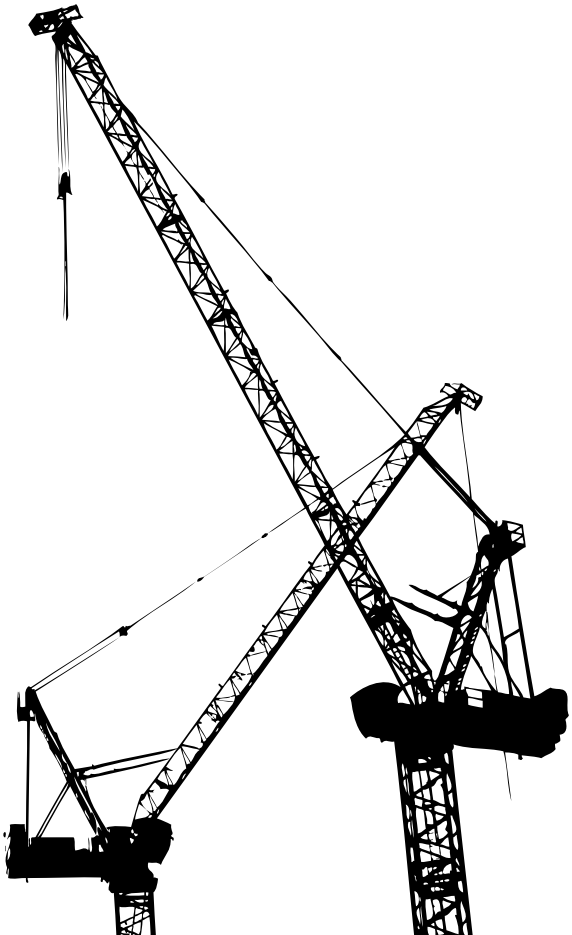
ARCHITEKTÚRA POČÍTAČOVÝCH SYSTÉMOV

2009/2010



APS 2009/2010
iPapiere
Vyučujúci: L. Hudec

Verzia 1.1 REVISITED
Hlavný manažér: iPanda
2010

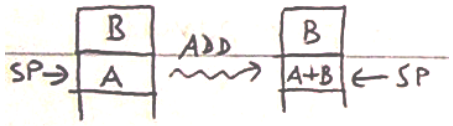
OBSAH

MNOŽINA INŠTRUKCIÍ - princípy a príklady

i. Klasifikácia architektúr množín inštrukcií

Požiadavkami sú: úplnosť inštrukčného súboru, základné inštrukcie musia byť ľahko použiteľné (efektívnosť), dedičnosť.

Podľa **typu vnútornej pamäte CPU** môžme architektúry deliť na:

Typ architektúry	Informácie	Výpočet C=A*B
zásobníková	stack pointer, operandy na vrch a pred ním 	PUSH A PUSH B ADD POP C
akumulátorová	akumulátor - zdrojové a cieľové operandy sú v ňom	LOAD A ADD B STORE C
architektúra s univerzálnymi registrami (GPR - General Purpose Registers)	k registrom sa pristupuje ľubovoľne prístupy: •REG-MEM (register-pamäť) •REG-REG (register-register)	REG-MEM: LOAD R1, A ADD R1, B STORE C, R1
		REG-REG: LOAD R1, A LOAD R2, B ADD R3,R1,R2 STORE C, R3

Hustota kódu - ak preložíme rovnaký program dvoma rôznymi CPU, dostaneme dva rôzne dlhé kódy, pričom ten kratší má väčšiu hustotu ako ten, ktorý je dlhší a má väčší počet inštrukcií.

Koncepcie GPR:

Počet pamäť. operandov	Max. počet operandov	Príklady
0	3	SPARC, MIPS, PowerPC
1	2	i80x86, Motorola 680x0
2	2	VAX stroje
3	3	

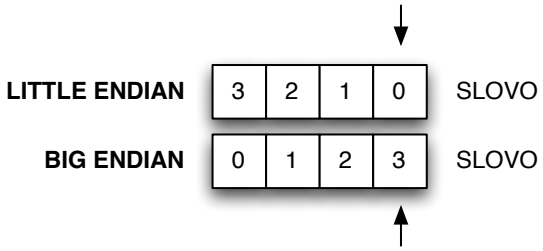
Aritmeticko-logické operácie typov GPR:

Typ (max. počet pamäťových operandov, počet operandov)	Výhoda	Nevýhoda
(0,3) [REG-REG] = [LOAD-STORE]	• jednoduché kódovanie inštrukcií • jednoduché generovanie kódu • výhodné pre prúdové spracovanie	• väčší počet inštrukcií ako pri pamäťových operandoch • niektoré inštrukcie sú krátke a nevyužívajú celý formát na kód
(1,2) [REG-MEM]	• operand môže byť sprístupnený bez zavedenia do registru CPU • formát inštrukcie môže poskytovať ľahké zakódovanie a dobrú hustotu kódu	• operandy nie sú ekvivalentné z hľadiska doby prístupu k operandu, kde býva i jeden prepísaný • rôzna doba realizácie inštrukcií
(3,3) [MEM-MEM]	• najkompaktnejšia architektúra • netreba internú pamäť CPU pre operandy	• veľké rozdiely vo veľkosti inštrukcií, rôzna doba realizácie inštrukcie • častý prístup do pamäti

ii. Adresovanie pamäti

Požiadavky: ako adresu interpretovať, ako ju špecifikovať, aká veľká je jedna pamäťová adresa objektu.

8-bit = SLABIKA
16-bit = POLSLOVO
32-bit = SLOVO
64-bit = DVOJSLOVO



Zarovnanie adresy (address alignment):

Width of object	0	1	2	3	4	5	6	7
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

iii. Adresovacie režimy operandov

Adresovací režim	Príklad inštrukcie	Význam	Kedy sa používa
registrový	ADD R4,R3	R4 ←R4+R3	keď sú operandy v registroch
bezprostredný (IMMEDIATE)	ADD R4,#3	R4 ←R4+3	pri narábaní s konštantou
posunutie	ADD R4,100(R1)	R4 ←R4+[100+R1]	prístup k lokálnym premenným
nepriamy registrový	ADD R4,(R1)	R4 ←R4+MEM[R1]	využívanie ukazovateľa alebo vypočítanej adresy
priamy (absolútny)	ADD R4,(1001)	R4 ←R4+MEM[1001]	pri globálnych premenných
nepriamy pamäťový	ADD R1,@(R3)	R1 ←R1+MEM[MEM[R3]]	ukazovateľ na ukazovateľ
autoinkrementačný	ADD R1,(R2)+	R1 ←R1+MEM[R2] R2 ←R2+d	prístup k položkám poľa
autodekrementačný	ADD R1,(R2)-	R2 ←R2-d R1 ←R1+MEM[R2]	pop
indexovaný	ADD R3,(R1+R2)	R3 ←R3+MEM[R1+R2]	prístup k položkám poľa
škálovateľný	ADD R1,100(R2)[R3]	R1 ←R1+MEM[100+R2+R3]	indexované polia

SPEC (System performance Evaluation Consortium) - balík štandardných aplikácií pre referenčné hodnotenie výkonnosti. Merá sa počet inštrukcií za sekundu (IPS - Instrucions per second) na 5 referenčných úloh floating point a 25 referenčných úloh integer (dohodnuté v SPEC ’89).

iv. Typy inštrukčných sád

Typmi inštrukčných sád sú:
- aritmeticko-logické
- presun údajov
- riadiace
- systémové
- dekadické
- v pohybovej rádovej čiarke
- reťazcové
- grafické.

Metódy pre príznaky (ako sa definuje podmienka pri podmienenom vetvení):

Spôsob	Ako je podmienka testovaná	Výhody	Nevýhody
kód podmienky CC (condition code)	špeciálne bity sú nastavené ALU operáciami, prípadne programovo	niekedy je stav CC nastavený samostatne	CC je extra stav
register podmienky Z-zero C-carry P-parity	inštrukcia testuje ľubovoľný register a zapisuje výsledok testu	jednoduchá	využíva register
porovnaj a vetvi	porovnanie je časťou inštrukcie vetvenia, často obmedzená na skupinu registrov	jedna inštrukcia	veľa práce na jednu inštrukciu

v. Typy a veľkosť operandov

Typy operandov:
- reťazce kódované pomocou ASCII
- integer
- floating-point

Floating-point môžu byť reprezentované jednoduchou presnosťou 32b, dvojnásobnou presnosťou 64b. Ostatné metódy a operácie s floating-point definuje norma IEEE 754, kde sú vnútorne reprezentované 80b a zaokrúhľované.

vi. Kódovanie inštrukcií

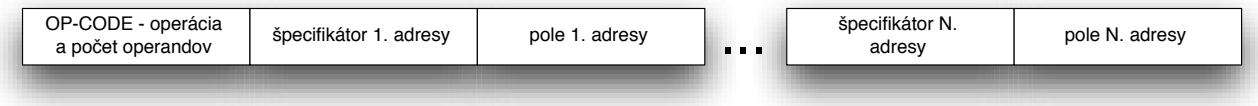
Kódovanie inštrukcií závisí od počtu adresovacích režimov, od stupňa nezávisosti OP-CODE (operačný kód), od počtu inštrukcií.

Ortogonalita OP-CODE - ak má inštrukcia aspoň jeden adresovací režim, tak môže použiť všetky spôsoby adresovacích režimov.

V prípade, že je väčší počet adresovacích režimov, používa sa režim špecifikátora adresy, čo je pole, ktoré definuje akým spôsobom je adresa špecifikovaná (CISC), alebo režim, kedy je adresovací režim definovaný priamo v OP-CODE.

Spôsoby kódovania inštrukcií:

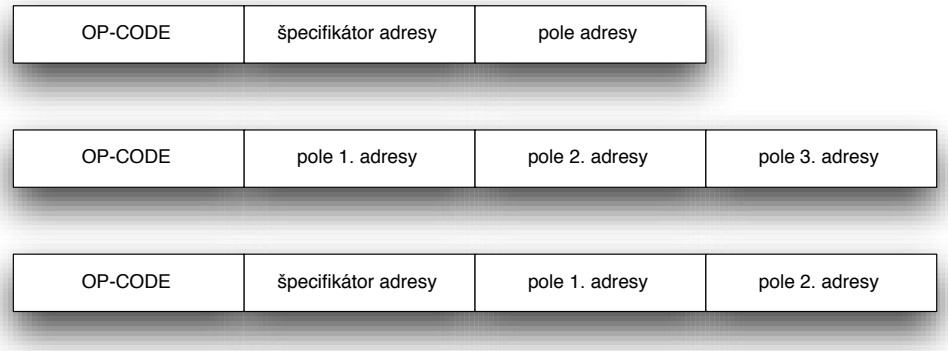
- s premenlivou dĺžkou (šetrenie pamäti)



- s pevnou dĺžkou (vysoký výkon)

strojové slovo 32/64 bit

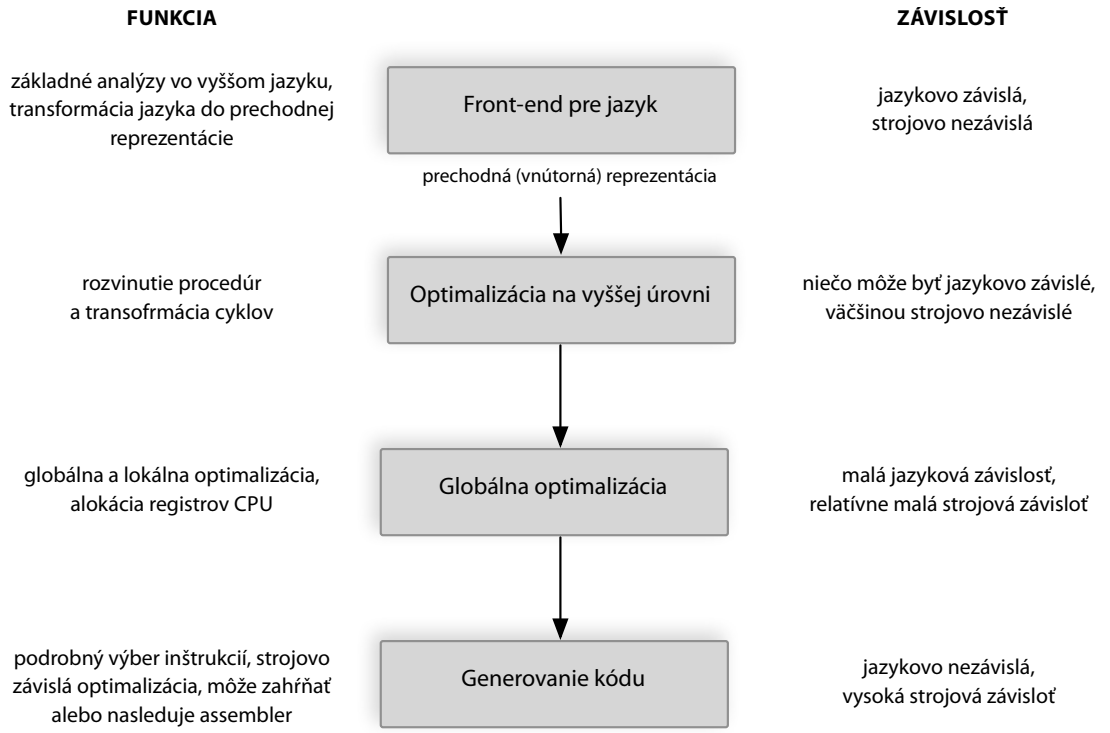
- hybridný spôsob (kombinácia)



vii. Postavenie a úloha prekladača

Vykonávané programy sú výsledkom prekladačov (vyšších jazykov), ktoré umožňujú korektne generovať optimálny kód.

Štruktúra súčasných prekladačov:



Dobré vlastnosti inštrukčného súboru pre jeho pisateľa:

- pravideľnosť - typická inštrukcia obsahuje OP-CODE, DAT_TYP, ADRES_REZIM, ortogonalita
- poskytnutie primitív a nie riešení - čo najjednoduchšie inštrukcie
- jednoduché porovnávanie medzi alternatívami - rýchlosť ≠ počet inštrukcií
- zabezpečiť v max. miere inštrukcie v etape programu - snaha obmedziť získavania parametrov v čase vykonávania

ARCHITEKTÚRA DLX (DeLuXe)

i. Charakteristika DLX, formát inštrukcií a inštrukčný súbor

DLX optimalizuje vykonávanie najčastejších inštrukcií, prúdové spracovanie inštrukcií a je optimálny pre vykonávanie SPEC benchmarkov. Je tu použitá koncepcia REG-REG architektúry GPR, pričom implementuje adresné režimy: bezprostredný operand, posunutie (priama adresa, nepriama registrová). Adresa ukazuje na slabiku (16b) vo forme Big Endian s 32b PC(program counter).

[R0=0] register 0 vždy obsahuje nulu
32x32bit GPR (R0...R31) - General Purpose Registers
32x32bit FPR (F0...F31) - Floating Point Registers

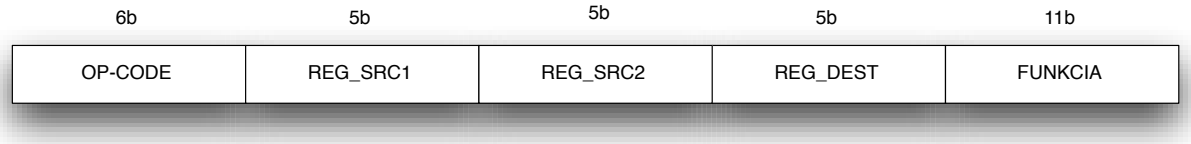
Dátové typy:
- 8b,16b,32b pre integer údaje
- 32b,64b pre floating point údaje.

Obľúbené však býva použitie 16b pre integer a 32b pre floating point skrz ich popularitu v jazyku C.

DLX implementuje pevný formát inštrukcií:
- I-typ:

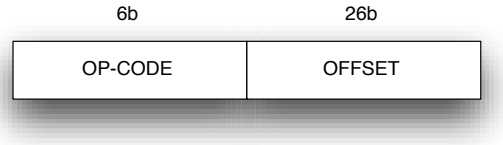
- DEST ← SRC1 op IMMEDIATE
- skoky s linkovým registrom, DEST=0, SRC1=destination, IMM=0
- podmienené skoky

- R-typ:



- DEST ← SRC1 funkcia SRC2

- J-typ:



- jump, jump on link, trap, return from exception

Format	Bits					
	31 26	**25 21**	**20 16**	**15 11**	**10 6**	**5 0**
R-type	0x0	Rs1	Rs2	Rd	*unused*	*opcode*
I-type	*opcode*	Rs1	Rd	*immediate*		
J-type	*opcode*	*value*				

Inštrukčný súbor DLX:

- load-store (REG-MEM)

```
LW R1,30(R2)      //R1 ← [R2+30], load word
LB R1,40(R3)      //R1 ← [R3+40], load byte a znamienkové rozšírenie
                   (doplnkový kód)
```

- aritmeticko logické operácie (REG-REG)

```
ADD R1,R2,R3      //R1 ← R2+R3
ADDI R1,R2,#3     //R1 ← R2+3
```

- vetvenie a skoky

```
JALR R2           //jump and link register
                  //vykona sa:
                  //R31 ← PC+4
                  //PC ← R2
JR R3             //PC ← R3
JR ADR            //R31 ← PC+4
                  //PC ← ADR
```

- floating point operácie

ii. Jednoduchá implementácia DLX

Neobsahuje floating point operácie, nazývaná tiež ako iteračný model. Inštrukcie sú vykonávané v 5 taktoch:

- **INSTRUCTION FETCH** - výber inštrukcie

```
IR ← MEM[PC]
NPC ← PC+4        //mame 4byte slovo
```

- **INSTRUCTION DECODE** - dekódovanie, výber zdrojového registra

```
A ← IR6-IR10
B ← IR11-IR15
IMM ← IR16-IR31  //znamienkove rozsirenie
```

- **EXECUTION** - vykonanie (záleží od inštrukcie)

```
ALUout ← A+IMM          //load/store - pamatova referencia
ALUout ← A op B         //aritmeticko-logicka REG-REG
ALUout ← A op IMM       //load/store - pamatova referencia
```

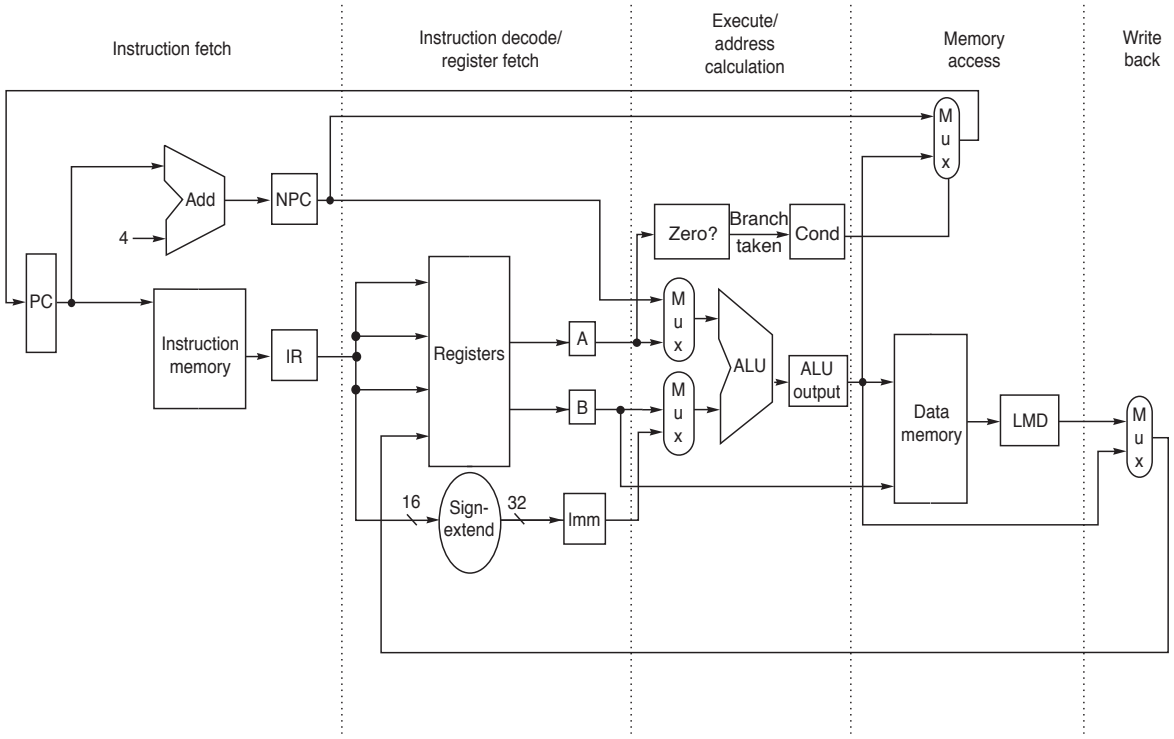
```
ALUout ← NPC + IMM        //vetvenie
condition ← A op 0        //nulovost
```

