PARALELNÍ ARCHITEKTURY

František Zbořil ml., Petr Hanáček
– 2024

Paralelizace úlohy

- Sekvenční vykonání úlohy – jedním procesorem jedny data.
- Pokud lze úloha rozdělit na **podúlohy**, které lze vykonávat paralelně, může být úloha splněna v kratším čase
- Pokud jeden dělník vykoná práci v čase $T_{sekvenční}$, pak N dělníků vykoná tu samou práci v čase $\frac{T_{sekvenční}}{N}$
- Práce (úloha) však nemusí být plně rozdělitelná (**paralelizovatelná**) mezi všechny dostupné dělníky
  - *Inicializace úlohy – **sekvenční část***

Zrychlení

Zrychlení výpočtu při paralelizaci úlohy

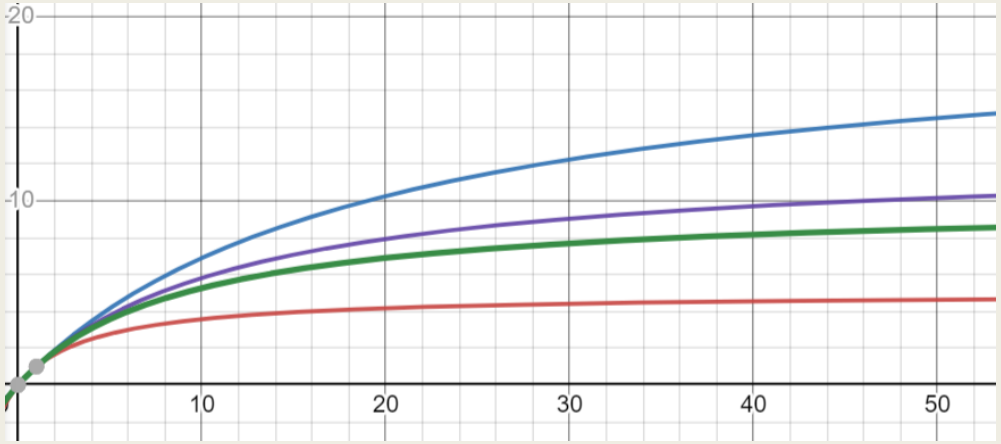
$$S = \frac{T_{neparalelizovaná\ úloha}}{T_{paralelizovaná\ úloha}}$$

Pokud část úlohy P paralelizovatelnou po N podúlohách označíme jako $\alpha P, \alpha \in < 0,1 >$, a zbývající část je vykonána sekvenčně

Amdahlův zákon

$$S(N) = \frac{1}{(1-\alpha)+\frac{\alpha}{N}}$$

$\alpha = 0.8; 0.9; 0.92; 0.95$



Paralelní architektury

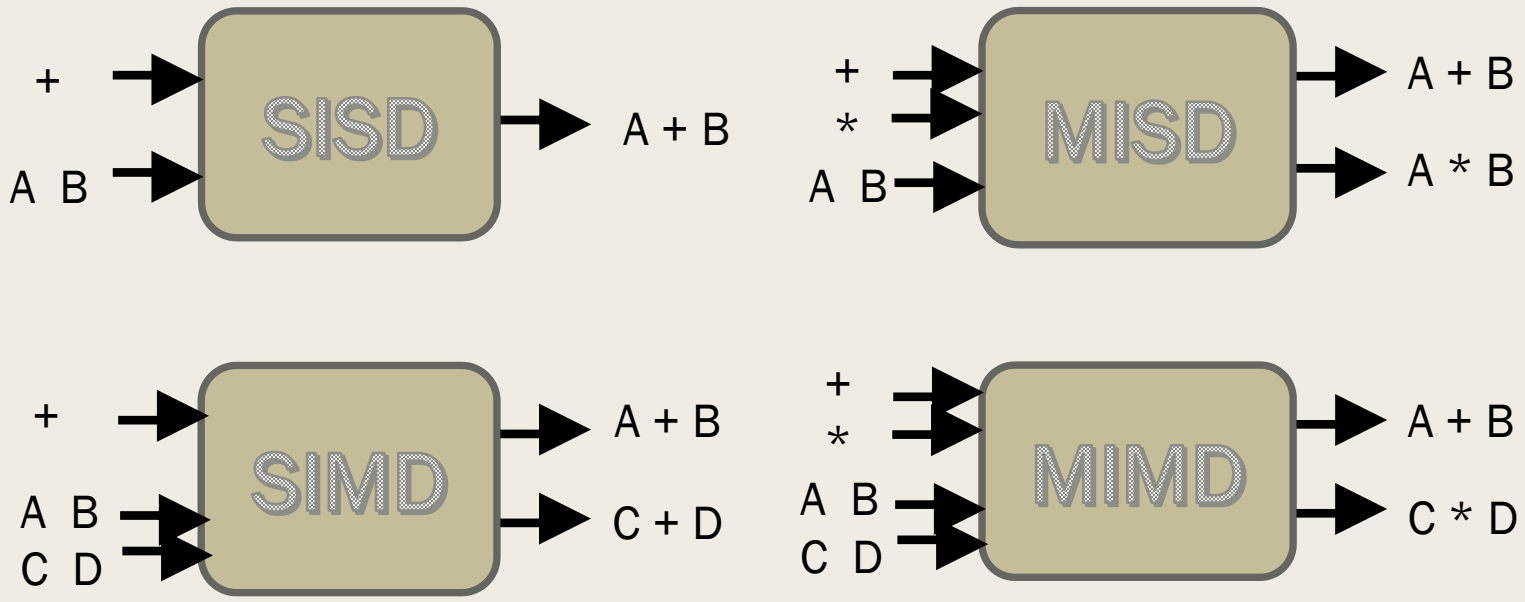
- **Von Neumanovské architektury**
  - *Flynova klasifikace*
- **Ne Vone Neumanovské architektury**
  - *Dataflow (řízené tokem dat)*
  - *Redukční počítače*
  - *Neuronové sítě*
  - *Atd.*

Flynova klasifikace

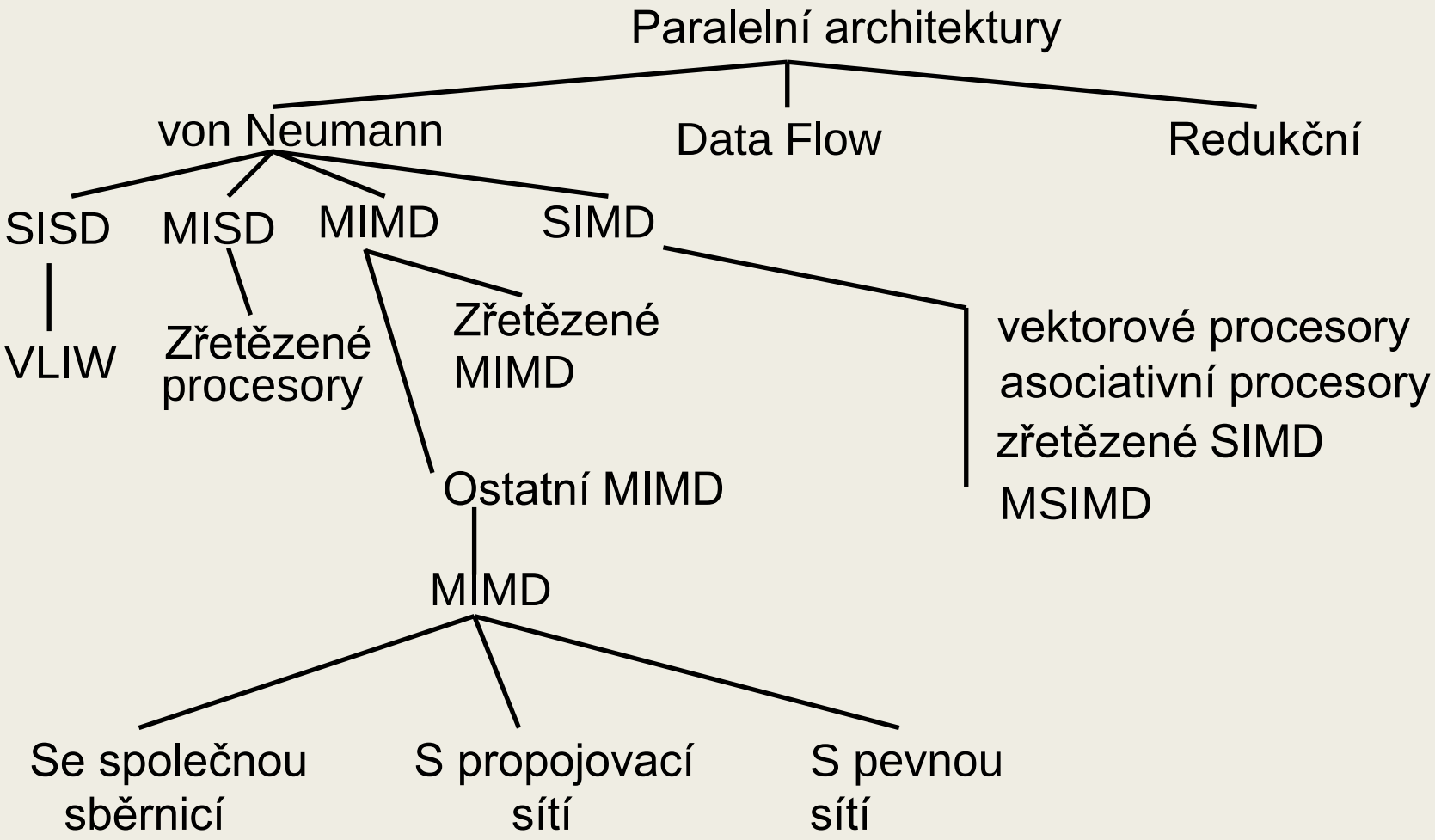
(Single / Multiple, Instruction / Data)

- **SISD** - Jedna instrukce pracuje s jedním datovým prvkem
- **SIMD** - Jedna instrukce pracuje s více datovými prvky
  - *Vektorové procesory*
  - *MSIMD*
- **MISD** - Více instrukcí pracuje s jedním datovým prvkem
- **MIMD** - Více instrukcí pracuje s více datovými prvky
  - *Vícevláknové procesory*
  - *Viceprocesorové systémy*
    - Distribuované systémy

Flynova klasifikace



Paralelní architektury



NE-VON NEUMANOVSKÉ ARCHITEKTURY

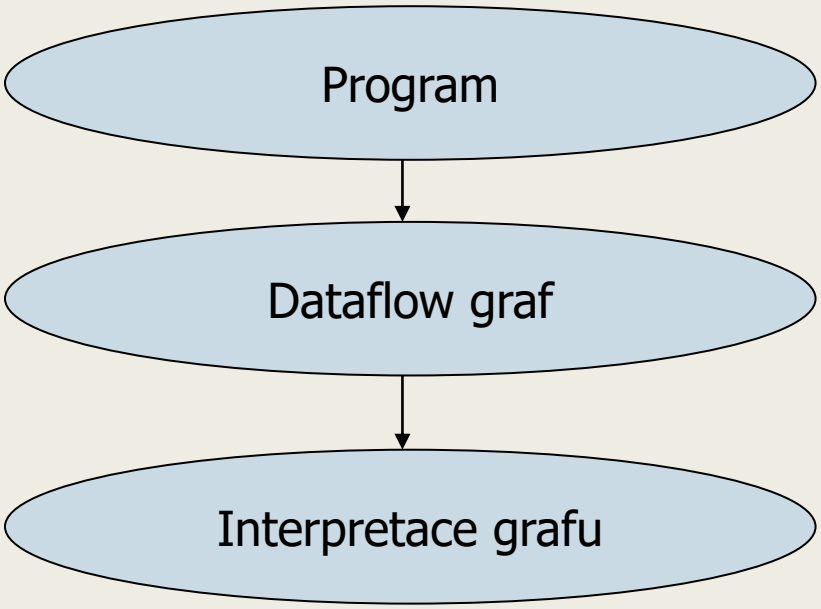
DATAFLOW

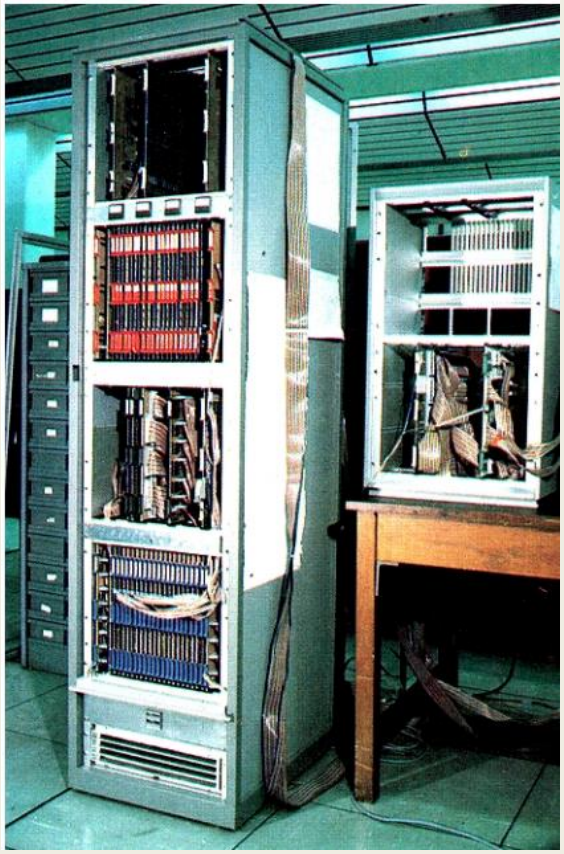
REDUKČNÍ

DATAFLOW ARCHITEKTURY

Dataflow architektura

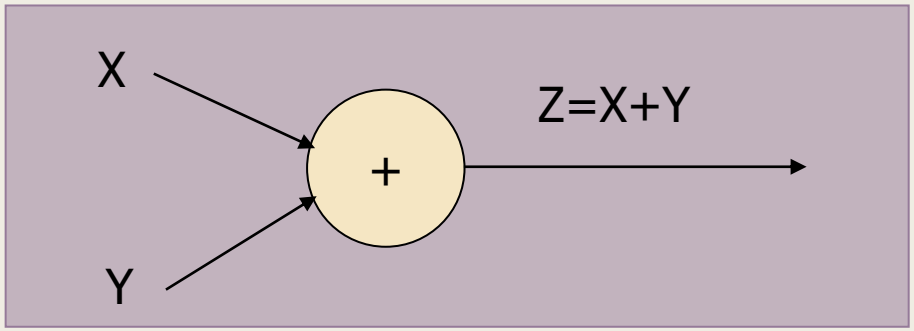
- není von Neumannovská architektura (nemá program a PC)
- provádí interpretaci grafu toku dat



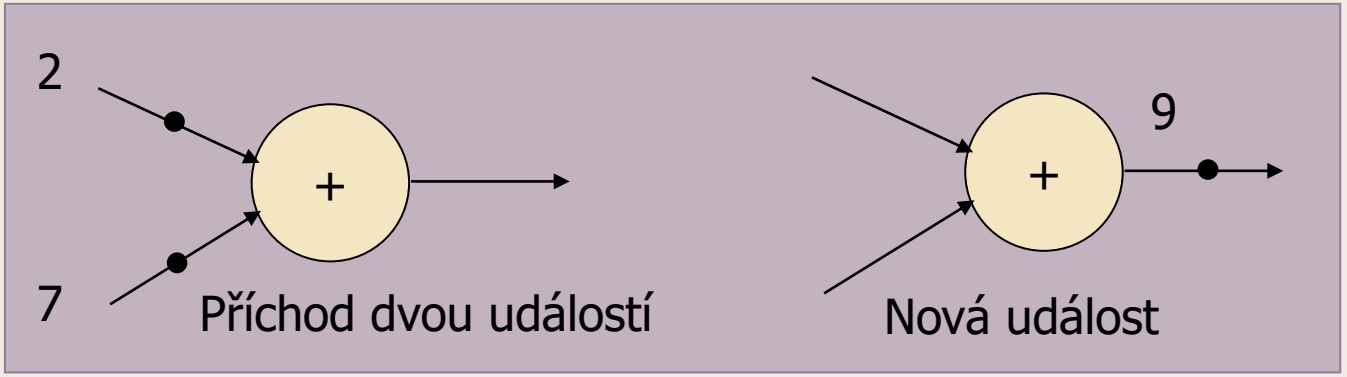


Graf toku řízení

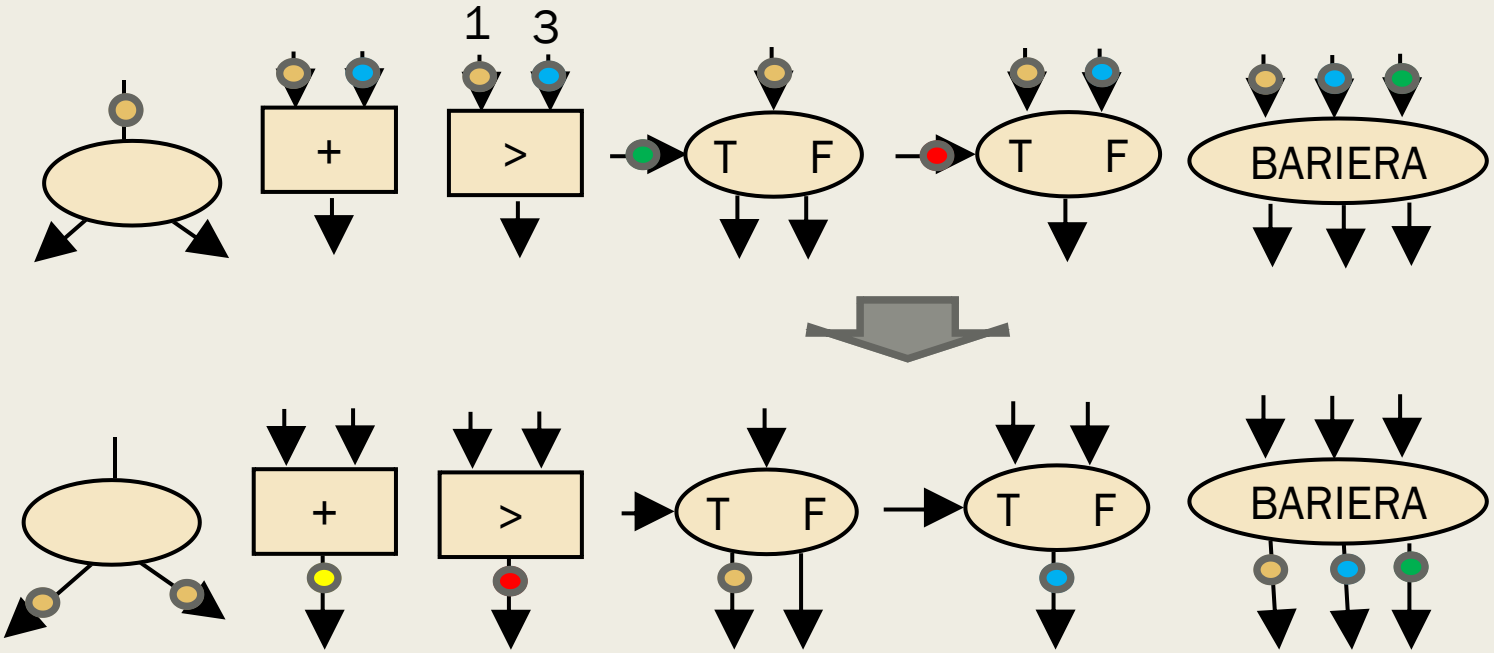
Uzel grafu



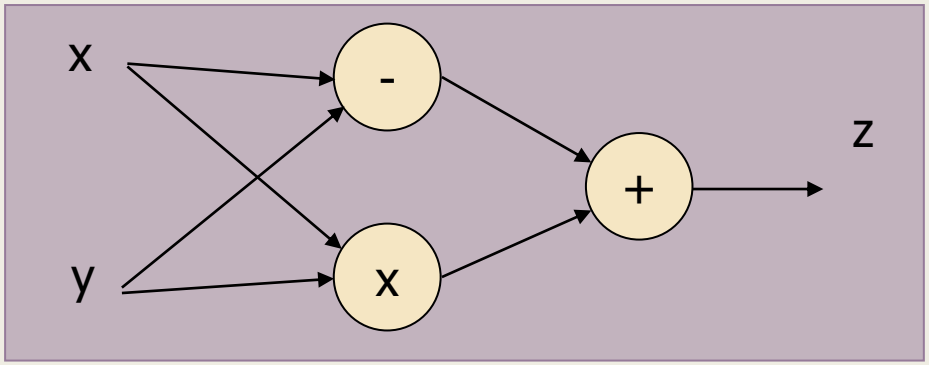
Provedení uzlu (firing)



