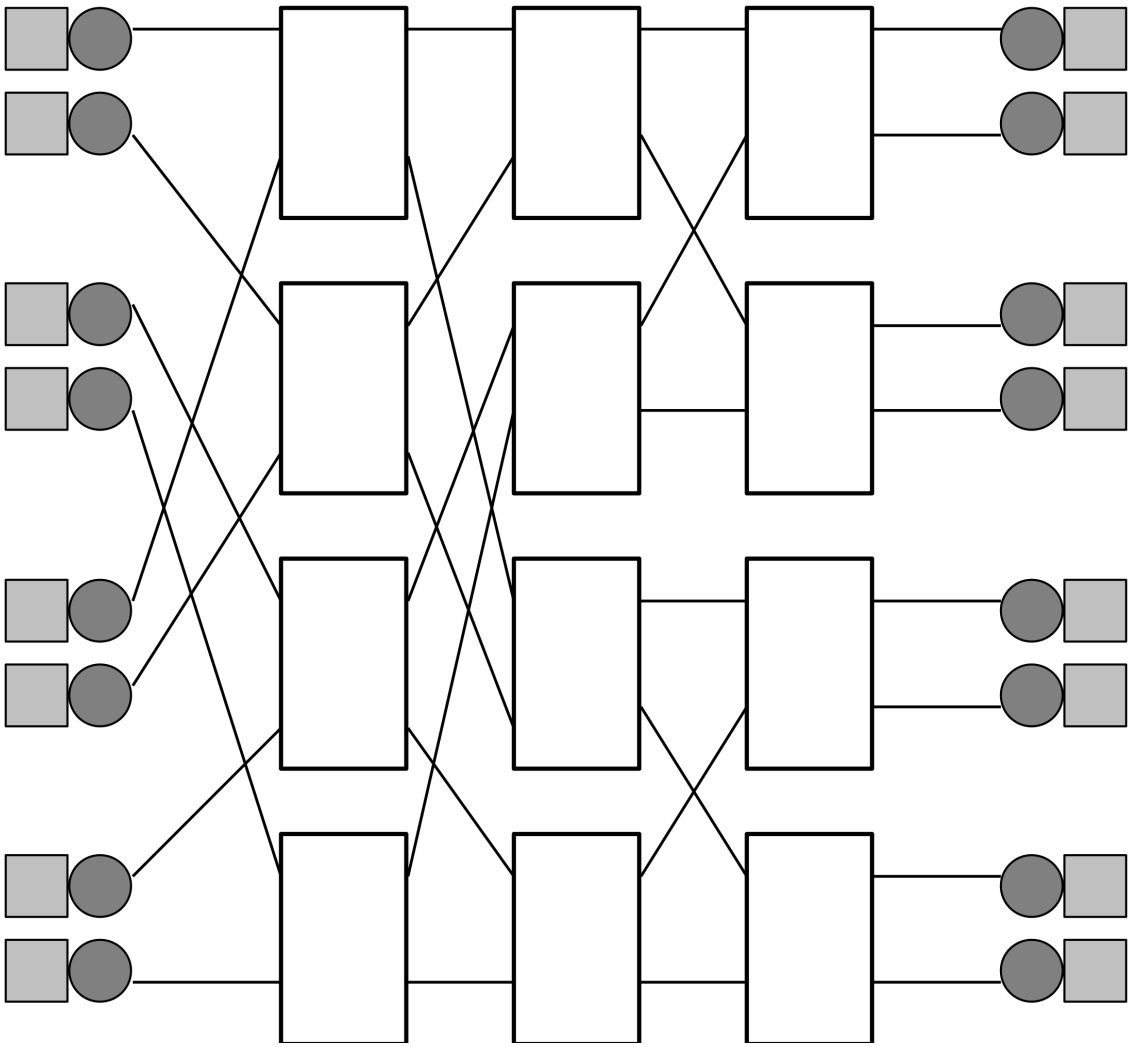
Propojovací sběrnicová síť



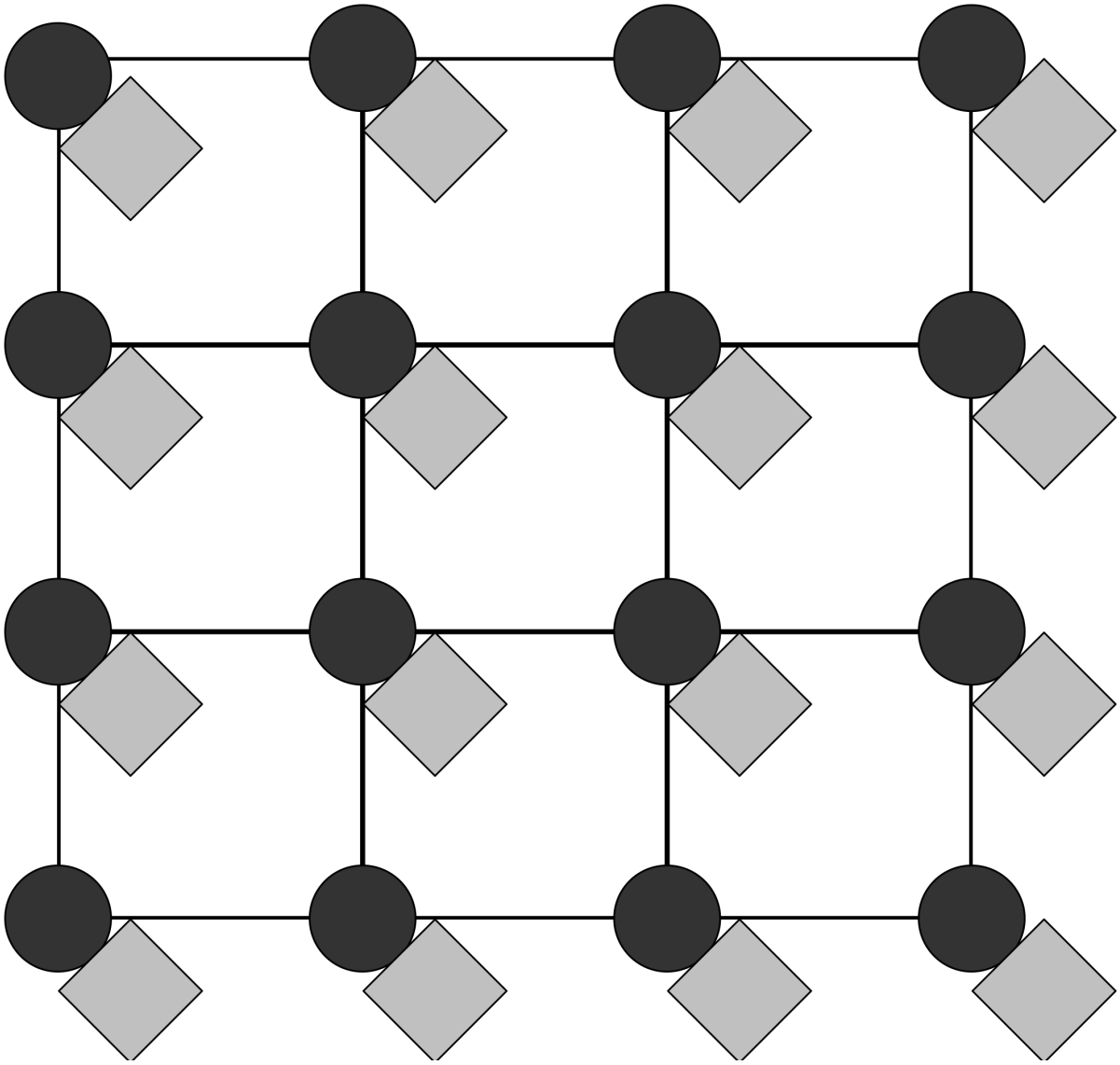
Propojovací síť s křížovým přepínačem



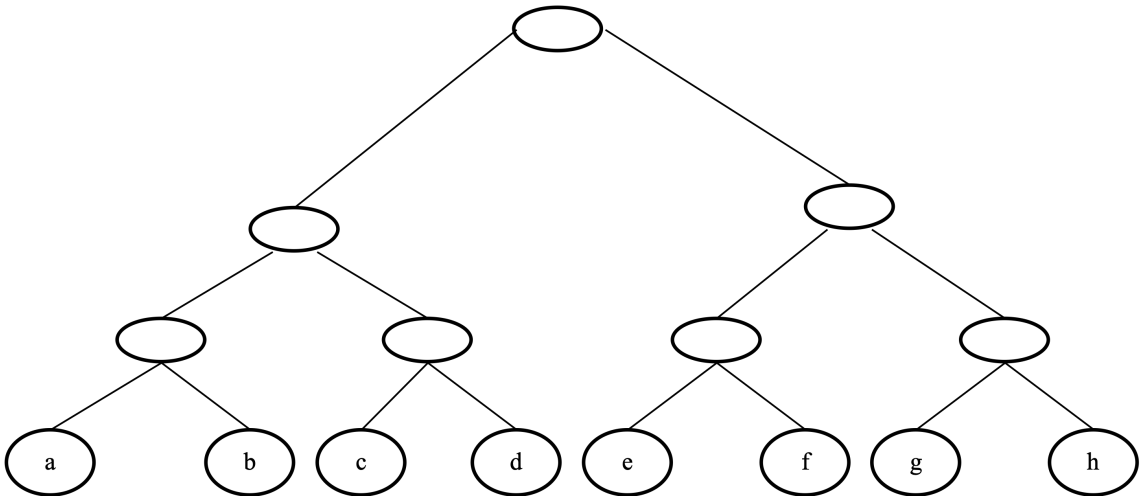
Nepřímá propojovací síť



Přímá propojovací síť



Odvození složitosti vychází ze struktury jednoduchého paralelního systému — sečtení hodnot a — h — ve třech taktech

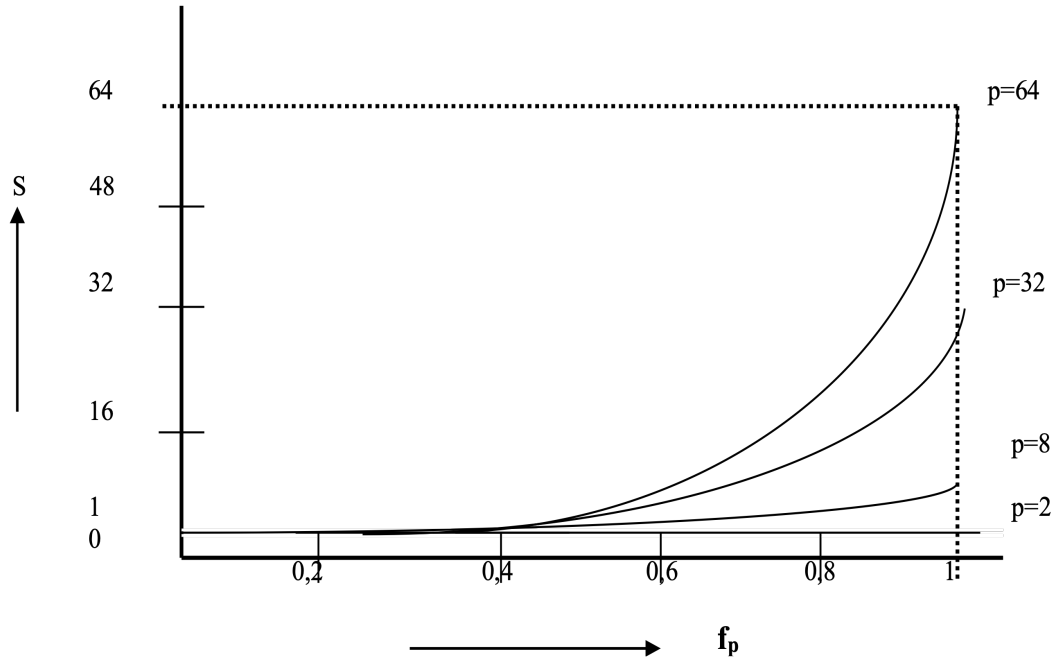


Odvození součinitele zrychlení pro paralelní výpočty

- f_s podíl délky části prováděné sériově (sekvenčně) k celkové délce výpočtu
- f_p podíl délky části prováděné paralelně k celkové délce výpočtu
- $f_s + f_p = 1$
- t doba sériového provádění výpočtu
- p počet procesorů
- Součinitel zrychlení S
- $S = t/(t.f_s + t.f_p/p)=1/((1 - f_p) + f_p/p)$

Amdahlův zákon

Amdahlův zákon popisuje závislost součinitele zrychlení na podílu paralelizované části výpočtu.