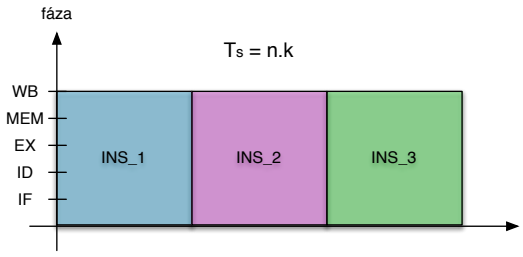
- **MEMORY** - prístup do pamäti/cyklus dokončenia vetvenia

```
LMD ← MEM[ALUout]         //pamatova referencia
MEM[ALUout] ← B           //pamatova referencia

PC ← ALUout               //vetvenie IF(condition)
PC ← NPC                  //vetvenie ELSE
```

- **WRITE BACK** - zápis späť

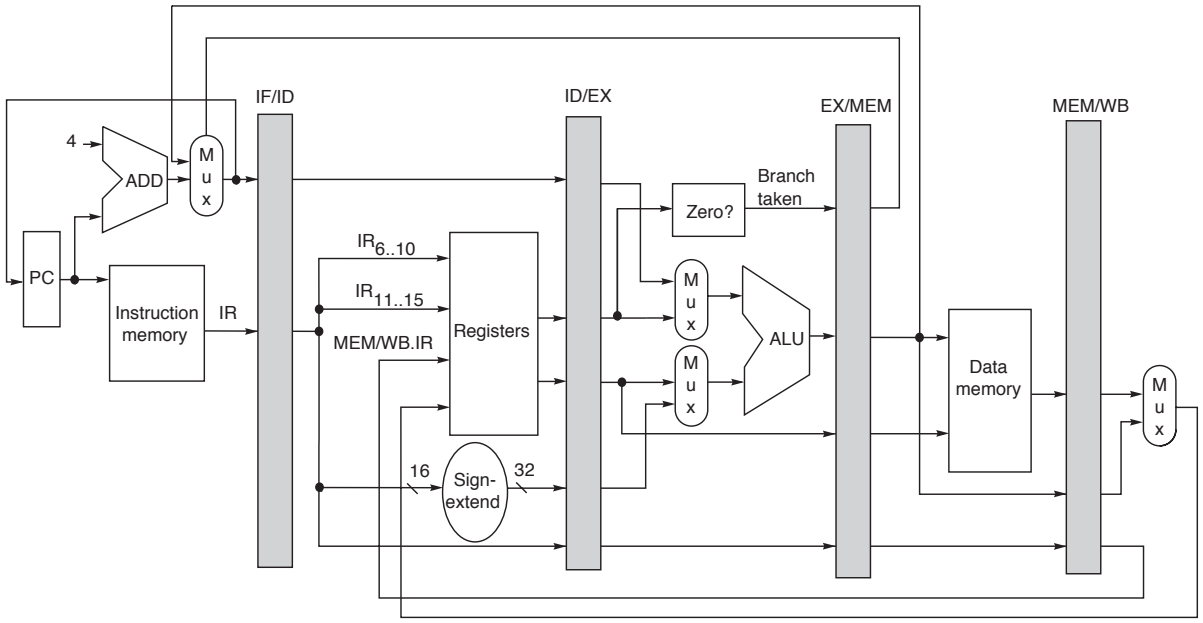
```
REG[IR16...IR20] ← ALUout     //aritmeticko-logicka REG-REG
REG[IR11...IR15] ← ALUout     //aritmeticko-logicka REG-IMM
REG[IR11...IR15] ← LMD        //instrukcne zavedenie, LMD je pomocny zachytny
                                register
```

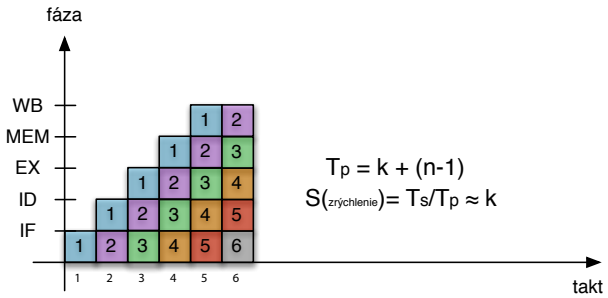




iii. Prúdová implementácia DLX (pipelling)

Úloha sa rozdelí na seba nadväzujúce podúlohy, ktoré sú realizované na rôznych prostriedkoch, ktoré sú nezávislé. Obsahuje registre prúdového spracovania IF|ID, ID|EX, EX|MEM, MEM|WB. Obsahuje hodiny, ktoré ich spúšťajú, ktoré musia byť vhodne načasované. V jednom momente je teda v CPU viacero inštrukcií, no každá v inej fáze vykonávania.





Stage	Any Instruction		
IF	IF/Id.IR <- Mem[PC]; IF/ID.NPC, PC <- (if EX/MEM.cond {EX/MEM.NPC} else {PC+4});		
ID	ID/EX.A <- Regs[IF/ID.IR$_{6..10}$]; ID/EX.B <- Regs[IF/ID.IR$_{11..15}$]; ID/EX.NPC <- IF/ID.NPC; ID/EX.IR <- IF/ID.IR; ID/EX.Imm <- (IR$_{16}$)^{16}##IR$_{16..31}$		
	ALU instruction	**Load or Store instruction**	**Branch instruction**
EX	EX/MEM.IR <- ID/EX.IR; EX/MEM.ALUoutput <- ID/EX.A op ID/EX.B; or EX/MEM.ALUoutput <- ID/EX.A op ID/EX.Imm; EX/MEM.cond <-0;	EX/MEM.IR <- ID/EX.IR; EX/MEM.ALUoutput <- ID/EX.A + ID/EX.Imm; EX/MEM.cond <- 0; EX/MEM.B <- ID/EX.B	EX/MEM.ALUoutput <- ID/EX.NPC + ID/ EX.Imm; EX/MEM.cond <- (ID/EX.A op 0;

MEM	`MEM/WB.IR <- EX/MEM.IR;`
	`MEM/WB.ALUoutput <-`
	`EX/MEM.ALUoutput;`
		`MEM/WB.IR <- EX/MEM.IR;`
		`MEM/WB.LMD <-`
		`Mem[EX/MEM.ALUoutput];`
		or
		`Mem[EX/MEM.ALUoutput]<-`
		`EX/MEM.B;`	

WB	`Regs[MEM/WB.IR`$_{16..20}$`]<-`
	`MEM/WB.ALUoutput;`
	or
	`Regs[MEM/WB.IR`$_{11..15}$`] <-`
	`MEM/WB.ALUoutput;`
		`Regs[MEM/WB.IR`$_{11..15}$`] <-`
		`MEM/WB.LMD;`	

iv. Hazardy v prúdovom spracovaní

Hazardy sú situácie, ktoré bránia využitiu ideálneho výkonu prúdového spracovania. Hazard vysporiadame tak, že zastavíme prúd vykonávania inštrukcií, až kým sa hazard nevyrieši. Prúdové spracovanie prináša problémy:

- existencia skokov nabúrava sled prúdu - riadiaci hazard
- súperenie o zdroje v stupňoch vykonávania, čítanie a zapisovanie naraz do pamäti - štrukturálny hazard
- dátová závislosť v dôsledku časového prekrývania - dátový hazard.

v. Štrukturálny hazard

Example: a machine has shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data-memory reference(load), it will conflict with the instruction reference for a later instruction (instr 3). To resolve this, we **stall** the pipeline for one clock cycle when a data-memory access occurs. The effect of the stall is actually to occupy the resources for that instruction slot. The following table shows how the stalls are actually implemented.

Clock cycle number									
Instr	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	MEM	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				stall	IF	ID	EX	MEM	WB

vi. Dátový hazard

Consider the pipelined execution of these instructions:

		1	2	3	4	5	6	7	8	9
ADD	**R1**, R2, R3	IF	ID	EX	MEM	**WB**				
SUB	R4, R5, **R1**		IF	**ID**$_{sub}$	EX	MEM	WB			
AND	R6, **R1**, R7			IF	**ID**$_{and}$	EX	MEM	WB		
OR	R8, **R1**, R9				IF	**ID**$_{or}$	EX	MEM	WB	
XOR	R10,**R1**,R11					IF	**ID**$_{xor}$	EX	MEM	WB

All the instructions after the ADD use the result of the ADD instruction (in R1). The ADD instruction writes the value of R1 in the WB stage (shown black), and the **SUB** instruction reads the value during ID stage (ID_{sub}). The SUB instruction will read the wrong value and try to use it.
The **AND** instruction is also affected by this data hazard. The write of R1 does not complete until the end of cycle 5 (shown black). Thus, the AND instruction that reads the registers during cycle 4 (ID_{and}) will receive the wrong result.

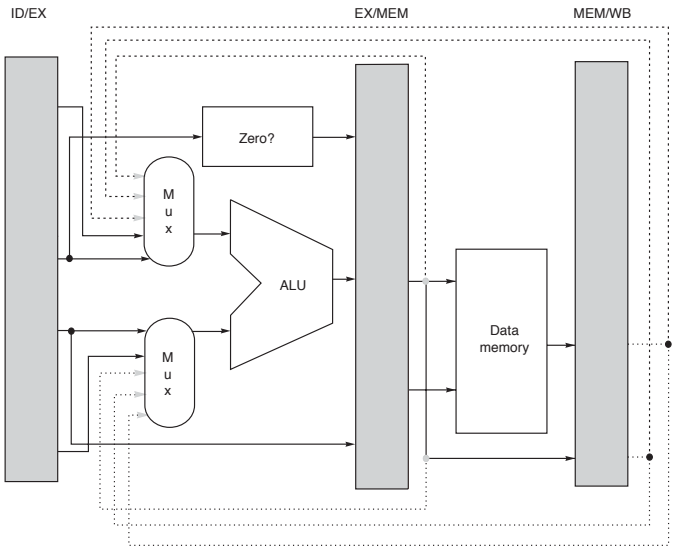
The **OR** instruction can be made to operate without incurring a hazard by a simple implementation technique. ***The technique*** is to perform register file reads in the second half of the cycle, and writes in the first half. Because both **WB** for ADD and ID_{or} for OR are performed in one cycle 5, the write to register file by ADD will perform in the first half of the cycle, and the read of registers by OR will perform in the second half of the cycle.

The **XOR** instruction operates properly, because its register read occur in cycle 6 after the register write by ADD.

A technique to eliminate the stalls for the hazard involving the SUB and AND instructions is called **data forwarding**.

Data forwarding:
The key insight in forwarding is that the result is not really needed by SUB until after the ADD actually produces it. The only problem is to make it available for SUB when it needs it.
If the result can be moved from where the ADD produces it (EX/MEM register), to where the SUB needs it (ALU input latch), then the need for a stall can be avoided.

Using this observation , forwarding works as follows:
- The ALU result from the EX/MEM register is always ***fed back*** to the ALU input latches.
- If the forwarding hardware detects that the previous ALU operation has written the register corresponding to the source for the current ALU operation, ***control logic*** selects the forwarded result as the ALU input rather than the value read from the register file.



Forwarding of results to the ALU requires the additional of three extra inputs on each ALU multiplexer and the addtion of three paths to the new inputs.

The paths correspond to a forwarding of:
(a) the ALU output at the end of EX,
(b) the ALU output at the end of MEM, and
(c) the memory output at the end of MEM.

Without forwarding our example will execute correctly with stalls:

		1	2	3	4	5	6	7	8	9
ADD	**R1**, R2, R3	IF	ID	EX	MEM	**WB**				
SUB	R4, R5, **R1**		IF	**stall**	**stall**	ID_{sub}	EX	MEM	WB	
AND	R6, **R1,** R7			**stall**	**stall**	IF	ID_{and}	EX	MEM	WB

As our example shows, we need to forward results not only from the immediately previous instruction, but possibly from an instruction that started three cycles earlier. Forwarding can be arranged from MEM/WB latch to ALU input also. Using those forwarding paths the code sequence can be executed without stalls:

		1	2	3	4	5	6	7
ADD	**R1**, R2, R3	IF	ID	EX_{add}	MEM_{add}	WB		
SUB	R4, R5, **R1**		IF	ID	EX_{sub}	MEM	WB	
AND	R6, **R1,** R7			IF	ID	EX_{and}	MEM	WB

The first forwarding is for value of **R1** from EX_{add} to EX_{sub} .
The second forwarding is also for value of **R1** from MEM_{add} to EX_{and}.
This code now can be executed without stalls.

Forwarding can be generalized to include passing the result directly to the functional unit that requires it: a result is forwarded from the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit.

Consider two instructions ***i*** and ***j***, with ***i*** occurring before ***j***. The possible data hazards are:

- **RAW (read after write)** - **j** tries to read a source before **i** writes it, so **j** incorrectly gets the old value. This is the most common type of hazard and the kind that we use forwarding to overcome.
- **WAW (write after write)** - **j** tries to write an operand before it is written by **i**. The writes end up being performed in the wrong order, leaving the value written by **i** rather than the value written by **j** in the destination.
- ***WAR (write after read)*** - ***j*** *tries to write a destination before it is read by* ***i*** *, so* ***i*** *incorrectly gets the new value.*

Unfortunately, not all potential hazards can be handled by forwarding.
Consider the following sequence of instructions:

		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
LW	**R1**, 0(R1)	IF	ID	EX	**MEM**	**WB**			
SUB	R4, **R1**, R5		IF	ID	EX_{sub}	MEM	WB		
AND	R6, **R1** R7			IF	ID	EX_{and}	MEM	WB	
OR	R8, **R1**, R9				IF	**ID**	EX	MEM	WB

The **LW** instruction **does not** have the data until the end of clock cycle 4 (**MEM**) , while the **SUB** instruction needs to have the data by the beginning of that clock cycle (EX_{sub}). For **AND** instruction we can forward the result immediately to the ALU (EX_{and}) from the MEM/WB register(**MEM**). **OR** instruction has no problem, since it receives the value through the register file (**ID**). In clock cycle no. 5, the WB of the LW instruction occurs "early" in first half of the cycle and the register read of the OR instruction occurs "late" in the second half of the cycle.

For **SUB** instruction, the forwarded result would arrive too late - at the end of a clock cycle, when needed at the beginning. The load instruction has a delay or latency that cannot be eliminated by forwarding alone. Instead, we need to add hardware, called ***a pipeline interlock***, to preserve the correct execution pattern. In general, a pipeline interlock ***detects*** a hazard and ***stalls*** the pipeline until the hazard is cleared. The pipeline with a stall and the legal forwarding is:

		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	9
LW	**R1**, 0(R1)	IF	ID	EX	**MEM**	WB				
SUB	R4, **R1**, R5		IF	ID	***stall***	EX_{sub}	MEM	WB		
AND	R6, **R1** R7			IF	***stall***	ID	EX	MEM	WB	
OR	R8, **R1**, R9				***stall***	IF	ID	EX	MEM	WB

The only necessary forwarding is done for R1 from **MEM** to $\mathbf{EX_{sub}}$. Notice that there is no need to forward R1 for **AND** instruction because now it is getting the value through the register file in ID (as **OR** above).

vii. Riadiaci hazard

Control hazards can cause a greater performance loss for DLX pipeline than data hazards. When a branch is executed, it may or may not change the PC (program counter) to something other than its current value plus 4. If a branch changes the PC to its target address, it is a ***taken branch***; if it falls through, it is ***not taken***. If instruction ***i*** is a ***taken branch***, then the PC is normally not changed until the end of MEM stage, after the completion of the address calculation and comparison. The simplest method of dealing with branches is ***to stall*** the pipeline as soon as the branch is detected until we reach the **MEM** stage, which determines the new PC. The pipeline behavior looks like :

Branch	IF	**ID**	EX	**MEM**	WB					
Branch successor		IF(stall)	***stall***	***stall***	IF	ID	EX	MEM	WB	
Branch successor+1						IF	ID	EX	MEM	WB

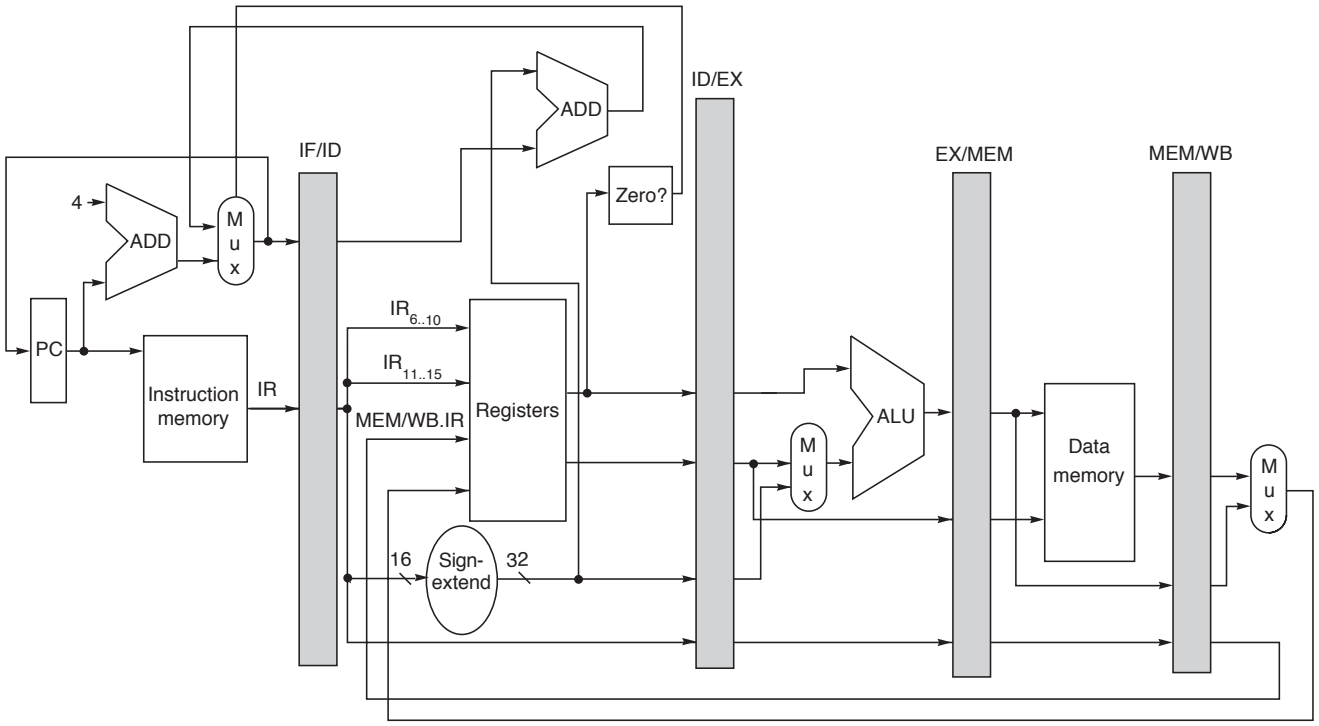
The stall does not occur until after **ID** stage (where we know that the instruction is a branch). This control hazards stall must be implemented differently from a data hazard, since the **IF** cycle of the instruction following the branch **must be repeated** as soon as we know the branch outcome. Thus, the first **IF** cycle is essentially a ***stall*** (because it never performs useful work), which comes to total 3 stalls. Three clock cycles wasted for every branch is a significant loss. With a 30% branch frequency and an ideal CPI of 1, the machine with branch stalls achieves only ***half*** the ideal speedup from pipelining!