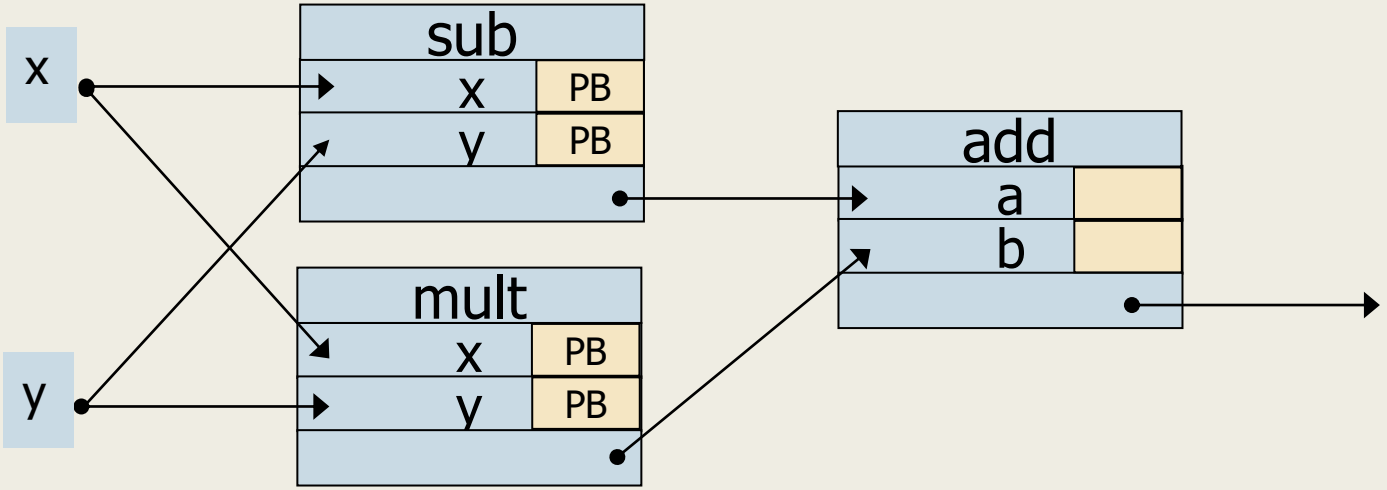
Typy uzlů



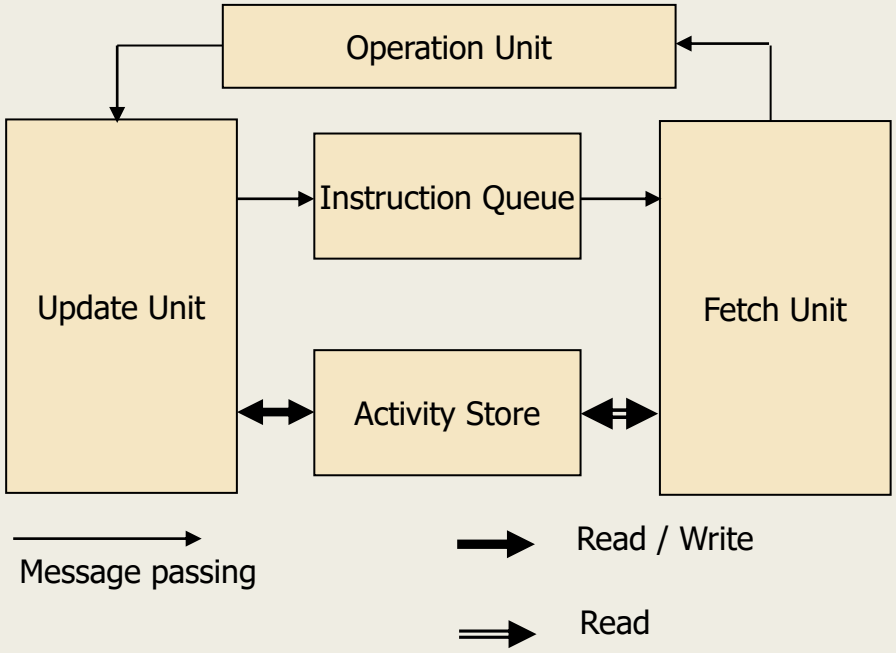
Příklad



Uložení grafu toku dat v paměti aktivit



Dataflow procesor, statický



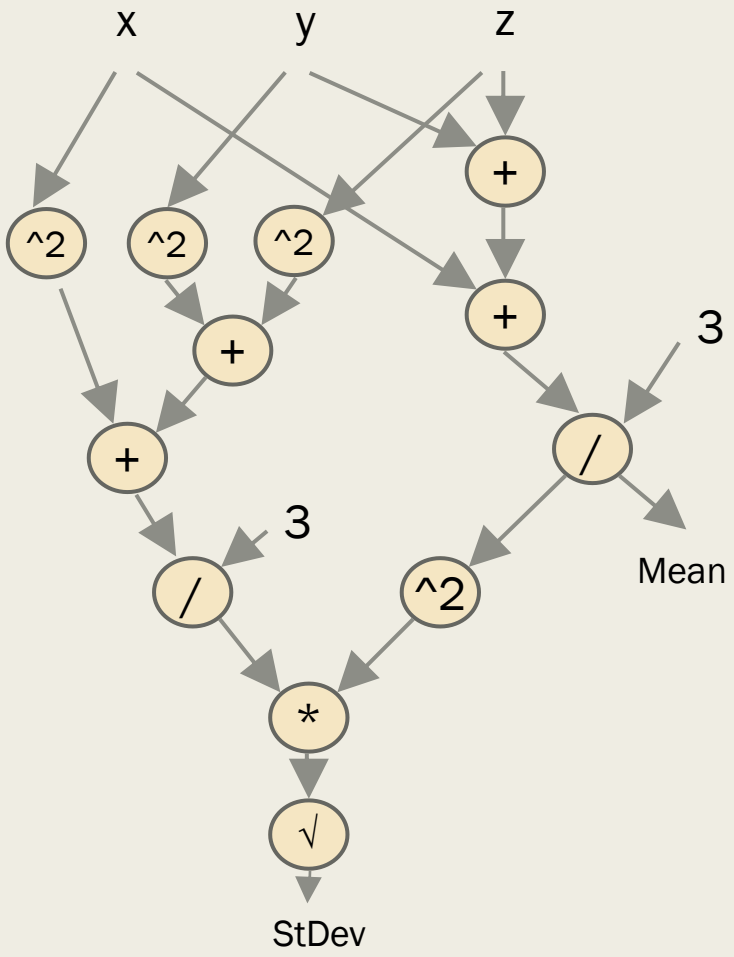
Static. The static dataflow model was proposed by Dennis and his research group at MIT.[2] Figure 1 shows the general organization of their Static Dataflow Machine. The activity store contains instruction templates that represent the nodes in a dataflow graph. Each instruction template contains an operation code, slots for the operands, and destination addresses (Figure 2). To determine the availability of the operands, slots contain presence bits. The update unit is responsible for detecting instructions that are available to execute. When this condition is verified, the unit sends the address of the enabled instruction to the fetch unit via the instruction queue. The fetch unit fetches and sends a complete operation packet containing the corresponding opcode, data, and destination list to one of the operation units and clears the presence bits. The operation unit performs the operation, forms result tokens, and sends them to the update unit. The update unit stores each result in the appropriate operand slot and checks the presence bits to determine whether the activity is enabled.

Jazyky pro Dataflow

- Hlavní požadavek – single assignment rule
  - *Proměnná se může vyskytovat na levé straně přiřazení pouze jedenkrát (v oblasti kódu, kde je používána)*
- Příklad jazyků
  - *VAL, Id, LUCID*
- Dataflow program je přeložen na dataflow graf, což je orientovaný graf s pojmenovánými **uzly**, které reprezentují **instrukce**, a **hranami**, reprezentujícími **závislosti** mezi instrukcemi
  - *(obdobný grafu závislostí, používaným v překladačích)*

Příklad v jazyce VAL

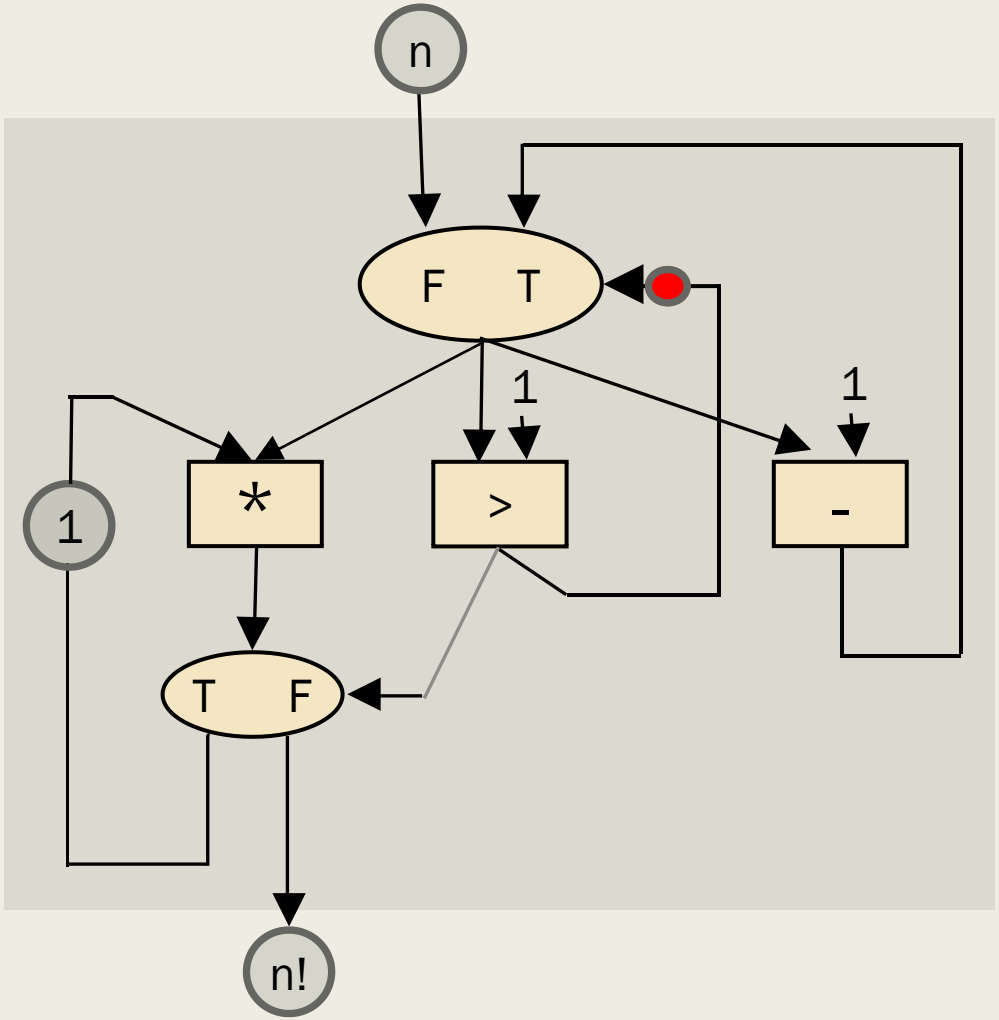
```
function Stats(x,y,z: real
          returns real,real);
  let
    Mean := (x + y + z)/3;
    StDev := SQRT((x**2 + y**2
                      + z**2)/3
                   - Mean**2);
   in
    Mean, StDev
  endlet
endfun
```



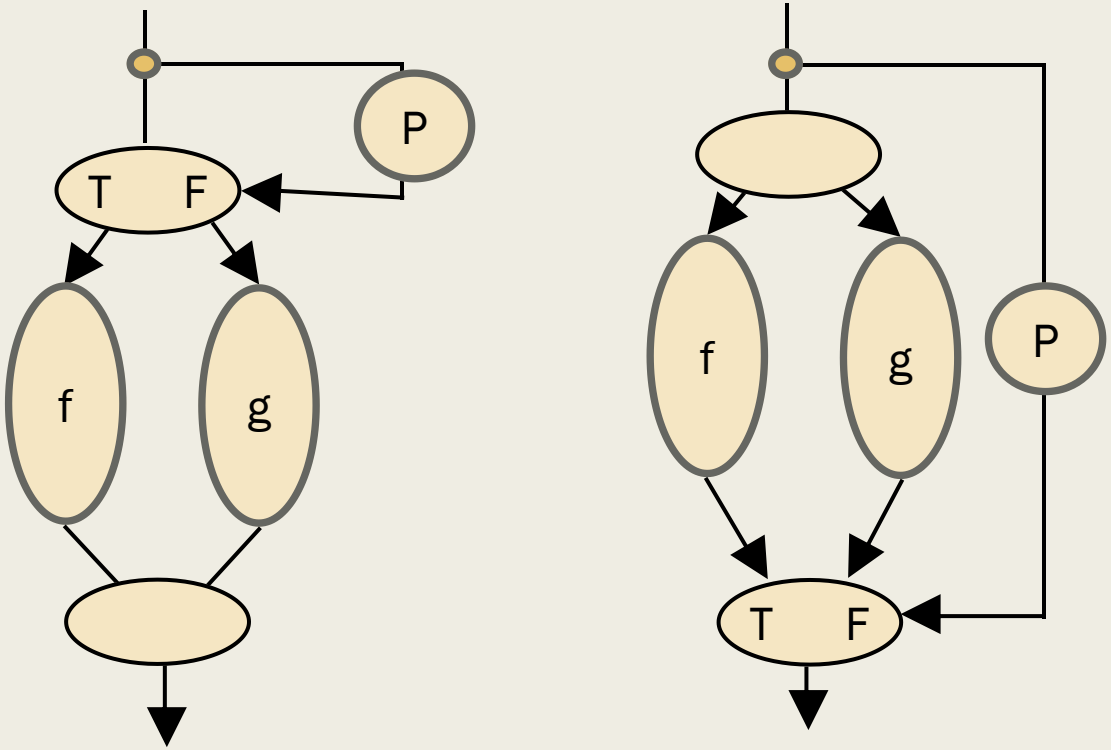
Příklad v jazyce ID, faktoriál

Program počítá faktoriál n! čísla n

```
( initial j <- n; k <- 1
  while j > 1 do
    new j <- j - 1;
    new k <- k * j;
  return k )
```



Implementace podmíněných výrazů



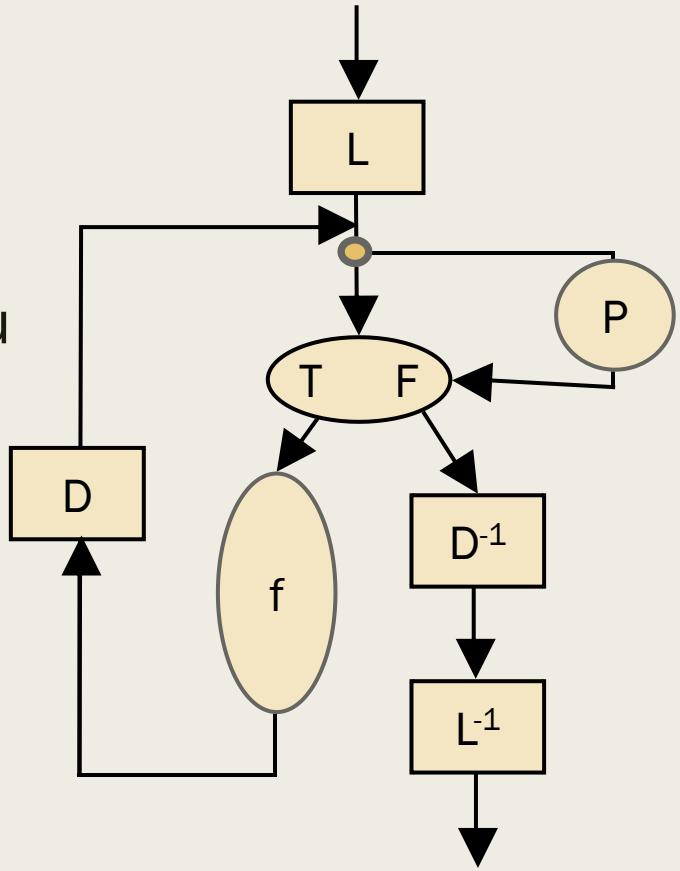
Implementace cyklů

L: Inicializace, nový kontext cyklu

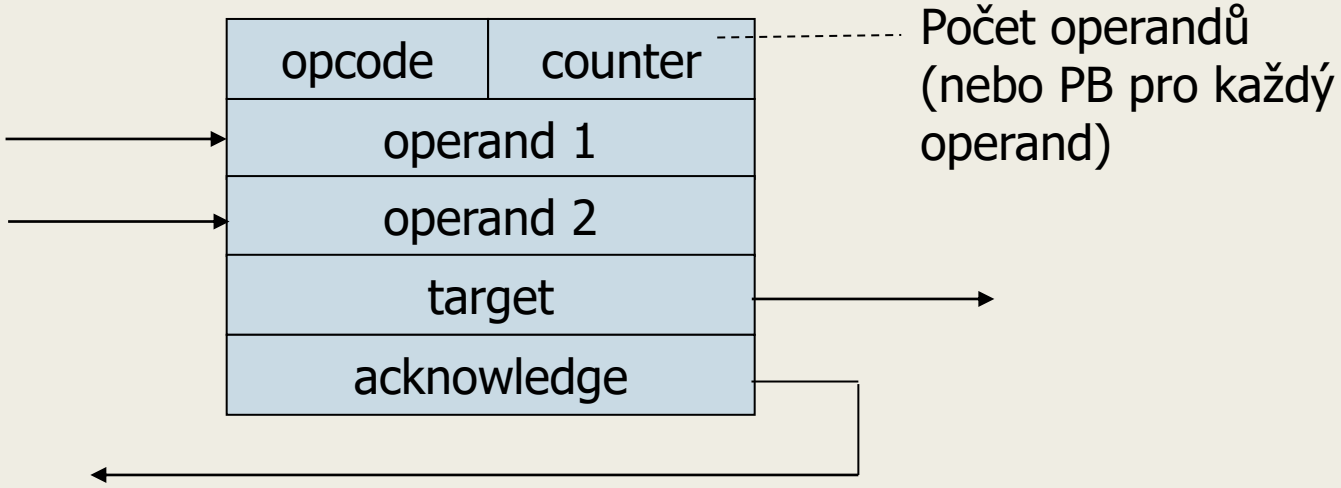
D: Inkrementace proměnné cyklu

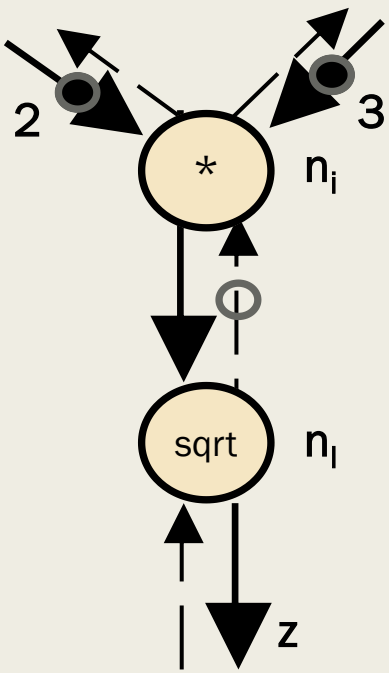
D^{-1}: proměnná cyklu na 1

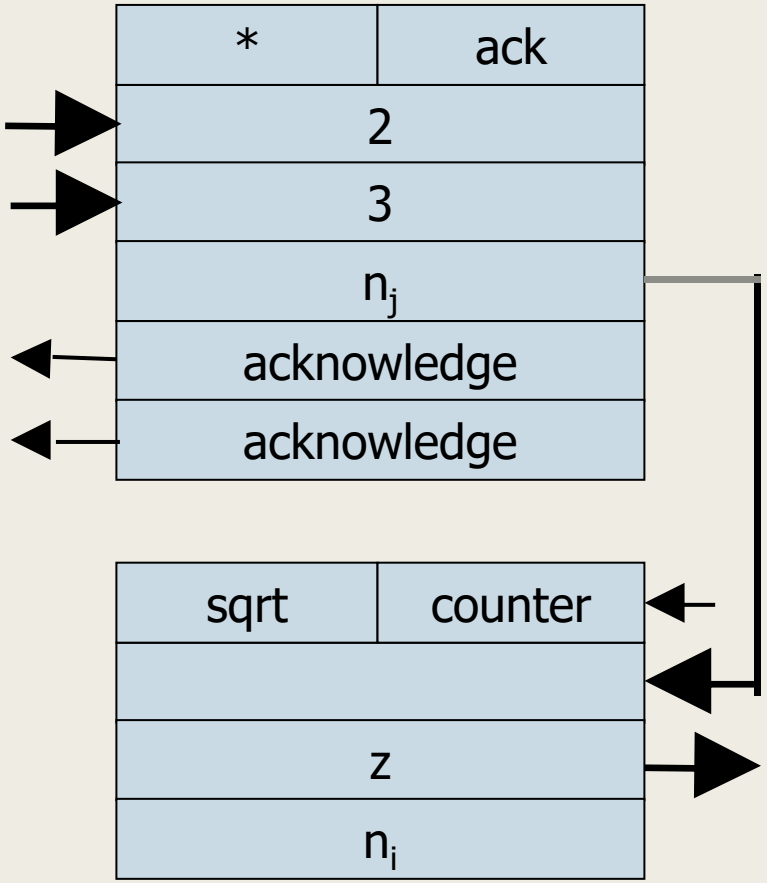
L^{-1}: nastavení původního kontextu

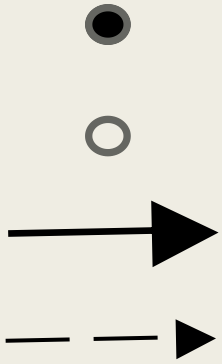


- Požadavek
  - *<opcode, operand, target>*
- Výsledek
  - *<result, target>*
- Zdroj paralelismu - větší počet operačních jednotek, které pracují paralelně
- Problém - vyskytnou-li se na jedné hraně grafu dvě události, může dojít ke změně jejich pořadí → potvrzení




2
3
*
n_i
sqrt
n_l
z

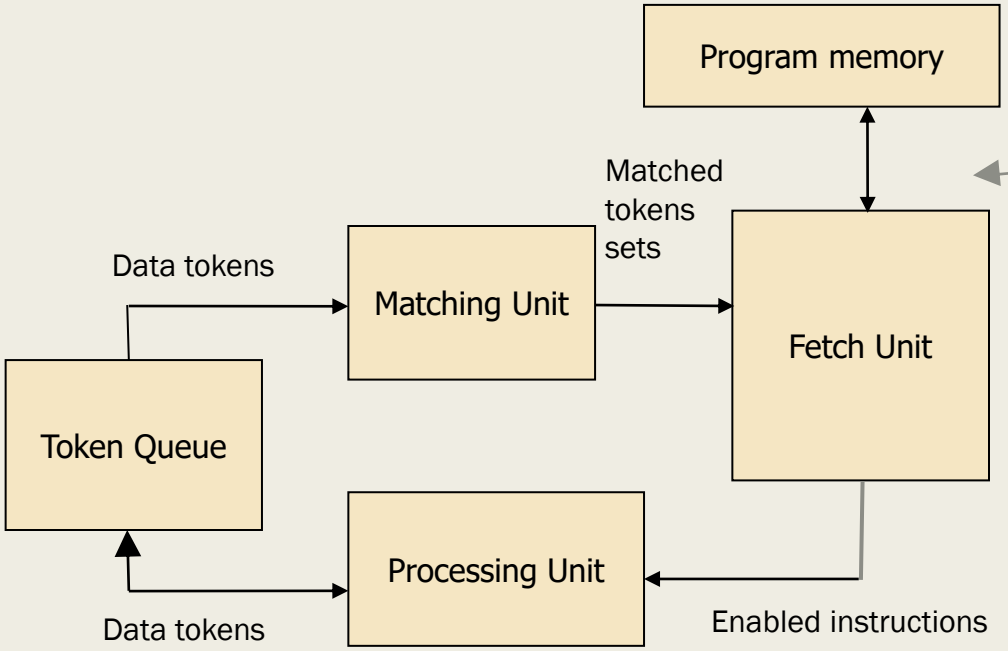

*
ack
2
3
n_j
acknowledge
acknowledge
sqrt
counter
z
n_i

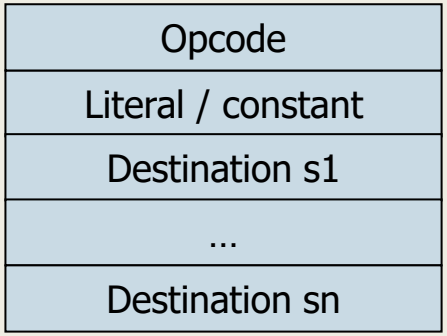


Dynamický Dataflow (Manchester architecture, 1985)

- Každá iterace smyčky nebo vyvolání podprogramu by měla být možná paralelně (jako samostatný podgraf)
- Jeho replikace je jen ‚konceptuální‘
- Každá značka má
  - adresu instrukce, pro kterou jsou data určena
  - kontextové informace
- Každá hrana je soubor (bag) který může obsahovat libovolné množství značek s různými tagy
- Operaci je možné vykonat, pokud má na vstupech všechna potřebná data (s identickými tag-y)

Dataflow procesor, dynamický

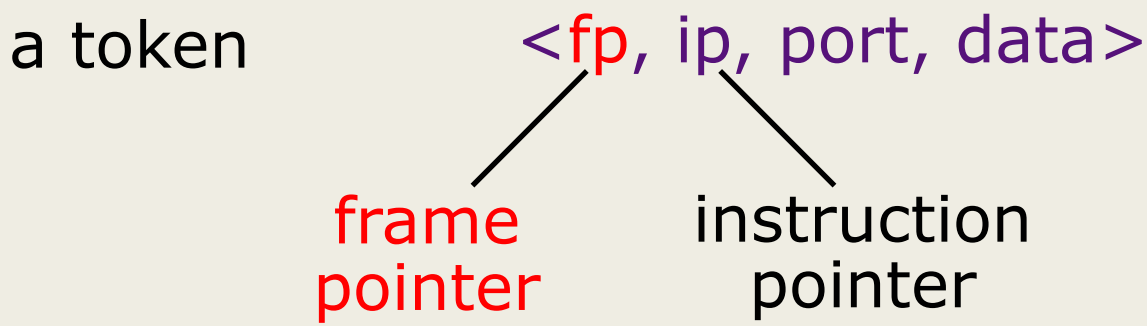




Dynamic. The dynamic dataflow model was proposed by Arvind at MIT[1] and by Gurd and Watson at the University of Manchester.[2] Figure 3 shows the general organization of the dynamic dataflow model. Tokens are received by the matching unit, which is a memory containing a pool of waiting tokens. The unit's basic operation brings together tokens with identical tags. If a match exists, the corresponding token is extracted from the matching unit, and the matched token set is passed to the fetch unit. If no match is found, the token is stored in the matching unit to await a partner. In the fetch unit, the tags of the token pair uniquely identify an instruction to be fetched from the program memory. Figure 4 shows a typical instruction format for the dynamic dataflow model. It consists of an operational code, a literal/constant field, and destination fields. The instruction and the token pair form the enabled instruction, which is sent to the processing unit. The processing unit executes the enabled instructions and produces result tokens to be sent to the matching unit via the token queue.