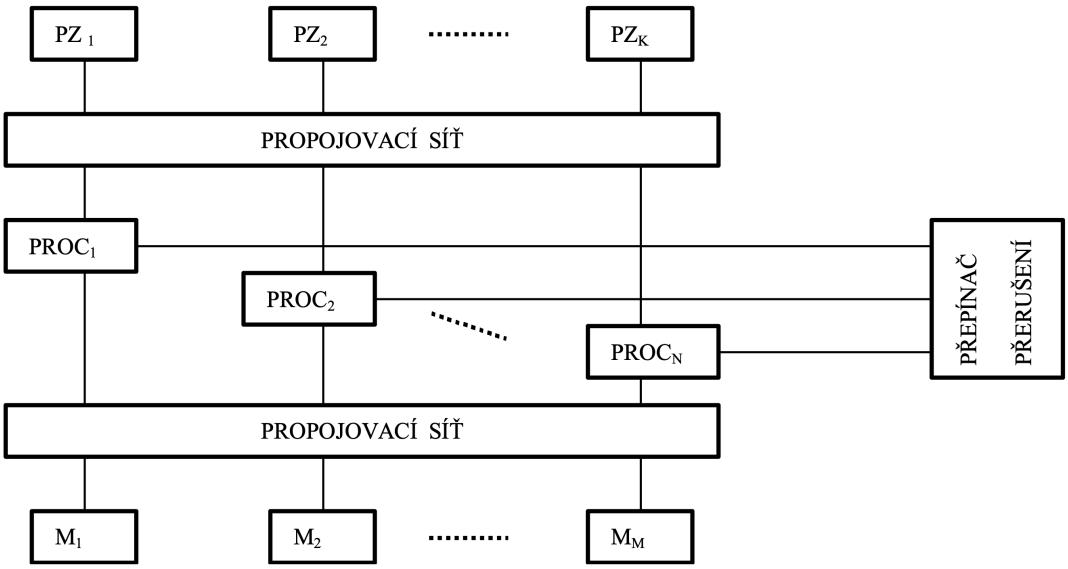
Multiprocesorové systémy

V užším slova smyslu jsou to **MIMD**:

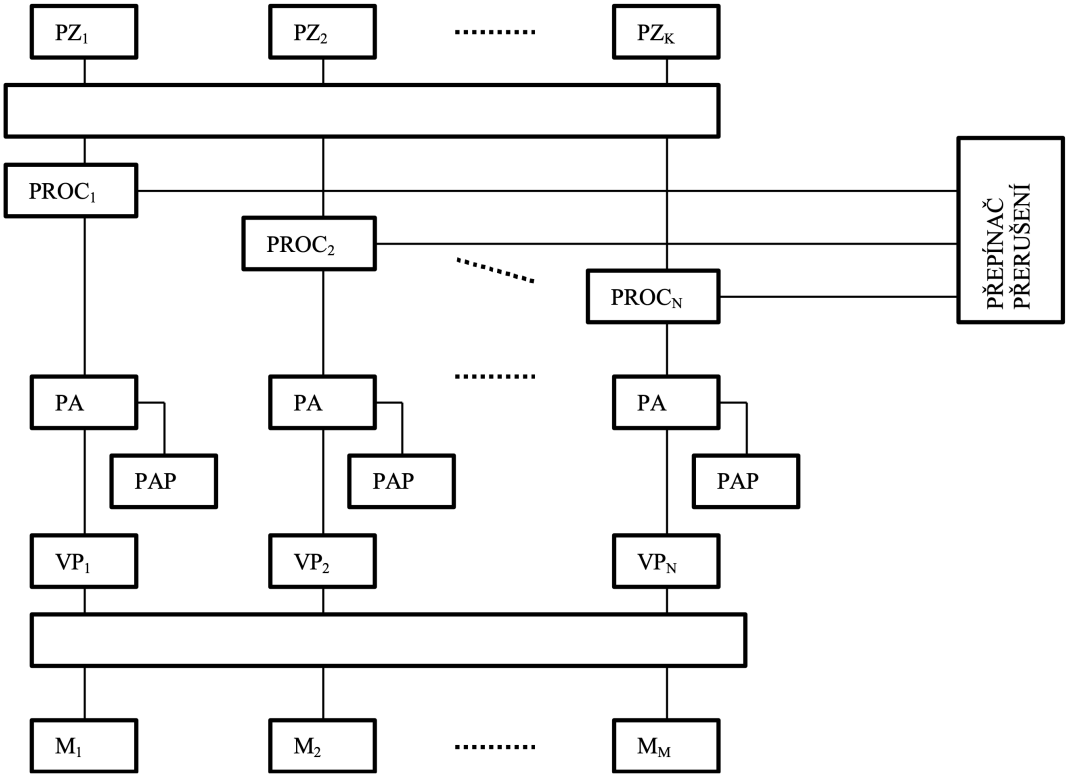
Těsně vázané vybaveny vlastní pamětí (může být velmi malá), malý stupeň autonomie (hlavní paměť a periferie jsou společné a využívají se střídavě)

Volně vázané velká lokální paměť, vlastní periferie, značný stupeň autonomie (blíží se počítačové síti)

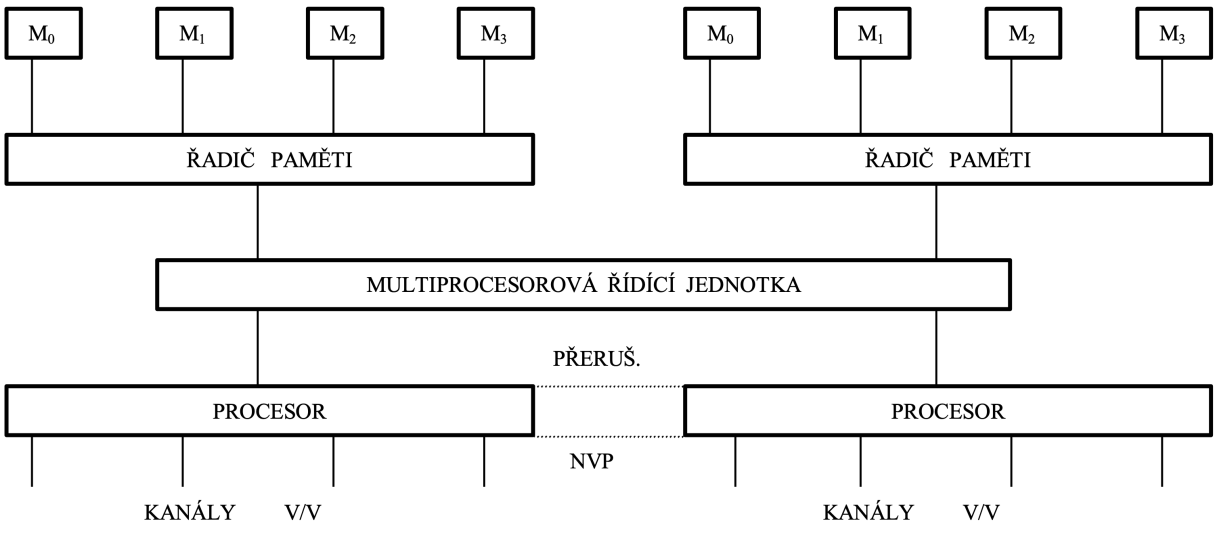
Obecná struktura těsně vázaného multiprocesorového systému



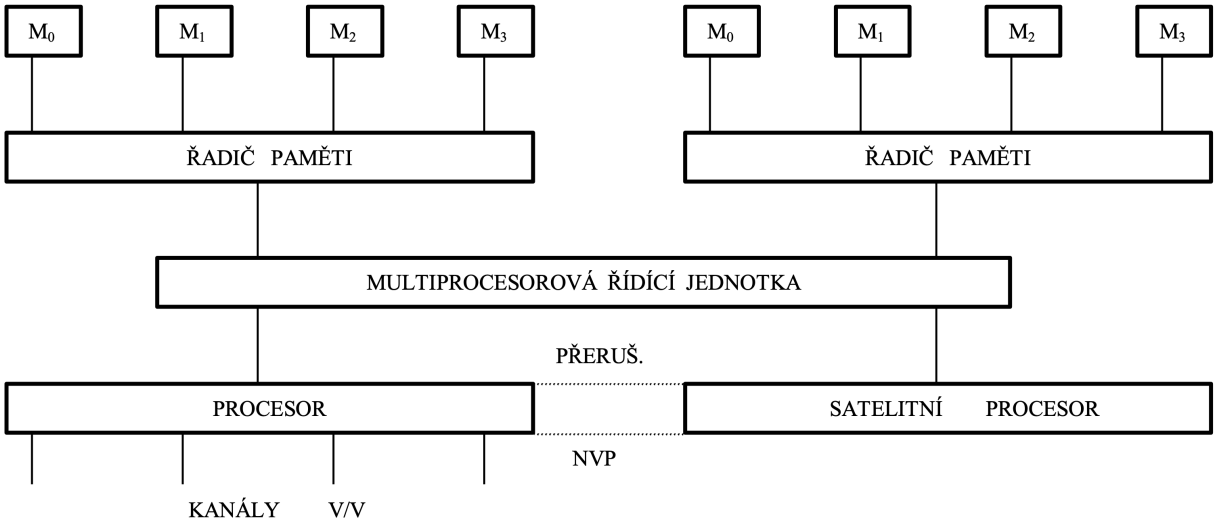
Těsně vázaný multiprocesorový systém s vyrovnávacími paměťmi