The number of clock cycles can be reduced by two steps:
- Find out whether the branch is taken or not taken ***earlier*** in the pipeline;
- Compute the taken PC (i.e., the address of the branch target) ***earlier***.

Both steps should be taken as early in the pipeline as possible.

By moving the zero test into the ID stage, it is possible to know if the branch is taken at the end of the ID cycle. Computing the branch target address during ID requires an additional adder, because the main ALU, which has been used for this function so far, is not usable until EX.

The revised datapath:



With this datapath we will need only one-clock-cycle stall on branches.

Branch	IF	**ID**	EX	MEM	WB		
Branch successor		IF(stall)	IF	ID	EX	MEM	WB

In some machines, branch hazards are even more expensive in clock cycles. For example, a machine with separate decode and register fetch stages will probably have a ***branch delay*** - *the length of the control hazard* - that is at least one clock cycle longer. The branch delay, unless it is dealt with, turns into a ***branch penalty***. Many older machines that implement more complex instruction sets have branch delays of four clock cycles or more. In general, the deeper the pipeline, the worse the branch penalty in clock cycles.

The simplest scheme to handle branches is to ***freeze*** or ***flush*** the pipeline, holding or deleting any instructions after the branch until the branch destination is known. A higher performance, and only slightly more complex, scheme is to predict the branch as not taken, simply allowing the hardware to ***continue as if the branch were not executed.*** Care must be taken not to change the machine state until the branch outcome is definitely known.

The complexity arises from:
- we have to know when the state might be changed by an instruction;
- we have to know how to "back out" a change.

The pipeline with this scheme implemented behaves as shown below:

Untaken Branch Instr	IF	ID	EX	MEM	WB		
Instr i+1		IF	ID	EX	MEM	WB	
Instr i+2			IF	ID	EX	MEM	WB

Taken Branch Instr	IF	ID	EX	MEM	WB			
Instr i+1		IF	***idle***	***idle***	***idle***	***idle***		
Branch target			IF	ID	EX	MEM	WB	
Branch target+1				IF	ID	EX	MEM	WB

When branch is not taken, determined during ID, we have fetched the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall one clock cycle.

An alternative scheme is to predict the branch as taken. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and ***begin fetching and executing at the target address.***

Because in DLX pipeline the target address is not known any earlier than the branch outcome, there is no advantage in this approach. In some machines where the target address is known before the branch outcome a predict-taken scheme might make sense.

In a delayed branch, the execution cycle with a branch delay of length n is:

```
Branch instr
sequential successor 1
sequential successor 2
. . . . .
sequential successor n
Branch target if taken
```

Sequential successors are in ***the branch-delay slots***. These instructions are executed whether or not the branch is taken. The pipeline behavior of the DLX pipeline, which has one branch delay slot is shown below:

Untaken branch instr	IF	ID	EX	MEM	WB				
Branch delay instr(i +1)		IF	ID	EX	MEM	WB			
Instr i+2			IF	ID	EX	MEM	WB		
Instr i+3				IF	ID	EX	MEM	WB	
Instr i+4					IF	ID	EX	MEM	WB

Taken branch instr	IF	ID	EX	MEM	WB				
Branch delay instr(i +1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target+1				IF	ID	EX	MEM	WB	
Branch target+2					IF	ID	EX	MEM	WB

The job of the compiler is to make the successor instructions ***valid*** and ***useful.***

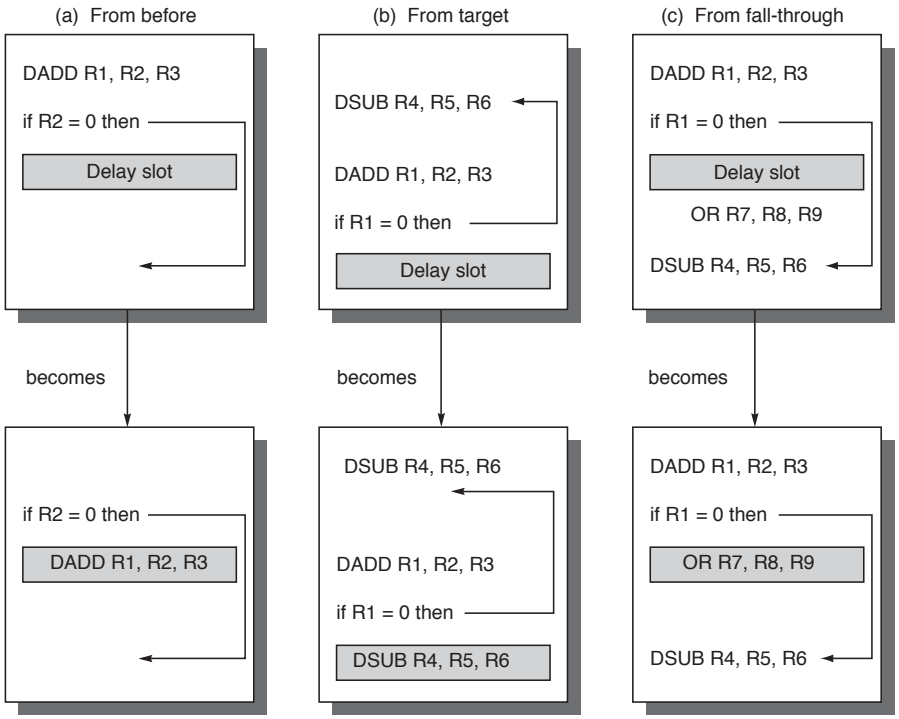
We will show three branch-scheduling schemes:
- From before branch
- From target
- From fall through

Scheduling the branch-delay slot:
In (a) the delay slot is scheduled with an independent instruction from before the branch. This is the best choice.

Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of R1 in the branch condition prevents the ADD instruction (whose destination is R1) from being moved after the branch.

In (b) the branch-delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path.

In (c) the branch-delay slot is scheduled from the not-taken fall through. To make this optimazation legal for (b) and (c), it must be OK to execute the SUB instruction when the branch goes in the unexpected direction. OK means that the work might be wasted but the program will still execute correctly.



Scheduling strategy	Requirements	When Improves Performance
From before branch	Branch must not depend on the rescheduled instructions	Always
From target	Must be OK to execute rescheduled instructions if branch is not taken	When branch is taken. May enlarge program if instructions are duplicated
From fall though	Must be OK to execute instructions if branch is taken	When branch is not taken

The limitations on delayed-branch scheduling arise from the ***restrictions on the instructions*** that are scheduled into the delay slots and our ***ability to predict*** at compile time whether a branch is likely to be taken or not.

viii. Rozšírenie prúdového prostriedku o viaccyklové operácie

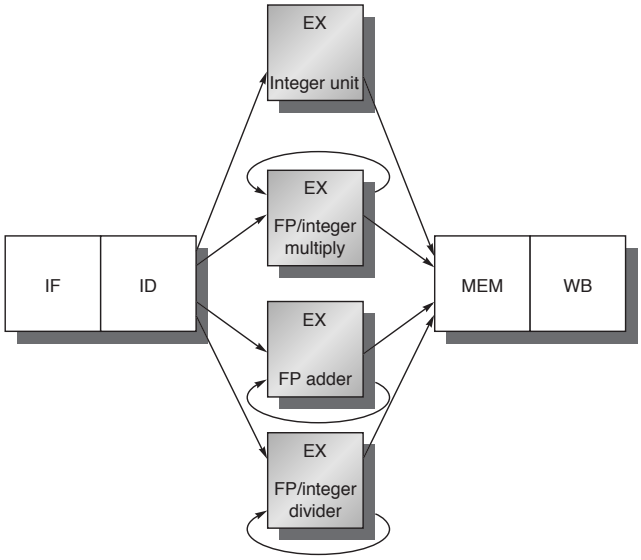
It is impractical to require that all DLX floating-point operations complete in one clock cycle, or even two. Doing it would mean accepting a slow clock, or using enormous amounts of logic in the floating-point units, or both. Instead, the floating-point pipeline will allow for a longer latency for operations. This is easier to grasp if we imagine the floating-point instructions as having the same pipeline as the integer instructions, with two important differences:

- The EX cycle may be repeated as many times as needed to complete the operation
- There may be multiple floating-point functional units.

Let's assume that there are four separate functional units :

- The main integer unit
- FP and integer multiplier
- FP adder (handles FP add, subtract, and conversion)
- FP and integer divider

If we also assume that the execution stages of these functional units are not pipelined, then the resulting pipeline looks like:

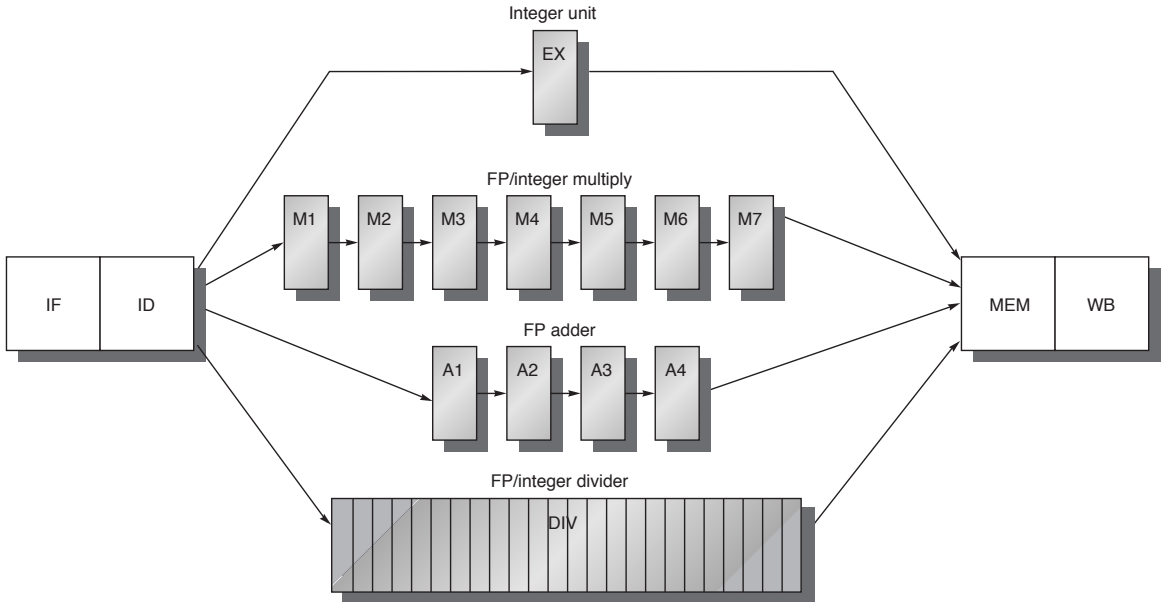


Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The floating-point operations simply loop when they reach the EX stage. In reality, the intermediate results are probably not cycled around the EX unit, but the EX pipeline stage has some number of clock delays larger then 1. We can generalize the structure of the FP pipeline to allow pipelining of some stages and multiple ongoing operations.

To describe such a pipeline we must define ***the latency*** of the functional units and ***the initiation interval***. ***Latency*** is defined as the number of intervening cycles between an instruction that produces a result and an instruction that uses the result. ***The initiation interval*** is the number of cycles that must elapse between issuing two operations of a given type.

Pipeline latency is essentially equal to one cycle less than the depth of the execution pipeline, which is the number of stages from the EX stage to the stage that produces the result. Thus, for the example pipeline above, the number of stages in an FP add is 4, while the number of stages in FP multiply is 7.

Extended pipeline is show below:



The example pipeline structure allows up to 4 outstanding FP adds, 7 outstanding FP/integer multiplies, and 1 FP divide. The FP multiplier and adder are fully pipelined and have a depth of 7 and 4 stages, respectively. The FP divider is not pipelined, but has 24 stages.

The pipeline timing of a set of independent FP operations:

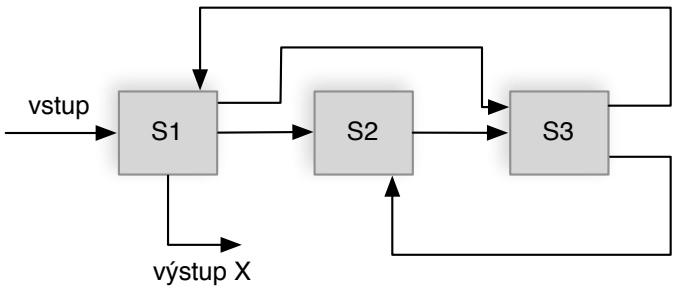
MULTD	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADDD		IF	ID	*A1*	A2	A3	**A4**	MEM	WB		
LD			IF	ID	*EX*	**MEM**	WB				
SD				IF	ID	*EX*	*MEM*	WB			

The stages in italic show where data is needed, while the stages in bold show where a result is available.

INÉ ARCHITEKTÚRY A MECHANIZMY

i. Prúdové a superskalárne techniky - Nonlinear pipelling

Zadefinujeme si všeobecný prúdový prostriedok, ktorý okrem prúdov umožňuje i spätnú i doprednú väzbu.



Reservation tables are two- dimensional charts used to show how successive pipeline stages are utilized (or reserved) for a specific function evaluation in successive pipeline cycles. Let X & Y be two functions evaluating in the above pipeline.

Rezervačná tabuľka pre funkcie X:

Funkcia X:

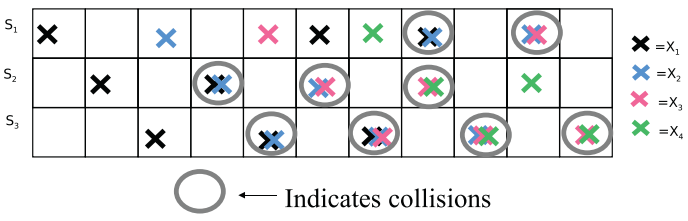
	1	2	3	4	5	6	7	8	*
S1	**X**					**X**		**X**	5,7,2
S2		**X**		**X**					2
S3			**X**		**X**		**X**		4,2

* - všetky možné latencie medzi všetkými inicializáciami (koľko políčok je medzi nimi vrátane druhej inicializácie)

Inicializáciou nazývame spustenie operácie. Latencia je počet cyklov medzi dvoma inicializáciami. Latencia *k* znamená, že dve inicializácie sú od seba vzdialené *k* cyklami. Poznáme dva druhy latencií:

- dovolené - nevznikne kolízia
- zakázané - vznikne kolízia.

Consider the same pipeline, lets see initiations in the pipeline for function X, with latency=2:



Pomocou zakázaných latencií vieme vypočítať minimálnu uzavretú cestu **MAL (Minimal Average Latency)**, ktorá dá prúdovému prostriedku najväčšiu efektivitu bez kolízii, čo sa anglicky nazýva Collision-free scheduling.

Kolízny vektor definuje zakázané a povolené kolízie, kde každý bit vektora definuje jednotkou zakázané latencie alebo nulou povolené. V každom cykle sa tieto bity posúvajú do prava. V našom prípade pre X a Y sú inicializačné kolízne vektory, bity sú číslované od 1:

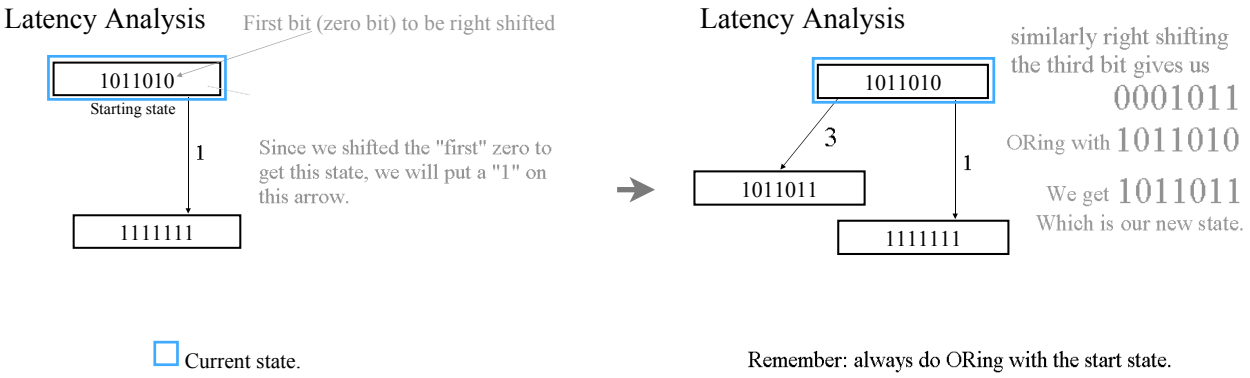
- pre X: 1011010 (7654321)
- pre Y: 1010 (4321).

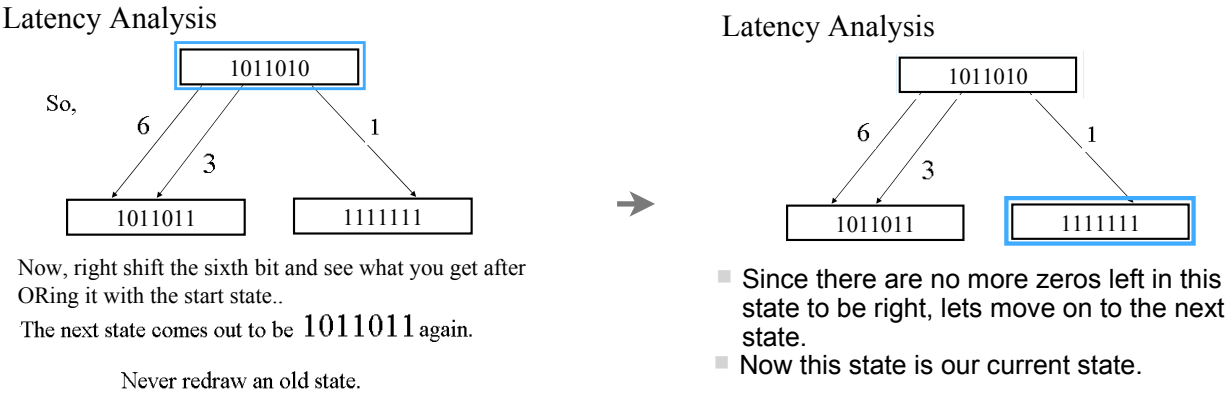
Let us first look at some basic terms:

- Latency sequence: a latency sequence is a sequence of permissible latencies between successive tasks initiations.
- Latency cycle: is a latency sequence which repeats itself.
- Constant cycle: which contains a single latency value.
- State diagram: will be used here to specify the permissible state transitions among successive tasks initiations.
- Collision vector: (collision array) is a combined set of permissible and forbidden latencies. Collision vector: C = (C_m C_{m-1} …… C_2 C_1), m: maximum forbidden latency, C_i : 0 permissible, C_i : 1 forbidden.

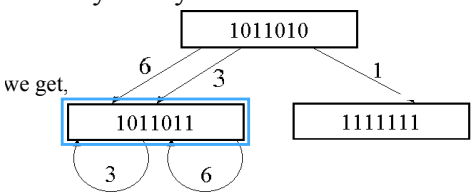
Construction of the state diagram:

- Collision vector is the initial position vector or the starting state of the sate diagram.
- Next state is obtained by right shifting the zeros of current state and then OR-ing the result with collision vector.



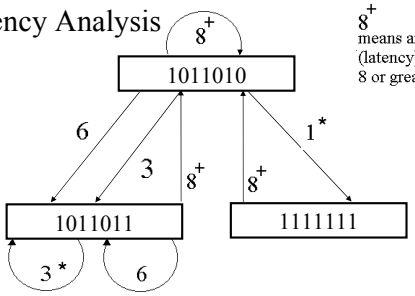


Latency Analysis



- In this state we don't have any zeros, so we move on to the next state. As our current state.
- Right shifting and ORing of the third and sixth bit gives the same state again.

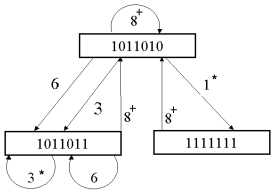
Latency Analysis



8^+ means any integer (latency) equal to 8 or greater than 8

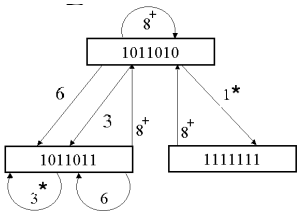
One last thing, when the number of shifts is m+1 (m: maximum forbidden latency), all the transitions are redirected back to initial state.
And mark the minimal latencies from each state with *.