Dynamic Dataflow Architectures

- následující tři slidy od Onur Mutlu, Computer Architecture: Dataflow (Part I)
- Allocate instruction templates, i.e., a frame, dynamically to support each loop iteration and procedure call
  - termination detection needed to deallocate frames
- The code can be shared if we separate the code and the operand storage



A Frame in Dynamic Dataflow

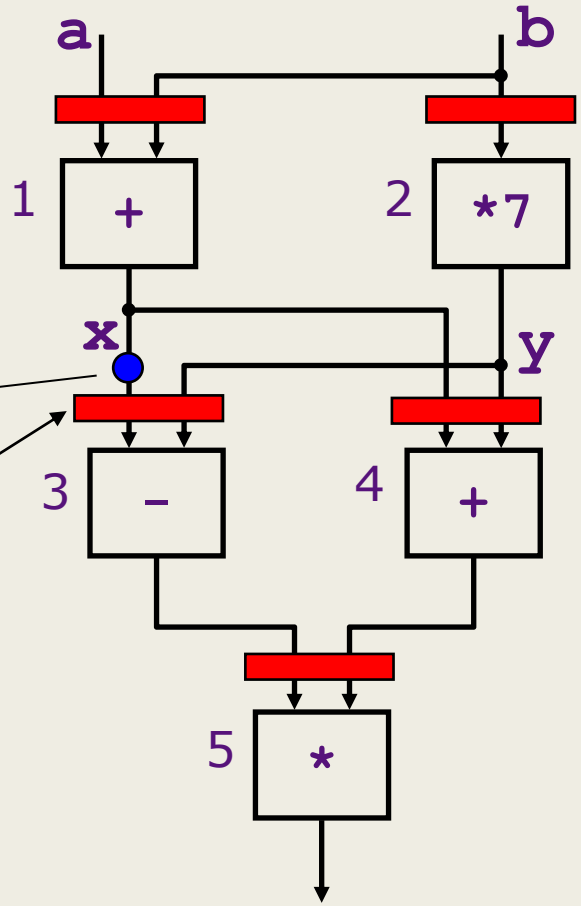
1	+	*1*	3L, 4L
2	*	*2*	3R, 4R
3	-	*3*	5L
4	+	*4*	5R
5	*	*5*	out

Program

<fp, ip, p , v>

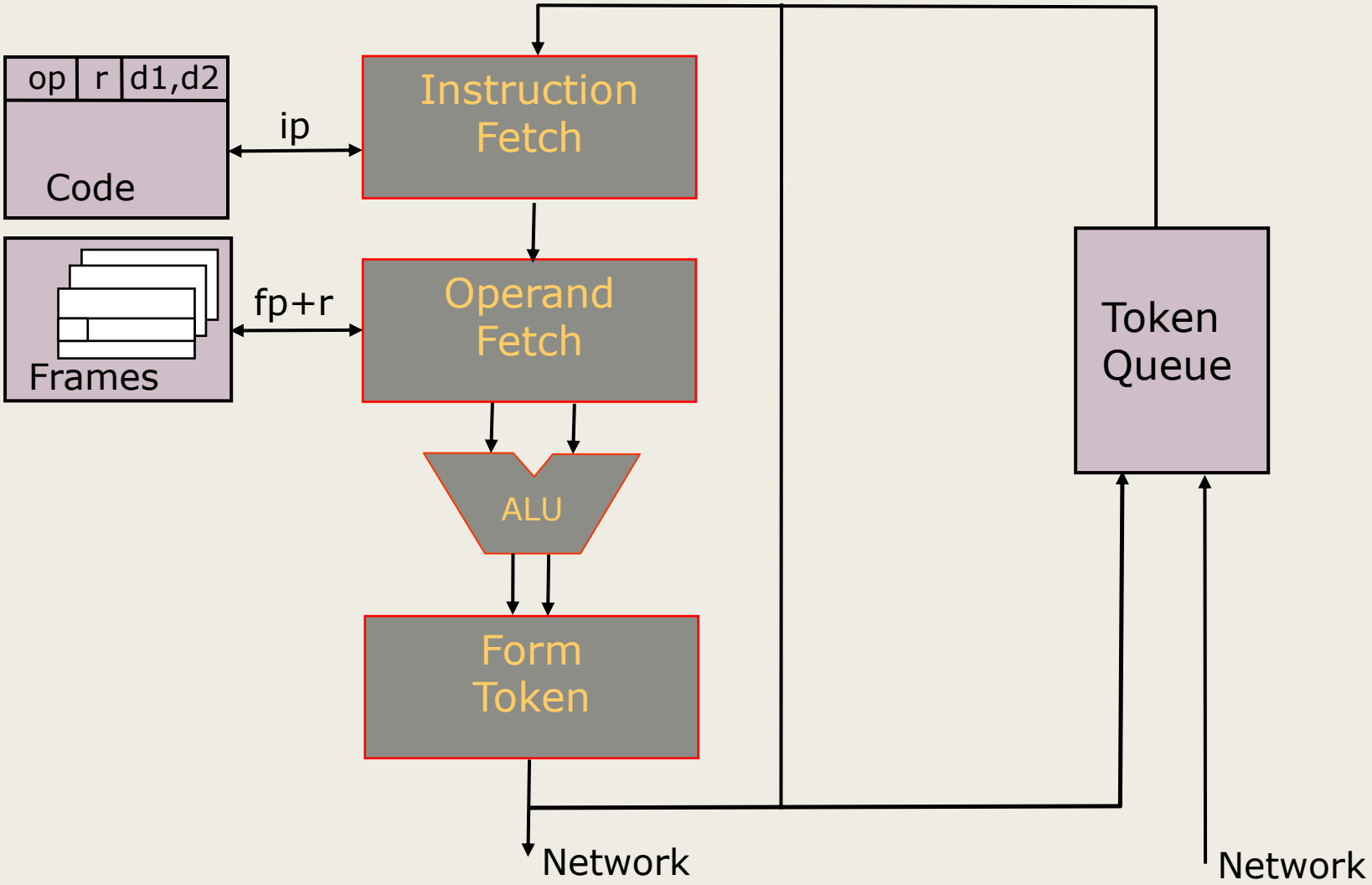
1		
2		
3	L	7
4		
5		

Frame



Need to provide storage for only one operand/operator

Monsoon Processor (ISCA 1990)



REDUKČNÍ POČÍTAČE

Redukční počítač

- Funkcionální programování / založené na redukci výrazů
- Redukce, nahrazení části výrazu jeho hodnotou
- Příklad

$$x * y + (x - y)$$

V jazyce LISP

(+ (* x y) (- x y))

Pro redukční počítač se rozdělí do sekvencí (vymezenou < ... >)

$$+< (*< x\ y >)(-< x\ y >) >$$

Pro vstupní hodnoty / argumenty x=3, y=2

$$+< (*< 3\ 2 >)(-< 3\ 2 >) >$$

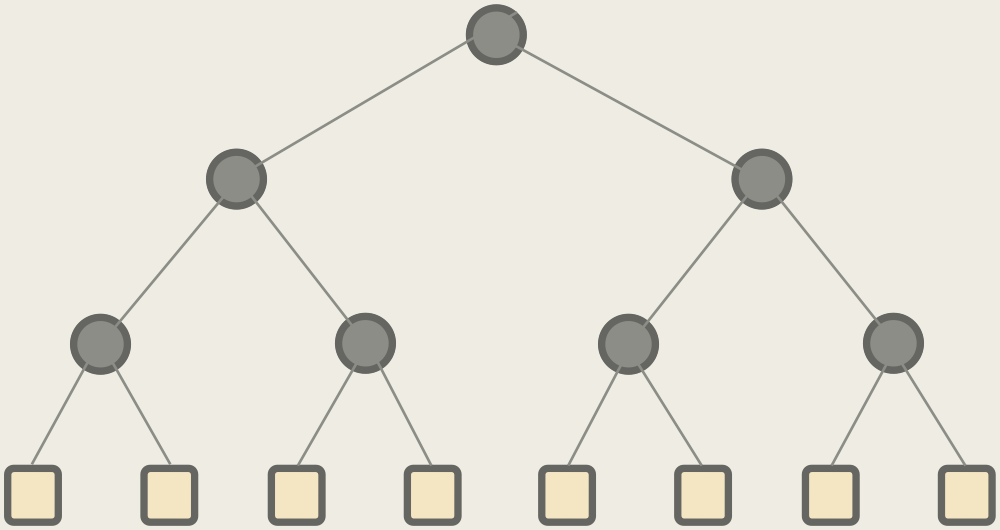
$$+< 6\ 1 >$$

$$7$$

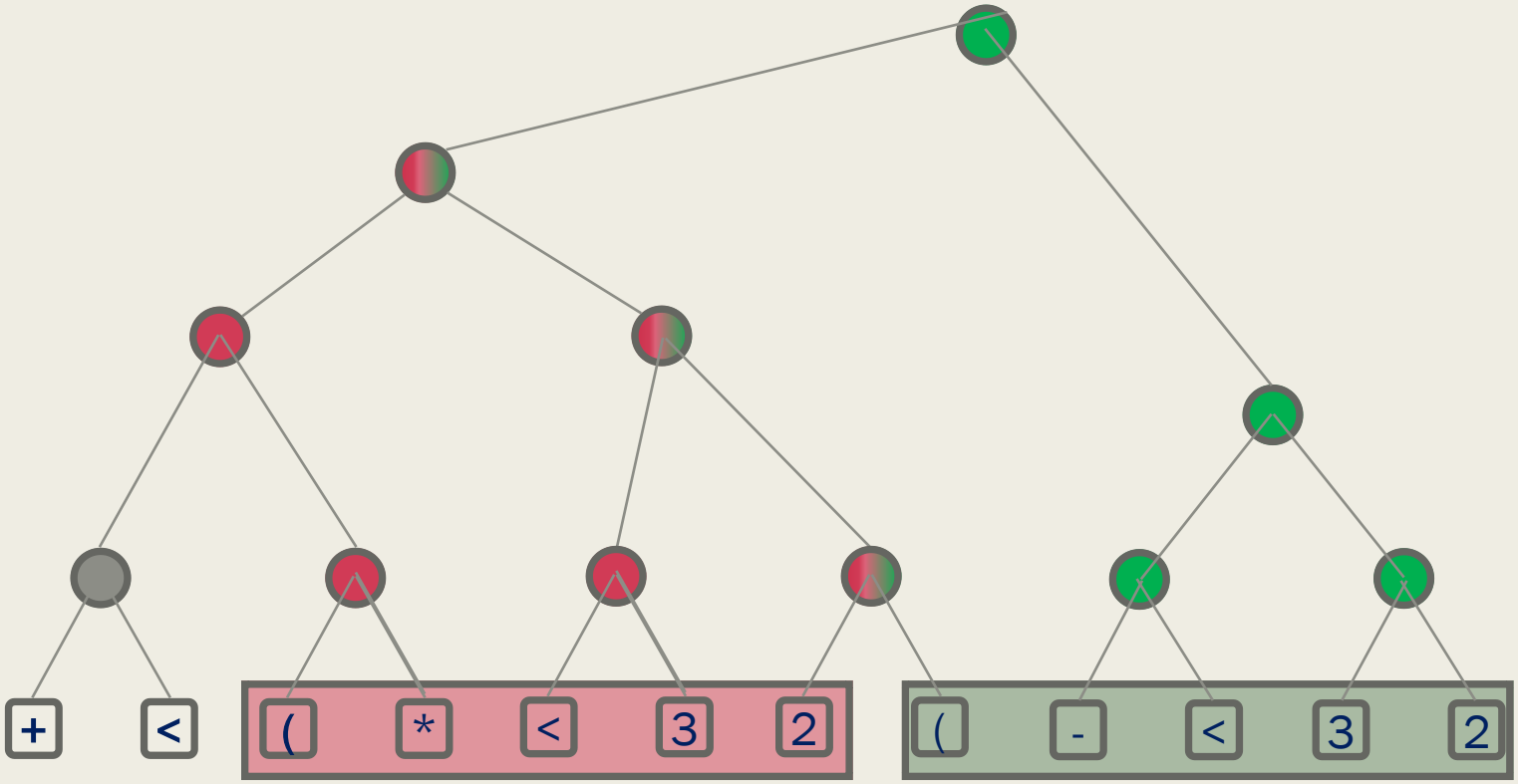
Redukční architektura, stromová

- T uzly: procesory a komunikace
- L uzly: paměti

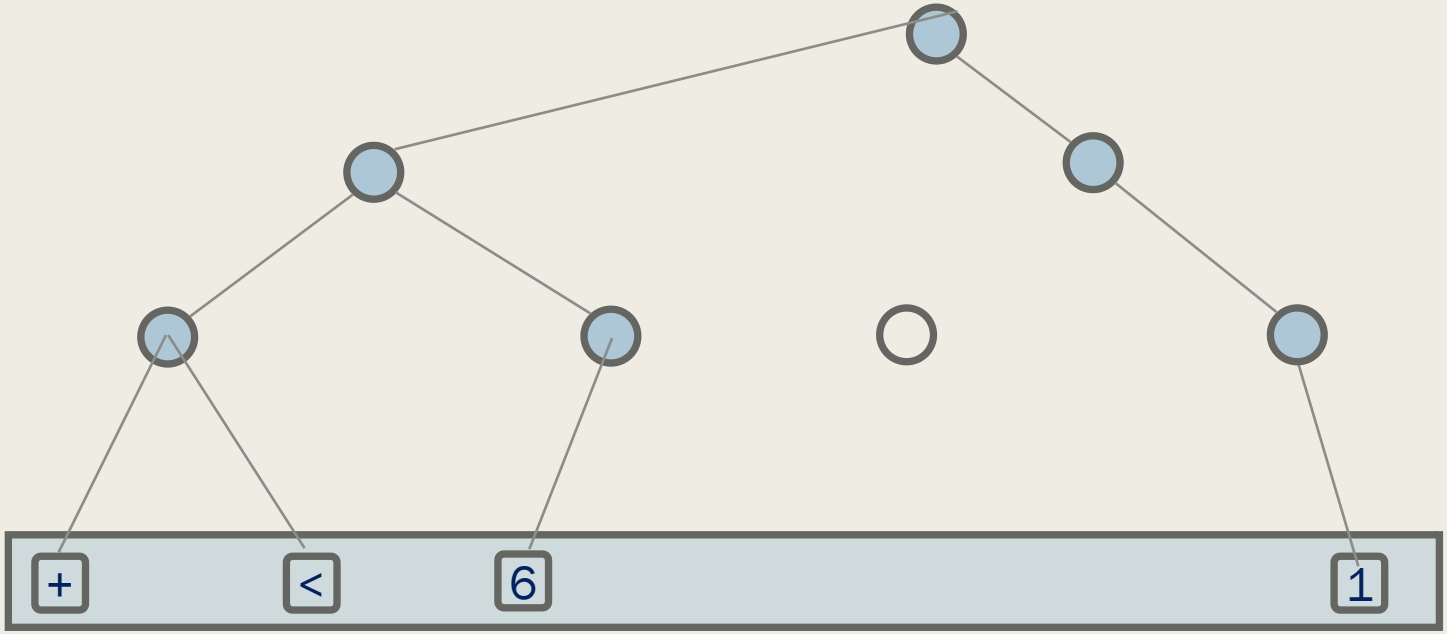
1, rozdělení L-uzlů na redukční pole
2, provedení redukce
3, posuv v paměti L-uzlů



Redukční architektura, stromová



Redukční architektura, stromová



7

Redukční počítače ALICE

- Založen na rekurzivně vyčíslitelných funkcích
- Funkce jsou zapsány pravidly
  - *Pravidla jsou prvního řádu a mohou zahrnovat rekurzi*

Příklad, připojení prvku do seznamu

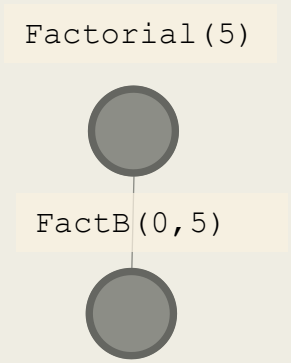
```
Append(x, nil) ← x::nil
Append(x, e::L) ← e:: Append(x, L)
```

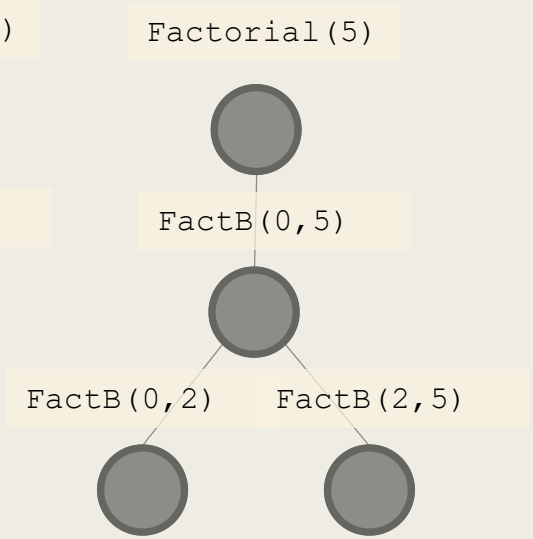
Příklad, výpočet faktoriálu (pro n>0)

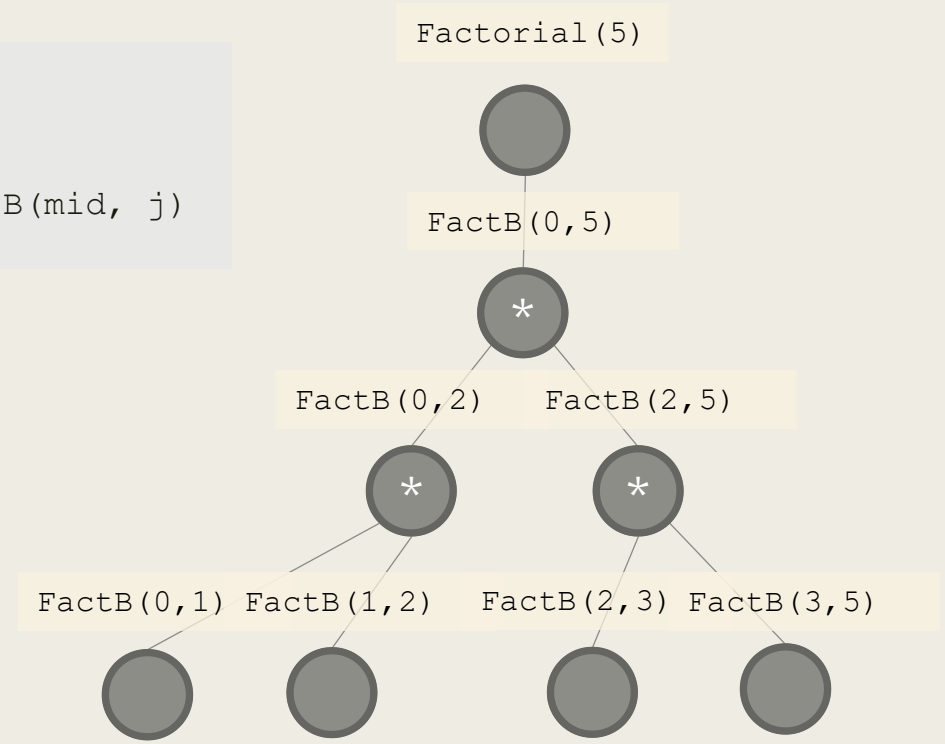
```
Factorial(n) ←  FactB(0, n)
FactB(i, i) ← i
FactB(i, i+1) ← i+1
FactB(i, j) ← FactB(i, mid) * FactB(mid, j)
     kde mid= (i+j) div 2
```

Výpočet faktoriálu rozkladem

```
R1 Factorial(n) ← FactB(0, n)
R2 FactB(i, i) ← i
R3 FactB(i, i+1) ← i+1
R4 FactB(i, j) ← FactB(i, mid) * FactB(mid, j)
```

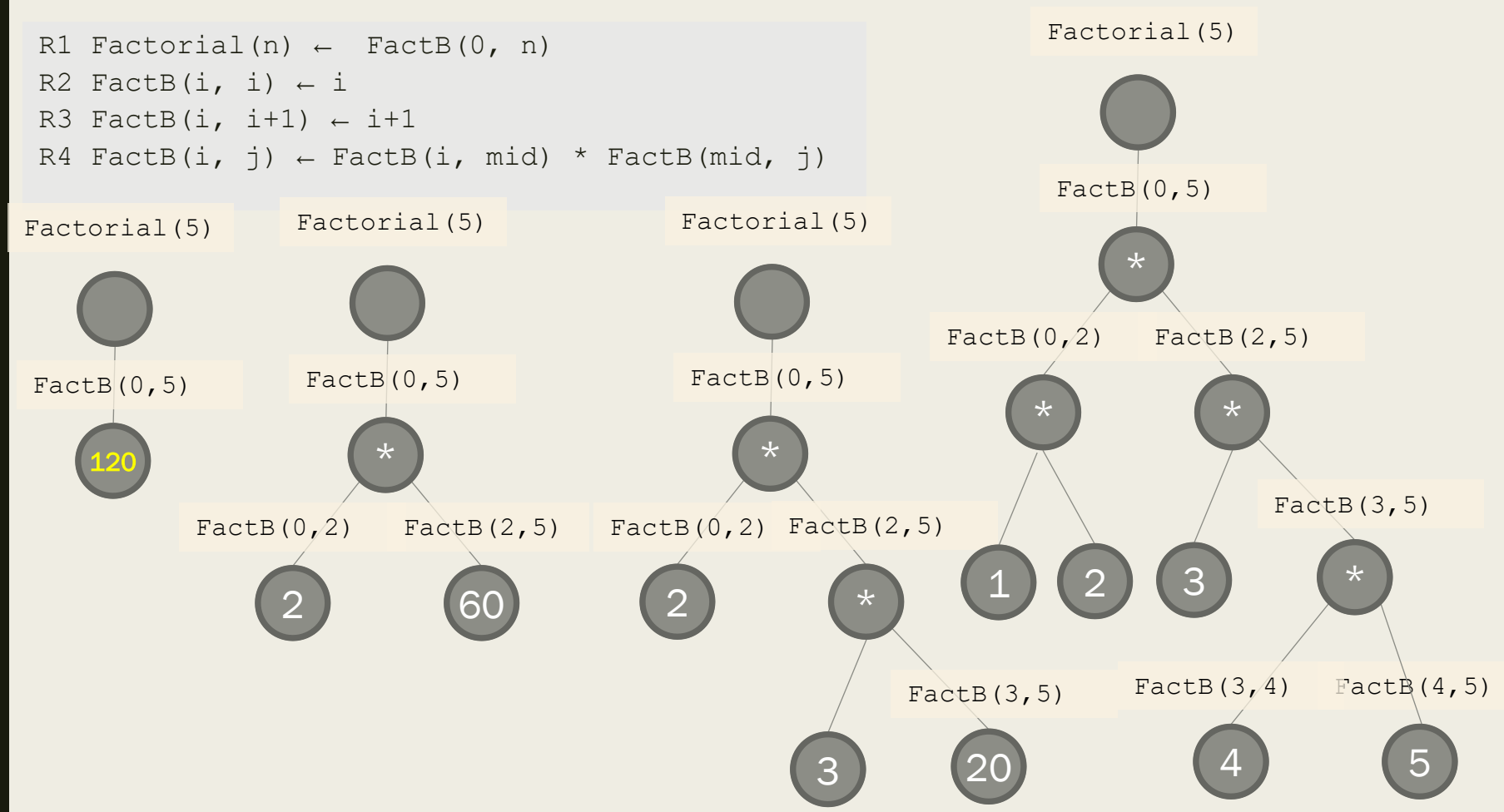






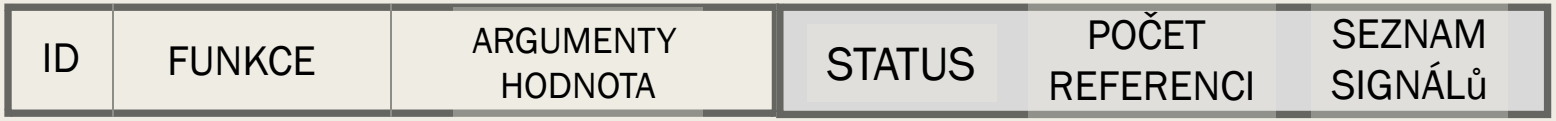
Výpočet faktoriálu rozkladem

```
R1 Factorial(n) ← FactB(0, n)
R2 FactB(i, i) ← i
R3 FactB(i, i+1) ← i+1
R4 FactB(i, j) ← FactB(i, mid) * FactB(mid, j)
```