Symetrický dvojprocesorový systém



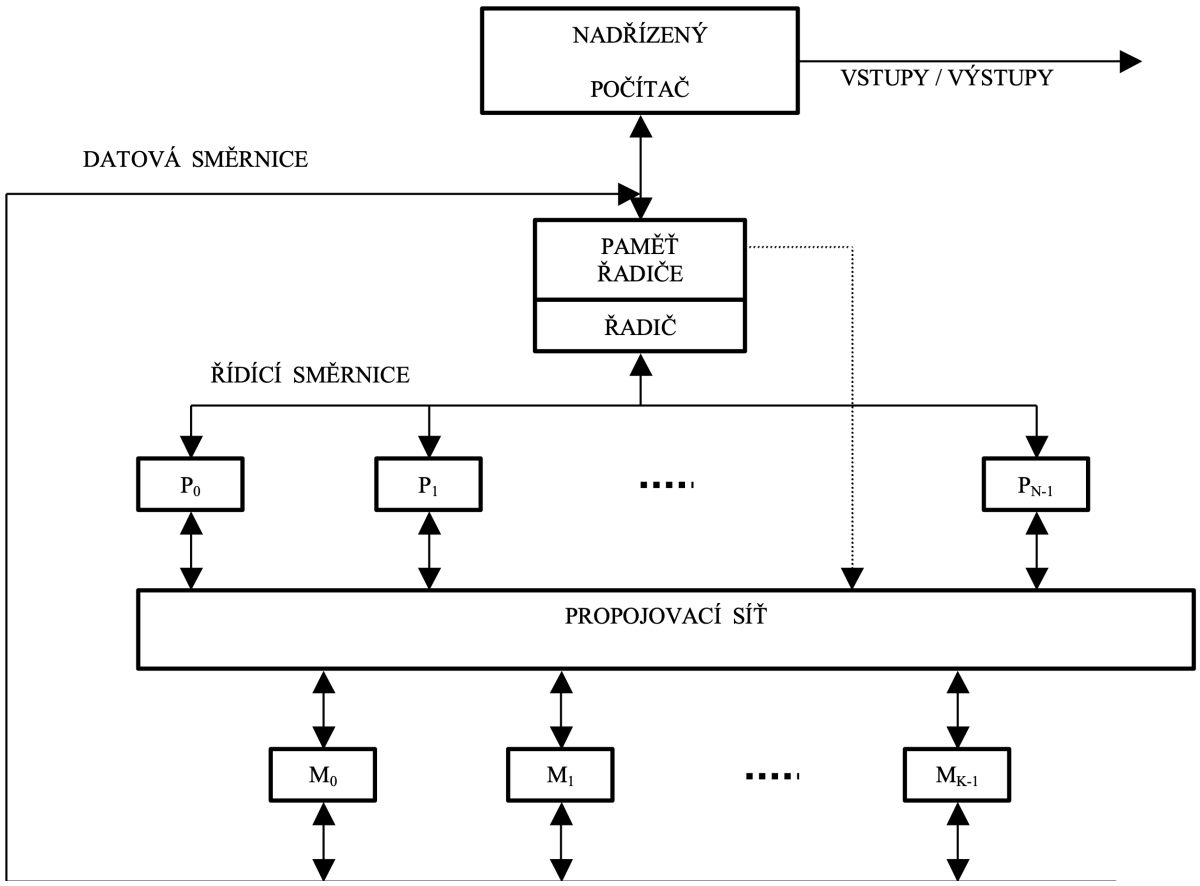
Asymetrický dvojprocesorový systém



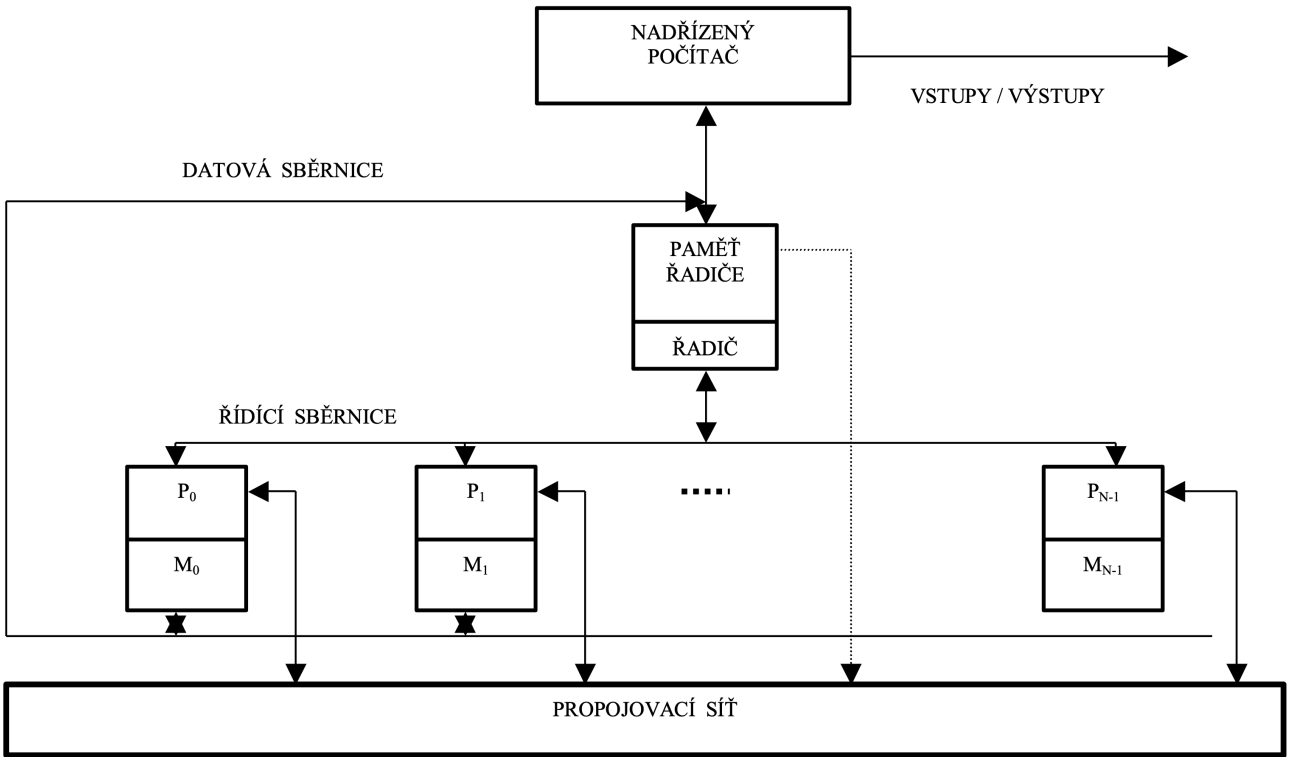
Maticové počítače

- Pro maticové operace
- Architektura SIMD
- Všechny procesory provádějí stejné operace
- Buď s lokálními pamětmi
- Nebo se sdílenou pamětí

Architektura systému SIMD se sdílenou pamětí



Architektura systému SIMD s lokálními paměťmi


NADŘÍZENÝ POČÍTAČ
VSTUPY / VÝSTUPY
DATOVÁ SBĚRNICE
PAMĚŤ ŘADIČE
ŘADIČ
ŘÍDÍCÍ SBĚRNICE
P0
M0
P1
M1
PN-1
MN-1
PROPOJOVACÍ SÍŤ

Masivně paralelní architektury

- Starší zjednodušení typu „nad 8 jader je systém neefektivní“ dnes už neplatí.
- Skutečný problém je složitější: škálování omezuje **paměťová propustnost**, **latence**, **koherence cache** a **propojovací síť**.
- Proto se ve větších systémech používají jiné topologie propojení a často i přístup **NUMA**.

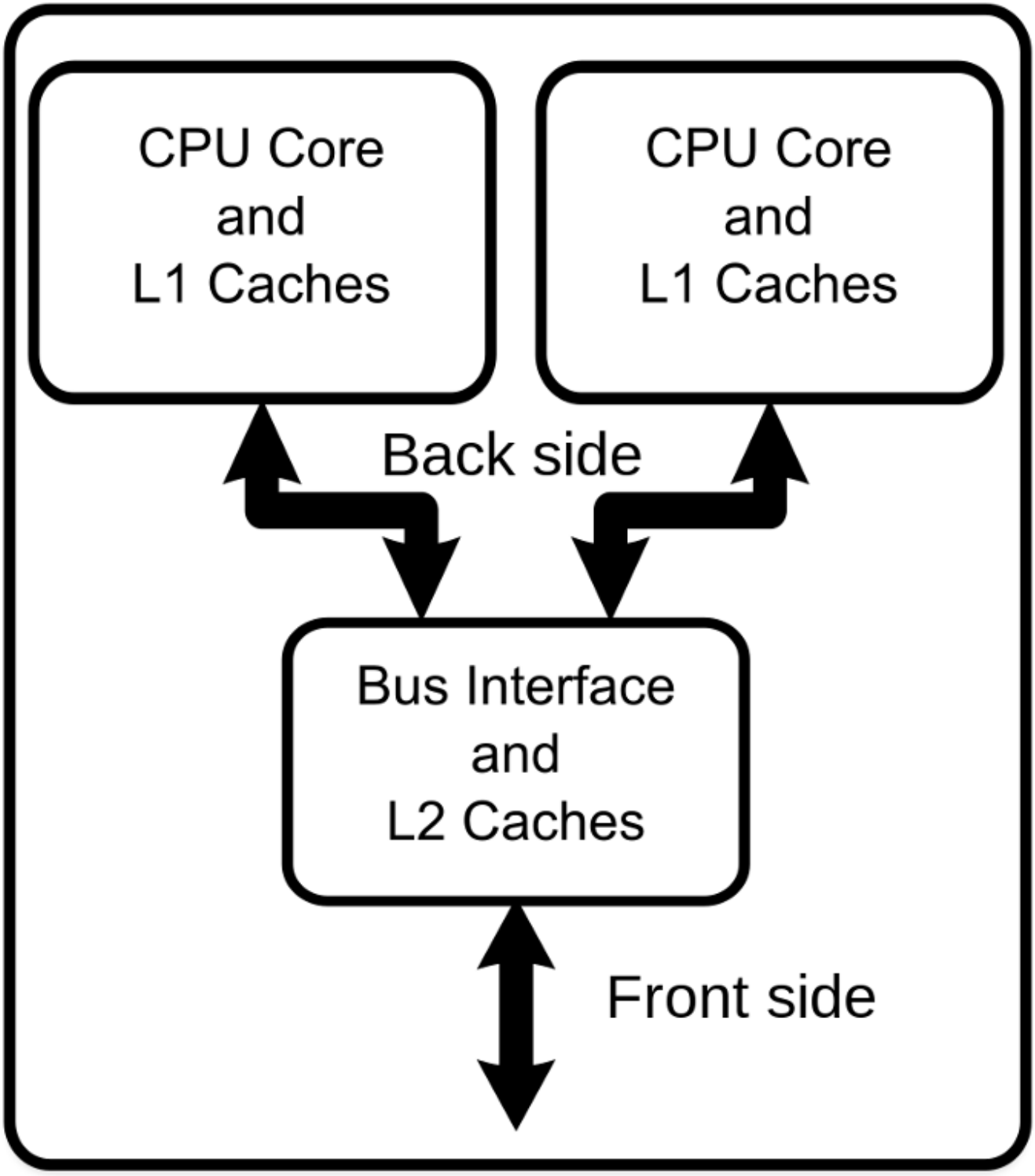
Vícejádrové procesory

- V jednom pouzdře nebo na jednom čipu integruje více CPU
- Spolupráce jader se obecně (v rámci mikropočítače) děje prostřednictvím operační paměti (RAM).
- Přirozené zefektivnění spolupráce jader nastává v případě společně sdílené paměti cache, která umožňuje další zlepšení spolupráce příslušných jader.



Základní přehled vícejádrových CPU, SMT a akcelerátorů už je v kapitole **5. Procesory**. Tady je důležité spíš to, že s rostoucím počtem jader roste význam koherence cache, interconnectu a paměťového subsystému.

Dvojjádrový procesor s lokální cache 1. úrovně, a sdílenou, integrovanou cache 2. úrovně

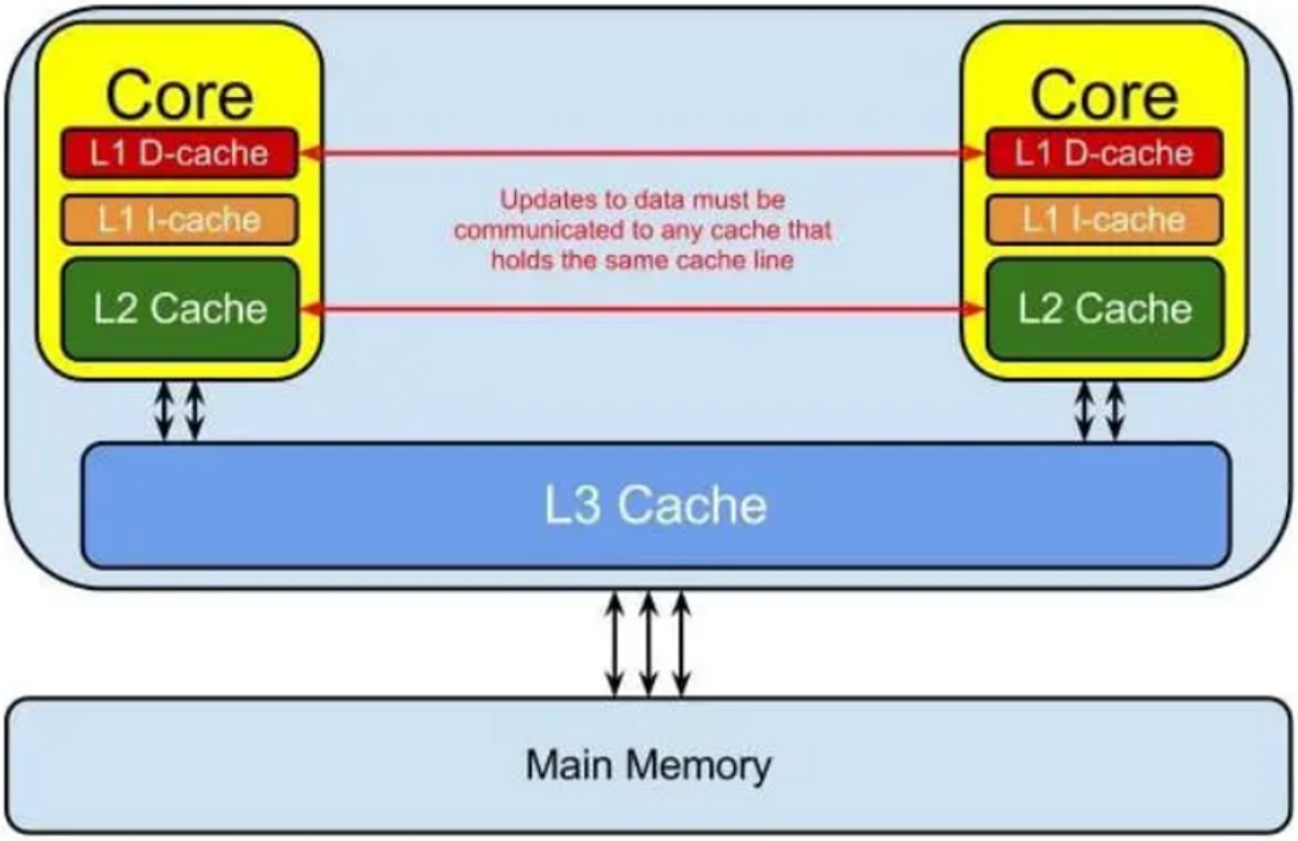


Dvojjádrový procesor s lokální cache 1. úrovně, a sdílenou, integrovanou cache 2. úrovně