Latency Analysis



- Our state diagram is complete now.
- This state diagram is used to characterize successive initiations of tasks in the pipeline in order to find the shortest *latency sequence* to optimize the control strategy.
- A state on the diagram is representing the contents of shift register after proper no. of shifts is made, which is equal to the latency between the current and next task initiations.

Latency Analysis



- The next step is to write the simple latency cycles.
- ***Simple Cycles:*** are those in which each state appears only once.

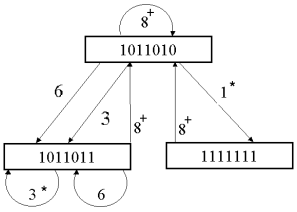
so for this state diagram we have:

(3), (6), (8), (1,8), (3,8), (6,8)

as our simple cycles.

- Some of these are called greedy cycles

Latency Analysis

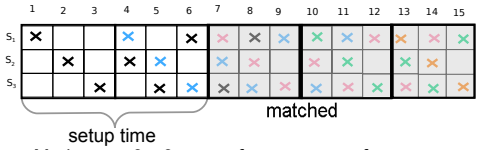


- ***Greedy Cycles:*** are those ones whose edges are all made with minimum latencies from their respective states (ones with *) .

so for this state diagram we have:

(3) & (1,8) as our greedy cycles

- The greedy cycle (1,8) has an average latency of (1+8)/2=4.5
- The greedy cycle (3) has a constant latency which equals the MAL.



- Mark every 3 x 3 array of squares as a frame.

(since greedy cycle is (3) & we have 3 stages)

- Keep on initializing new tasks until consecutive frames begin to match.
- The time until there appears no match is called the Setup Time.
- Calculating the efficiency: With in a frame we have,

Total stages=9
Stages busy=8 (take from the matched frames)

$$\text{efficiency: } \eta = \frac{\text{busy stages}}{\text{total stages}} \times 100$$

$$\eta = \frac{8}{9} \times 100$$

$$= 88.89\%$$

Efektívnosť je inými slovami stupeň využitia.

Priepustnosť je: 1/MAL * frekvencia hodín.

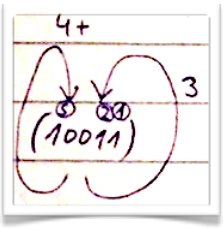
ii. Optimalizácia plánovania prúdového prostriedku

Pravidlá pre MAL:
- MAL ≥ maximálny počet obsadení jedného stupňa
- MAL ≤ strednej latencii minimálneho jedného cyklu ≤ počet 1tiek v kolíznom vektore+1.

Majme funkciu Z:

	1	2	3	4	5	6	*
S1	**Z**					**Z**	5
S2		**Z**		**Z**			2
S3			**Z**				
S4				**Z**	**Z**		1

Kolízny vektor: 10011.

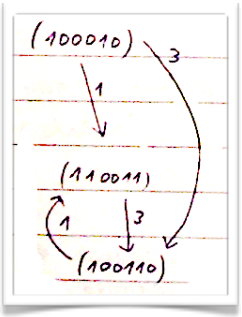


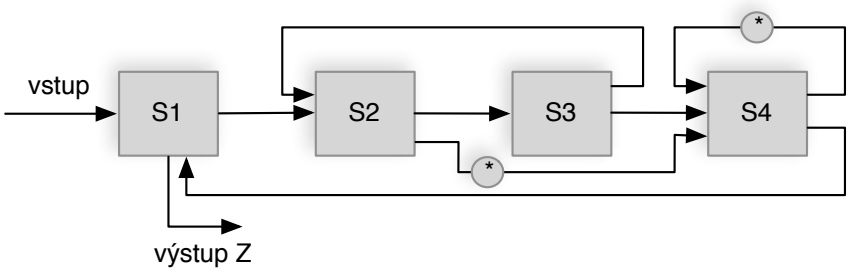
MAL=3

Vkladaním čakacích stavov zmenšíme počet 1tiek v kolíznom vektore, vznikne nám i modifikovaný prúdový prostriedok:

	1	2	3	4	5	6	7	*
S1	**Z**						**Z**	6
S2		**Z**		Z				2
S3			**Z**					
S4				Z		**Z**		2
D1					Z			
D2					Z			

Kolízny vektor: 100010.



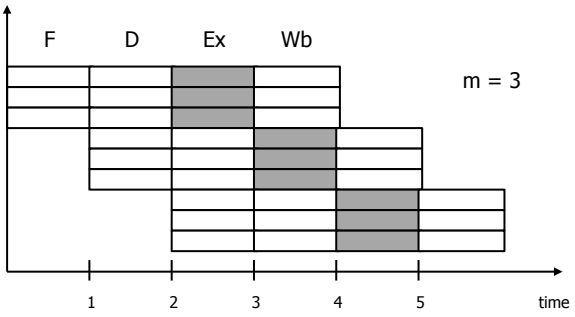


MAL = (1+3)/2 = 2
P = 1/MAL * 10^6 * f[MHz]

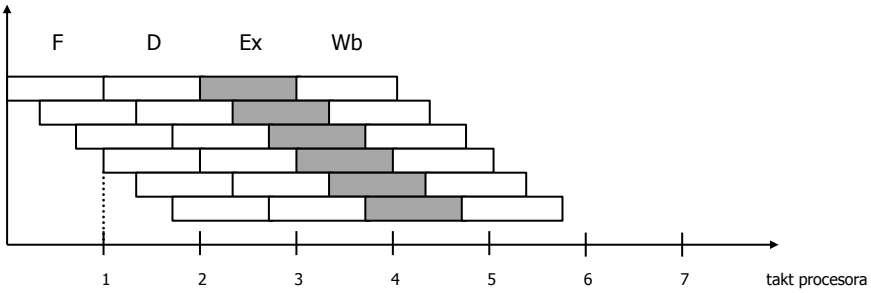
iii. Superskalárne a superprúdové spracovanie

Základnú koncepciu DLX (základný skalárny stroj) môžme vylepšiť na tzv. superskalárny stroj alebo superprúdový stroj.

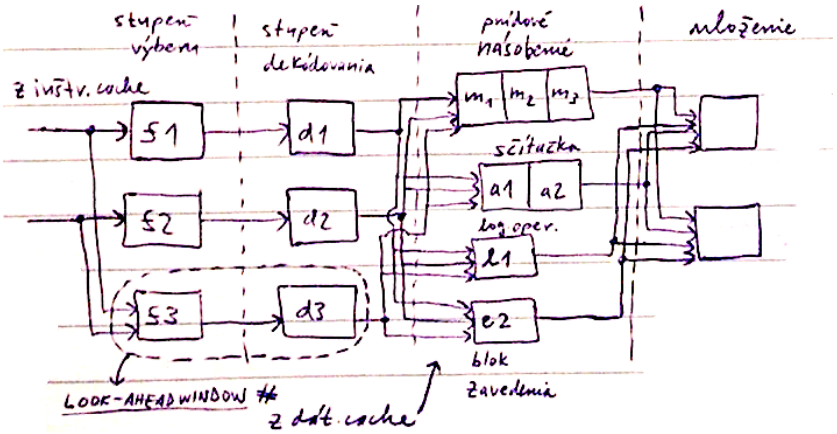
Superskalárny stroj typu n=3:
Má viacero základných skalárnych strojov, ktoré pracujú paralelne.



Superprúdový stroj typu n=3:
Má superprúdové spracovanie, 3x rýchlejšie zadávanie inštrukcií, skrz rozdelenie hodín, kde je každý stupeň rozdelený na 3 podstupne.



Superprúdový stroj na ďalšom obrázku obsahuje 2 prúdy, 4 funkčné jednotky a llokahead window, ktoré umožňuje zadávať inštrukcie mimo poradia.



Inštrukcie môžme zadávať:
- v poradí - nie je problém s dátovou závislosťou, ale je slabšia priepustnosť
- mimo poradia.

Príklady:

```
I1    LOAD R1,A   R1 ← MEM(A)
I2    ADD R2,R1   R2 ← R2+R1
I3    ADD R3,R4   R3 ← R3+R4
I4    MUL R4,R5   R4 ← R4xR5
I5    CMPL R6     R6 ← R6 negovane
I6    MUL R6,R7   R6 ← R6xR7
```

Zadávanie a dokončenie v poradí:

	1	2	3	4	5	6	7	8	9
I1	F1	D1	E2	S1					
I2	F2	D2		A1	A2	S2			
I3		F1	D1		A1	A2	S1		
I4		F2	D2	M1	M2	M3	S2		
I5			F1	D1	E1			S1	
I6			F2	D2		M1	M2	M3	S2

F_i-fetch, D_i-decode, E_i-execute, A_i-add, M_i-multiply, S_i-store

Zadávanie v poradí dokončenie mimo poradia:

	1	2	3	4	5	6	7	8	9
I1	F1	D1	E2	S1					
I2	F2	D2		A1	A2	S2			
I3		F1	D1		A1	A2	S1		
I4		F2	D2	M1	M2	M3	S2		
I5			F1	D1	E1	S1			
I6			F2	D2		M1	M2	M3	S2

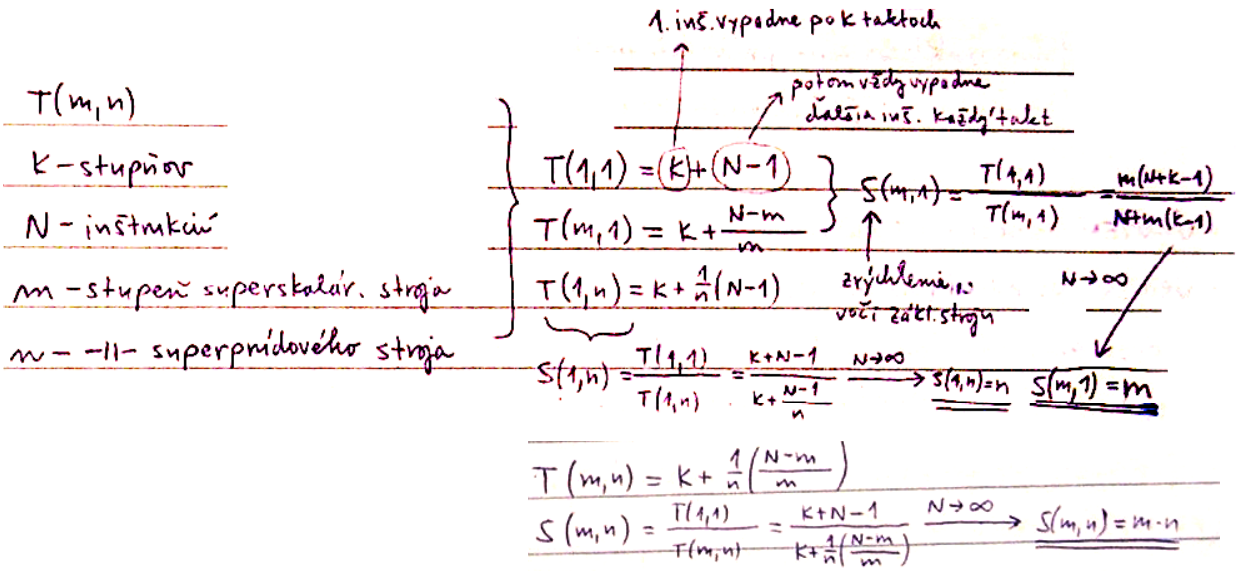
Zadávanie aj dokončenie mimo poradia:

	1	2	3	4	5	6	7	
I1		F2	D2	E2	S2			PIPE2
I2			F2	D2	A1	A2	S2	PIPE2
I3	F1	D1	A1	A2	S1			PIPE1
I4	F2	D2	M1	M2	M3	S2		PIPE2
I5	F3	D3	E1	S1				LOOK AHEAD W.
I6		F1	D1	M1	M2	M3	S1	PIPE1

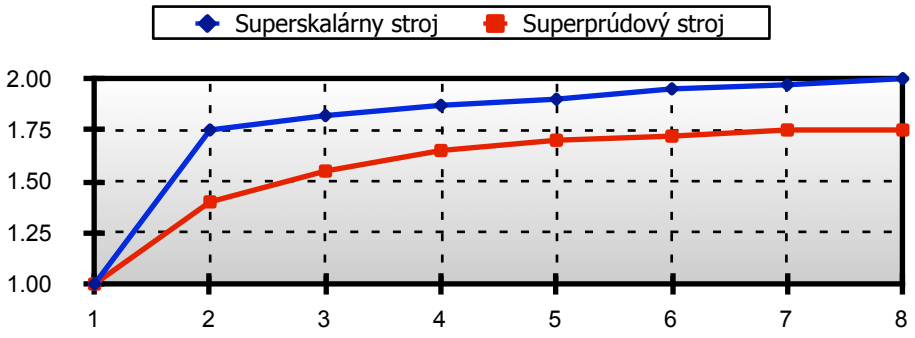
Výkonnosť:

Typ stroja	Základný skalárny k-stupňový stroj T(1,1)	Superskalárny stroj stupňa m T(m,1)	Superprúdový stroj stupňa n T(1,n)	Superskalárny superprúdový stroj stupňa T (m,n)
Cyklus prúdového spracovania	1	1	1/n	1/n
Frekvencia zadávania inštrukcií	1	m	1	m
Latencia základných inštrukcií	1	1	n*1/n	n*1/n
Paralelizmus plne využívajúci prúdový prostriedok	1	m	n	m*n

Odvodenia výkonnosti:



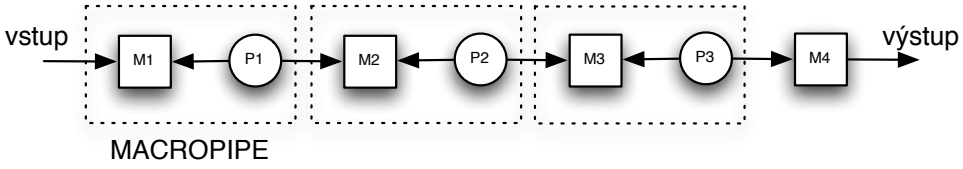
Príklad výkonnosti:
Porovnanie pre superskalárny (1 ≤ m ≤ 4) a superprúdový (1≤ n ≤ 6) stroj. Superprúdové spracovanie má pomalší nábeh (väčšie straty vo výkonnosti pri vetvení). Superskalárne má obmedzenie pri paralelnom spracovaní inštrukcií. Superprúdové stroje majú obmedzenie predstihom hodín a obvodmi na čipe.



iv. Klasifikácia prostriedkov podľa úrovne spracovania

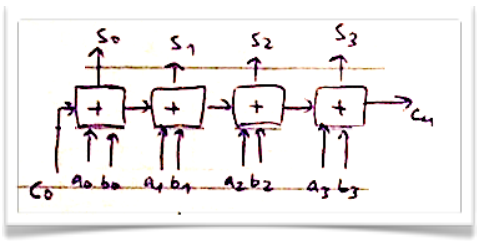
Inými slovami i Handlerova klasifikácia:
- úroveň funkčných blokov
- úroveň inštrukcií
- úroveň procesov (proces sa rozdelí na nadväzujúce podprocesy a tie realizujeme rôznymi prostriedkami)

Stupne pri spracovávaní procesov:

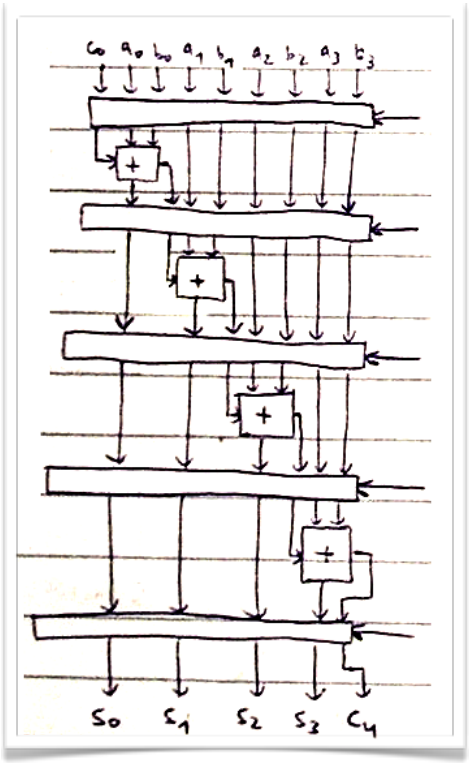


Zostrojenie prúdovej sčítačky:

Klasická 4x1bit sčítačka je nevýhodná, musíme čakať 4x2 taktov skrz prenos carry bitov ak predpokladáme, že každá sčítačka potrebuje 2 takty na presnos. Naopak v prúdovej sčítačke môžme v do nej tlačiť v každom takte nové operandy, prvý súčet máme po 4 taktoch, nasledovné súčty každý ďalší takt.

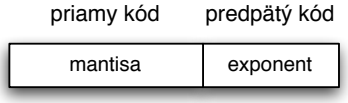


klasická



prúdová

Sčítavanie v pohyblivej rádovej čiarke:



Napr. 8 bit exponent:
127 1111 1111
-128 0000 0000

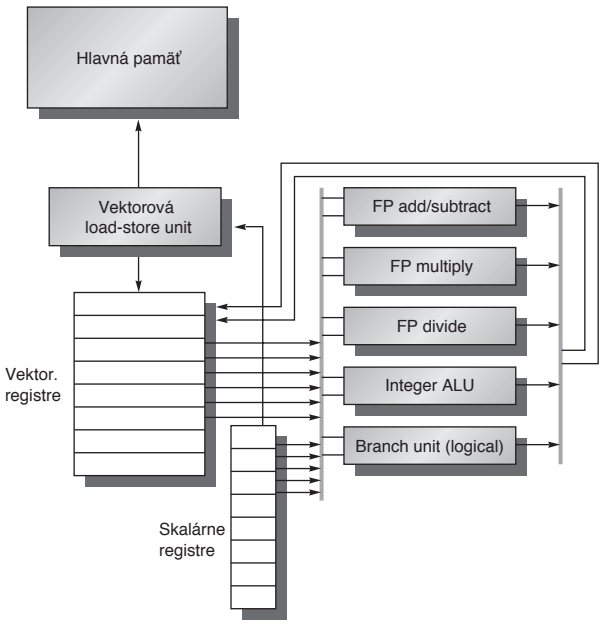
1. stupeň:
    1) zarovnanie exponentov
    2) denormalizácia mantisy s menším exponentom
2. stupeň:
    3) súčet mantís
    4) ošetrenie pretečenia mantisy multiplexom
3. stupeň:
    5) normalizácia mantisy

v. Klasifikácia prostriedkov podľa konfigurácie a riadenia

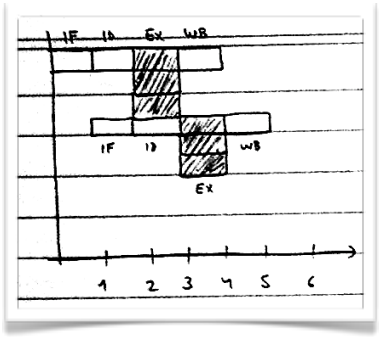
Podľa konfigurácie a riadenia môžme prostriedky klasifikovať na:
- **jednofunkčné** (väčšinou statické) a **viacfunkčné** (viaceré funkcie súčasne, alebo v rôznych časoch)
- **statické** a **dynamické** (ide o schopnosť menenia konfigurácie, poradia vstupného spracovania)
- **skalárne** a **vektorové** (skaláry - operandy, ktoré nemajú medzi sebou súvis)

vi. Architektúra VLIW (Very Long Instruction Word)

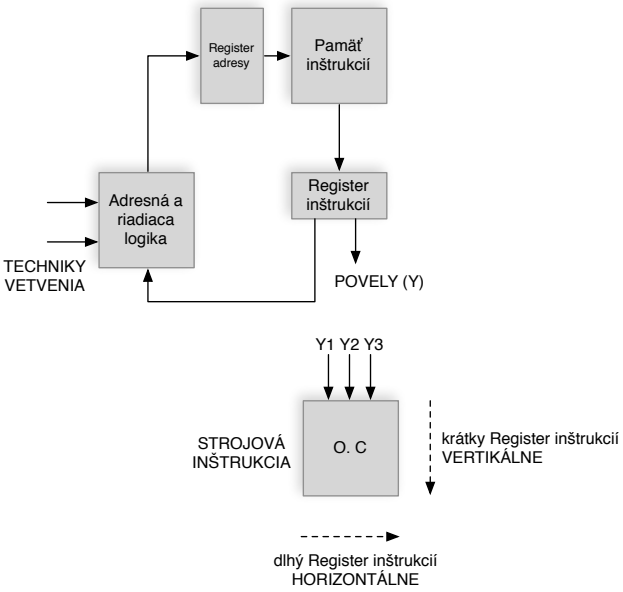
Mikroprogramovanie môžme realizovať dvoma spôsobmi:
- vertikálne mikroprogramovanie - v jednom takte jedna mikro operácia, viacej taktov
- horizontálne mikroprogramovanie - v jednom takte viacero mikro operácii, menej taktov.