Redukční počítače, ALICE

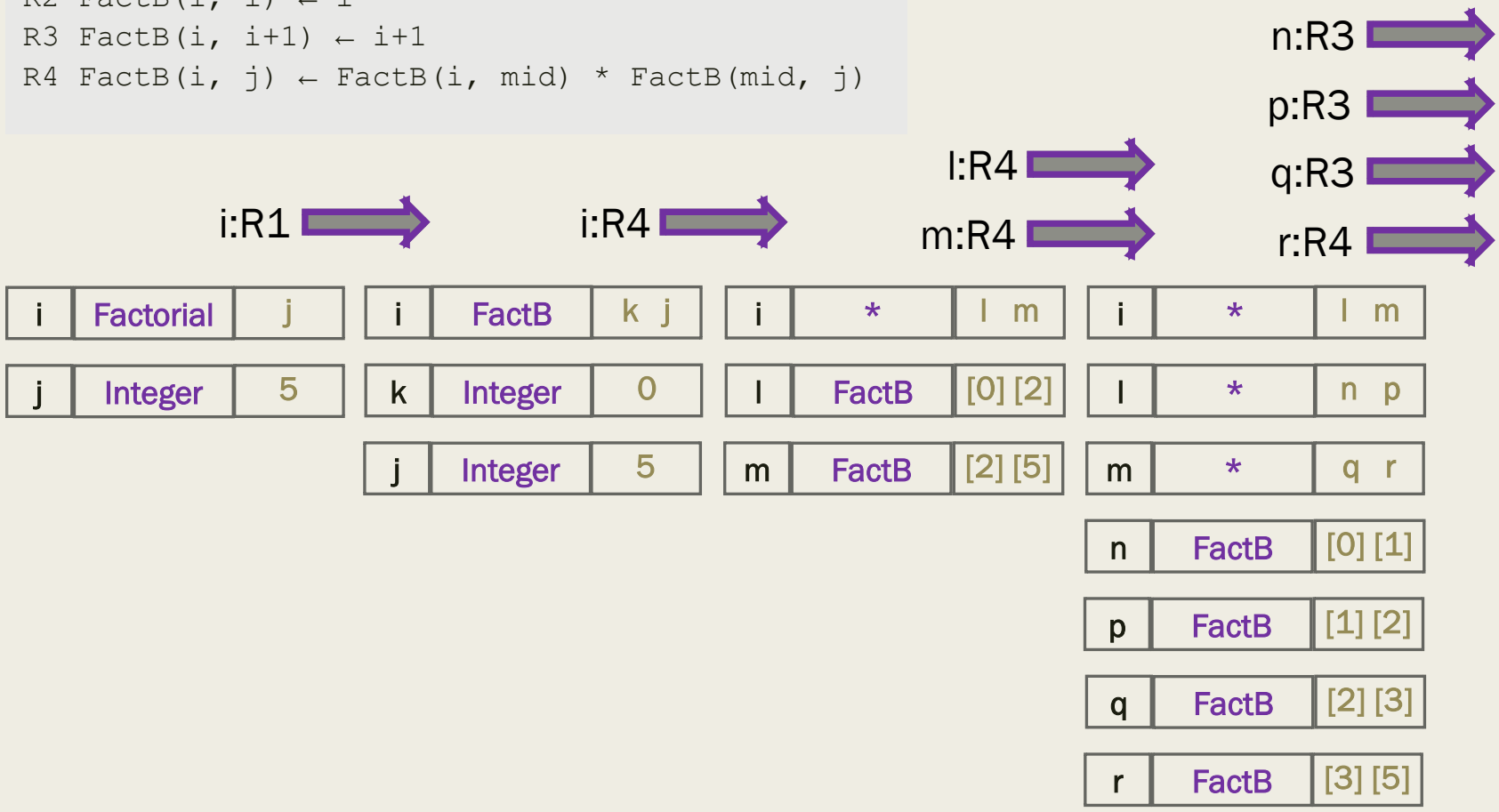
- Funkce je realizována **pakety**



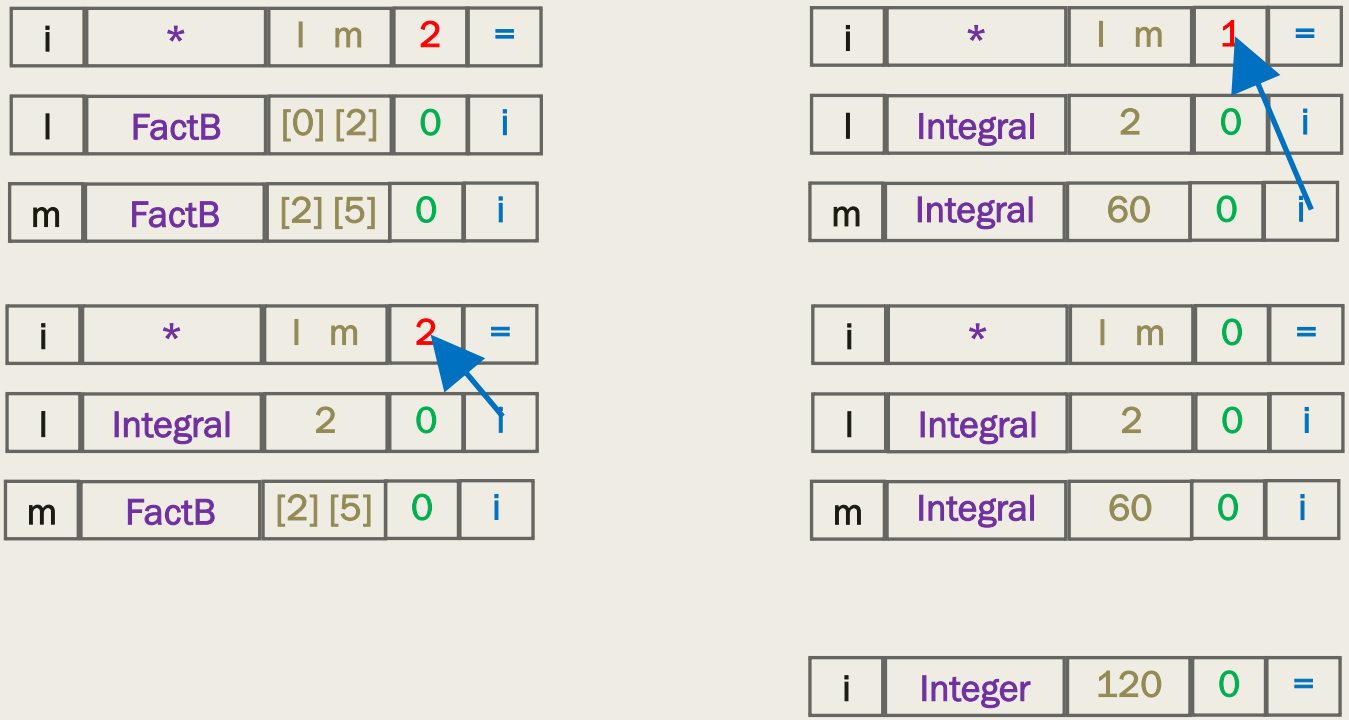
- Funkce je redukovatelná, pokud má k dispozici všechny argumenty
- Pokud nemá, musí čekat, dokud neobdrží signály, že jsou již k dispozici (v počtu referencí)
- Pokud je redukovaná, je k dispozici výsledná hodnota – signály všem, kteří ji čekají (v seznam signálů)

Redukční počítače, ALICE

```
R1 Factorial(n) ← FactB(0, n)
R2 FactB(i, i) ← i
R3 FactB(i, i+1) ← i+1
R4 FactB(i, j) ← FactB(i, mid) * FactB(mid, j)
```



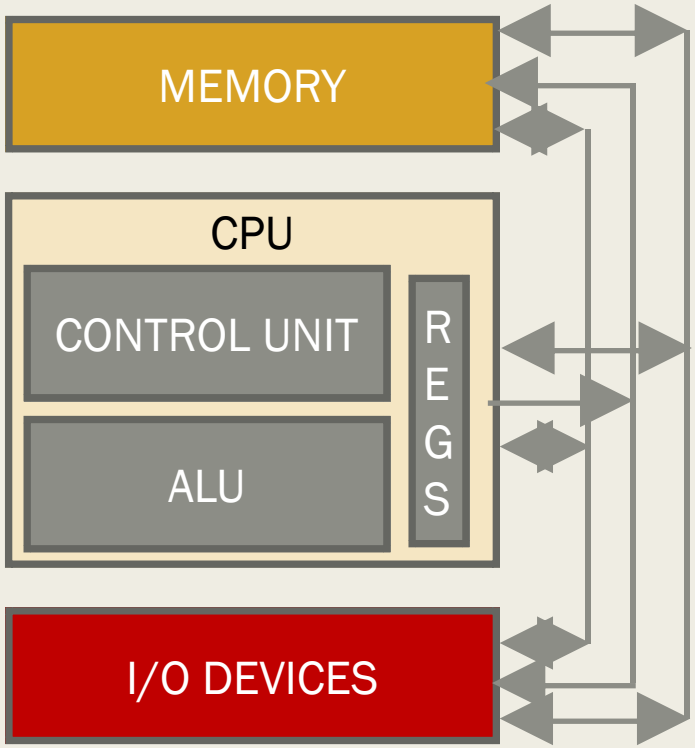
Redukční počítače, ALICE
Čekání na operandy



VON NEUMANOVSKÉ ARCHITEKTURY

VLIW, VEKTOROVÉ, ZŘETĚZENÉ
MIMD, PROPOJOVACÍ SÍTĚ

Von Neumannovská architektura



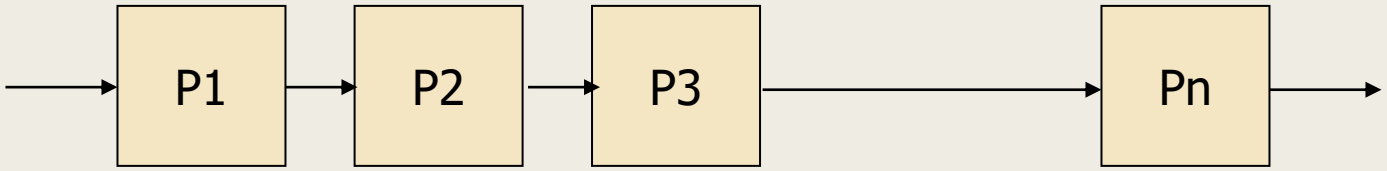
CPU – Central Processing Unit

- Řídící jednotka: řídí fungování procesoru, obsahuje PC a IR, obstarává signály
- ALU – Aritmeticko logická jednotka, provádí operace profesoru.
- Registry – akumulátor, dále například datové či adresové registry, registr příznaků…
- Sběrnice – adresová, datová, řídící

ZŘETĚZENÉ PROCESORY

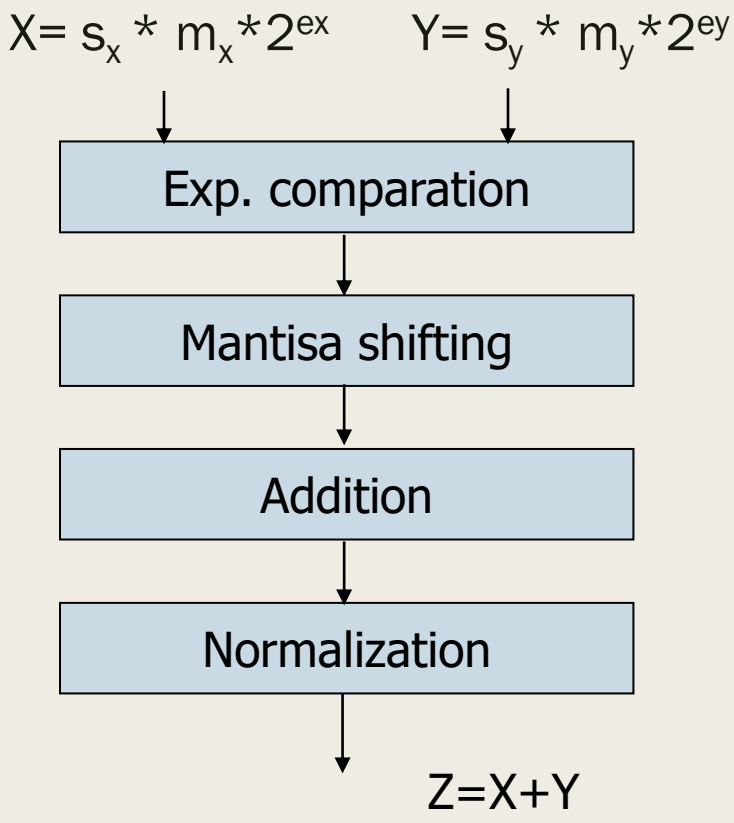
Zřetězené procesory

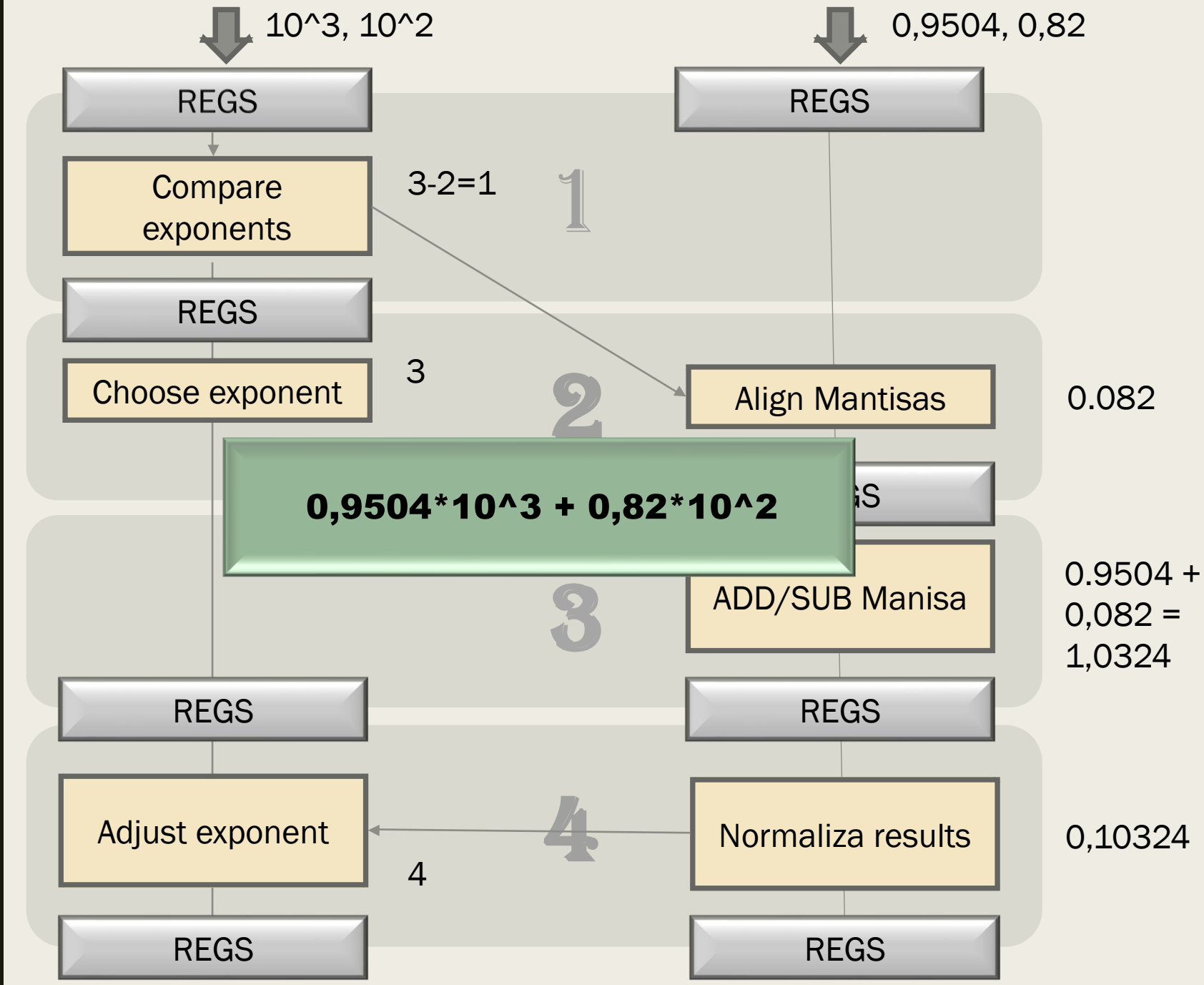
- Lineárně propojené procesory
- Řešení úloh s proudovým charakterem
- Data procházejí postupně jednotlivými procesory
- Aritmetické zřetězení / Zřetězení instrukcí



Zřetězené procesory, aritmetické zřetězení

Sčítání reálných čísel




10^3, 10^2
0,9504, 0,82
REGS
REGS
Compare exponents
3-2=1
1
REGS
Choose exponent
3
2
Align Mantisas
0.082
0,9504*10^3 + 0,82*10^2
GS
ADD/SUB Manisa
3
0.9504 + 0,082 = 1,0324
REGS
REGS
Adjust exponent
4
4
Normaliza results
0,10324
REGS
REGS

RISC, pětifázové zpracování

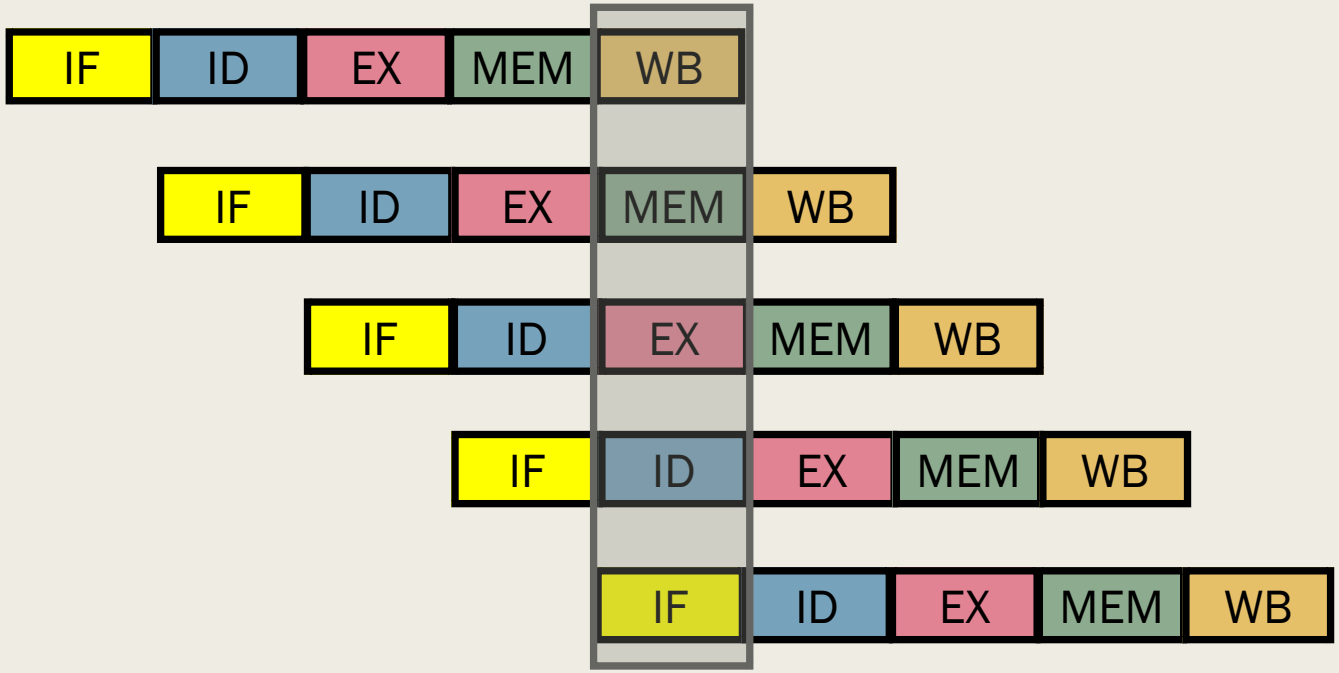
IF: Načtení instrukce
ID: Dekódování instrukce
- *Specifické pro RISC,*
- *Určení operandů, načítání z lok. paměti / registrů, adresy cílů skoků*

EX: Vykonání instrukce
MEM: Alokace dat v paměti, pokud je třeba
- *jednocyklové instrukce -> pouze posílají výsledky dále*
- *dvoucyklové instrukce (R1 [adresa]) načítají data z paměti*

WB: Zápis do lok. paměti / registrů, další např. řídící akce

Pipeline

Zřetězené procesory, RISC

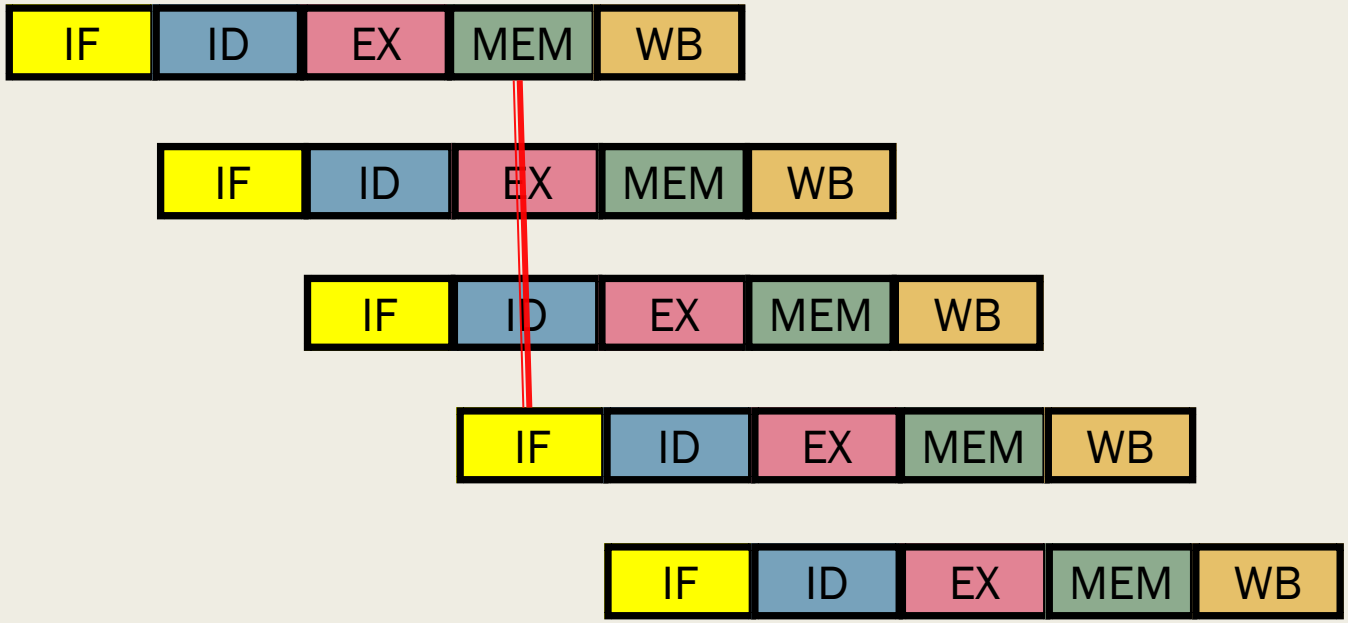


Zřetězené procesory, problémy

- Strukturální hazardy
  - *Instrukce v zřetězené lince požadují stejný prostředek (přístup do paměti)*
- Datové hazardy
  - *Výsledek předchozí instrukce v zřetězené lince je očekáván jinou – následující instrukcí*
- Řídící hazardy
  - *Podmíněné skoky*