L1 cache
- **Instrukční cache** se zabývá informacemi o operaci, kterou má procesor provést,
- **datová cache** uchovává data, nad kterými má být operace provedena.

L2 cache ukládá kopie dat přečtených z adresy v operační paměti

Mezipaměť L1, L2, L3



Architektura současných počítačů

Parametr	Původní von Neumann	Současný počítač
Sběrnice a paměť	Jedna společná sběrnice pro data i instrukce	Modifikovaný Harvard, oddělené L1 cache, bodové spoje
Zpracování instrukcí	Striktně sekvenční	Pipeline, out-of-order, superskalární jádra
Počet výpočetních jednotek	Jedno hlavní CPU	Vícejádrové CPU a specializované akcelerátory
Správa V/V	CPU přímo řídí přenosy	DMA, dedikované řadiče, rychlá I/O propojení

Modifikovaná harvardská architektura a kešování

- Na úrovni hlavní paměti se stále používá společný adresní prostor.
- Uvnitř jádra ale CPU často odděluje **I-cache** a **D-cache**.
- Tím se zmírňuje *von Neumannovo úzké hrdlo* a zlepšuje se plynulost pipeline.

Paralelismus na úrovni instrukcí (ILP)

- Jde o techniky, které zvyšují výkon **uvnitř jednoho jádra**.
- Patří sem zejména **pipeline**, **superskalárnost**, **out-of-order execution** a **register renaming**.

Vykonávání mimo pořadí a přejmenování registrů

- CPU nečeká zbytečně na instrukce, které mohou běžet až později.
- Místo toho hledá takové operace, které lze vykonat dříve bez porušení správnosti programu.
- **Přejmenování registrů** pomáhá odstranit falešné závislosti a lépe využít výpočetní jednotky.

SMT a jemnější paralelismus v jádře

- **SMT** (*Simultaneous Multithreading*) umožňuje jednomu fyzickému jádru obsluhovat více vláken.
- Nejde o totéž jako další jádro, ale o lepší využití nečinných částí pipeline.
- Typickým příkladem je **Hyper-Threading** u některých procesorů Intel.

SIMD a vektorové instrukce

- Vedle ILP existuje i *datový paralelismus*.
- Jedna instrukce může provádět tutéž operaci *nad více prvky současně* vektorové výpočty, SIMD instrukce, GPU, NPU.

- To je důležité pro multimédia, šifrování, numerické výpočty i strojové učení.

Vícejádrové zpracování a akcelerátory

- Místo neustálého zvyšování frekvence jednoho jádra se výkon zvyšuje i přidáváním dalších jader.
- Běžné je také delegování specializovaných úloh na **GPU** nebo **NPU**.
- Moderní procesory a SoC (např. Apple Silicon nebo mobilní platformy Arm) integrují CPU, GPU, NPU a další řadiče do jednoho pouzdra či čipu.
- Dnešní počítače jsou proto stále častěji **heterogenní systémy**.

Koherence cache a NUMA

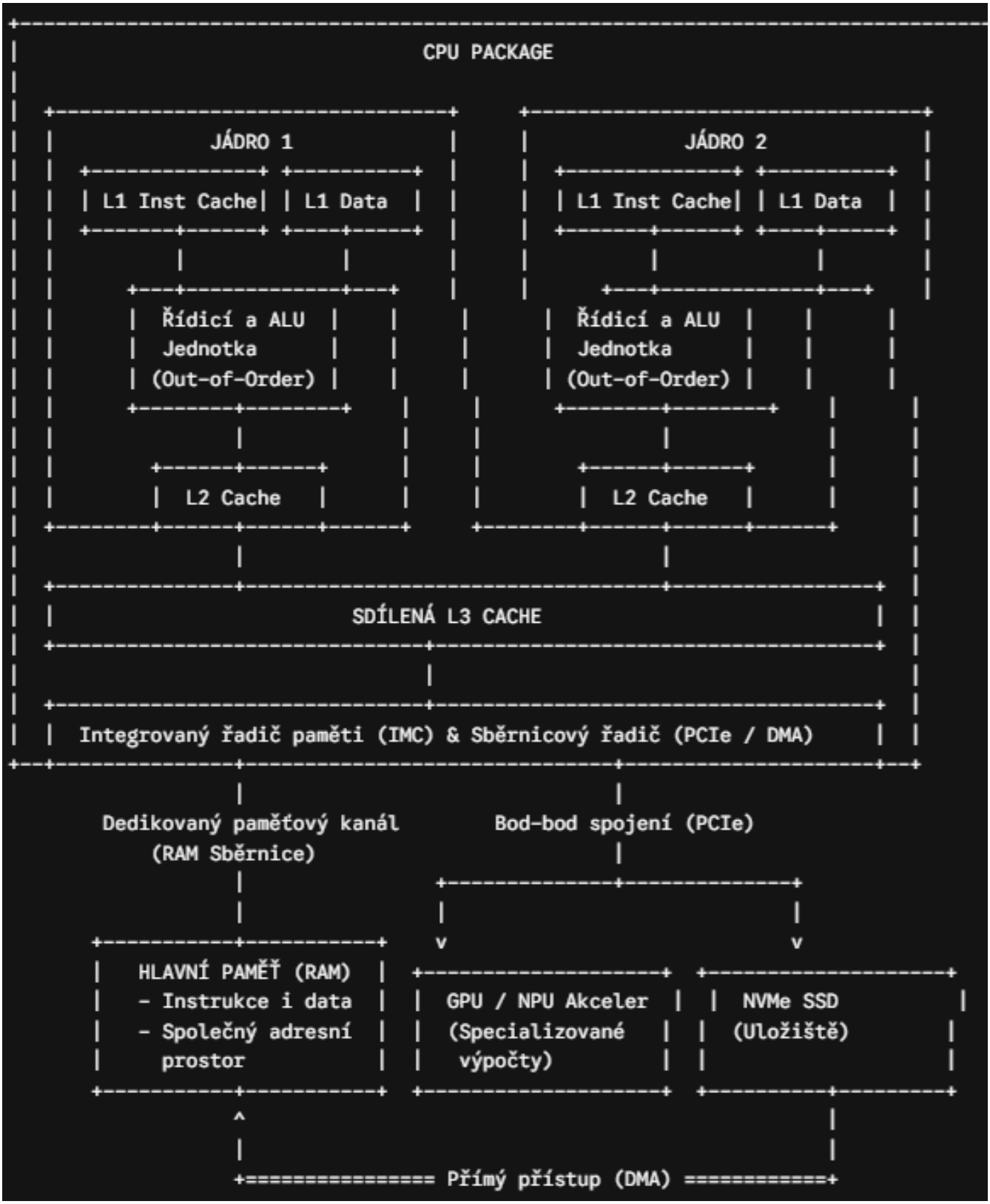
- Ve vícejádrovém systému je nutné hlídat, aby různá jádra viděla konzistentní data.
- Tomu se věnuje **koherence cache**.
- U větších serverů se navíc řeší **NUMA** (*Non-Uniform Memory Access*), kde není přístup do celé RAM stejně rychlý.

Sběrnice typu Point-to-Point a DMA

- Tradiční sdílené sběrnice byly postupně nahrazeny rychlými **point-to-point** spoji.
- Typickým příkladem je **PCI Express**.
- **DMA** dovoluje periferiím přesouvat data do RAM bez toho, aby každý krok musel řídit procesor.

Klíčové vazby na schématu

- Dvojitá L1 cache
- Hierarchie keší L1 → L2 → L3
- Bodové propojení zařízení
- DMA kanál



Aktuální směry (2026)

- **Heterogenní výpočet**: CPU + GPU + NPU / maticové akcelerátory
- **Chipletové** a dlaždicové návrhy místo jednoho velkého monolitického čipu
- Pokročilé **vektorové a maticové instrukce** pro AI, HPC a multimédia
- Stále důležitější **výkon na watt** a řízení spotřeby

- Rychlé point-to-point propojení CPU, paměti a akcelerátorů
- V serverech roste význam **CXL** a paměťové disaggregace

Vektorové a maticové rozšíření ISA

- Klasické SIMD je doplňováno instrukcemi specializovanými na vektorové a maticové výpočty.
- U **Armv9** sem patří např. **SVE2** a **SME/SME2**; cílem je vyšší propustnost u AI, multimédií a numerických úloh.
- Podobný trend existuje i u x86 a u specializovaných GPU/NPU: více práce se provede nad blokem dat jednou instrukcí nebo jedním výpočetním příkazem.

CPU už není jediný hlavní výpočetní prvek

- Operační systém a runtime stále častěji plánují práci mezi různými typy výpočetních jednotek.
- CPU je vhodné pro obecné a řídicí úlohy, GPU pro masivní datový paralelismus a NPU pro vybrané operace strojového učení.
- Výkon systému proto nelze hodnotit jen počtem CPU jader nebo taktovací frekvencí.

Bezpečnostní důsledek spekulace

- Vyšší výkon přinášejí techniky jako predikce skoků a spekulativní vykonávání.
- Tyto optimalizace ale ukázaly i bezpečnostní rizika.
- Známým příkladem jsou zranitelnosti typu **Spectre** a **Meltdown**.

Asociativní procesory

- Maticové procesory se sdílenou pamětí někdy využívají asociativní výběr z paměti (podle obsahu) — hlavně CACHE paměti
- Paralelní, sériové po bitech, sériové po slovech



Jde hlavně o specializované téma. Pro běžné procesory je dnes prakticky důležitější znalost **asociativního vyhledávání v cache** než představa samostatného „asociativního počítače“ jako hlavního směru vývoje CPU.

Asociativní procesory: komentář k obrázku

- Komentář k obrázku
  - K = Hledaný klíč
  - M = Maska 1111
  - P = paměťové pole
  - V = registr výsledku
  - Ti = pomocné registry mezivýsledku
  - R = vybraná informace

Asociativní operace

- Hledání *extrémů* (max, min, medián — mezi ostatními podle daného uspořádání)
- Hledání *ekvivalence* (shoda s klíčem, neshoda s klíčem, podobnost s klíčem v toleranci)
- Hledání *sousednosti* (nebližší nižší, vyšší)
- Hledání podle *prahu* (menší než klíč, menší nebo rovný, větší než klíč, větší nebo rovný)
- Hledání podle dvou a více prahů (*interval*)
- *Řazení* (vzestupné, sestupné)

Počítače s netradičním řízením

„Netradiční“ = jiné než principiálně navrhl John von Neumann

- Počítače řízené *tokem dat*
- Počítače řízené tokem *požadavků*
- *Systolické* systémy
- *Neuronové* počítače



Většina těchto architektur není dnes běžným základem osobních počítačů nebo serverových CPU. Jsou ale důležité historicky a některé principy se vracejí ve specializovaných akcelerátorech.