Formát inštrukcie obsahuje informácie pre riadenie pre každú UNIT procesora : LS UNIT | FP ADD | ... | INT ALU | BRANCH UNIT. VLIW je zovšeobecnená koncepcia horizontálneho mikroprogramovania a superskalárneho spracovania. Synchronizáciu naplánovanie a platnosť údajov v jednotkách procesora rieši prekladač.



Mikroinštrukcie:



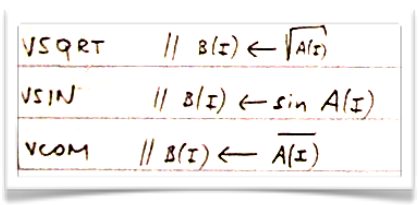
Odlišnosti VLIW od superskalárneho prevedenia:

- dekódovanie VLIW inštrukcií je jednoduchšie
- hustota kódu superskalárneho stroja je vyššia
- superskalárny stroj môže byť kompatibilný s veľkým počtom neparalelných strojov
- VLIW môže byť chápaný ako extrémny prípad superskalárneho stroja, v ktorom sú všetky nezávislé inštrukcie synchrónne zopakované.

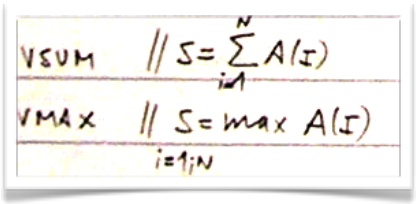
vii. Vektorové procesory

Operandy inštrukcií môžu byť vektory, inštrukcie teda podporujú vektorové operácie:

- vektor-vektor:



- vektor-skalár:



- vektor x vektor -> vektor:

```
VADD    // C(I) = A(I) + B(I)
VMPLY   // C(I) = A(I) * B(I)
VAND    // C(I) = A(I) and B(I)
```

- vektor x skalár -> vektor:

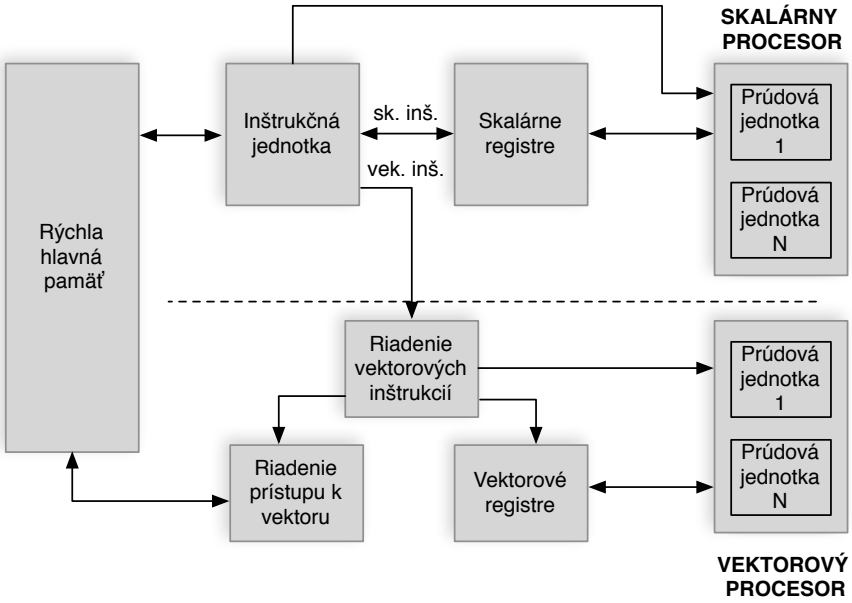
```
SADD    // B(I) ← A(I) + S
SDIV    // B(I) ← A(I) / S
```

Formát vektorových inštrukcií:
- pole operačného kódu - počet zložiek vektora a veľkosť zložky
- operandy (adresujúce vektor/register v procesore, v pamäti - bázová adresa, počet zložiek vektora, veľkosť zložky vektora).

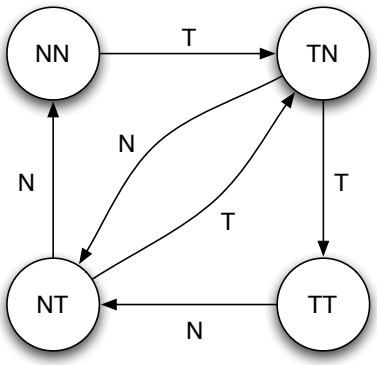
Vektorové prenosy:
- z pamäti do pamäti (TI-ASC, CDC CYBER) - prístup pomocou superslov (128b, 256b)
- z vektorových registrov do vektorových registrov (CRAY, FUJITSU VP200).

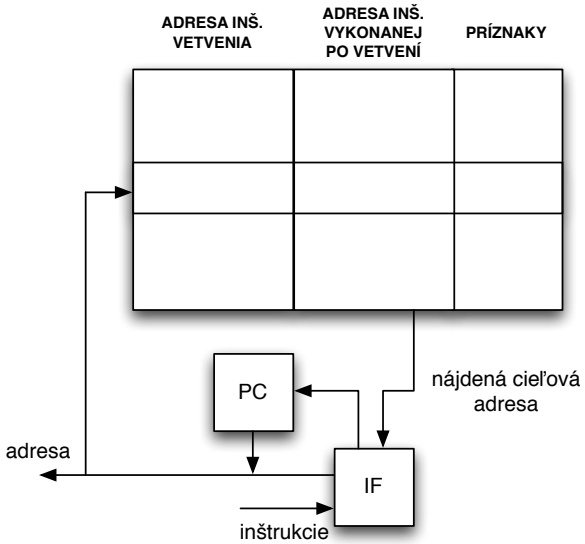
```
      DO 100 I=1,N
      A(I) = B(I) + C(I)
100   B(I) = 2 * A(I+1)
      ↓ zmeloží sa ako

SKALÁRNY STROJ                VEKTOROVÝ STROJ

      INIT I=1                A(1:N) = B(1:N) + C(1:N)
10    READ B(I)               TEMP(1:N) = A(2:N+1)
      READ C(I)               B(1:N) = 2 * TEMP(1:N)
      ADD B(I), C(I)
      STORE A(I)
      READ A(I+1)
      MPLY 2 * A(I+1)
      STORE B(I)
      INCREMENT I
      IF I≤N GOTO 10
      STOP
```

Architektúra typického vektorového procesora:



viii. Predpovedná logika a história vetvenia





T - branch taken - vetvenie realizované
N - branch not taken - vetvenie nerealizované
NN, TN, NT, TT - last two branches not taken / taken

Príznaky môžu uchovávať históriu, inštrukcia sa bude vykonávať viackrát.

HODNOTENIE VÝKONNOSTI POČÍTAČOV

Hodnotenie výkonnosti počítačov je miera schopnosti vykonávať funkciu, na ktorú je určený. Nevyjadruje iba jednu hodnotu, vo všeobecnosti je to množstvo za jednotku času. Väčšinou hovoríme o vektorovej výkonnosti - za akú dobu pri zmene vstupov dostaneme zmenu výstupu (priepustnosť systému, doba odozvy, stupeň využitia...).

i. Gibsonov (inštrukčný) mix

Test, aby sa zabránilo subjektívnym hodnoteniam, prvé hodnotenie výkonnosti HW systémov.

Pre dve základné oblasti výpočtov si zvolíme štatistiku vykonaných inštrukcií programom: l_i (inštrukcia), p_i (pravdepodobnosť výskytu), t_i (doba realizácie). T je stredná doba realizácie danej inštrukcie.

$$p = \frac{1}{T}$$

$$T = \frac{\sum_{i=1}^{n} p_i t_i}{\sum_{i=1}^{n} p_i}$$

ii. Spôsob posudzovania výkonnosti

Dnes posudzujeme výkonnosť pomocou programov, ich úrovne sú:

- reálne programy - odskúšanie v praxi (Ccad, ls)
- jadrá - vybrané časti programov, ktoré sú kritické pre danú aplikáciu (Livermoore loop, Limpack)
- benchmarky - hry
- syntetické benchmarky - podobne ako jadrá, snažia sa simulovať úlohy rôznych aplikačných oblastí - vedecko technické výpočty, oblasť prekladačov (Dhystone, Whetstone)
- systém SPEC

CPI = počet cyklov CPU / počet inštrukcií programu

CPU_{time} = počet inštrukcií programu * CPI * perióda hodín CPU

CPU_{time} = počet cyklov CPU * perióda hodín CPU (ako dlho je úloha realizovaná procesorom)

Počet inštrukcií programu závisí od architekrúry množiny inštrukcií a technológie prekladača. Perióda hodín CPU závisí od technológie použitej na výrobu čipu a organizácie procesora.

RISC CPI = 1, CISC CPI > 1, Superskalárne stroje < 1.

iii. Monitorovanie výkonnosti

Monitor je softvérové riešenie:

- programový monitor - skresľuje reálny beh programu, je lacnejší, nedá sa ísť na nižšiu úroveň ako inštrukcie
- technický monitor - paralelné pripojenie do monitorovaného počítačového systému, presné merania nezávislé od OS, dá sa monitorovať zbernica, komunikačný kanál, nutnosťou je však špeciálne zariadenie a chudobná funkcionalita
- hybridný monitor - kombinácia, vidíme ho ako IO zariadenie, monitoruje vstupy a výstupy

Úlohy riešené monitorovaním:

- meranie na konfigurácii
- meranie činnosti OS
- meranie činnosti používateľa.

PAMÄŤOVÝ PODSYSTÉM POČÍTAČA

Pamäťová hierarchia:
- interná pamäť (elektronická)
  - 0 registre - t_i = 1ns, správa prekladačom
  - 1 cache (L1, L2) - t_i = 10-50 ns, správa technickými prostriedkami
  - 2 hlavná pamäť - t_i = 50-70 ns, správa OS
- externá pamäť (elektro-mechanická)
  - 3 pevný disk - t_i = 5 ms, správa OS/používateľ
  - 4 CD, flash, USB - t_i

Základné parametre:
- t_i - doba prístupu
- s_i - kapacita
- c_i - cena za bit
- b_i - priepustnosť
- x_i - množstvo dát na jeden prístup

Hierarchia:
- $t_i > t_{i+1}$
- $s_i < s_{i+1}$
- $c_i > c_{i+1}$
- $b_i > b_{i+1}$
- $x_i > x_{i+1}$

i. Správa pamäti

Treba zabezpečiť:
- aby v cache bola vždy prítomná požadovaná informácia
- ako uvoľnovať miesto v pamäti nižšej úrovne
- ako zabezpečiť zhodu rôznych inštancií tej istej premennej (koherenciu dát)

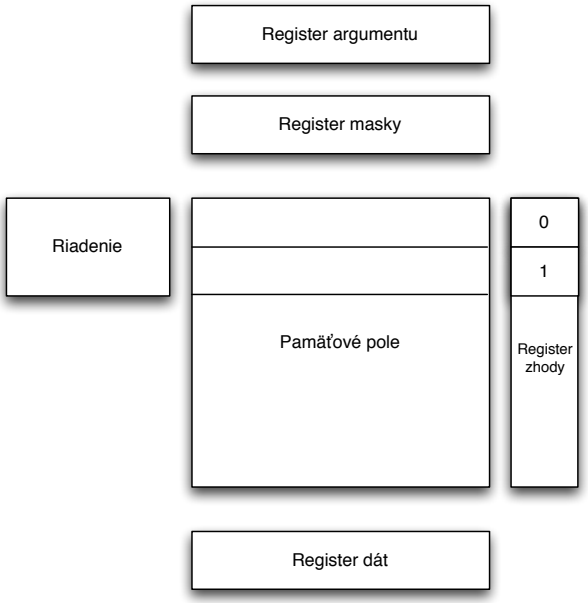
Hit ratio h_i /Miss ratio m_i - pravdepodobnosť, že na i-tej úrovni sú požadované informácie nájdené/ nenájdené.

$m_i = 1 - h_i$
$0 \leq h_0 \leq h_i \leq h_{i+1} \leq h_4 = 1$

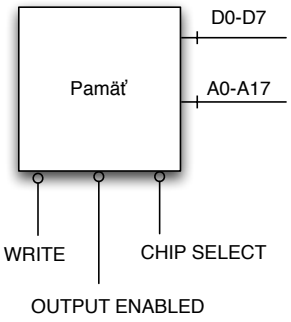
ii. Klasifikácia pamäti podľa prístupovej metódy

Klasifikujeme:
- RAM (Random access Memory) - ľub. prístup
- SAM (Sequentially Access Memory) - páska
- DASD (Direct access storage devices) - disk
- CAS (Content access memory) - tabuľky - asociatívna pamäť

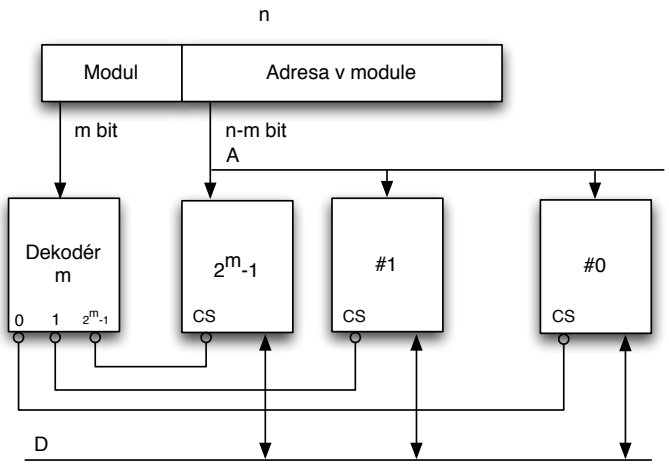
Asociatívna pamäť:



iii. Adresovacie schémy hlavnej pamäti

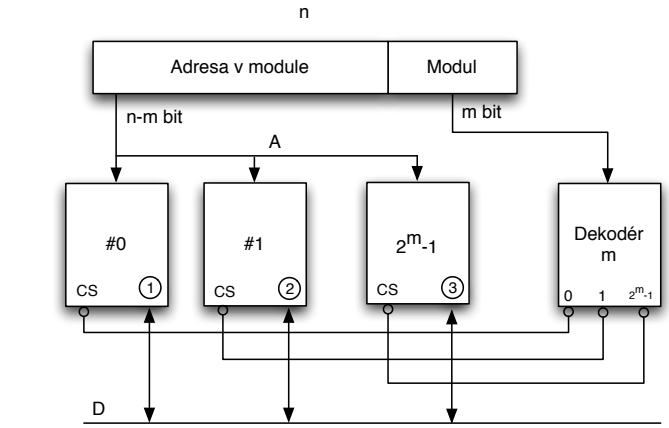


Prepletenie horných bitov adresy:



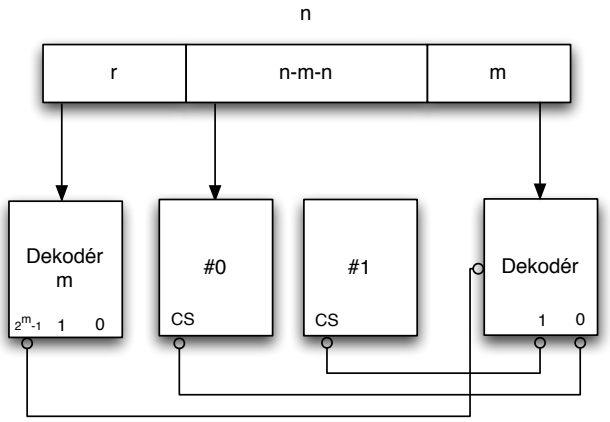
Doba prístupu ako do modulu, je škálovateľný.

Prepletenie dolných bitov adresy:



Doba prístupu je znížená pri lineárnom prístupe k údajom (čím viac modulov, tým rýchlejšie), ak však vypadne jeden modul, tak prestane fungovať. Číta sa paralelne. Doba prístupu je doba prístupu do modulu / počet modulov.

Hybridné koncepcie:

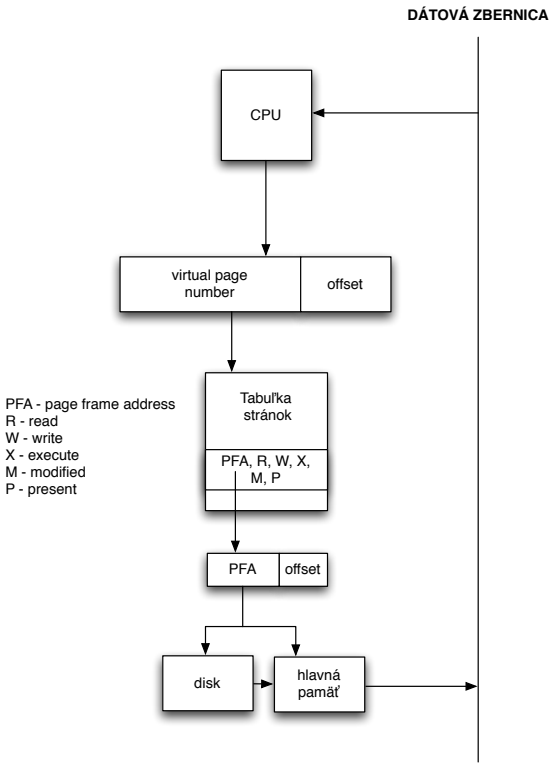


Prvá (vyššia) úroveň predstavuje prepletenie horných bitov, druhá (nižšia) úroveň prepletenie dolných bitov. Je tu realizované dvojúrovňové dekódovanie. Doba prístupu je znížená pri lineárnom prístupe k údajom. Pamäť je škálovateľná.

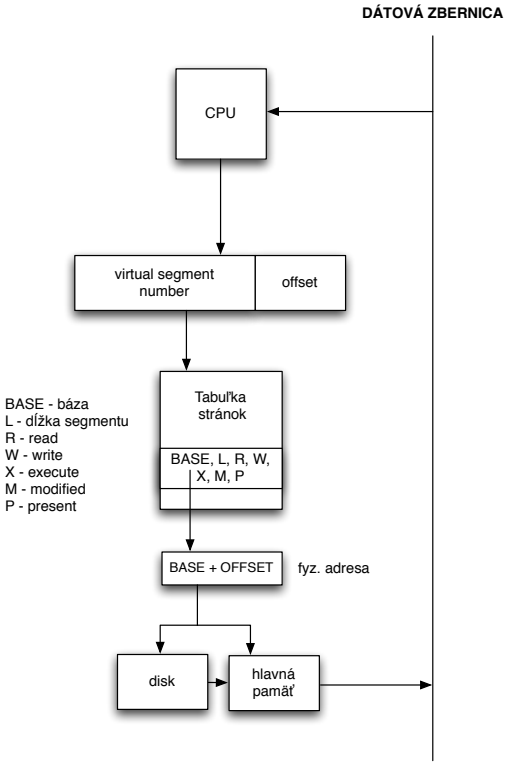
iv. Virtuálne pamäťové systémy - stránkovanie, segmentácia

Koncepciu virtuálnej pamäte má na starosti Memory management unit. Koncepcie mapovania virtuálnej pamäti na fyzickú sú stránkovanie (časti pamäti - stránky sú rovnako veľké) a segmentácia (časti kódu - do fyzickej pamäti zavádzame naraz celý segment rozdielnych dĺžok).