Strukturální hazardy

Současný přístup ke zdroji (paměti, sběrnici) na různých stupních zřetězení



Řešení: pozdržení vykonávání
oddělení instrukční a datové paměti

Datové hazardy, typy

- RAW (Read after Write)

```
ADD R1, T1, T2
SUB T3, R1, T4
```

  write SUB musí být zapsáno až po zapsání ADD

- WAR (Write after Read)

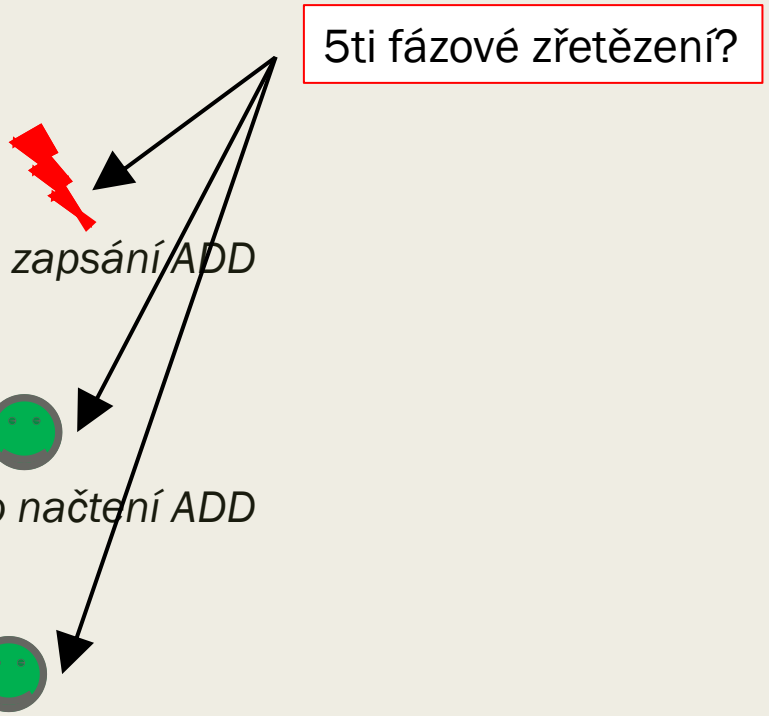
```
ADD T1, R1, T2
SUB R1, T3, T4
```

  write SUB musí být zapsáno až po načtení ADD

- WAW (Write after Write)

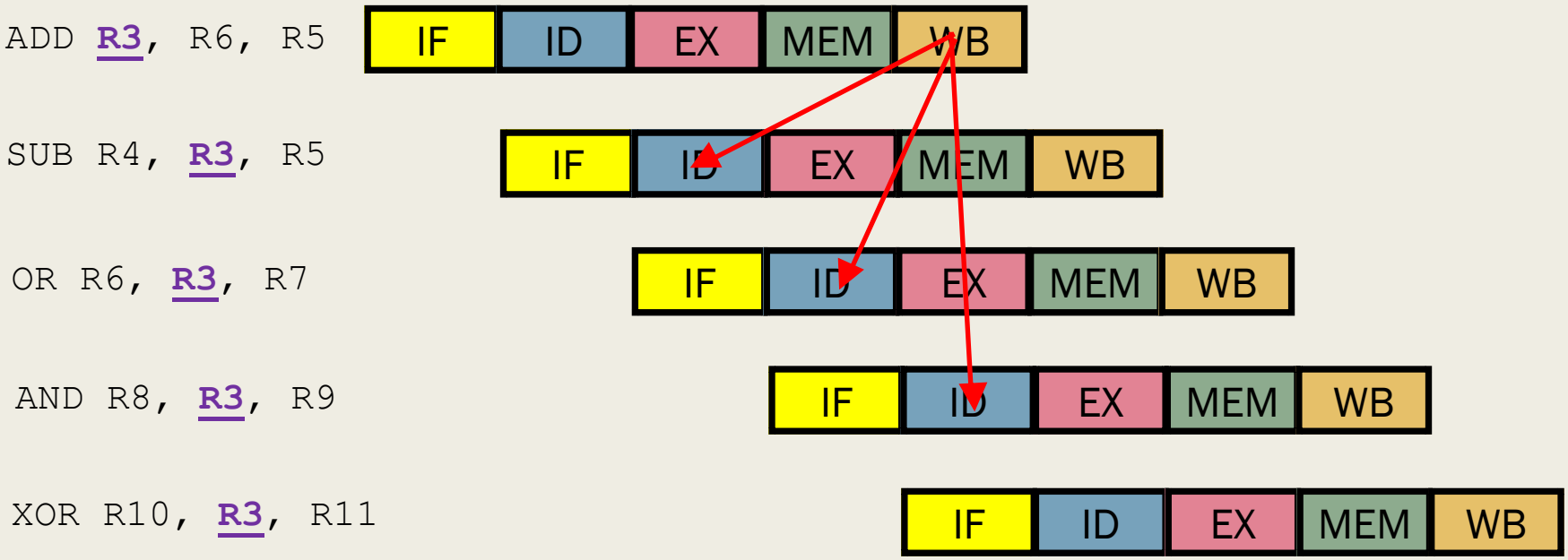
```
ADD R1, T1, T2
SUB R1, T3, T4
```

  write SUB musí být načteno až po zapsání ADD



Datové hazardy, řešení

Přeposíláním



Pozastavením (stall)

Snižuje propustnost linky

Překladačem

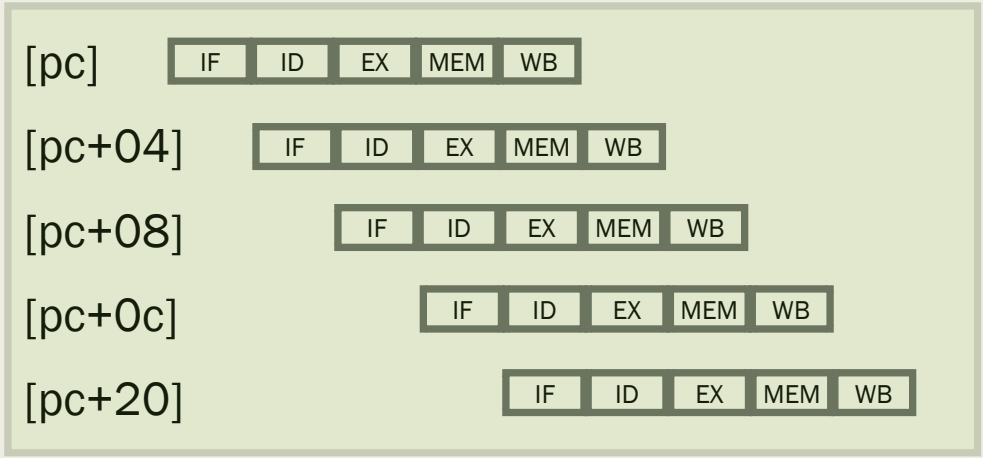
Překladač zamění provádění instrukcí tak, aby data byla k dispozici včas

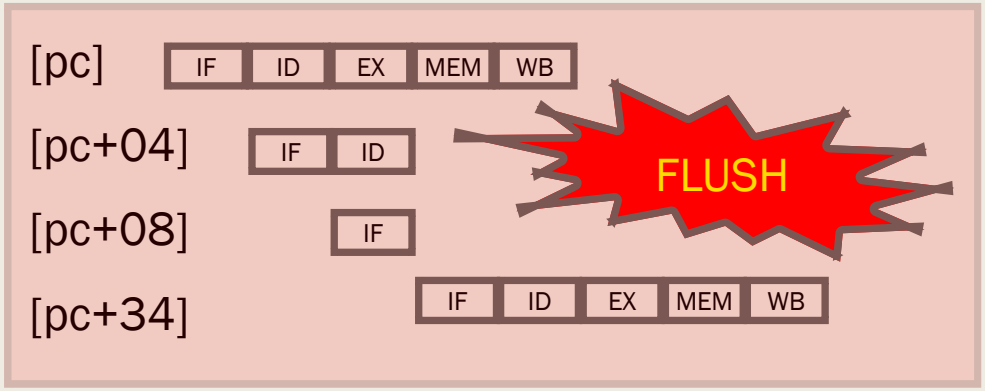
Překladač vloží instrukci **nop**, snižuje propustnost linky

Řídící hazardy

- Pokud se zpracovává podmíněný skok, následné instrukce nemají být vykonány
- Vyprázdnění (flush)

  [pc]: beq t1,t2,30 …

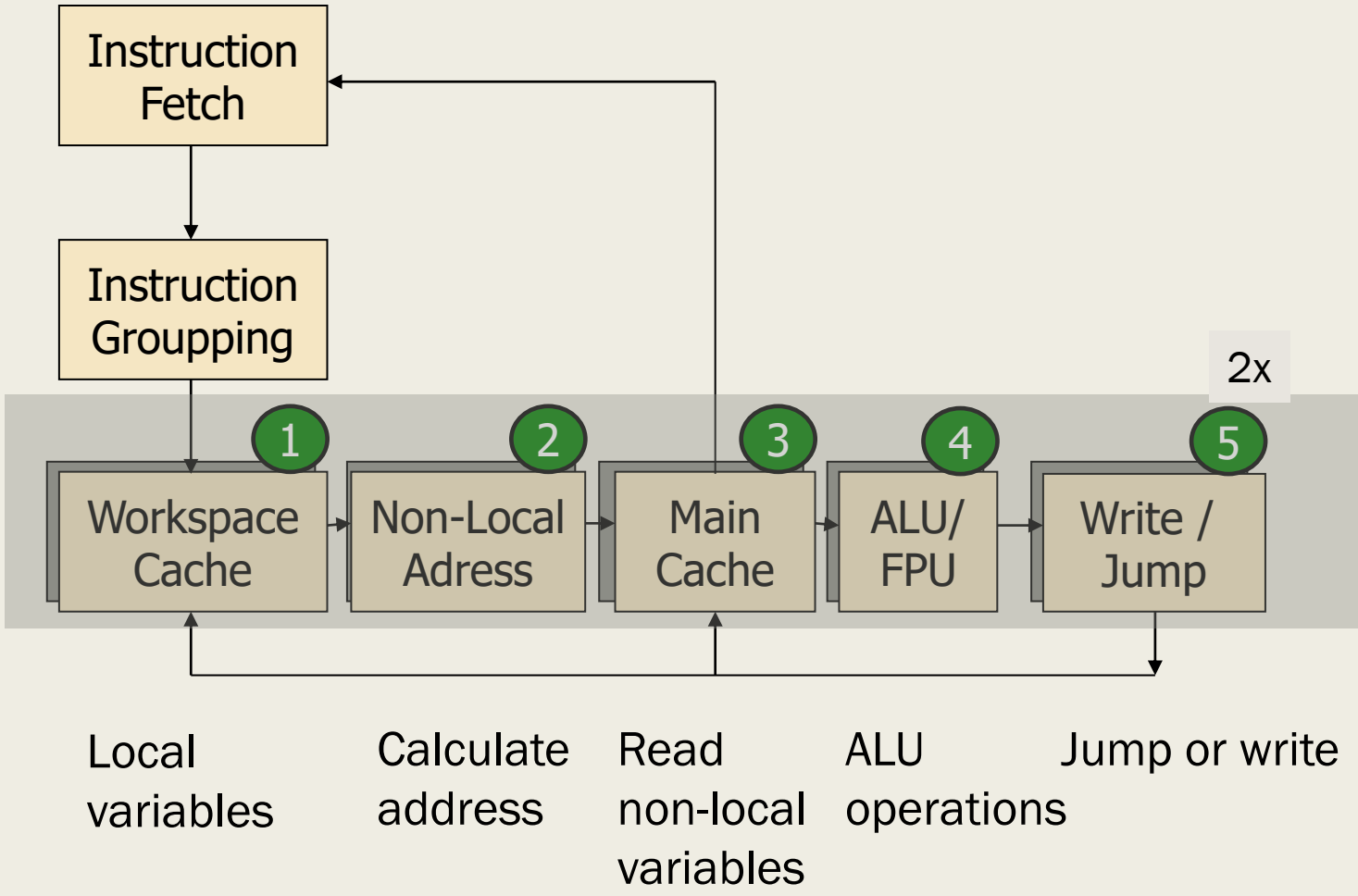




Řídící hazardy, řešení

- Odložení vykonání následujících instrukcí (stall)
- Předvídání, dynamické předvídání
  - *Vytváření a uchovávání záznamů o provedení – neprovedení skoku pří vykonání instrukce*
- Zaměněním pořadí instrukcí
  - *Instrukce následně vykonané neovlivní provádění programu*

Zřetězené procesory TRANSPUTER T9000



Zřetězené procesory

a[i+1]=b[j+15]+c[k+7]

ldl j	1
ldl b	1
wsub	2
ldnl 15	2,3
---	---
ldl k	1
ldl c	1
wsub	2
ldnl 7	2,3
add	4
---	---
ldl i	1
ldl a	1
wsub	2
stnl 1	2,5

INSTRUKCE

Integer registry

Areg, Breg, Creg

Instrukce

ldl n (load local)
Areg´ ← word[Wptr @ n]
Breg´ ← **Areg**
Creg´ ← **Breg**

ldnl n (load non local)
Areg´ ← word[**Areg** @ n]

stnl n (store non local)
word´[**Areg** @ n] ← **Breg**
Areg´ ← **Creg**

wsub (word subscript)
Areg´ ← **Areg** +word[**Breg**]
Breg´ ← **Creg**
Creg´ ← undefined

add
Areg´ ← **Breg** + **Areg**
Breg´ ← **Creg**
Creg´ ← undefined

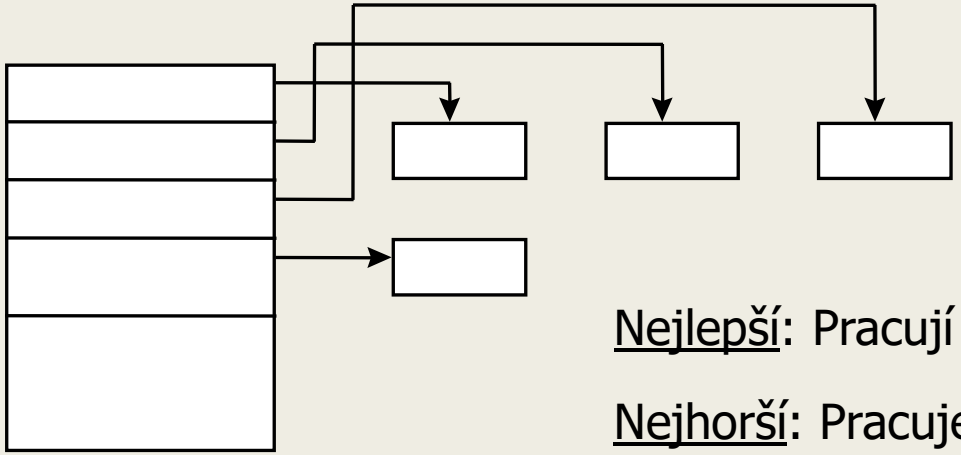
VLIW

VLIW - Very Long Instruction Word

- jediný tok řízení, který řídí všechny procesory
- každá instrukce - samostatné pole s operačním kódem pro všechny procesory
  - *jediný PC, který prochází program*
  - *instrukce jsou prováděny sekvenčně*
  - *instrukce také určují, jak probíhá komunikace mezi procesory. Jsou v nich uloženy informace o odesílateli, příjemci a velikosti dat. Tyto informace s ohledem na časování generuje kompilátor*

VLIW (Very Long Instruction Word)

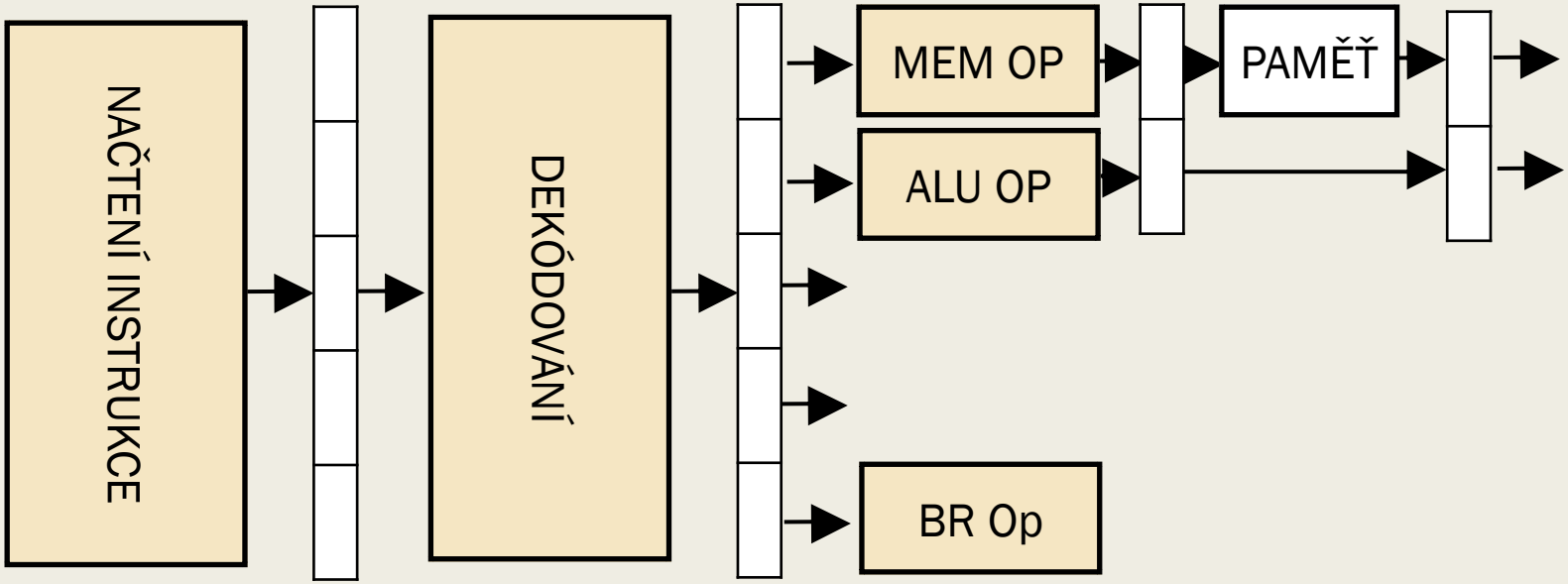
- **Překladač** organizuje nezávislé instrukce do jednoho dlouhého instrukčního slova
  - *Zvyšuje nárok na překladač, ale snižuje ‚složitost' HW*
- Například ELI-512 (512 bitů)

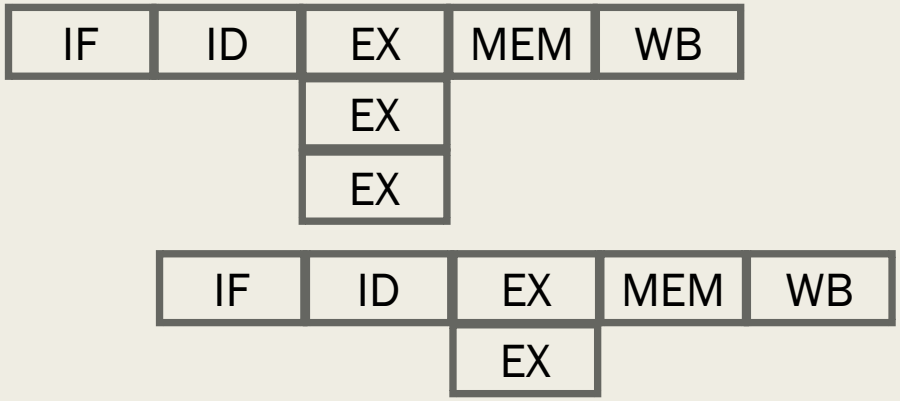


Nejlepší: Pracují všechny procesory

Nejhorší: Pracuje jeden procesor

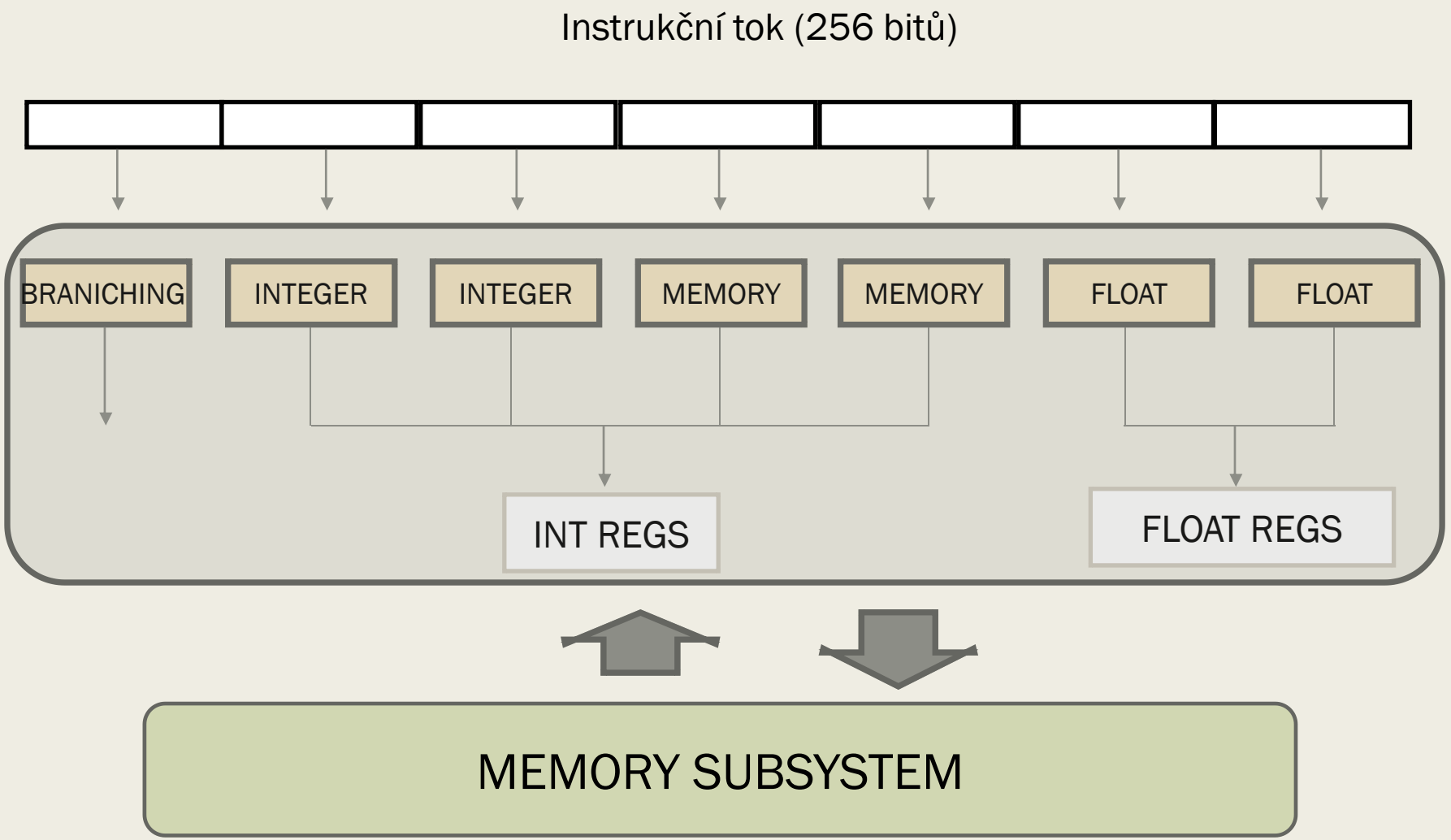
VLIW - Very Long Instruction Word





...