Počítače řízené tokem dat

- Data flow computer
- Instrukce se spouští ve chvíli, kdy jsou k dispozici všechny operandy potřebné pro její provedení
- Počítače řízené tokem dat nemají čítač instrukcí

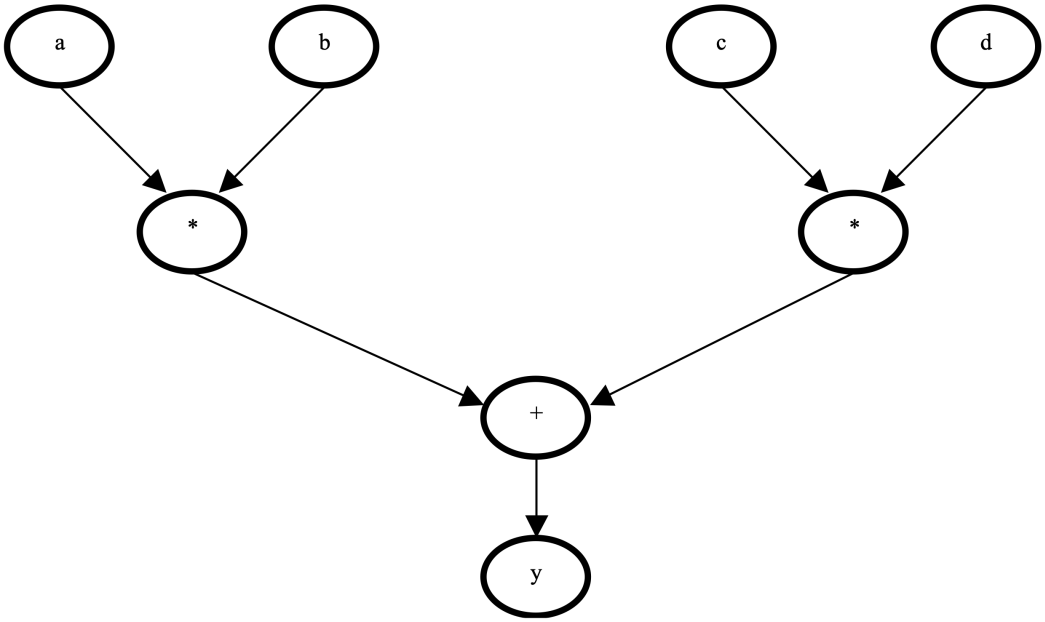
Příklad `y=a*b+c*d`

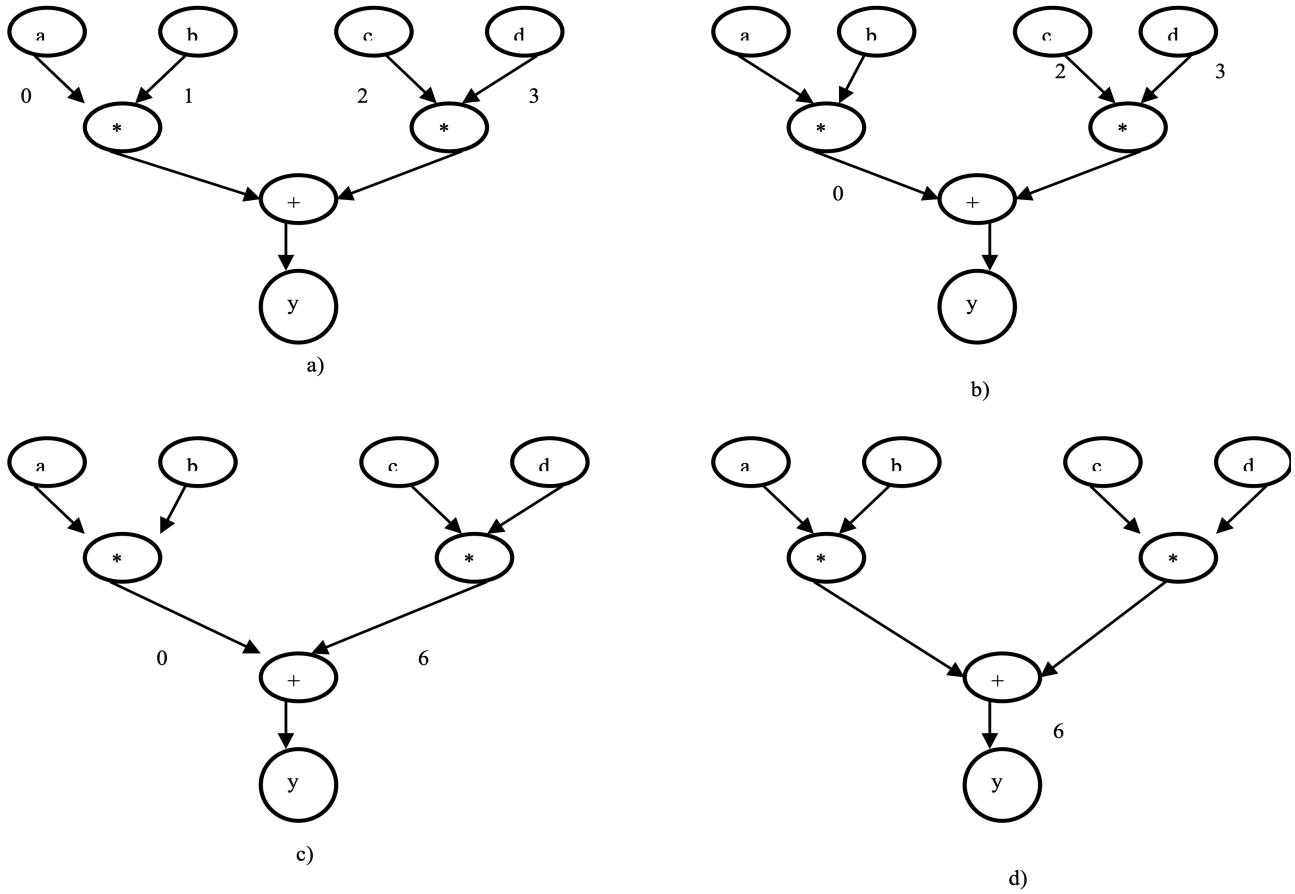
Pro jednoadresní počítač:

```
Přesuň1 a  a    -> R1     ; ①
Násob1  b  R1*b -> R1
Přesuň2 c  c    -> R2
Násob2  d  R2*d -> R2
Sečti12    R1+R2-> R1
Ulož1   y  R1   -> y
```

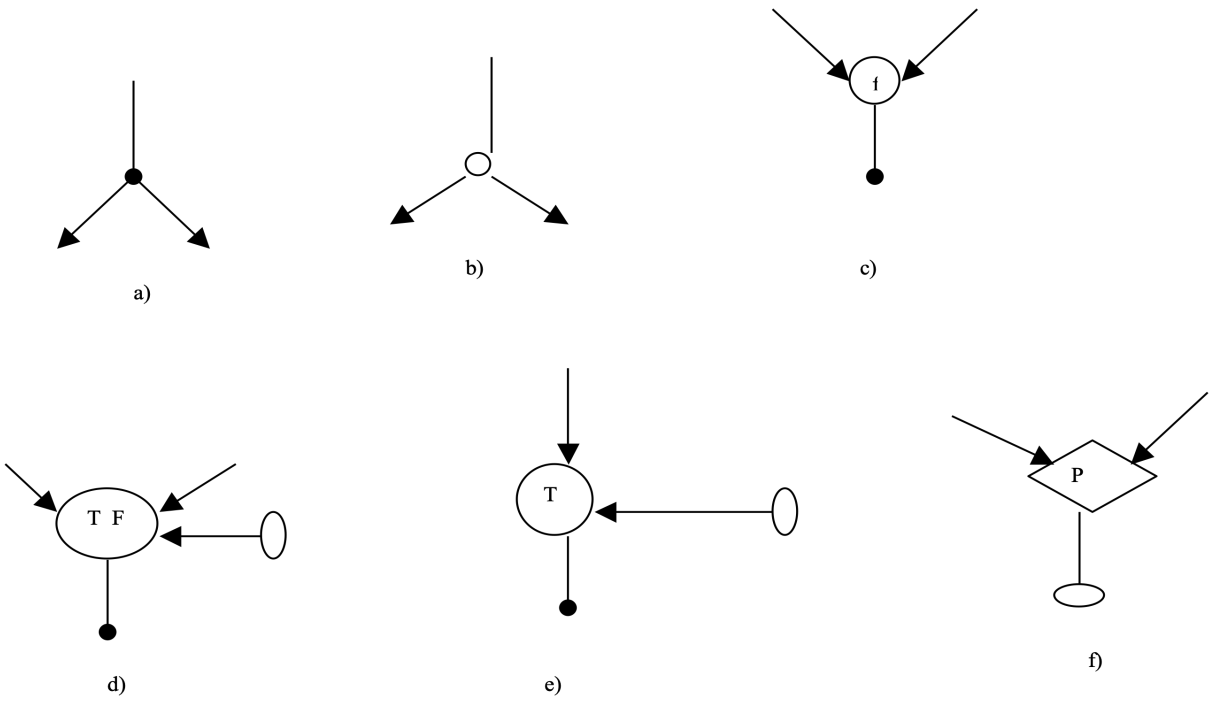
① "Přesuň1" znamená instrukci přesouvající hodnotu zadanou jako parametr do registru 1.

Popis algoritmu orientovaným grafem


a
b
c
d
*
*
+
y

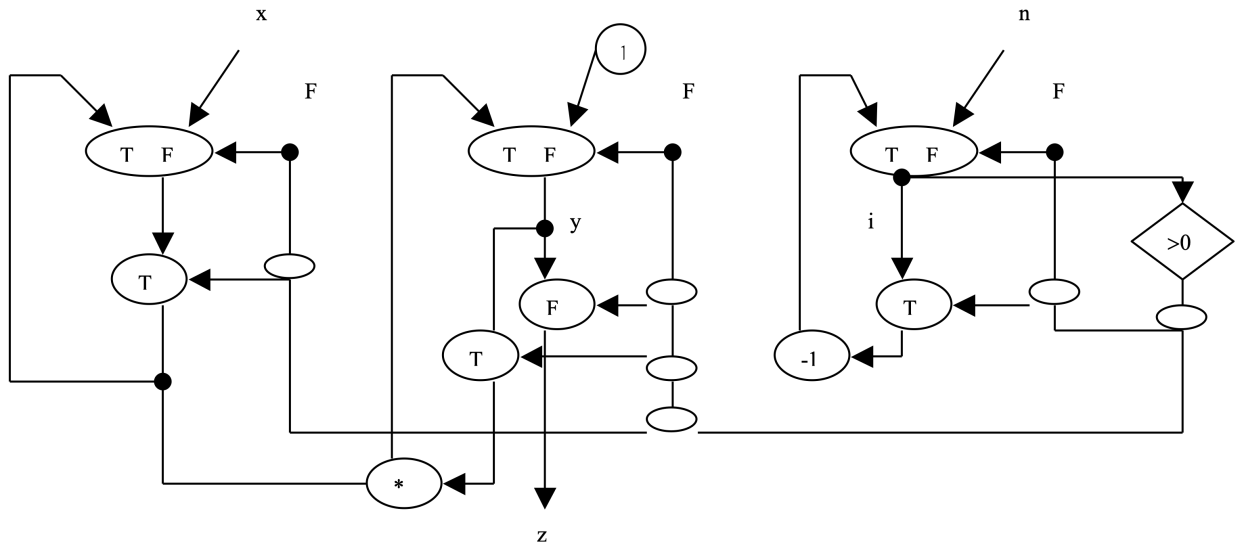


Symboly používané při popisu algoritmů grafem



Obr. 5. a- větvení datového spoje, b- větvení logického spoje, c- symbol operace, d- rozhodovací blok, e- hradlo, f- vyhodnocení logické podmínky (splnění podmínky pro skupinu dat)