Stránkovanie:



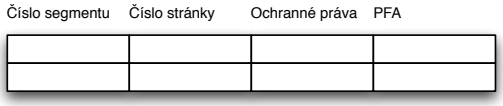
Segmentácia:



Page fault - bit present určuje, či je stránka v pamäti alebo nie, ak nie je nastáva page fault a PFA adresa stránky ukazuje na disk, z ktorého je potrebné stránku načítať a vložiť do hlavnej pamäti.

Segment fault - podobne ako page fault.

Stránkovaný segment:
Je kombináciou predošlých dvoch: preklad virtuálnej adresy -> segment -> stránka.



Správy pamäti:
Intuitívna koncepcia programovej lokality - procesor generuje nenáhodné adresy, zložky:
- časová - generované adresy majú časovú tendenciu
- priestorová - generované adresy majú následnú tendenciu
- sekvenčná.

Vieme voliť veľkosť jednotky, počet jednotiek.

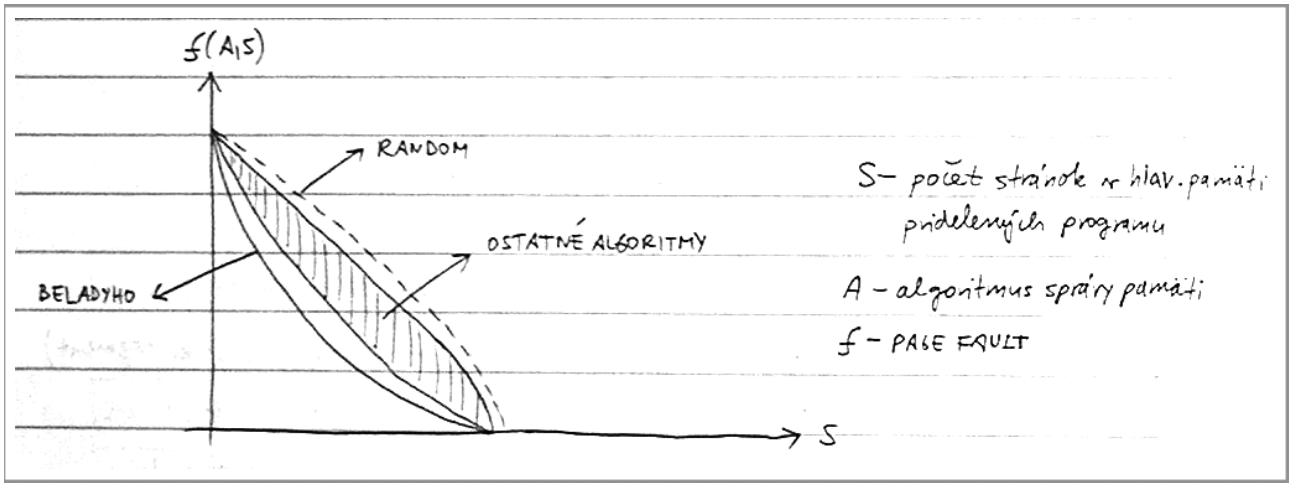
Tabuľka stránok/segmentov sa môže nachádzať v hlavnej pamäti, disku, čiastočne v Memory Management Unit procesora.

Vo fyzickej pamäti vzniká fragmentácia:
- interná fragmentácia - čiastočné zaplnenie stránok
- zavádzanie tabuľky do pamäti.

Správa stránkovanej pamäti:
Predpokladajme, že poznáme postupnosť stránok, v akej sa bude program vykonávať, máme obmedzenú kapacitu stránok v hlavnej pamäti, nastavajú PAGE FAULTy.
Beladyho optimálny algoritmus:
- nutný predpoklad vedomosti o vykonávaní programu
- nepraktický
- slúži však na porovnanie s ostatnými algoritmami.

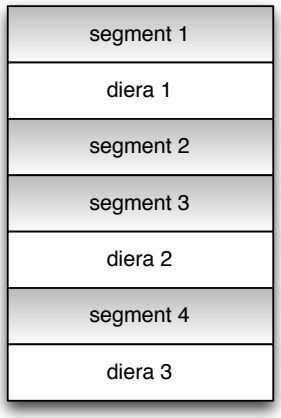
Používané algoritmy:
- LRU (Least Recently Used) - vyhodíme stránku, ku ktorej sme pristupovali najďalej v minulosti
- LFU (Least Grequently Used) - vyhodíme tú, do ktorej sme najmenej pristupovali od posledného Page Faultu
- FIFO, LIFO
- Random

Závislosť page fault od použitého algoritmu a počtu stránok f(A,s):



Technikou redukujúcu page fault je predvýber stránky - ak vyberáme stránky z disku, nevyberáme po jednej, ale vo viacerom počte.

Správa segmentovanej pamäti:

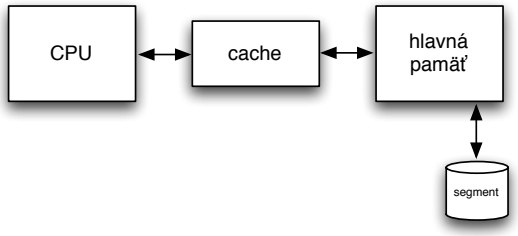


Algoritmy zavádzania segmentov do pamäti:
- best fit - segment vložíme do diery najbližšie podobnou veľkosťou
- worst fit - segment uložíme do najväčšej diery
- first fit - vložíme do diery, ktorá je prvá väčšia ako segment
- buddy algoritmus - snažíme sa nájsť dieru 2x veľkú ako segment

Kompakcia hlavnej pamäti - dáme diery za seba a vytvoríme tak veľkú súvislú dieru a ak stále ani tam nemožno uložiť zavádzaný segment, tak nájdeme segment, ktorý z pamäti vyhodíme.

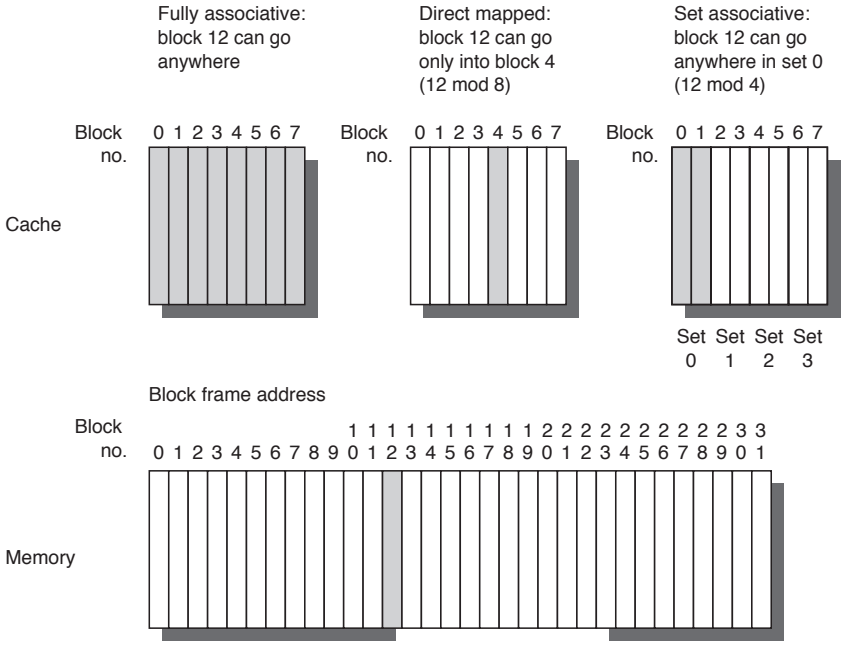
v. Rýchla vykonávacia pamäť (cache)

Slúži na vyrovnávanie rýchlosti prístupu rôznych pamätí, napr. medzi CPU a hlavnou pamäťou.



Blok cache je elementárna jednotka pamäti v cache, obsahuje niekoľko slov, teda je veľký n-násobkom slova. Poznáme rôzne mapovania:

- plne asociatívne mapovanie - fyz.adresa = tag (5b) + offset, blok hlavnej pamäti môže byť na ľubovoľnej pozícii vrámci cache, výhodou je maximálna priepustnosť, nevýhodou však nutnosť asociatívnej pamäti
- čiastočne asociatívne mapovanie - fyz adresa = tag (3b) + index (2b) + offset, vrámci jedného SETU aplikujeme plne asociatívne mapovanie ale medzi setmi aplikujeme priame mapovanie
- priame mapovanie - fyz adresa = tag (2b) + index (3b) + offset, blok hlavnej pamäti môže byť v len v práve jednom bloku v cache, nižšia priepustnosť, ale jednoduchá správa
- sektorové mapovanie - fyz adresa = tag (3b) + blok (2b) + offset, súvislý počet blokov vytvára sektor, sektor môže byť v cache ľubovoľne umiestnený, ale pozície blokov v rámci sektora sú fixné.



Block fault - na výber bloku pri nedostupnom bloku môžme použiť algoritmy:

- RANDOM - náhodný výber bloku
- LRU (Least Recently Used) - HW realizácia je pomocou matice preklápacích obvodov n*n, ak máme prístup do i-tého bloku, tak nastavíme preklápací obvod v i-tom riadku na 1 a v i-tom stĺpci preklápacieho obvodu vynulujeme, vyhadzovaný blok tak má najmenej jednotiek v riadku
- FIFO

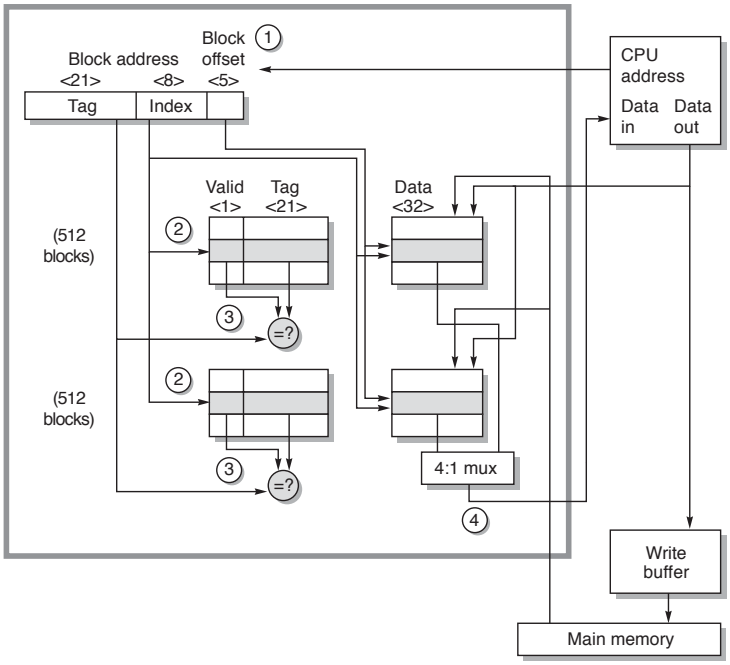
Miss ratio cache pri použitých algoritmoch:

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Zápis do cache (blok je v cache):
- WRITE THROUGH (zápis cez cache) - zároveň sa zapisuje aj do hlavnej pamäti a tá je pomalšia - write buffer (slúži na bufferovanie požiadaviek do zápisov do hlavnej pamäti)
- WRITE BACK (zápis späť) - ak zmeníme blok v cache nastaví sa mu dirty bit a pri jeho obetovaní pri block fault sa zapíše späť do hlavnej pamäti - efektivita.

Zápis do cache (blok nie je v cache):
- WRITE ALLOCATE - ak nám blok chýba, zavedieme ho do cachce, väzba s WRITE BACK
- NON-WRITE ALLOCATE (zápis okolo) - ak nám blok chýba tak ho nedávame do cache, ale rovno do hlavnej pamäte, väzba s WRITE THROUGH.

Príklad cache na procesore Alpha AXP21064:



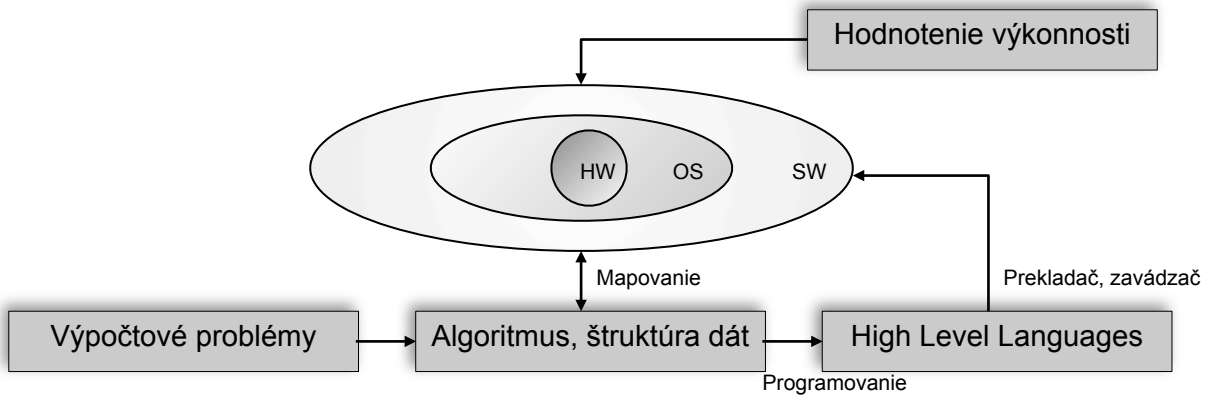
Opatrenia zvyšujúce priepustnosť cache:
- redukcia Miss ratio - pomocou architektúry cache
- redukcia Miss penalty - pomocou správy pamäti
- redukcia času v prípade nájdenia informácii v cachce - prípad hitu.

Multi-level caches generally operate by checking the smallest Level 1 (L1) cache first; if it hits, the processor proceeds at high speed. If the smaller cache misses, the next larger cache (L2) is checked, and so on, before external memory is checked. L1 je cache priamo na čipe, L2 mimo čipu.

MODELY PARALELNÝCH POČÍTAČOV

Typy paralelného spracovania:
- sumiltánnosť
- prúdovosť
- súbežnosť.

Prvky moderných PC:



Výpočtové problémy paralelného spracovania:
- numerické extenzívne výpočty
- spracovanie tranzakcií
- rozsiahle databázy
- logické odvodzovanie.

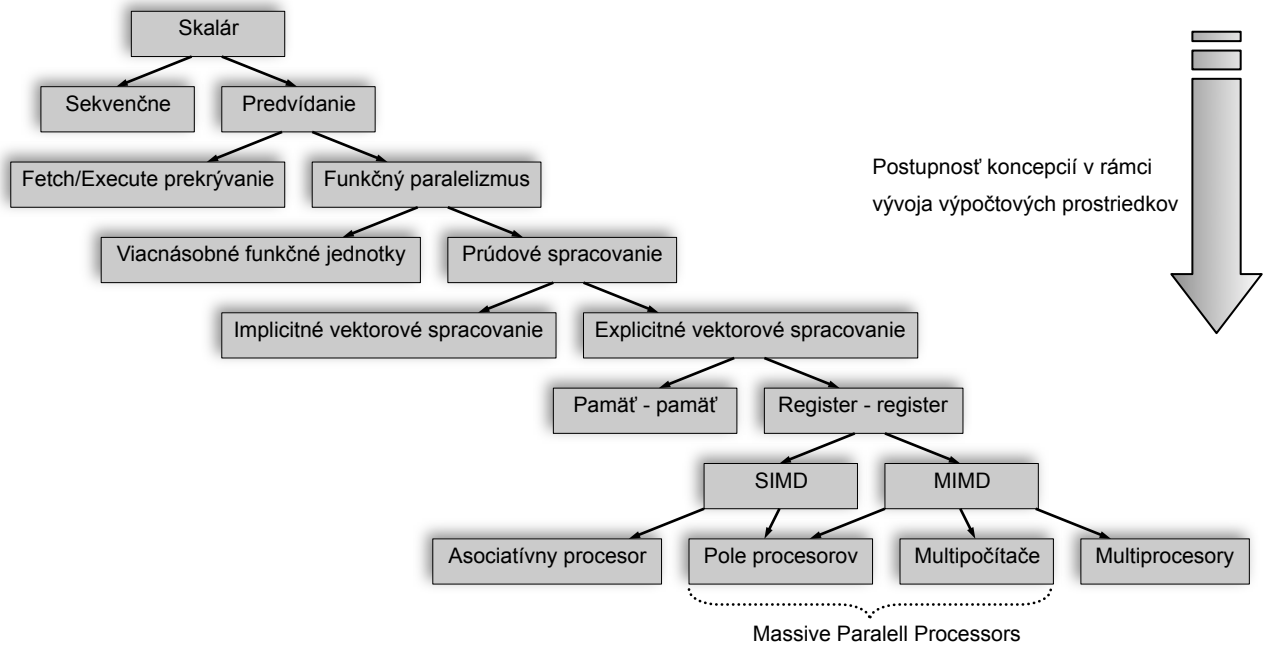
Algoritmy a dátové štruktúry:
- numerické aplikácie - determinanty alg., pravidelné štruktúry dát
- spracovanie symbolov.

Paralelizmus aplikácie:
- návrh algoritmu
- v čase programovania
- v čase prekladu
- v čase behu.

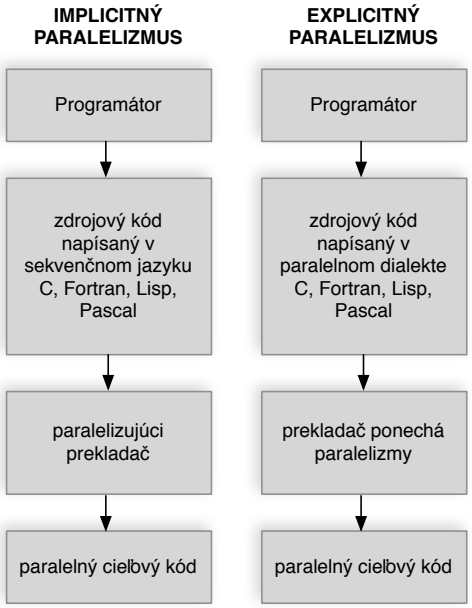
Podporné nástroje:
- programovacie jazyky - nové jazyky (OCCAM), paralelné rozšírenie klasických programovacích jazykov (Paralel C).

Podpora prekladača:
- preprocesor
- prekompilátor
- paralelizujúci prekladač.

Vrstvy vývoja paralelných aplikácií:



Implicitný a explicitný paralelizmus pri programovaní paralelných systémov:



i. Multiprocesory a multipočítače - MIMD

Multiprocesory komunikujú cez spoločné premenné, viaceré procesory sú ovládané jedným OS. Multipočítače komunikujú pomocou správ.

FLYNN: procesor, riadiaca jednotka, pamäť, tok dát, tok inštrukcií.

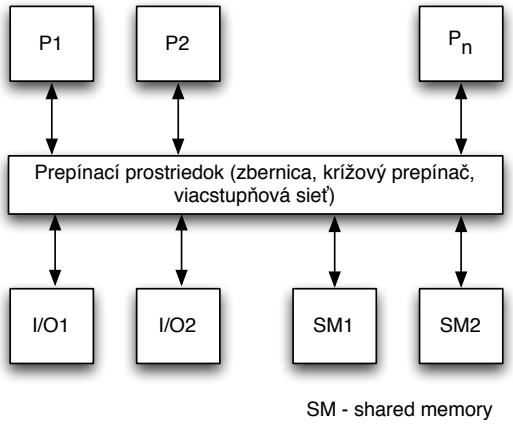
Kategorizácia strojov MIMD (Multiple Instruction stream, Multiple Data stream):
- multiprocesory (jednotný adresový priestor, spoločná pamäť):
  - s distribuovateľnou pamäťou (škálovateľné)
  - s centrálnou pamäťou (neškálovateľné)
- multipočítače (viacnásobný adresový priestor, posielanie správ)
  - distribuovateľné (škálovateľné)
  - centrálne (škálovateľné)

Multiprocesor so spoločnou pamäťou:

Ak majú všetky procesory rovnaký prístup k I/O zariadeniam, tak je to symetrický multiprocesor, nesymetrické delíme na Master a Attached.

Rozdeľujeme modely:
- UMA - uniform memory access
- NUMA - non uniform memory access
- COMA - cache only memory access.