TRACE 7/300

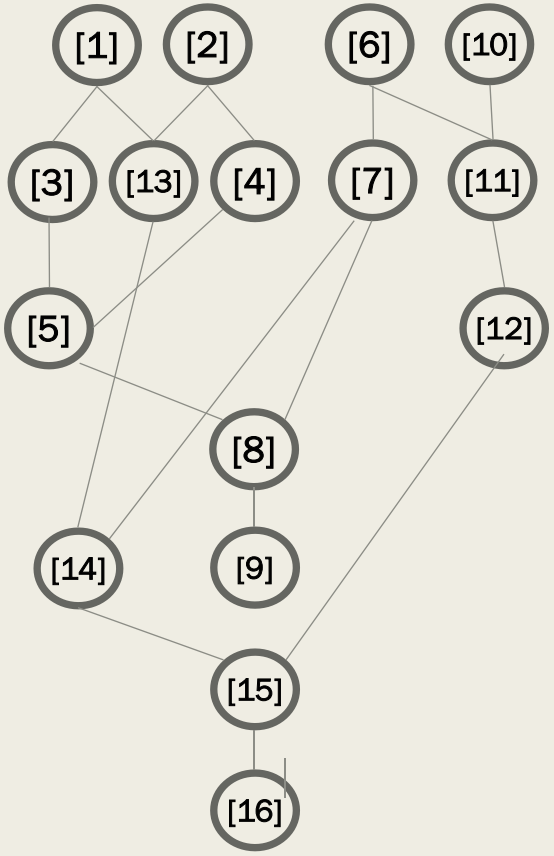

Instrukční tok (256 bitů)
BRANICHING
INTEGER
INTEGER
MEMORY
MEMORY
FLOAT
FLOAT
INT REGS
FLOAT REGS
MEMORY SUBSYSTEM

Překlad pro VLIW

C=(2A+3B)*2I
Q=(C+A+B)-4(I+J)

[1] LD A
[2] LD B
[3] t1=A*2
[4] t2=B*3
[5] t3=t1+t2
[6] LD I
[7] t4=2*I
[8] C=t4*t3
[9] ST C
[10] LD J
[11] t5= I+J
[12] t6= 4+t5
[13] t7=A+B
[14] t8=C+t7
[15] Q=t8-t6
[16] ST Q

GRAF ZÁVISLOSTÍ



DLOUHÉ INSTRUKCE

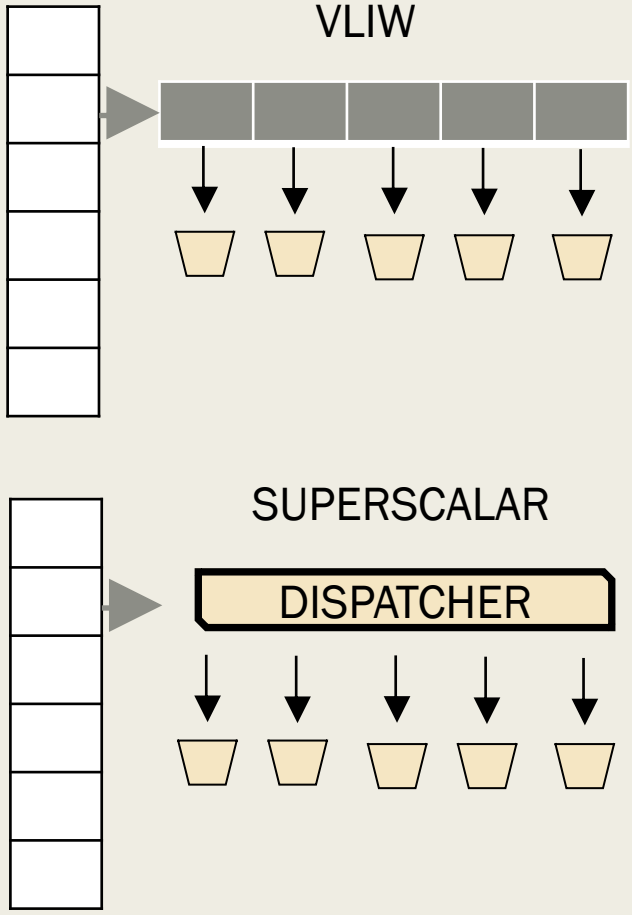
BRA	IN0	IN1	ME0	ME1	FL0	FL1
			LD A	LD B		
			LD I	LD J		
					2*A	3*B
	2*I	I+J			t1+t2	A+B
	4*t5				t4*t3	C+t7
			ST C		t8-t6	
			ST Q			

VLIW - Vlastnosti

- Výhody
  - Jednoduchý hardware
  - Dobře škálovatelné
- • Nevýhody
  - Podmíněné skoky (když jedna instrukce provede skok, ostatní instrukce mají problém)
  - Problém toku dat (instrukce zpracovávané v jednom kroku nemohou navzájem používat své výsledky)
    - Některé závislosti nejsou staticky rozpoznatelné v době překladu (např. aliasing ukazatelů, oddělený překlad, dynamické sestavování) => překladač musí být konzervativní a výsledná výkonnost je nižší než při schopnosti rozpoznat všechny závislosti
    - Některé latence nejsou odhadnutelné v době překladu (např. latence instrukcí LOAD a STORE kvůli případnému Hit nebo Miss v cache)
    - některé VLIW používají softwarově řízenou lokální paměť místo datové cache; jinde překladač předpokládá 100% Hit-Rate
  - Velikost programu - synchronizační NOPy jsou u VLIW navíc
  - VLIW nejsou softwarově zpětně kompatibilní jako superskalární procesory (i když existují cesty, jak toho dosáhnout)

VLIW vs superskalární procesory

- Statické superskalární procesory (Static, In-Order Superscalar)
  - *Zpracovávají více instrukcí paralelně ale v programovém pořadí*
  - *Typická šířka 3-4 instrukce*
  - *Paralelní provádění může zahájit pouze limitovaná kombinace typů instrukcí („párovatelné instrukce“)*
- VLIW procesory
  - *Instrukce obsahuje více paralelních operací*
  - *Konflikty detekuje a řeší překladač (téměř výlučně)*
  - *Šířka instrukcí typicky 6-10 paralelních operací*
- Dynamické superskalární procesory (Dynamic, Out-Of-Order Superscalar)
  - *Zpracovává více instrukcí paralelně i mimo programové pořadí*
  - *Dnes typicky podporuje spekulativní provádění instrukcí za skokem a někdy i spekulativní provádění Load/Store instrukcí*



SIMD

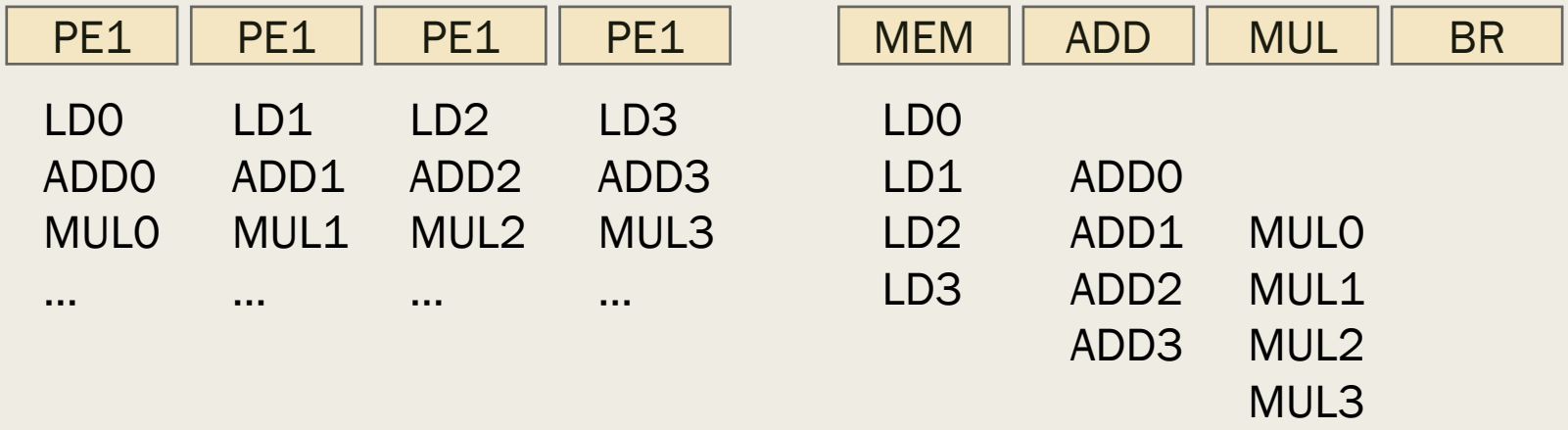
VEKTOROVÉ PROCESORY

SIMD

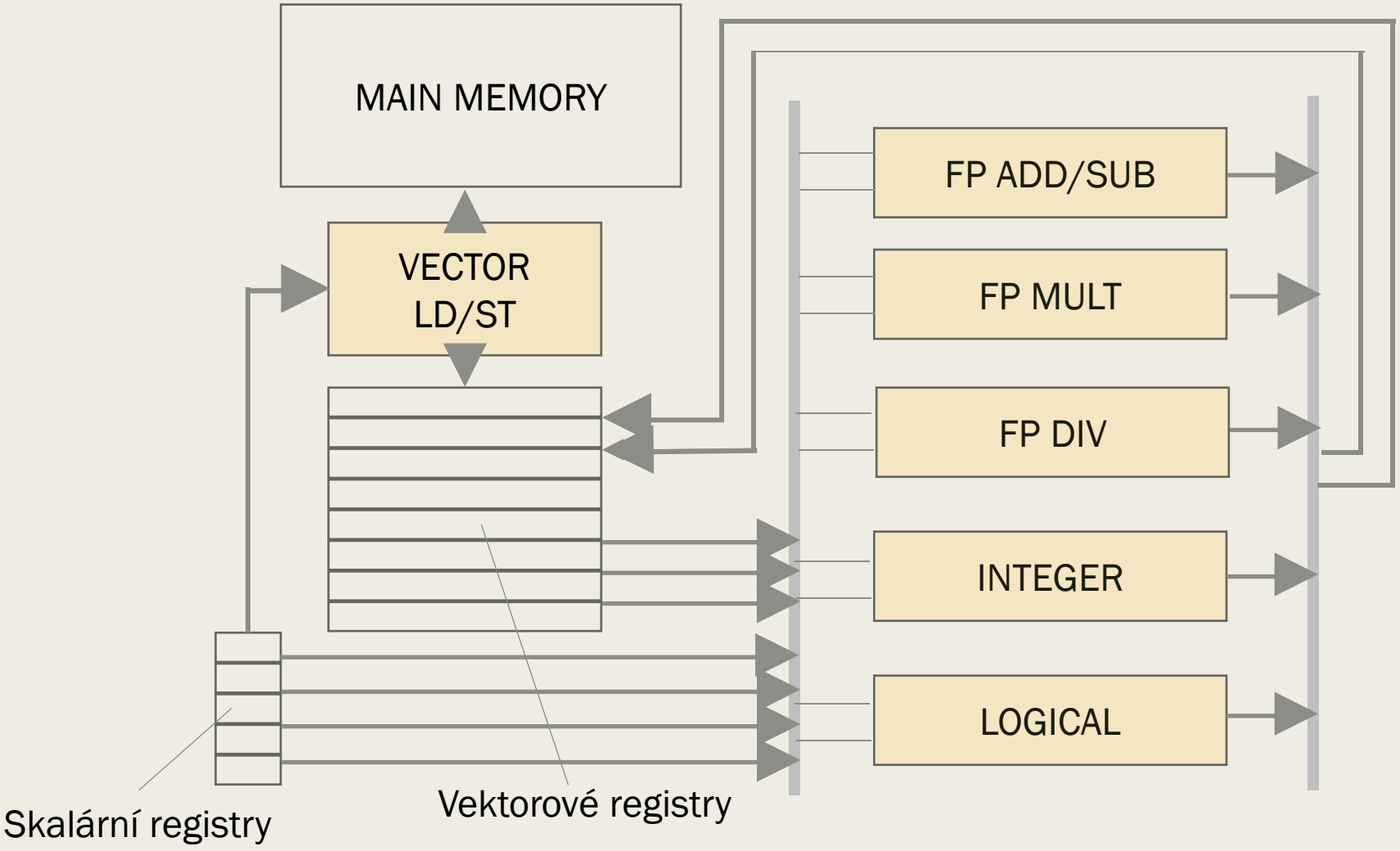
- **Vektorové procesory** (ale nejen ony, array computers)
- Obsahuje instrukce pro práci s poli – vektory dat
- 1976: Cray-1 Superpočítač (Seymour Cray)
- Použití dříve u GPU, dnes již běžná součást architektur

Vector / array computers

- **Array computers (mřížkové počítače)**
  - *Obsahuje jednotky (**processing elements**) zvládající ALU, a Floating point + registry*
  - *Pracují paralelně a synchronizovaně*
- **Vector computers (vektorové počítače)**
  - *Jednotky jsou jednodušší, specializované*
  - *Pracuji zřetězeně (deep pipelining, pipelined SIMD).*



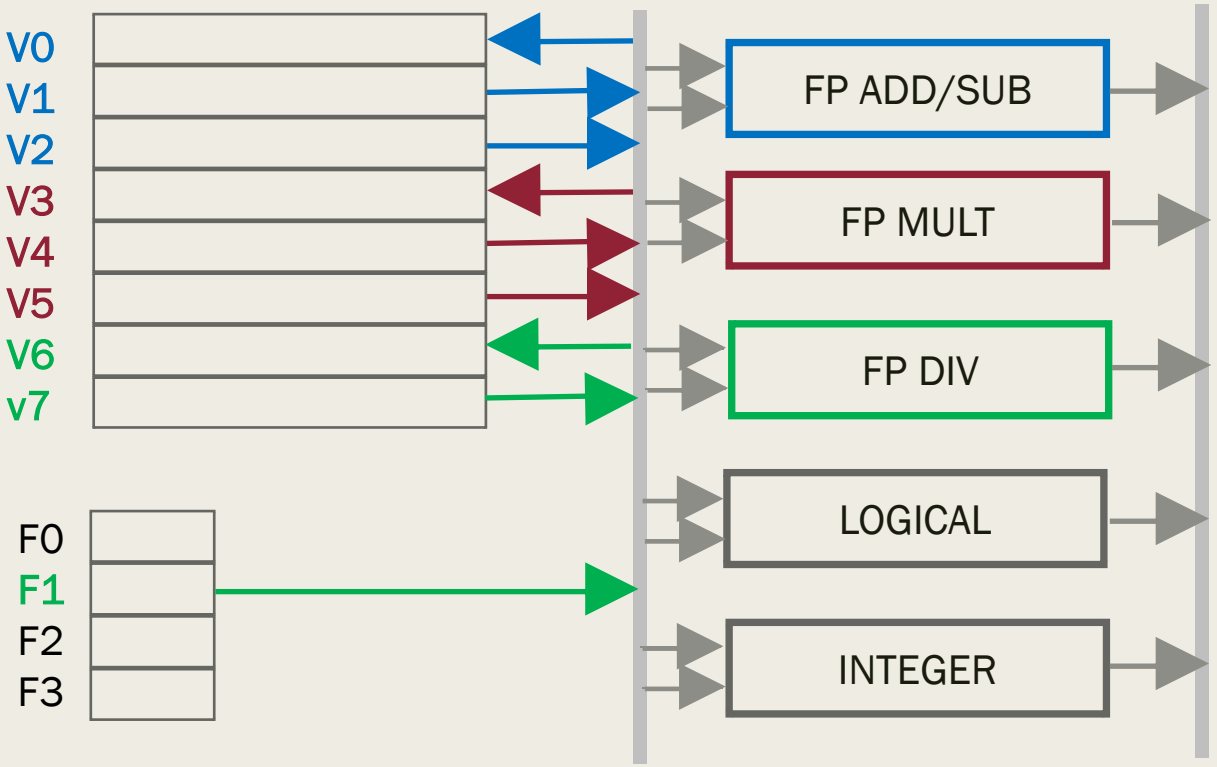
Vektorový procesor (VMIPS)



Vektorový procesor (VMIPS)

```
ADDVV.D V0,V1,V2
MULVV.D V3,V4,V5
DIVVS.D V6,V7,F1
```

Instrukční sada



Příklad: výpočet *Y = a * X + Y*

Sekvenční

```
LD F0, a
ADDI R4, Rx, #512 ; last address to load
Loop:
LD F2, 0(Rx) ; load X(i)
MULTD F2, F0, F2 ; a * X(i)
LD F4, 0 (Ry) ; load Y(i)
ADDD F4, F2, F4 ; a x X(i) + Y(i)
SD F4, 0 (Ry) ; store into Y(i)
ADDI Rx, Rx, #8 ; increment index to X
ADDI Ry, Ry, #8 ; increment index to Y
SUB R20, R4, Rx ; compute bound
BNZ R20, Loop ; check if done
```

Vektorové

```
LD F0, a ; load scalar a
LV V1, Rx ; load vector X
MULVS.d V2, F0, V1 ; vector-scalar
                   ;  multiply
LV V3, Ry ; load vector Y
ADDVV.d V4, V2, V3 ; add
SV Ry, V4 ; store the result
```

Načítání vektoru, Stride

C=A x B

A

0	1	2	3	4	5

B

0					
6					
12					
18					
24					
30					

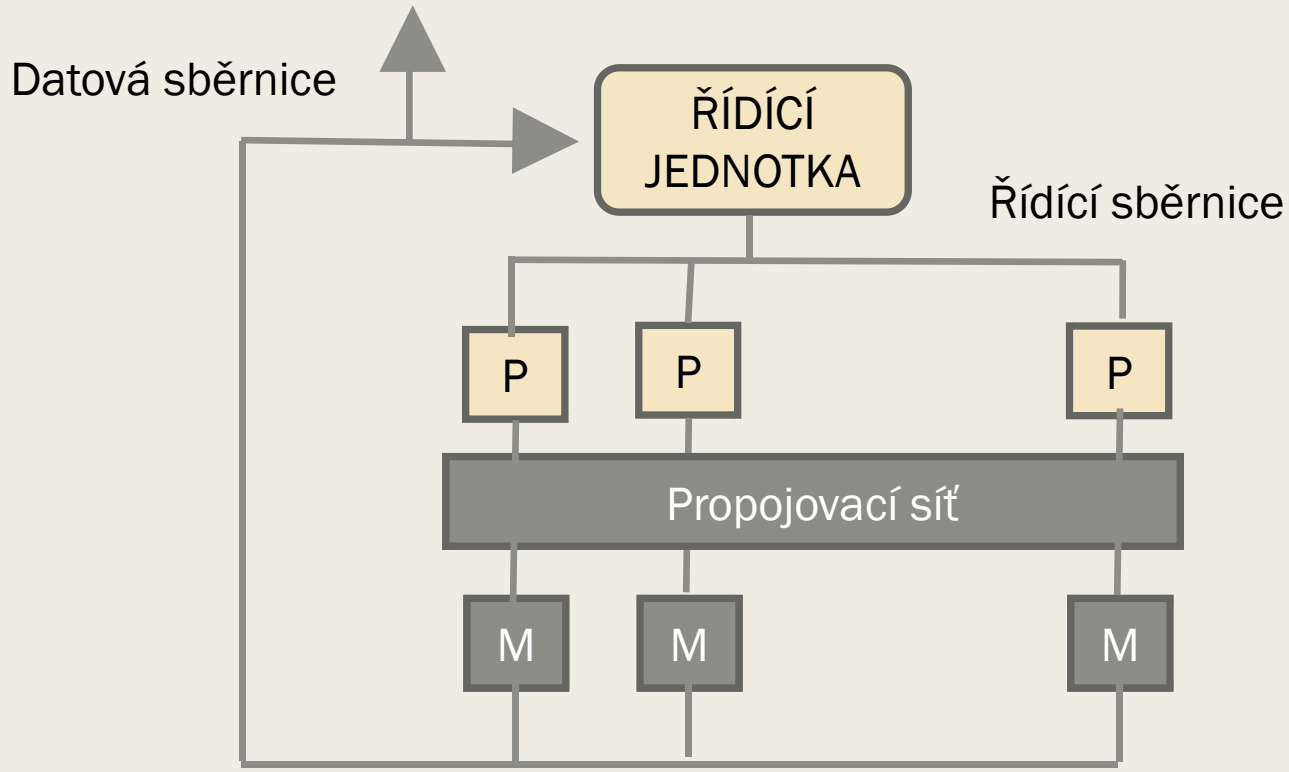
C[0]=A[0]*B[0]+A[1]*B[6]+A[2]*B[12] + … A[5]*B[30]