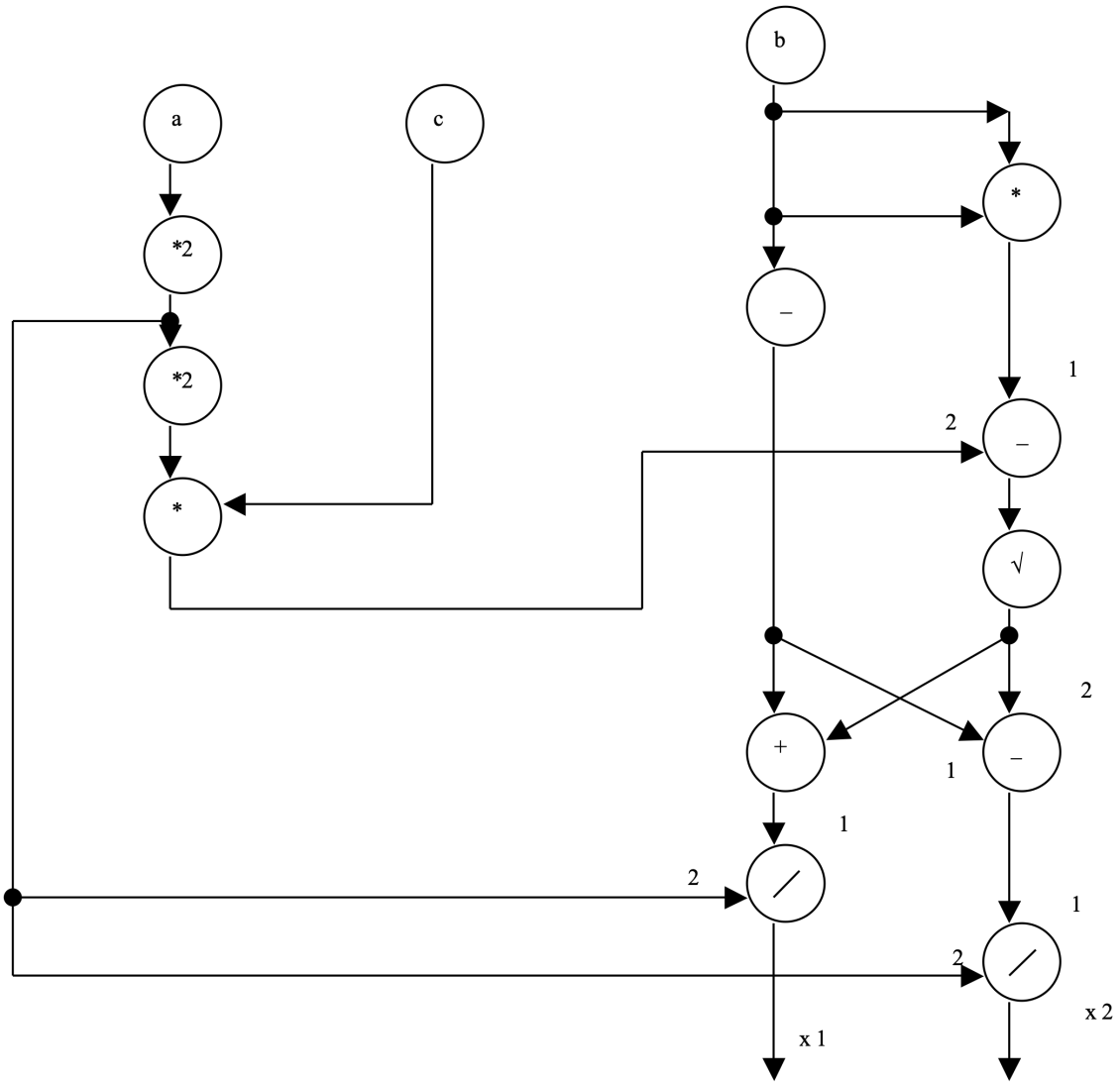
Grafický zápis algoritmu pro výpočet x^(n)



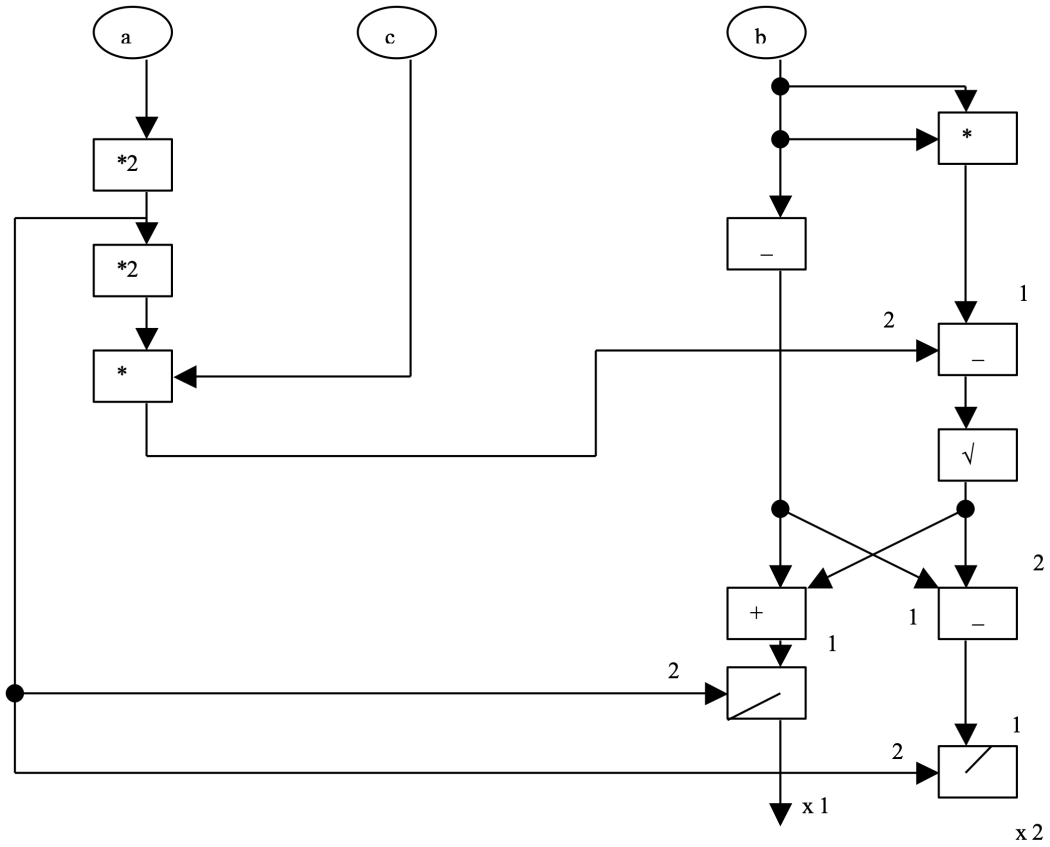
Příklad jednoúčelového počítače řízeného tokem dat

- Podobné analogovým počítačům
- Př. Výpočtu kořenů kvadratické rovnice
- $a*x^2 + b*x + c = 0$
- $X = (-b +/- SQRT(b^2 - 4*a*c))/(2*a)$

Výpočet kořenů kvadratické rovnice



Struktura jednoúčelového procesoru

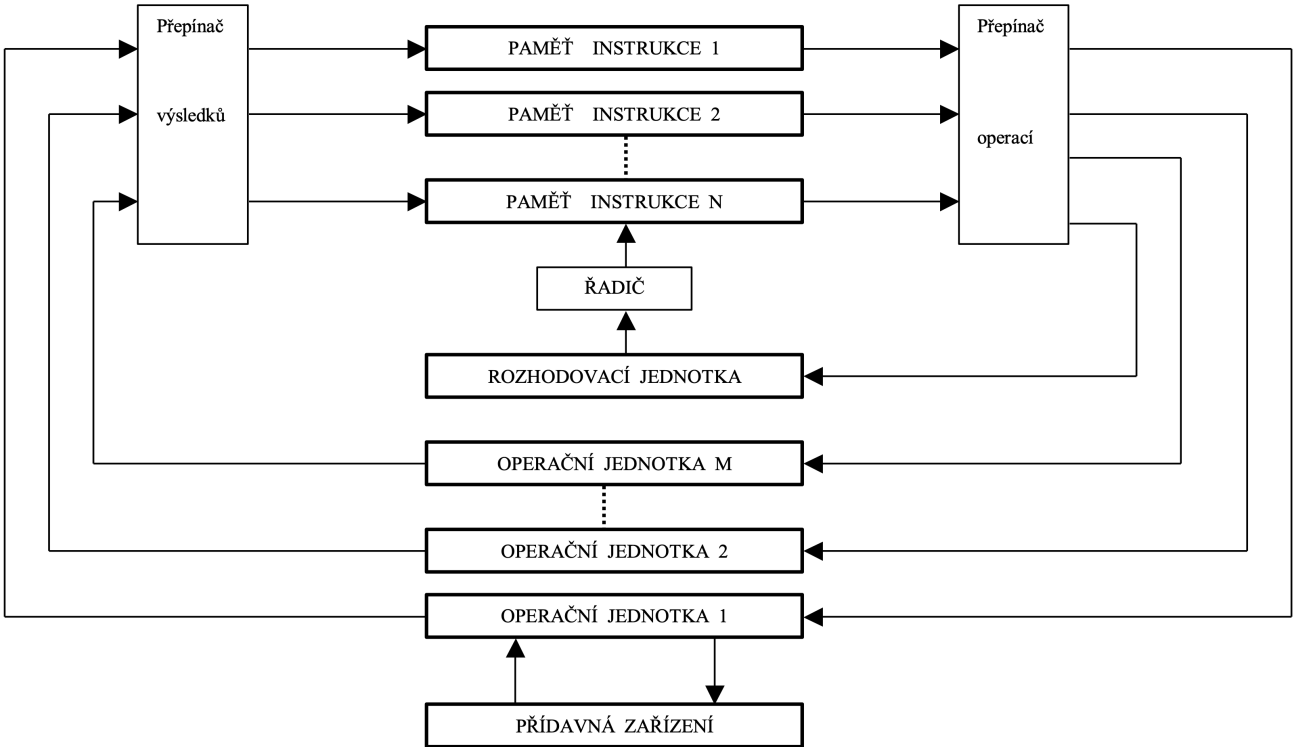


Univerzální počítače řízené tokem dat

Statické není schopen rozlišovat příslušnost operandů k různým úlohám nebo k různým cyklům v rámci jedné úlohy. Proto se musí zajistit rozlišení, aby se na vstupu operační jednotky nesetkaly operandy, které nemají být zpracovány v jedné operaci

Dynamické každý operand je vybaven příznakem příslušnosti k určité úloze — tím umožněno paralelní zpracování