Model UMA:
Procesor pristupuje k pamäti rovnakým a rovnocenným spôsobom.



Príklad: Odhadnite výkonnosť multiprocesora, ktorý vykonáva sekvenčný program na UMA. L2, L4, L6 trvajú 1 strojový cyklus + k strojových cyklov na každú medziprocesorovú komunikáciu na spoločnú pamäť, L1, L3, L5, L7 sú pre jednoduchosť zanedbané. Kód sa nachádza v cache a dáta v hlavnej pamäti a cache.

```
L1    DO 10 I=1,N
L2    A(I)=B(I)+C(I)
L3    CONTINUE
L4    SUM=0
```

```
L5    DO 20 J=1,N
L6    SUM=SUM+(A(J))
L7    CONTINUE
```

Na jednoprocesorovom systéme sa L2 vykoná N-krát, zaberie teda N taktov. Vynulovanie sumy L4 sa vykoná iba raz, zaberie 1 takt. Nakoniec pripočítavanie položky L6 poľa k sume sa vykoná N krát, zaberie teda N taktov. Celý program teda na jednoprocesorovom systéme pri uvažovaní uvedených zjednodušení trvá 2N+1 taktov ~ 2N. Na multiprocesorov systéme UMA s M procesormi rozdelíme cykli realizácie do M sekcií s dĺžkou L=N/M:

```
DO    ALL   k=1,M
            DO 10 I=L*(K-1)+1, K*L
            A(I)=B(I)+C(I)
      10    CONTINUE
            SUM(k)=0
            DO 20 J=1,N
            SUM(k)=SUM(k)+(A(L*(K-1)+J))
      20    CONTINUE
END   ALL
```

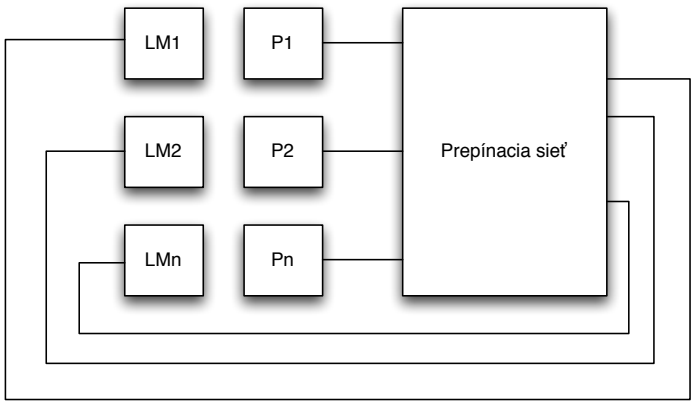
Jednotlivé parciálne súčty sa nakoniec nasčítavajú do výsledku (systémom binárneho stromu) – pokiaľ **z** je počet úrovní binárneho stromu (hĺbka), **k** je doba prístupu do spoločnej pamäte, potom súčet parciálnych súm zaberie z.(k+1). V našom prípade je hĺbka stromu $\log_2 M$.

Program vykonáme ako $2L + z*(k+1) = 2N/M + (k+1)\log_2 M$.

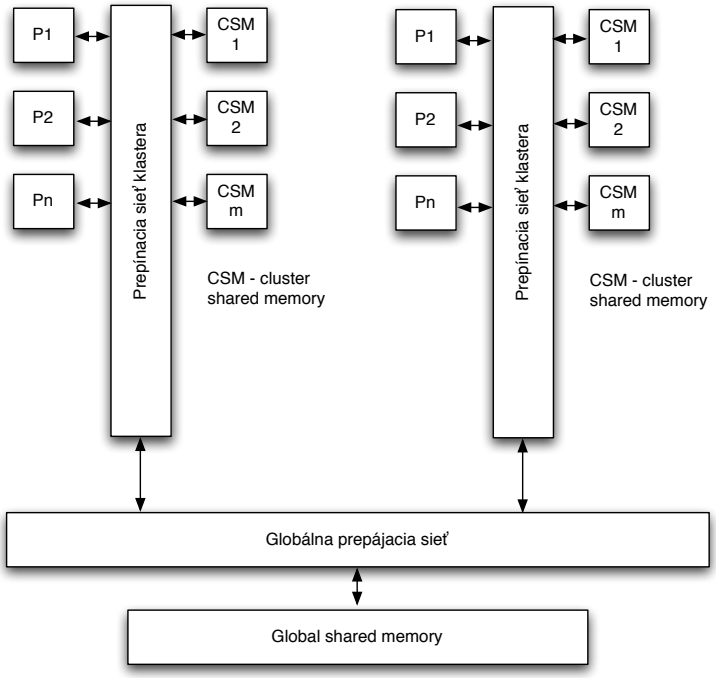
Pre $N=2^{20}$:
- sekvenčné vykonanie = 2^{21}
- paralelné vykonanie pri k 200 cyklov a M 256 = $2.20^{20}/2^8 + (201) * 8 = 2^{13} + 1608 = 9800$ cyklov.

Zrýchlenie máme $2^{21}/9800 = 214$, účinnosť 214/256 = 84%.

Model NUMA:
Každý procesor vidí všetky dáta, ale do jednej pamäti má rýchlejší prístup ako do druhej. Každý procesor pristupuje k lokálnym pamätiam cez prepínaciu sieť. Jednotlivé lokálne pamäte vytvárajú globálnu pamäť (zdieľanú). Daný procesor vidí globálnu pamäť ako súhrn všetkých lokálnych pamätí.

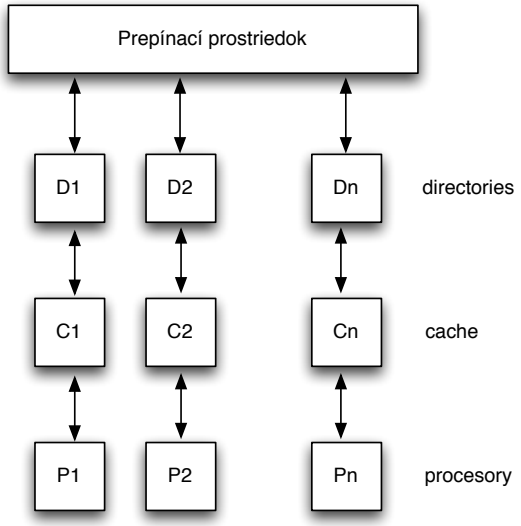


LM - local memory, vytvárajú spoločnú pamäť.



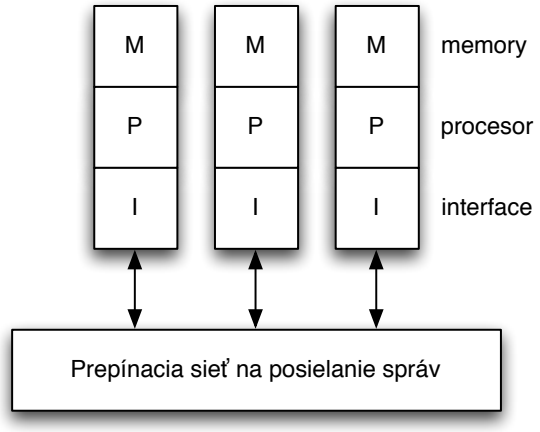
Ďalším spôsobom realizácie NUMA sú klusterové architektúry. Všetky procesory vidia GSM (Global Shared Memory). Jeden procesor teda vidí global shared memory a svoj cluster shared memory, nevidí teda lokálne pamäti ostatných procesorov, z toho vyplýva, že medzi klastrami je zakázaný prístup do LM.

Model COMA:



Spoločná pamäť je distribuovaná do cache, všetka cache vytvára adresovateľný priestor, ktorý sa adresuje pomocou distribuovaného adresára.

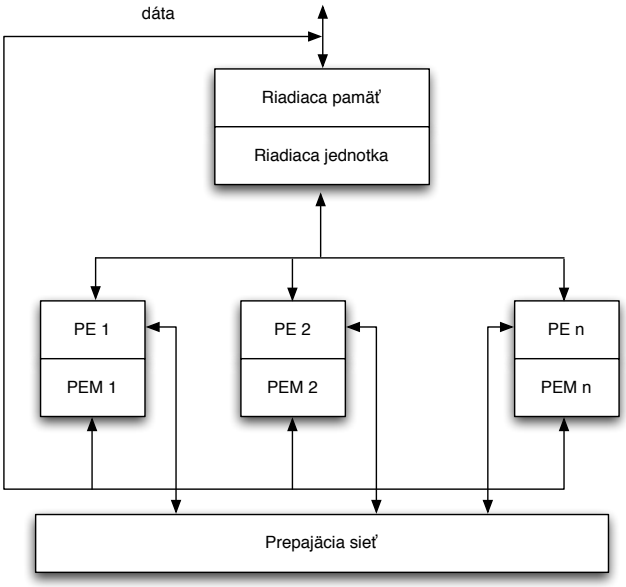
Multipočítač s distribuovanou pamäťou:



Každý uzol je autonómny počítač, procesor vie pristúpiť iba k vlastnej pamäti - model NORMA (No Remote Memory Access). Dobrá škálovateľnosť, ale ťažšia práca programátorov, vyžadujú sa efektívne prekladače a distribuované OS.

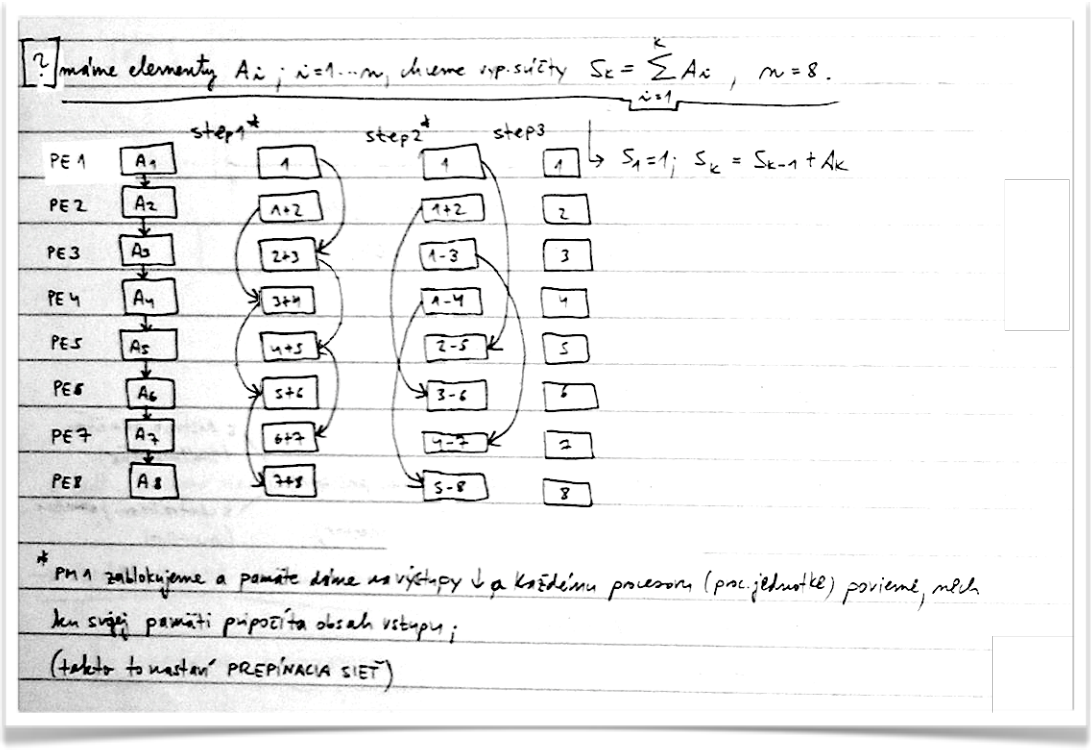
ii. Maticové procesory - SIMD

Po skončení vykonávania sa dáta vyzbierajú z lokálnej pamäti, sú vhodné na rýchle, špecializované výpočty.

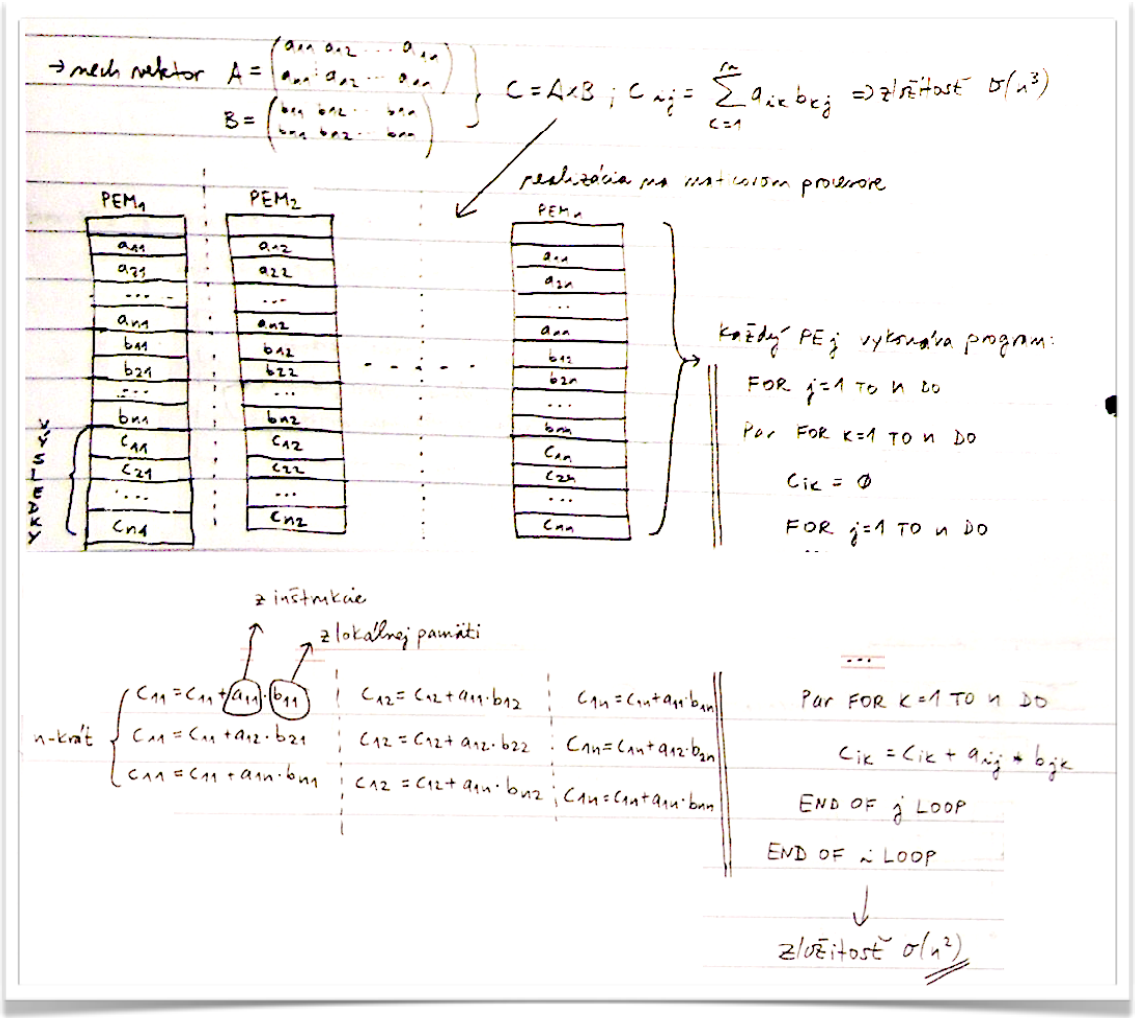


PE - procesorový element
PEM - lokálna pamäť

Príklad:



Príklad:



iii. Charakteristika prepínacích sietí

Režim činnosti:
- synchrónny - typický pre SIMD
- asynchrónny - požidavky na komunikáciu prichádzajú asynchrónne, typický pre MIMD
- kombinovaný.

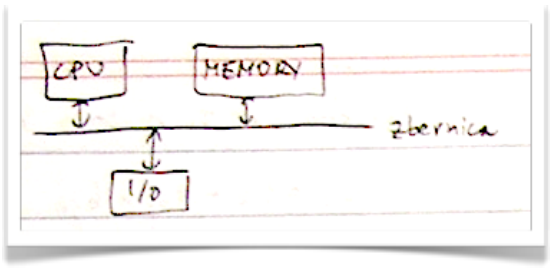
Riadiace stratégie:
- centralizovaná - špeciálny prvok riadi riadenie, typický pre SIMD
- distribuovaná - každý prvok si na základe zistenej infraštruktúry rozhodne sám, typický pre MIMD.

Prepínacia metóda:
- obvodové prepínanie - medzi vysielačom a prijímačom sa vytvorí spojenie, typický pre SIMD
- prepínanie paketov - vysielanie paketov a čakanie na odpoveď, typický pre MIMD
- integritné prepínanie - kombinácia.

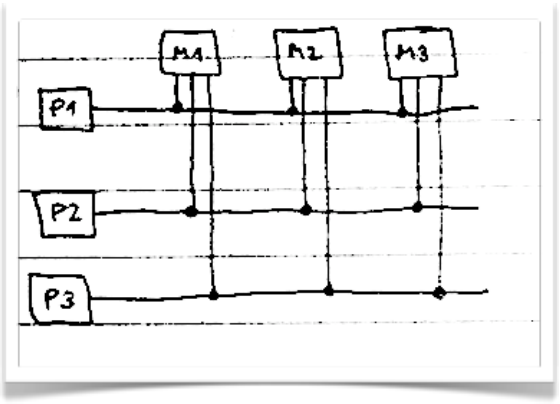
Topológia siete:
- statické - prvky sú fixné
- dynamické - rekonfigurovateľné prvky, typické pre SIMD.

Prepínacie prostriedky pre SIMD:
- zbernica - malá priepustnosť, preto existujú viaczbernicové systémy, protokol pre pridelovanie, porucha znamená zlyhanie systému - single point of failure
- viacportové moduly - prepínanie riešené v moduloch, nutné veľa prepojení
- krížový prepínač - podporuje všetky permutácie prepojení, drahé, ale rýchle
- viacstupňové siete - základný stavebný element, má charakter krížového prepínača, viacero stupňov:
  - blokujúce (Baseline) - nedá sa prepojiť voľný vstup s voľným výstupom, prestavením sa dá odstrániť blokovanie
  - prestaviteľné (Benešove, Crossbar switch) - dá sa prepojiť ľub. vstup s ľub. výstupom
  - neblokujúce (Closet) - najdokonalejšie.