C[0]=A[0]*B[0]+A[0+1]*B[0+6]+A[1+1]*B[6+6] + … A[4+1]*B[24+6]

```
LV V1, 0(Ra)
LVWS V2, 0(Rb), 6
MULVV V3, V1, V2
```

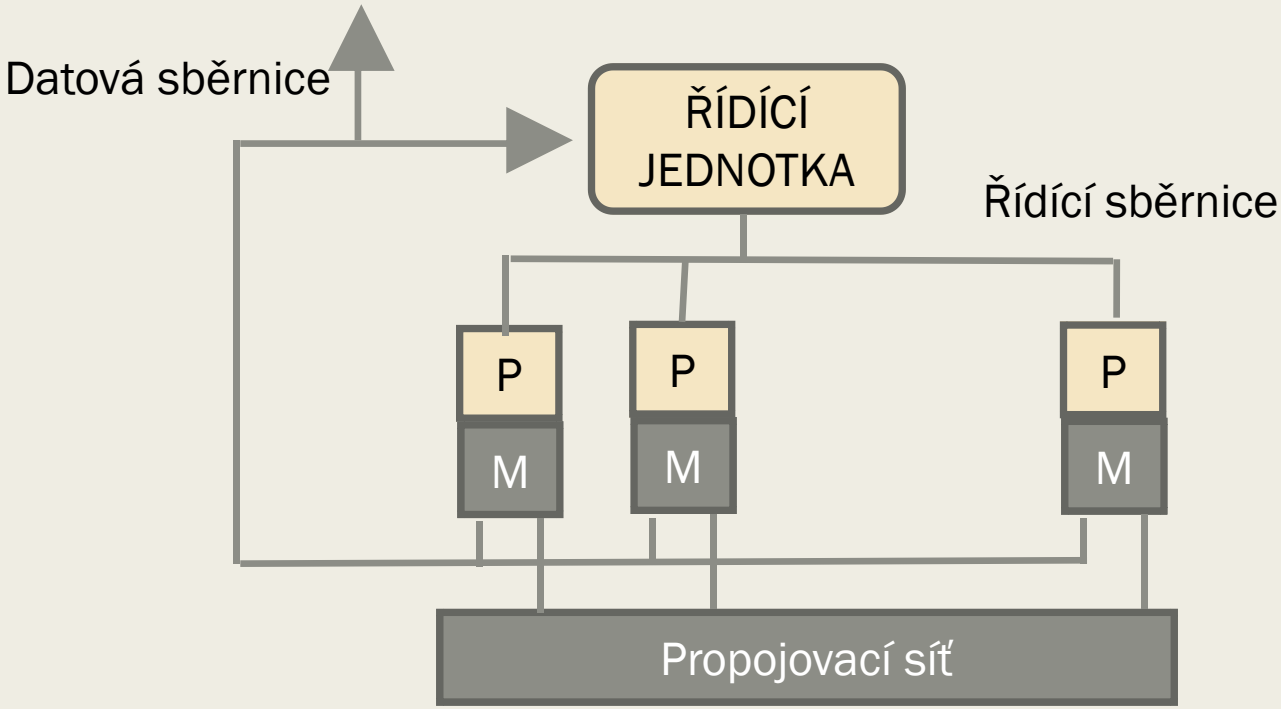
SIMD – Vektorový procesor

UMA –Uniform Memory Access

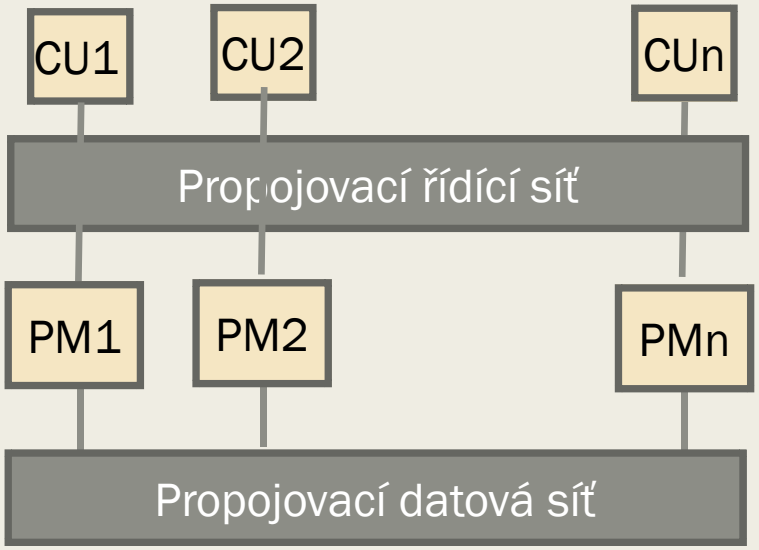


SIMD – Vektorový procesor

NUMA – Non-Uniform Memory Access



MSIMD – Multiple SIMD



- Dělení procesorů na nezávislé skupiny
- Dynamické alokování procesorů

Výhody / nevýhody SIMD

Výhody

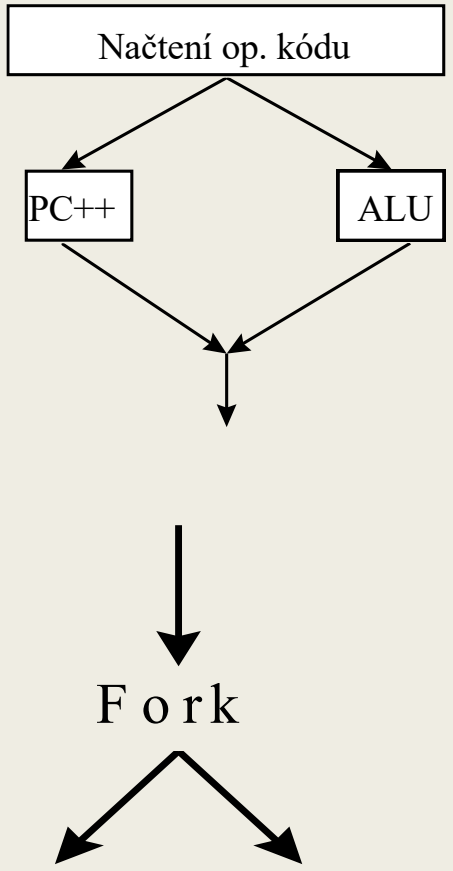
- Jednodušší než MIMD
- Menší nároky na paměť
- Jeden instrukční tok a synchronizace zjednodušuje programy
- Odpadá režie se složitou synchronizací mezi procesy, která je u MIMD
- Rychlejší komunikace mezi procesory než u MIMD
  - *Menší latence*
  - *Menší režie na struktury komunikačních paketů, směrování atd.*

Nevýhody

- Ne všechny problémy jsou datově paralelizovatelné
- Pokles výkonnosti u programů s mnoha podmíněnými skoky
- Nejsou vhodné při malém počtu procesorů (neexistuje něco jako „starter kit“)
- Vyžadují ne úplně běžné procesory

Granularita paralelismu

- uvnitř instrukcí - INTRA INSTRUCTION
  - *nejjemnější paralelismus*
- mezi instrukcemi INTER INSTRUCTION
  - *některé instrukce se provádějí paralelně, ale není to vidět na úrovni programovacího jazyka*
  - *zřetězení u RISC*
  - *více jednotek (T10000)*
- mezi příkazy
  - *vektorové počítače*
  - *specializované koprocesory (FPU)*
- mezi bloky procesu (vlákna, thready)
  - *parbegin*
  - *thread1;*
  - *thread2;*
  - *parend*
- mezi procesy
  - *zpravidla nemají sdílenou paměť*
  - *mohou být odděleně kompilovány*



INTERAKCE V PARALELNÍCH SYSTÉMECH

Petr Hanáček, František Zbořil ml.

– 2024

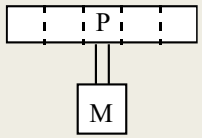
Komunikace

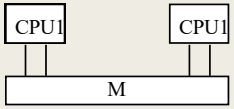
- přenáší data
  - *„Předávání informací mezi procesy (procesory, vlákny, …)“*
- Prostředky pro komunikaci
  - *Sdílená paměť*
    - skutečná x simulovaná
    - boj o sběrnici, cache, lokalita odkazů (busy waiting !)
    - řešení konfliktů při zápisu
    - obtížně použitelná pro synchronizaci
  - *Zasílání zpráv*
    - kanály
      - *synchronní x asynchronní (kapacita)*
      - *jednosměrný x obousměrný (ACK)*
    - volání vzdálených procedur (RPC)
    - všesměrové vysílání (broadcasting)
      - *úmyslné posílání zpráv všem*
      - *vysílání každému procesu*
      - *záplava - na jednu zprávu odpoví procesy jinou b. zprávou*

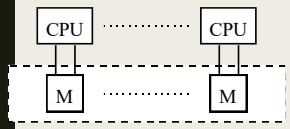
Synchronizace

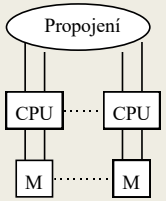
- nepřenáší se data
  - *„Zajištění požadovaných časových vztahů mezi událostmi“*
- Prostředky
  - zasílání zpráv (nejlépe synchronní)
  - randevous (RPC - remote procedure call)
  - semafor
  - monitor
  - bariéra
- Typické synchronizační úlohy
  - *soupeření*
    - vzájemné vyloučení
    - čtenáři x písaři
  - *kooperace*
    - dohoda
    - producent - konzument
- Synchronizace v distribuovaných systémech
  - *Logický čas*
  - *Časová razítka / tokeny*

Sdílená paměť a předávání zpráv u MIMD









	Předávání zpráv	Sdílená
■ Multitasking		
– *1 cpu*		
– *přepínání kontextu*		
– *virtuální procesory*	*simulováno SW*	*Ano*
■ sdílená paměť		
– *Cache*		
– *těsně vázané*		
– *boj o sběrnici*	*simulováno SW, HW*	*Ano*
■ virtuální sdílená paměť		
– *all cache*	*simulováno SW, HW*	*simul. HW*
– *spojení caches komunikačními kanály*		
– *navenek se tváří jako společný (jediný) adresový prostor*		
■ předávání zpráv		
	Ano	*simul. SW*
– *volně vázaná architektura*		
– *počítačové sítě*		
– *propojeny pouze procesory*		

Vlastnosti sdílené paměti / předávání zpráv

- Sdílená paměť
  - *všechny procesy mají přístup do společného paměťového prostoru*
  - *řešení současného přístupu k jedné buňce paměti*
    - Exclusive-read, Exclusive-Write (EREW)
    - Concurrent-Read, Exclusive-Write (CREW)
    - Exclusive-Read, Concurrent-Write (ERCW)
    - Concurrent-Read, Concurrent-Write (CRCW)
- Předávání zpráv
  - *každý procesor má svůj adresový prostor*
  - *procesory mají vlastní paměť - komunikace pomocí zpráv*

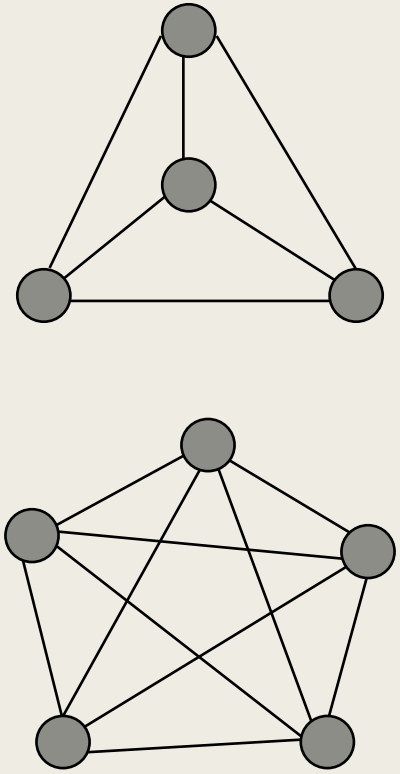
Propojovací sítě

- Použití propojovacích sítí
  - *Propojit procesory se sdílenou pamětí*
  - *Propojit procesory navzájem*
- Typy propojovacích médií
  - *Statické*
  - *Dynamické*
  - *Sdílené (sběrnice)*
  - *Přepínané*
- Vlastnosti propojovací sítě ovlivňují vhodnost jednotlivých typů algoritmů a ovlivňují efektivnost toku dat

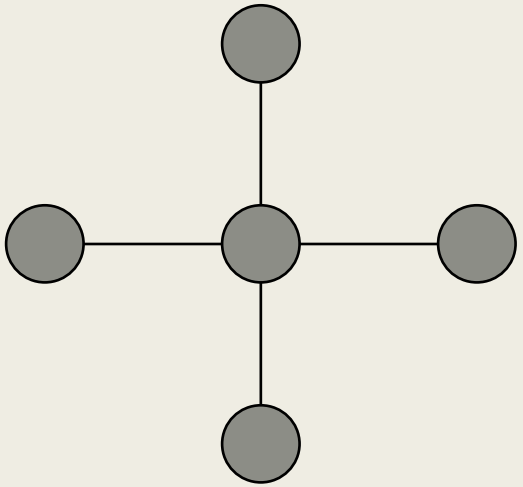
Statické sítě

- Všechny uzly jsou procesory
- Kanály jsou spojnice mezi uzly (procesory)
- Používají se pro architektury bez sdílené paměti
- Vlastnosti
  - ***Průměr*** *(Diameter): Délka nejdelší z nejkratších cest mezi všemi dvojicemi uzlů*
  - ***Šířka bisekce*** *(Bisection width): Minimální počet hran, které spojují dvě přibližně stejně velké poloviny sítě*
  - ***Konektivita*** *(Arc connectivity): Minimální počet hran, které je nutné odstranit pro rozdělení sítě na více částí, také stupeň uzlu*
  - ***Velikost*** *(Network size): Počet uzlů v síti*
  - ***Cena*** *(Cost): Počet hran v síti*

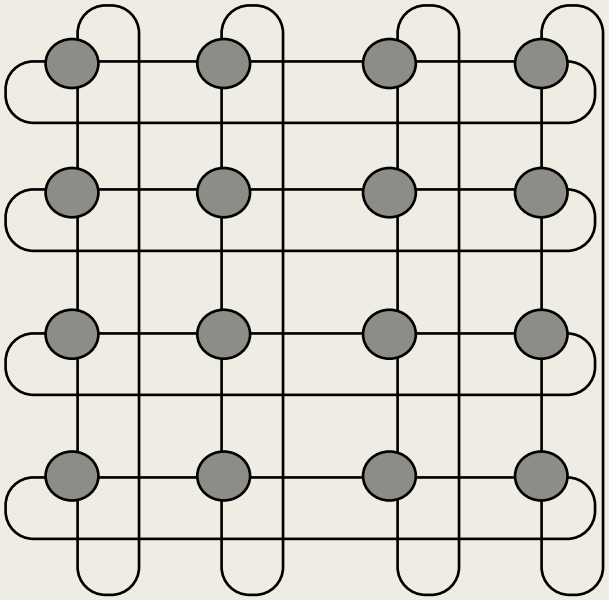
- Úplné propojení
  - *Diametr = 1*
  - *Konektivita = p-1*
  - *Šířka bisekce = $p^2/4$*



- Hvězda
  - *Diametr = 2*
  - *Konektivita = 1*
  - *Šířka bisekce = (p-1)/2*



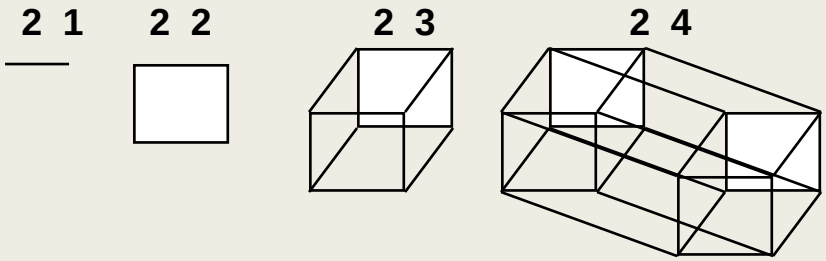
k-ární *n*-rozměrná kostka



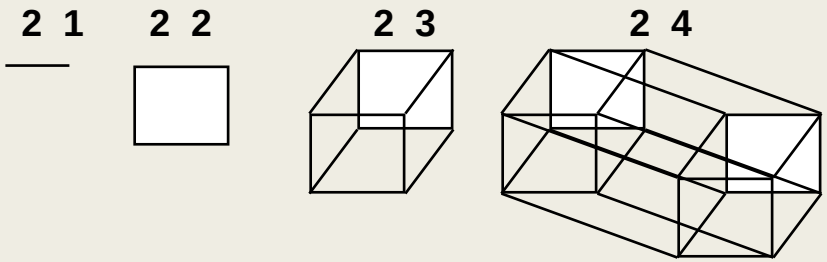
Kartézký součin n lineárních polí s *k* uzly

Počet uzlů k^n

Například: **Hyperkostka**, kružnice, torus,...



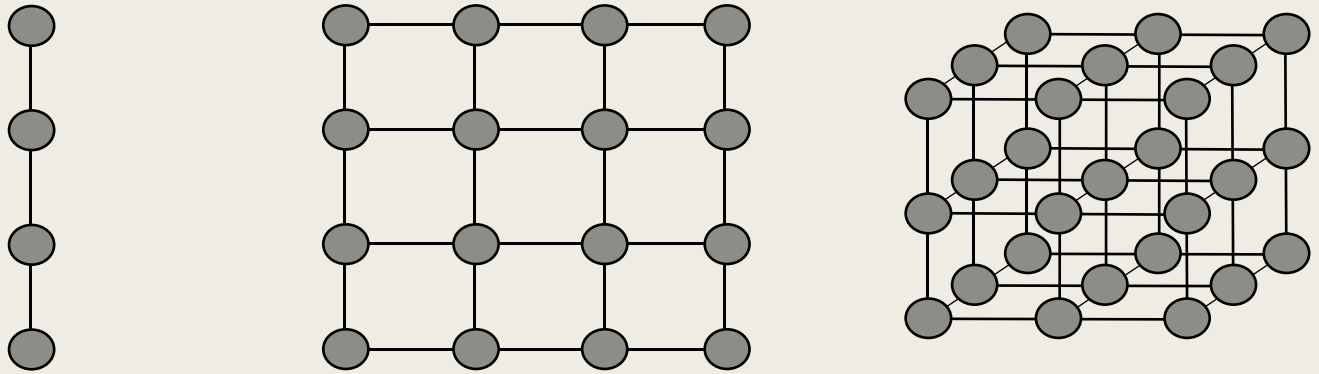
n-rozměná kostka (k=2)



Populární v starších počítačích s předáváním zpráv
(intel iPSC, NCUBE)

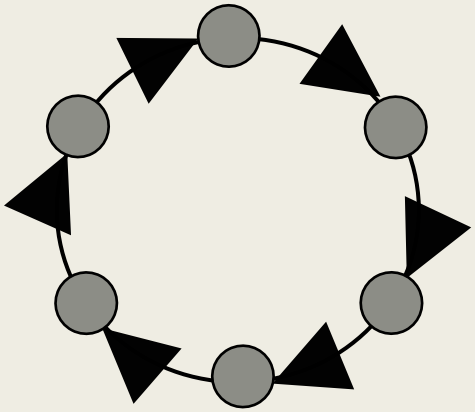
- Počet uzlů $|N| = 2^n$
- Diametr = n
- Konektivita = n
- Šířka bisekce = 2^{n-1}

k-árni *n*-rozměná kostka (mřížka)

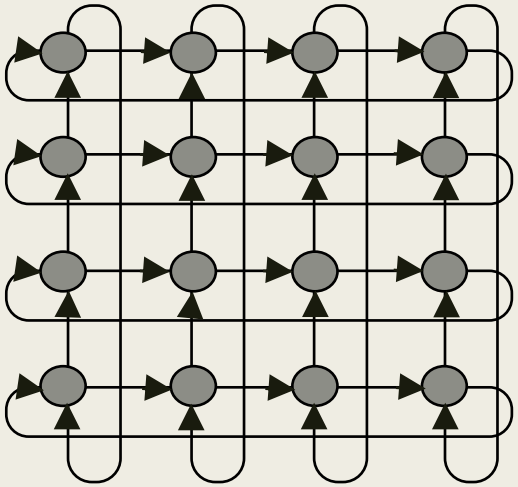


- Počet uzlů $|N| = k^n$
- Diametr = n(k-1)
- Konektivita = $2n$
- Šířka bisekce = k^{n-1}

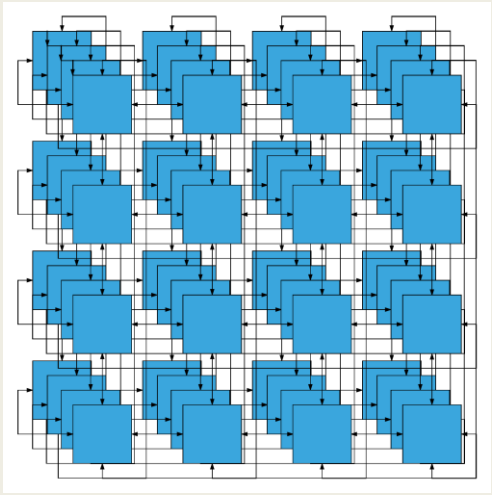
k-árni *n*-rozměná kostka (torus)



n=1 jednosměrný



n=2, jednosměrný



n=3, obousměrný

JEDNOSMĚRNÝ

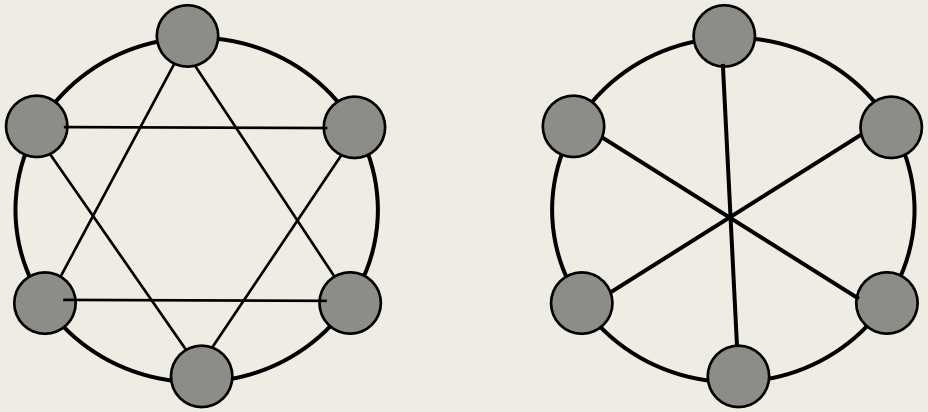
- Počet uzlů $|N| = k^n$
- Diametr = n(k-1)
- Konektivita = $2n$
- Šířka bisekce = k^{n-1}

OBOUSMĚRNÝ

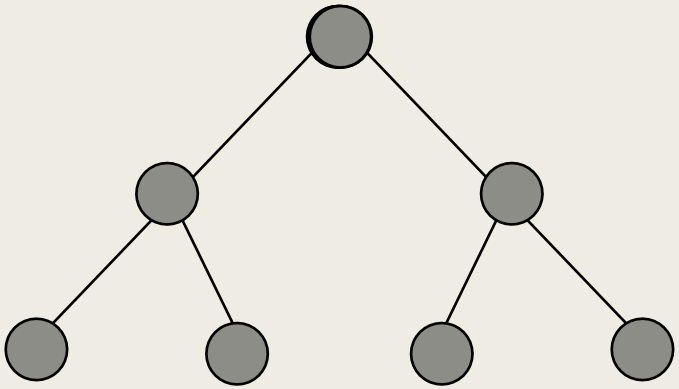
- Počet uzlů $|N| = k^n$
- Diametr = n[k/2]
- Konektivita = $4n$
- Šířka bisekce = $2k^{n-1}$

Ring, Chordal ring

- Ring o *k* uzlech je *k*-ární 1rozměrný Torus
- Chordal ring (ring + propojení uzlů ve vzdálenosti ‚*l*‘, případně ve vzdálenostech z množiny $L = \{l_1, l_2 \dots l_k\}$



d-arní strom



Strom, kde žádný uzel nemá více než **d** potomků

Obvykle binární (d=2)

- Diametr = 2h (pro hloubku stromu h)

p pro jakýkoliv strom

$2*(\log_d p)$ pro vyvážené stromy

- Konektivita = 1
- Šířka bisekce = 1

Dynamické sítě

- Prvky jsou buď procesory, paměťové buňky, nebo přepínače
- Často se používají pro implementaci architektur se sdílenou pamětí
- Sběrnice, křížové přepínače (crossbar), dynamické sítě, fat tree
- Příklad: **sběrnice** (bus)
  - *Cena* $\Theta(p)$*, propustnost* $\Theta(1)$
- Příklad: **křížový přepínač** (crossbar)
  - *Cena* $\Omega(p^2)$*, propustnost* $O(p)$*, neblokující*
  - *Diameter = 1, Konektivita = 1, Bisekce = p*
- Neblokující
  - *Lze spojit vždy libovolné dva uzly, které jsou k dispozici*
- Přenastavitelné
  - *Lze změnit trasu propojení pro vyřešení konfliktu*
- Blokující
  - *Nelze současně propojit libovolné dva uzly*