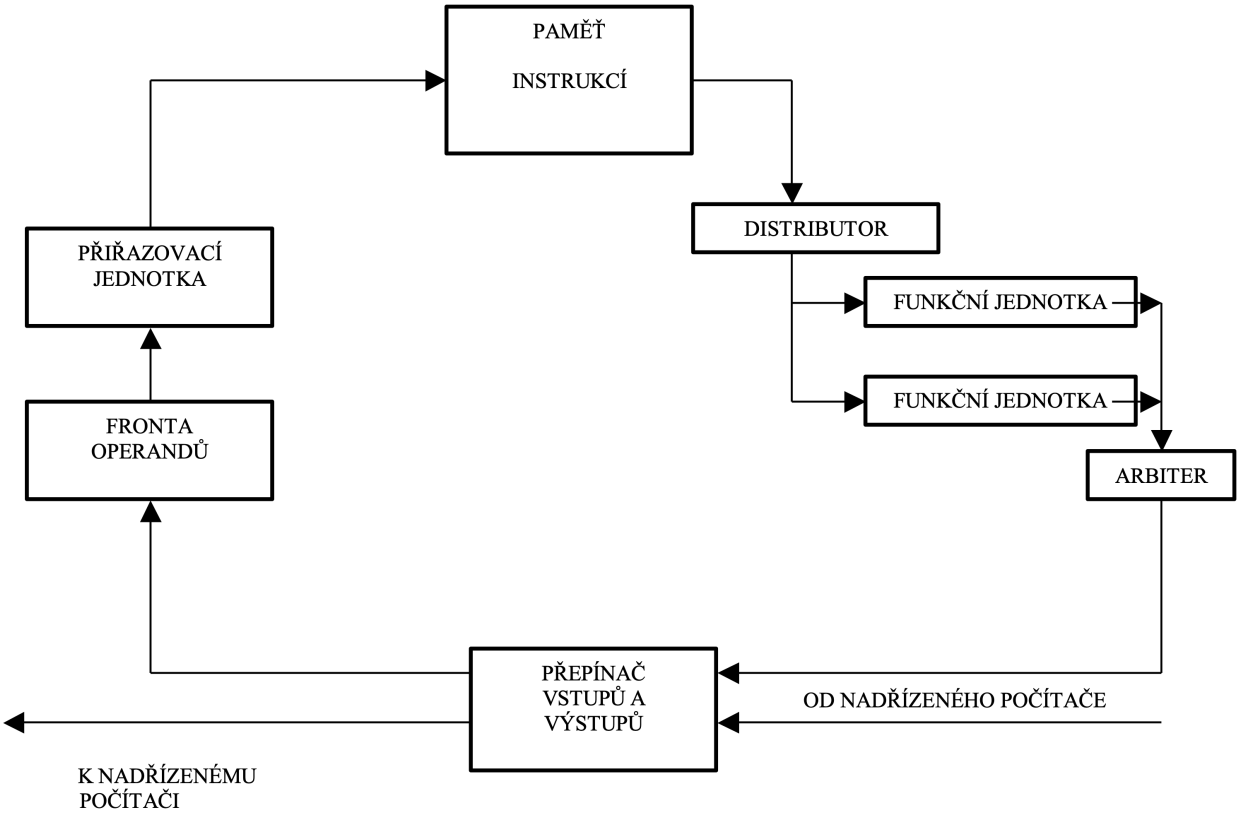
Struktura statického počítače řízeného tokem dat



Paket instrukce



Struktura dynamického počítače řízeného tokem dat



Použitelnost počítačů řízených tokem dat

- Řešení úloh aritmetické povahy
- Provádění podmíněných operací a operací závislých na povaze zpracovávaných dat
- Nevhodné pro realizaci procedur
- Nemožnost použití vyšších programovacích jazyků
- Proto speciální jazyky se snadným překladem do strojového kódu

Počítače řízené tokem požadavků

- Počítače řízené tokem dat mnohdy vytvářejí výsledky „do zásoby“, bez ohledu na to, kdy budou skutečně použity
- Každý mezivýsledek se vypočítá až tehdy, kdy je požadován pro další výpočet

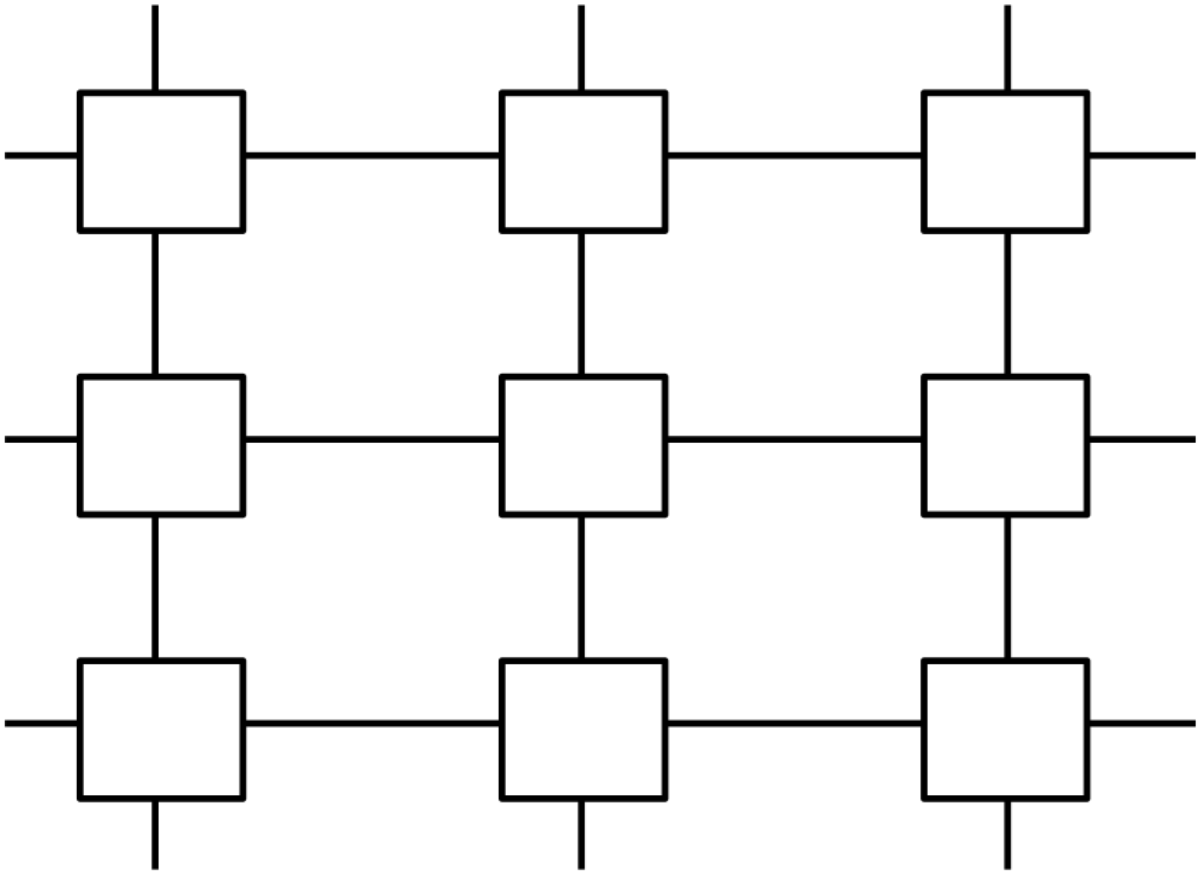
Systolické počítače

- Pevně propojená síť funkčních jednotek, určená pro řešení jedné úlohy nebo jedné třídy úzce příbuzných úloh
- Její propojení je vlastně obvodová realizace algoritmu výpočtu (proto není řízen programem)
- Myšlenka vznikla z redukce nároků na rychlost paměti

Vlastnosti systolických struktur

- mřížkové struktury spec. procesních prvků,
- zpracování dat ve více (nejčastěji 1-2) dimenzích
- na rozdíl od řetězeného zpracování, strukturou proudí vstupní data i mezivýsledky
- data mohou proudit několika směry a různými rychlostmi
- buňky jsou jen jednoho nebo nejvýš několika málo typů
- kombinace intenzivní lokální komunikace a výpočtů
- provádění operací rytmickým, inkrementálním, buňkovým a opakovaným způsobem
- mohou fungovat jako koprocesor (back-end) k hostiteli
- vysoké požadavky na I/O paměťový subsystém mezi buňkami.

Prvky 2-D systolických polí



Aplikace systolických polí

- maticová aritmetika
- zpracování signálů
- zpracování obrazů
- rozpoznávání řeči
- operace s relačními databázemi
- manipulace s datovými strukturami
- manipulace s řetězy znaků

Systolické buňky

Systolické buňky a topologie

pevné, programovatelné, rekonfigurovatelné

Jednoúčelová systolická pole

Zadrátované systolické prvky ušité na míru specifické aplikaci. Desítky nebo stovky buněk se mohou vejít na jeden čip.

Univerzální systolická pole

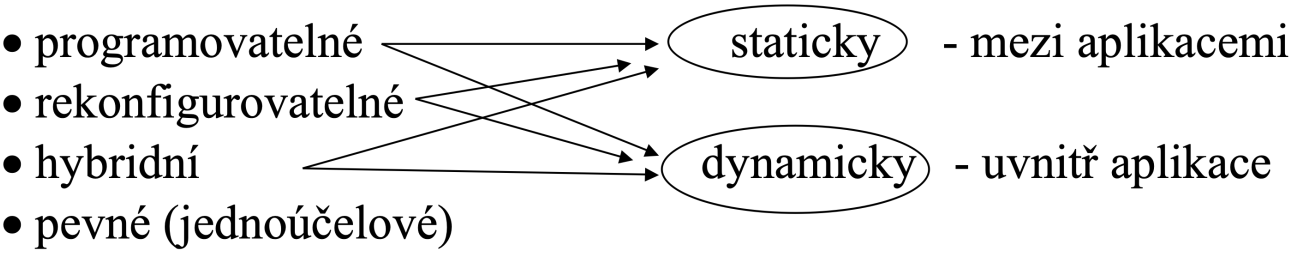
Mohou se adaptovat na řadu aplikací programováním nebo rekonfigurací.

Programovatelná systolická pole

Buď propojení polí nebo každá procesní jednotka je programovatelná.