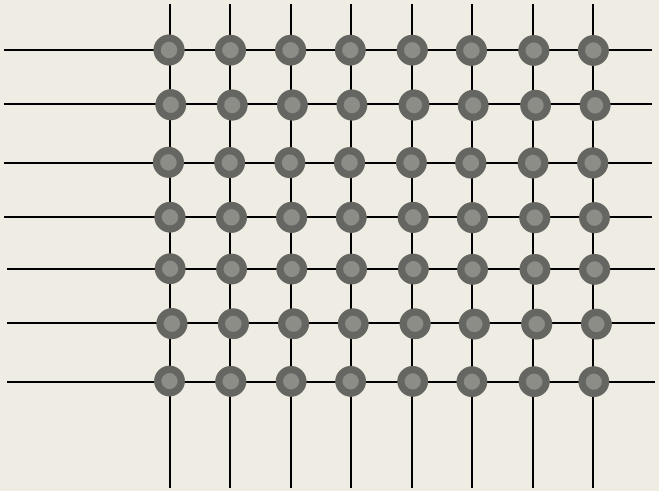
Křížový přepínač

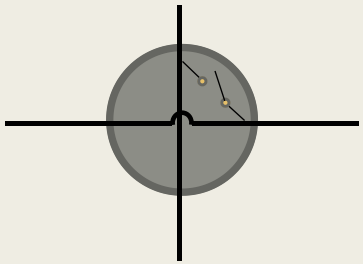
Příklad: **křížový přepínač** (crossbar)

Cena $\Omega(p^2)$, propustnost $O(p)$, neblokující

Diameter = 1, Konektivita = 1, Bisekce = p

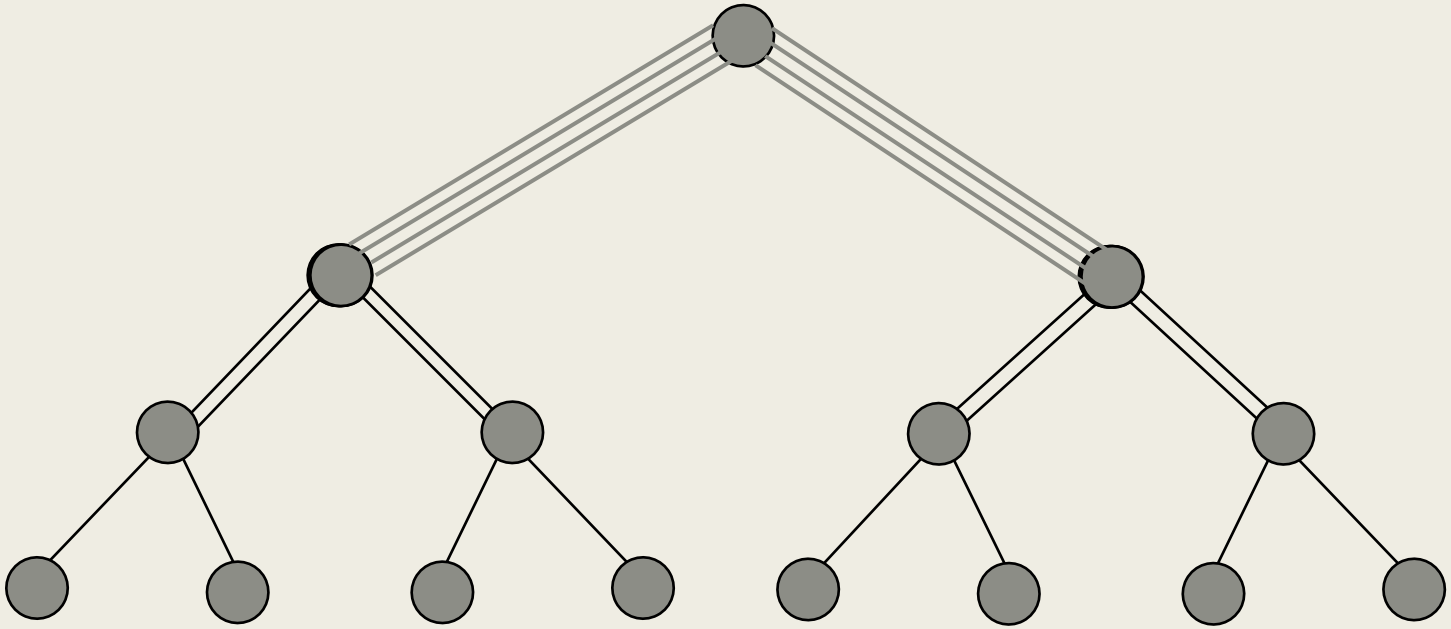
Pro p>m některé procesory nemají možnost přistupovat k paměti





Fat tree

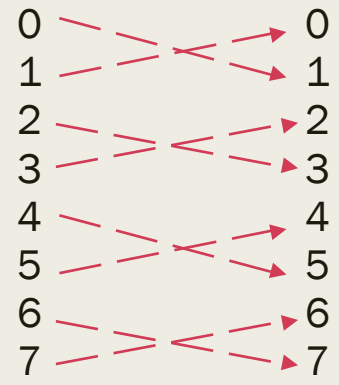
- Řeší problém větší zátěž komunikací blíže ke kořenu stromu
- Dynamická volba kanálu pro komunikaci



Shuffle and Exchange

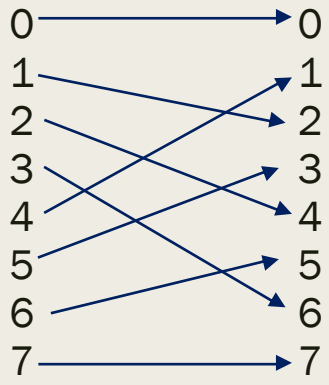
Exchange – inverze posledního bitu

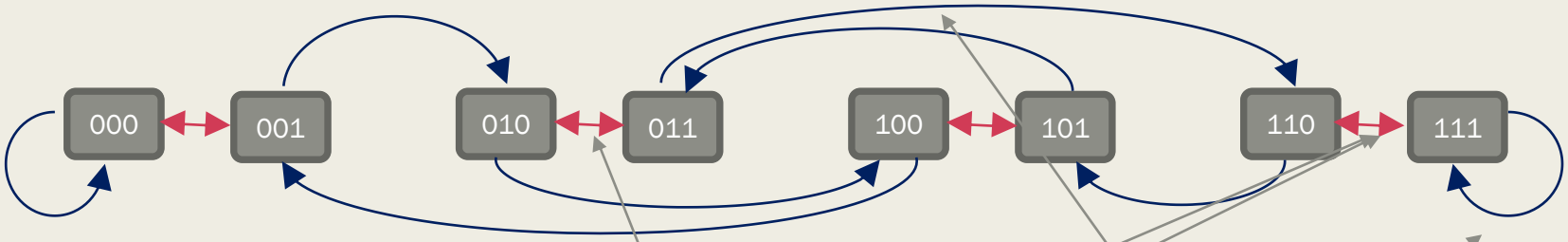
0	000	->	001
1	001	->	000
2	010	->	011
3	011	->	010
4	100	->	101
5	101	->	100
6	110	->	111
7	111	->	110



Shuffle – levý shift

0	000	->	000
1	001	->	010
2	010	->	100
3	011	->	110
4	100	->	001
5	101	->	011
6	110	->	101
7	111	->	111





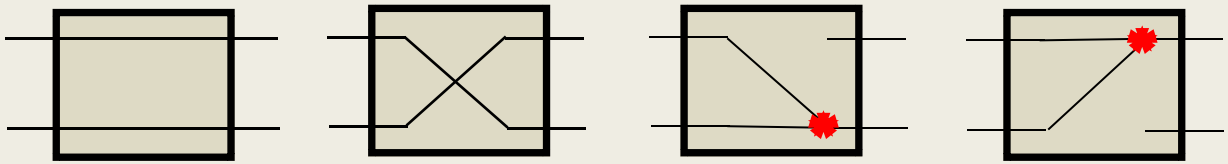
Exchange dle bitů cílového uzlu

Příklad: cílový **110** z uzlu **010**

```
EXCHANGE (pro 110) 010 -> 011, SHUFFLE 011  -> 110
EXCHANGE (pro 110) 110 -> 111, SHUFFLE 111  -> 111
EXCHANGE (pro 110) 111 -> 110
```

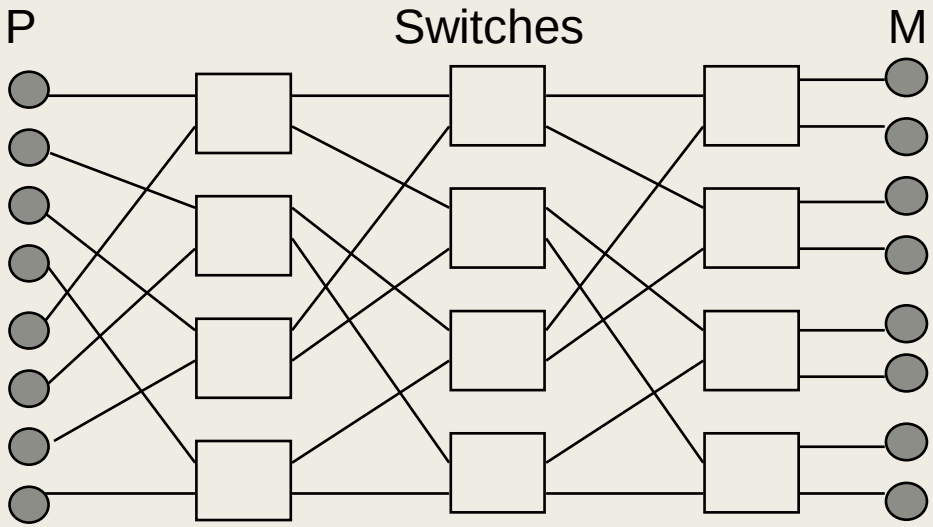
Přepínače

- Data přechází přes přepínače, ne přes uzly (např. hyperkostka)
- Přepínač … přeposílá vstup na jeden nebo více výstupů
- Realizace, fyzický, multiplexor / demultiplexor
- Buffer v případě konfliktů
- Možné směrovací mechanismy

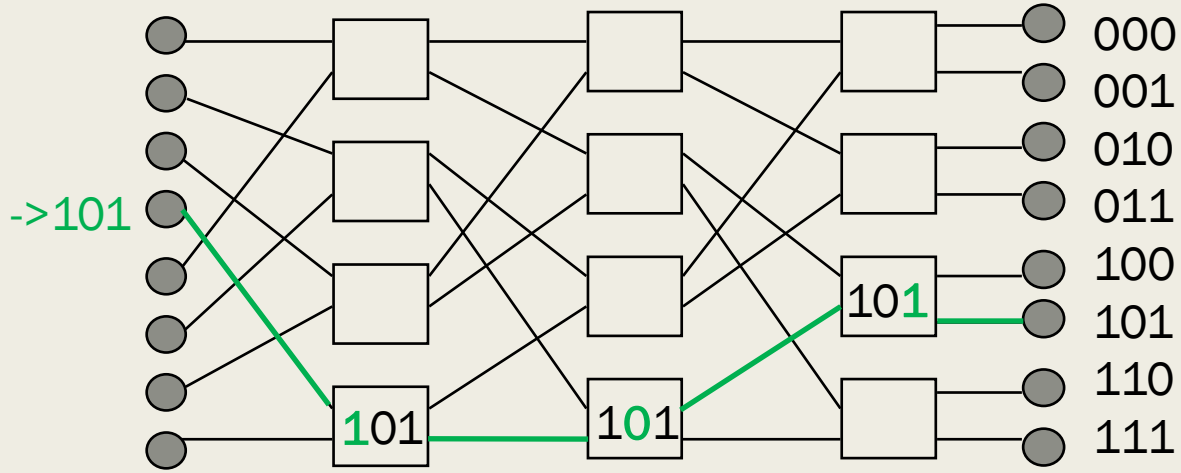


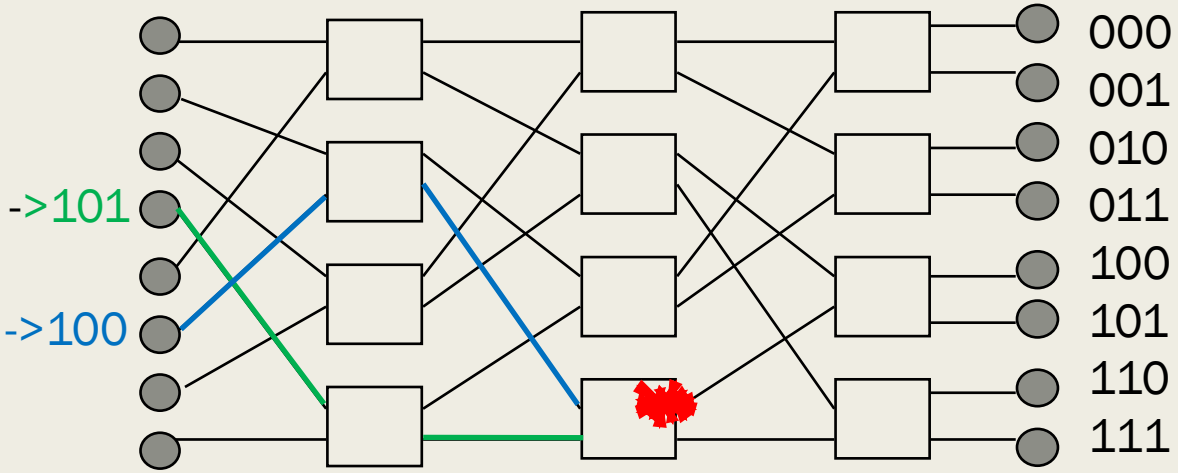
Multistage indirect networks

- Many such networks: Omega, Butterfly, Benes, …
  - *Connect p processors to p memory banks with $\Theta(p \lg p)$ switches*
  - *$\Theta(\lg p)$ stages of $\Theta(p)$ switches each*
- Blocking network
  - *Even if processors address distinct memory banks (permutation routing), can still have contention for switching elements*
- Omega network (p = 8)



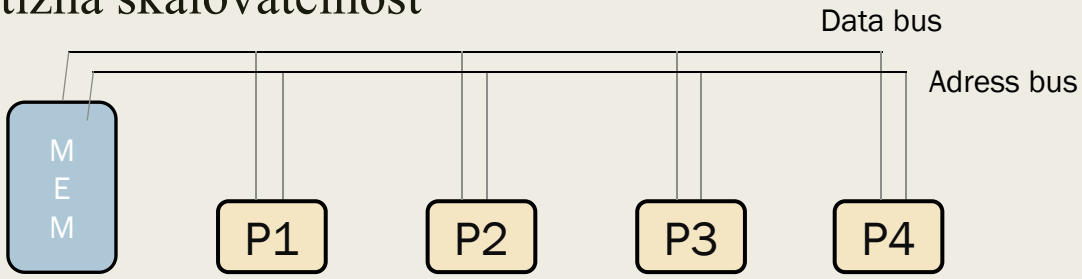
Síť Omega, konflikty



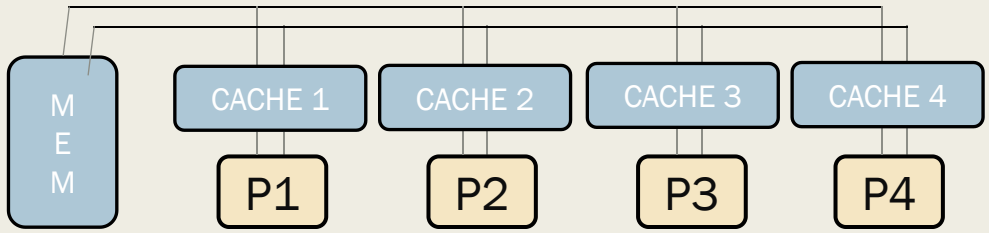


Sběrnice

- Obvykle pro sdílený prostředek (paměť)
- Cena $\Theta(p)$, propustnost $\Theta(1)$, diametr=1
- Obtížná škálovatelnost

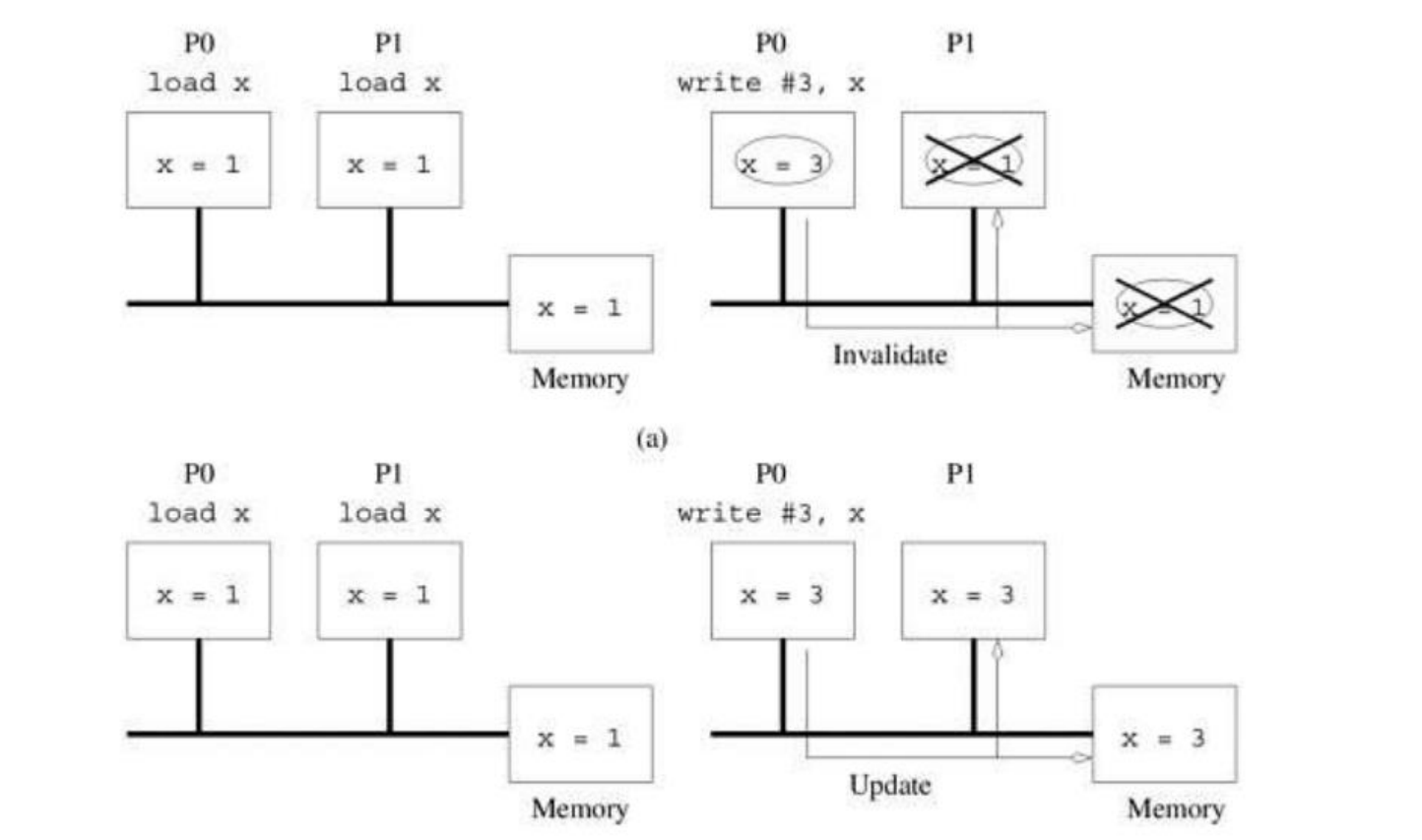


- Zvýšení propustnosti – lokálni cash

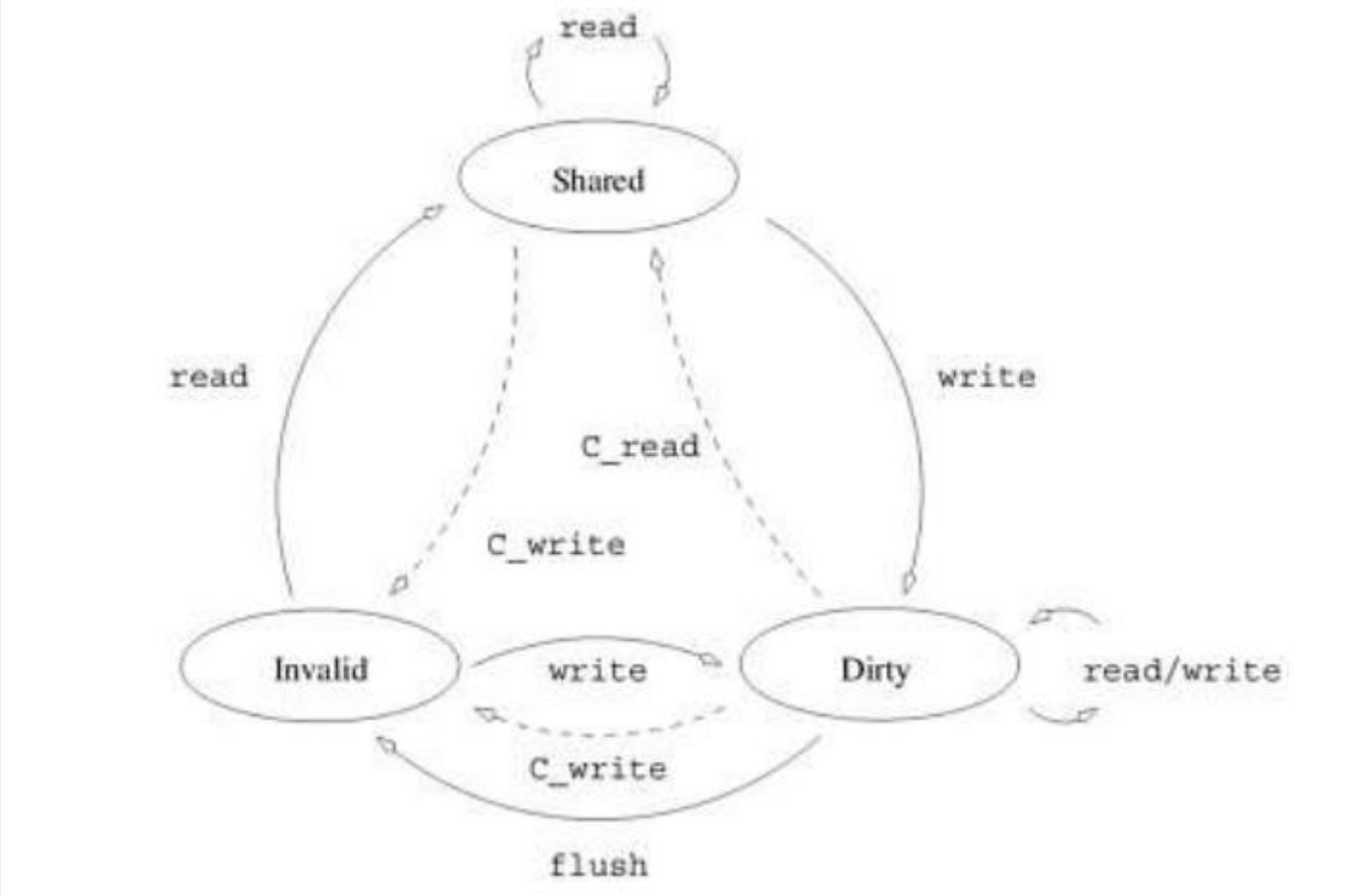


p procesorů, přístup k datům v paměti / cache t_{cycle}
Každý proces načítá k dat, 50% dat v cache
$0.5 * t_{cycle} * k + 0.5 * t_{cycle} * kp \rightarrow 0.5 * t_{cycle} * kp$ pro velké p

Cache, koherence



Cache, koherence



Cache, koherence

Time ↓

Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
				x = 5, D y = 12, D
read x		x = 5, S		x = 5, S
	read y		y = 12, S	y = 12, S
x = x + 1		x = 6, D		x = 5, I
	y = y + 1		y = 13, D	y = 12, I
read y		y = 13, S	y = 13, S	y = 13, S
	read x	x = 6, S	x = 6, S	x = 6, S
x = x + y		x = 19, D	x = 6, I	x = 6, I
	y = x + y	y = 13, I	y = 19, D	y = 13, I
x = x + 1		x = 20, D		x = 6, I
	y = y + 1		y = 20, D	y = 13, I