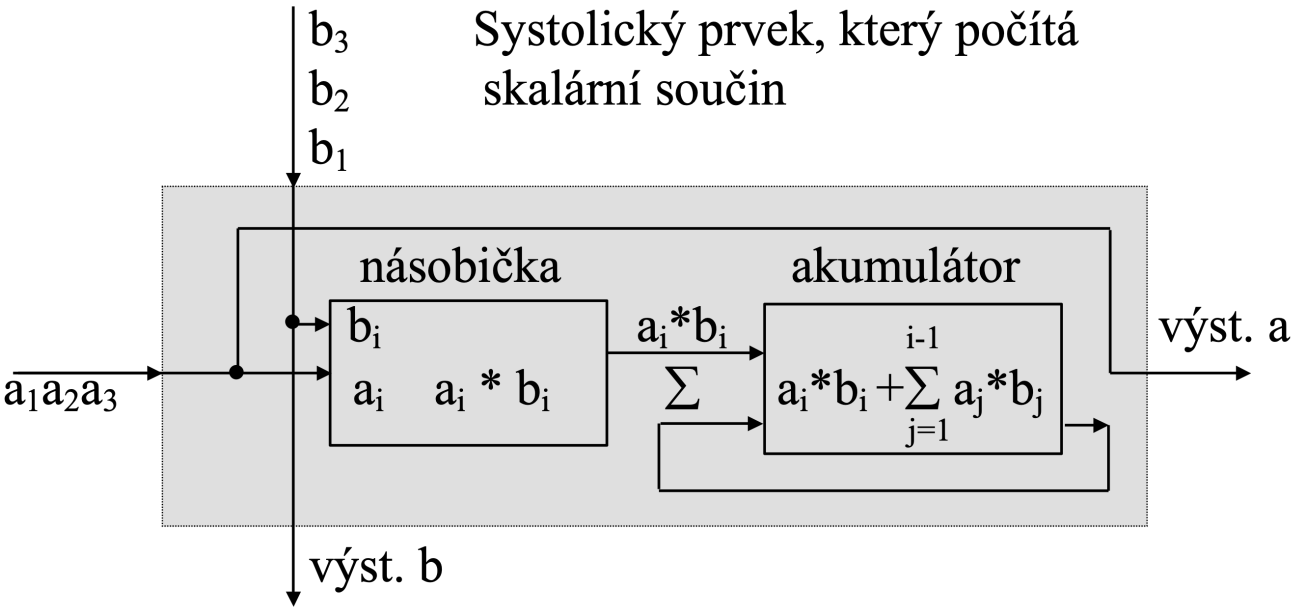
Topologie a typy systolických polí

Buňky a topologie **se mohou měnit:**

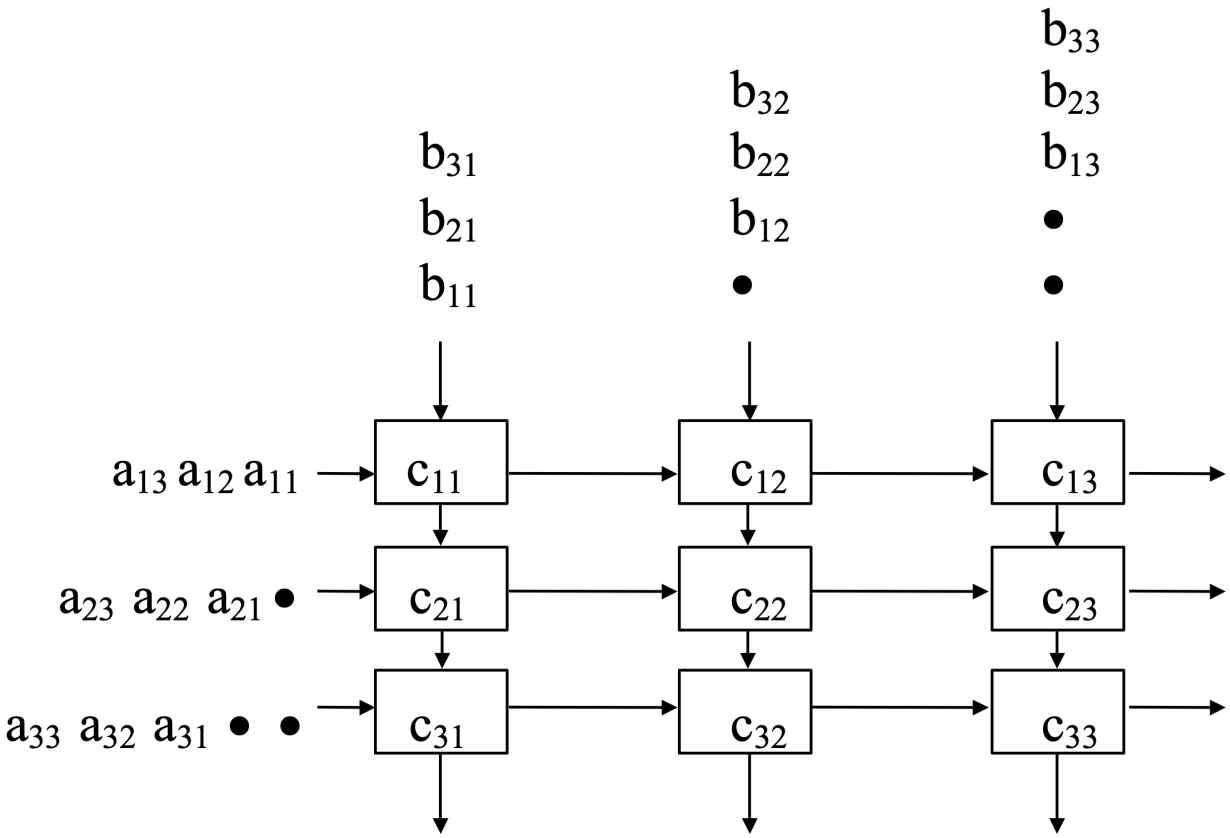


FPGA - field-programmable gate arrays
FPID - field-programmable interconnect devices
CSoC – Konfigurovatelné systémy na čipu (XPP, …)

Systolické násobení matic

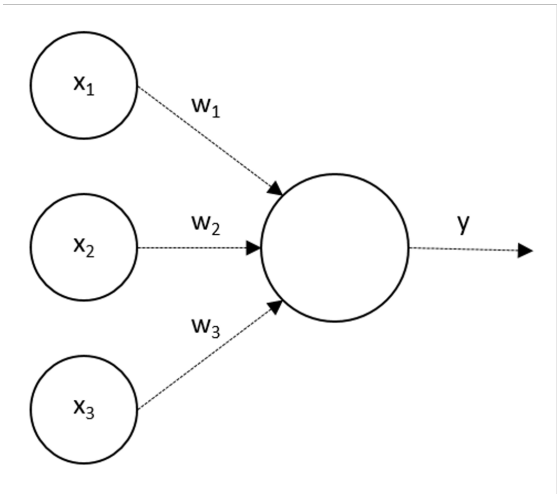


Systolický součin dvou matic 3 x 3


b33
b32 b23
b31 b22 b13
b21 b12
b11
a13 a12 a11
a23 a22 a21
a33 a32 a31
c11 c12 c13
c21 c22 c23
c31 c32 c33

Neuronové počítače

- Základní myšlenkou je síť propojených výpočetních prvků (**umělých neuronů**).
- Historicky se často začínalo modelem prahového logického prvku.
- Dnes se tyto principy uplatňují hlavně v **neuronových sítích** a ve specializovaných akcelerátorech typu **GPU**, **TPU** nebo **NPU**.

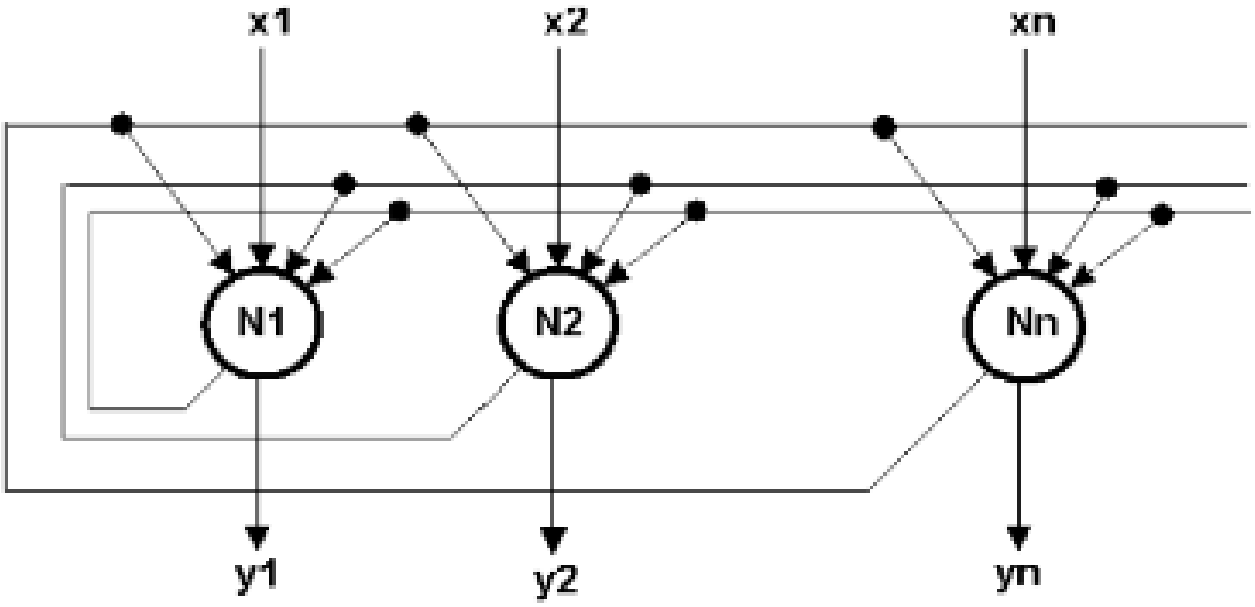


$$y = f\left(\sum_{i=1}^{n} w_i \cdot x_i - \theta\right)$$

$x_i \ldots vstupy$
$w_i \ldots váhy$
$\theta \ldots práh$
$f \ldots přenosová\ funkce$
$y \ldots výstup$

Fáze činnosti

1. **Učení**: nastaví se na požadované chování (váhy, práh a struktura neuronů na základě přiřazení vzorku vstupů k požadované odezvě sítě)
2. **Aktivní činnost sítě**: podle naučeného chování



Vlastnosti

- model se neučí klasickým sekvenčním programem, ale úpravou **vah** podle dat

- dobrá schopnost aproximace složitých závislostí
- schopnost zobecnění na podobná, ne zcela shodná data
- výsledek silně závisí na datech, architektuře a způsobu učení

Příklady použití

- rozpoznávání obrazu, řeči a textu
- klasifikace, odhad, predikce a detekce vzorů
- zpracování dat tam, kde je ručně psaný algoritmus obtížný nebo křehký

Neuronové počítače

Neuronová síť	Počítač
Je učena nastavováním vah, prahů a struktury	Je programován instrukcemi, (if, then, go to,...)
Paměťové a výkonné prvky jsou uspořádány spolu	Proces a paměť pro něj jsou separovány
Paralelismus	Sekvenčnost
Tolerují odchylky od originálních informací	Netolerují odchylky
Samoorganizace během učení	Neměnnost programu

Kvantové počítače

- Jde o výpočetní model založený na principech **kvantové mechaniky**.
- Základní jednotkou dat je **qubit**.
- Jeho základní stavy se zapisují jako `|0>` a `|1>`.
- Qubit může být v **superpozici** těchto stavů; nejde jen o „libovolnou hodnotu mezi 0 a 1“ v běžném číselném smyslu.



Kvantový počítač není „rychlejší klasický procesor“. Je to specializovaný model vhodný jen pro některé typy úloh a dnes rozhodně nenahrazuje běžná CPU.

Klíčové principy kvantového počítání

Superpozice	Qubit může být ve stavu `|0>`, `|1>` nebo v jejich kombinaci.
Propletení	Stav více qubitů nelze vždy rozdělit na nezávislé stavy jednotlivých částí.
Interference	Kvantové algoritmy využívají skládání amplitud tak, aby zvýraznily správné výsledky a potlačily jiné.

Jak kvantový počítač počítá

- Kvantový algoritmus připraví stav qubitů, provede posloupnost kvantových operací a nakonec systém změří.
- Výhoda se neprojeví u každé úlohy; záleží na existenci vhodného kvantového algoritmu.
- Často uváděné oblasti:
  - rozklad velkých čísel,
  - simulace molekul a materiálů,
  - některé optimalizační a vyhledávací úlohy.

Proč ještě nejsou běžné?

- Qubity jsou velmi citlivé na rušení z okolí.
- Kvantové počítače často potřebují *extrémně nízké teploty* (blízko absolutní nuly).
- Je těžké je škálovat, tedy vytvořit kvantový počítač s *mnoha stabilními qubity*.

Příklad kvantového počítání

Příklad specializované úlohy

- Smyslem kvantového výpočtu není „zázračně vyřešit vše jedním krokem“.
- Typický příklad z literatury je

    - **Shorův algoritmus** pro rozklad čísla nebo
    - **Groverův algoritmus** pro vyhledávání.
- Pro výuku zde stačí chápat, že kvantová výhoda vzniká jen u některých úloh a za cenu velmi náročné fyzické realizace.

Klasický procesor (binární)

Základní jednotka	Bit – může být 0 nebo 1
Zpracování	Provádí operace nad bity pomocí logických bran (AND, OR, NOT…)
Architektura	Sekvenční – instrukce se vykonávají krok za krokem
Paměť	RAM, cache, registry – vše klasické
Použití	Všechny běžné počítače, mobily, servery

Kvantový procesor

Základní jednotka	Qubit – může být 0, 1, nebo superpozice obou
Zpracování	Využívá kvantové jevy: superpozice, propletení, interference
Architektura	Speciální kvantový model, ne běžný sekvenční procesor
Paměť	Kvantová paměť – uchovává kvantové stavy (velmi citlivá)
Použití	Speciální úlohy: kryptografie, simulace molekul, optimalizace

Hlavní rozdíly

- Klasické CPU zůstává univerzálním základem běžných počítačů.
- Kvantové počítače jsou zatím experimentální a specializované.

🏁 Co si z kapitoly odnést

- Výkon CPU neurčuje jediná veličina; rozhodují frekvence, IPC, paměťová hierarchie, paralelismus i charakter úlohy.
- Pipeline, superskalárnost, predikce, out-of-order execution a register renaming zvyšují **ILP**.
- Vícejádrové CPU, SMT, SIMD, GPU a NPU přidávají další úrovně paralelismu.
- Amdahlův zákon připomíná, že sekvenční část programu omezuje maximální zrychlení.
- Moderní počítač je heterogenní systém s hierarchickou pamětí a rychlými point-to-point propojeními.
- Netradiční architektury ukazují, že von Neumannův model není jediný možný způsob organizace výpočtu.

Na další kapitolu: Sběrnice

HTML · Slides · PDF | Přehled předmětu