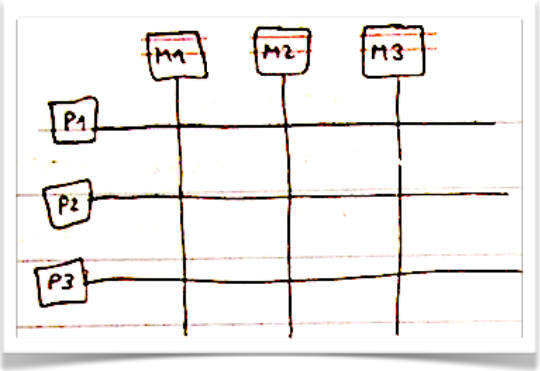
Zbernica:



Viacportové moduly:

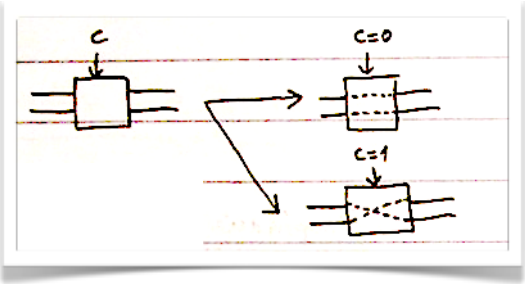


Krížový prepínač (NxM):



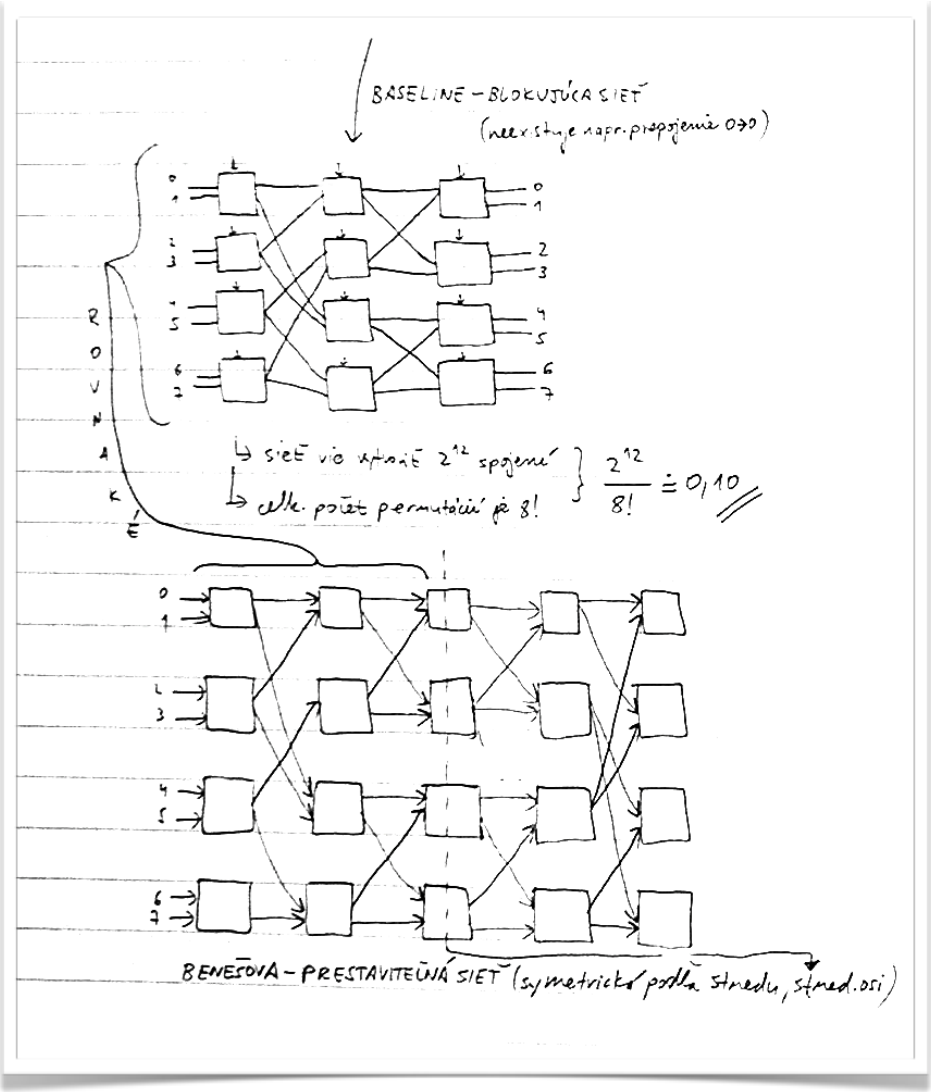
v danom čase zabezpečí jednu z permutácii (1,2,3), (3,2,1), (2,3,1), (1,3,2), (2,1,3), (3,1,2).

Viacstupňové siete:

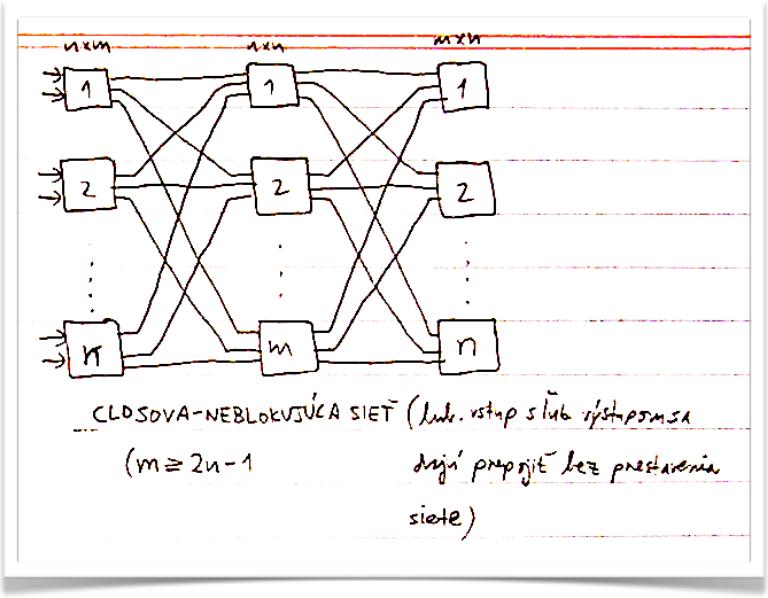


kompromis medzi krížovým prepínačom a zbernicovým systémom, prvky sú elementárne krížové prepínače

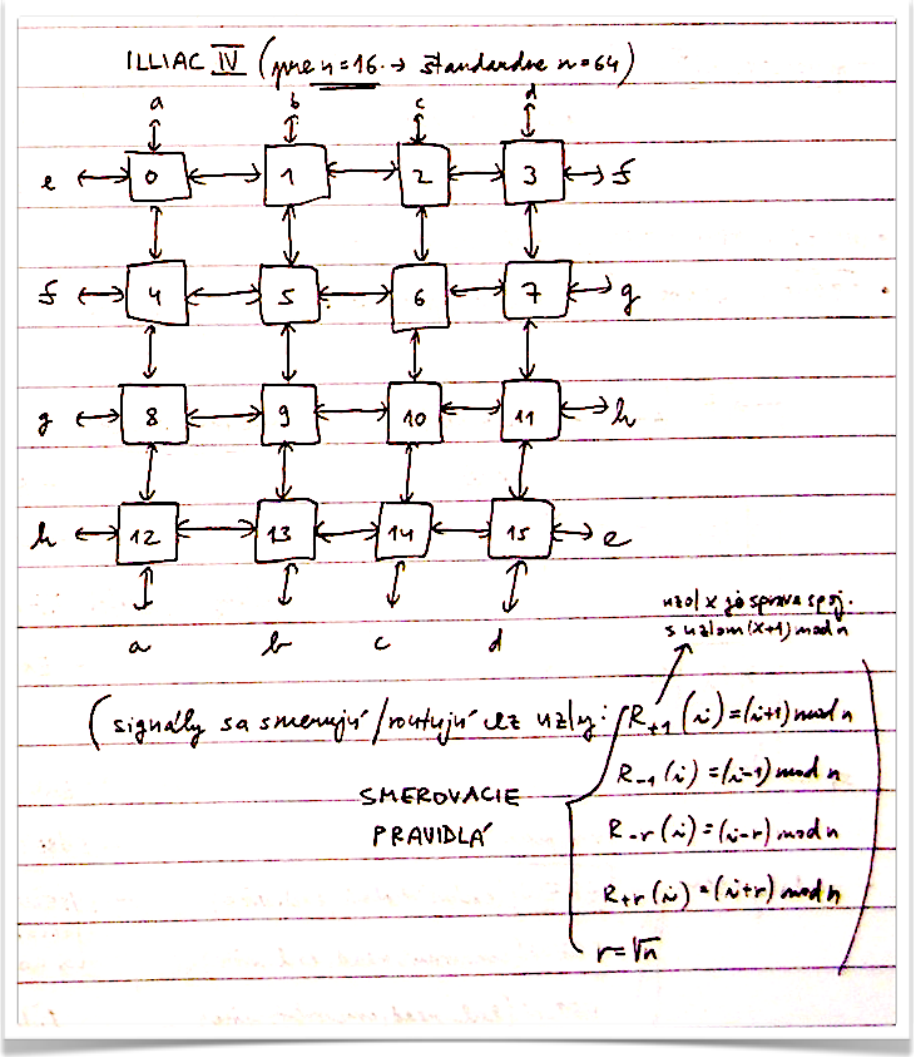
Blokujúca sieť a Benešova sieť:

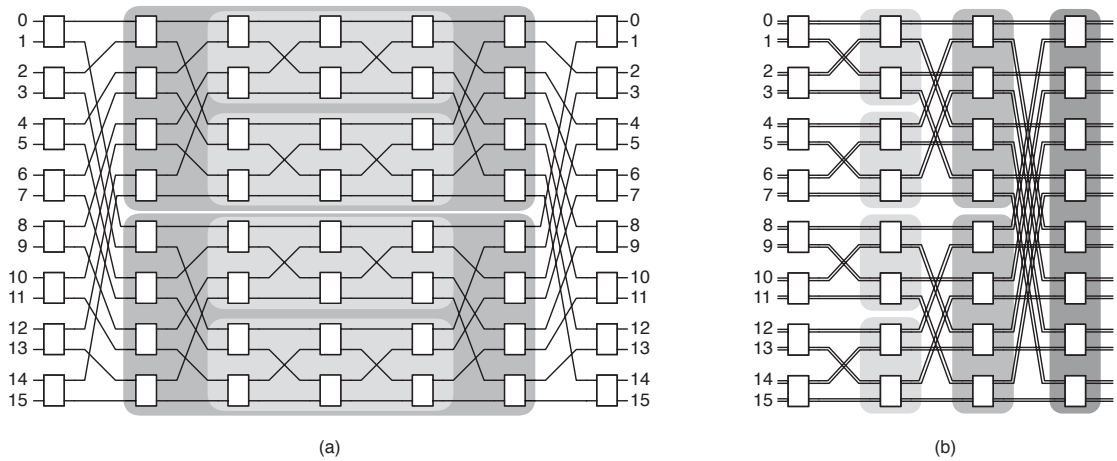


Closova neblokujúca sieť:



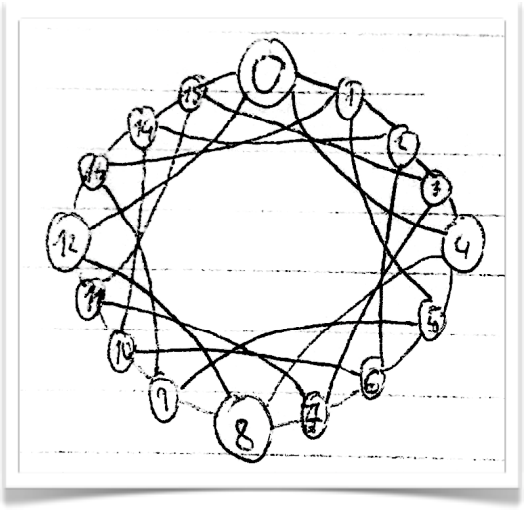
Prepínacia sieť ILIAC IV:





Two Beneˇs networks. (a) A 16-port Clos topology, where the middle-stage switches shown in the darker shading are implemented with another Clos network whose middle-stage switches shown in the lighter shading are implemented with yet another Clos network, and so on, until a Beneŝ network is produced that uses only 2 × 2 switches everywhere. (b) A folded Beneŝ network (bidirectional) in which 4 × 4 switches are used; end nodes attach to the innermost set of the Beneŝ network (unidirectional) switches. This topology is equivalent to a fat tree, where tree vertices are shown in shades.

Premostený kruh:



iv. Modely PRAM

PRAM (Parallel random access machine), je to teoretický model nezávislý od fyzickej architektúry. Je to idealizovaný paralelný PC, ktorý má 0 prístupovú dobu a aj oneskorenie (SIMD).

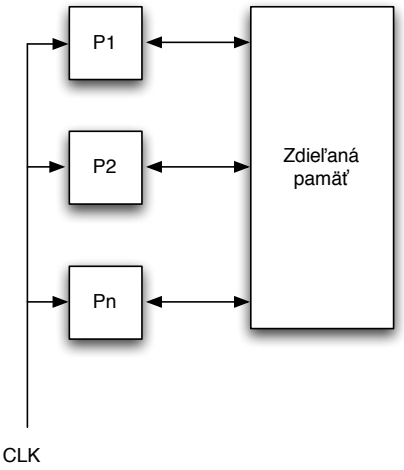
Časová zložitosť algoritmu:

- veľkosť problému S
- časová závislosť g(s) je rádu $\sigma(f(s))$ ak existujú kladné konštanty c a s_o také, že g(s) ≤c * f(s) pre všetky s > s_0
- f(s) môže byť polynóm (alg. zložitosti triedy P), alebo konštanta alebo exponenciála.

Priestorová zložitosť algoritmu:
- odhad vyžadovanej pamäti pre algoritmus
- odvodenie analogicky ako časová.

Triedy algoritmov:
- P polynomiálna zložitosť
- NP nepolynomiálna zložitosť - neexistuje algoritmus na vypočítanie výsledku, ale ak ho poznáme, dokážeme ho v čase P overiť.

P < NP
NP-P nie je výpočtovo zvládnuteľná
NP-P=NPC je alebo nerovné nule ?



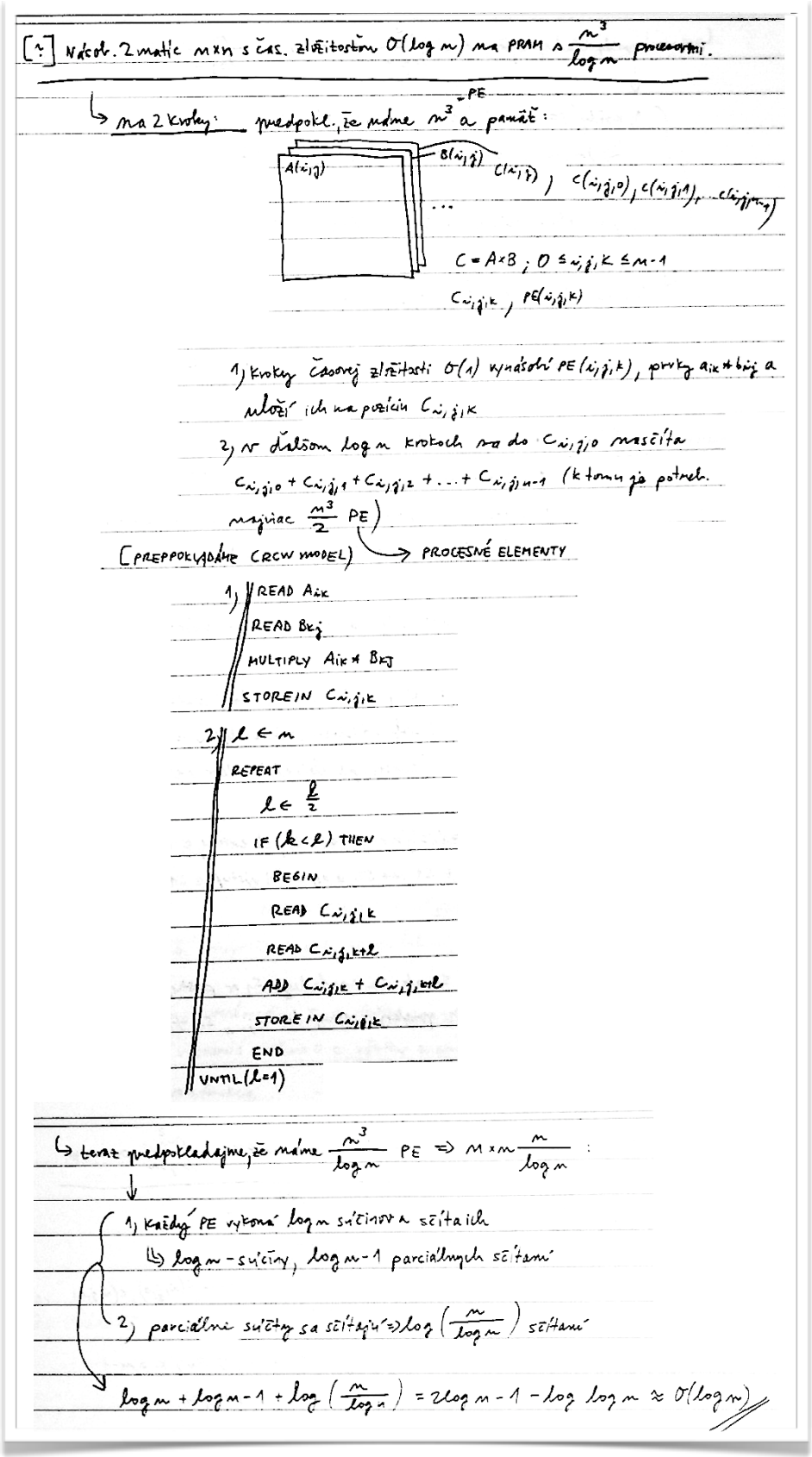
Režimy činnosti:
- READ MEMORY
- EXECUTE
- WRITE MEMORY

Modely súvislých čítaní z pamäti:
- EXCLUSIVE READ (ER) - najviac 1 CPU číta
- EXCLUSIVE WRITE (EW)
- CUNCURENT READ (CR)- súbežné čítanie
- CONCURENT WRITE (CW) - súbežný zápis, ak sa zapisujú na to isté miesto dve hodnoty zostane v pamäti rovnaká hodnota, ľubovoľného z nich alebo hodnota zapísaná CPU s vyššou prioritou

Modely PRAM:
- EREW
- CREW
- ERCW
- CRCW - algoritmy na CRCW môžu byť simulované algoritmami EREW.

Najlepší EREW algoritmus na n procesoroch bude nie menej ako $\sigma(\log n)$ krát pomalší ako algoritmus CRCW, teda EREW=CRCW*logn.

Príklad:



Použijme PRAM model s procesormi.

Predpokladajme mód CREW. Sledujme ako klesne zložitosť.

Najprv uvažujme o n^3 procesných elementov, PE(i,j,k) definuje jednoznačne konkrétny PE
A(i,k) x B(k,j) [i-ty riadok matice A násobený j-tym stĺpcom matice B] = C(i,j,k)

Za log n taktov sa nasčítava C(i,j,0) = C(i,j,0) + C(i,j,1) + ... + C(i,j,n-1)

Použijeme teda $n^3/2$*PE v prvom cykle nasčítavania, dokopy n^2*(n - 1) súčtov.

PE(i,j,k) ... n * n * n/log n , log n súčinov a sčitaní.

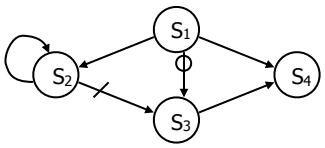
v. Programové a sieťové vlastnosti paralelných PC

Podmienky (ovplyvnenie) paralelizmu:
- výpočtový model pre paralelné počítanie
- medziprocesorová komunikácia v paralelnej architektúre
- systémová integrácia pre zahrnutie paralelného systému do všeobecného výpočtového prostredia.

Dátová závislosť, závislosť prostriedkov - ak preskúmame závislosť medzi príkazmi (grafickej závislosti), tak analýzou grafu vieme zistiť možnosti paralelizácie alebo vektorizácie:
- toková závislosť - príkaz S2 ke tokovo závislý na príkaze S1, ak existuje vykonateľná postupnosť príkazov z S1 na S2 a aspoň jeden výstup z S1 vstupuje do príkazu S2, S1 $\rightarrow$S2
- antizávislosť - S2 je antizávislý na S1 ak S2 nasleduje S1 v postupnosti vykonávaných príkazov a ak výstup S2 prekrýva vstup S1. S1 $\not\rightarrow$ S2
- výstupná závislosť - S1 a S2 sú výstupne závislé, ak produkujú do tej istej premennej, S1 $\circ\!\rightarrow$ S2
- I/O závislosť - ak S1 a S2 pristupujú do toho istého súboru
- neznáme závislosti - ostatné závislosti

Majme príkazy S_1 .. S_4. A, B sú lokácie v pamäti, R sú registre.

```
S1 ... Load R1,A ;  S2 ... Add R2,R1;
S3 ... Move R1,R3;  S4 ... Store B,R1;
```



Riadiaca závislosť - prejaví sa počas behu programu runtime, môže sa vyskytovať v cykle, medzi premennými cyklu:

Majme program vo FORTRANe:
- nezávislá - iterácie nie sú závislé z pohľadu riadenia

```
DO 10  I = 1, N
   A(I) = C(I)
   IF (A(I) .LT. 0) A(I) = 1
10 CONTINUE
```

- závislá - iterácie sú z pohľadu riadenia závislé, nevieme to zorganizovať tak, aby výpočet bežal paralelne

```
DO 10  I = 1, N
   IF (A(I-1) .EQ. 0) A(I) = 0
10 CONTINUE
```

vi. Berštеinove podmienky

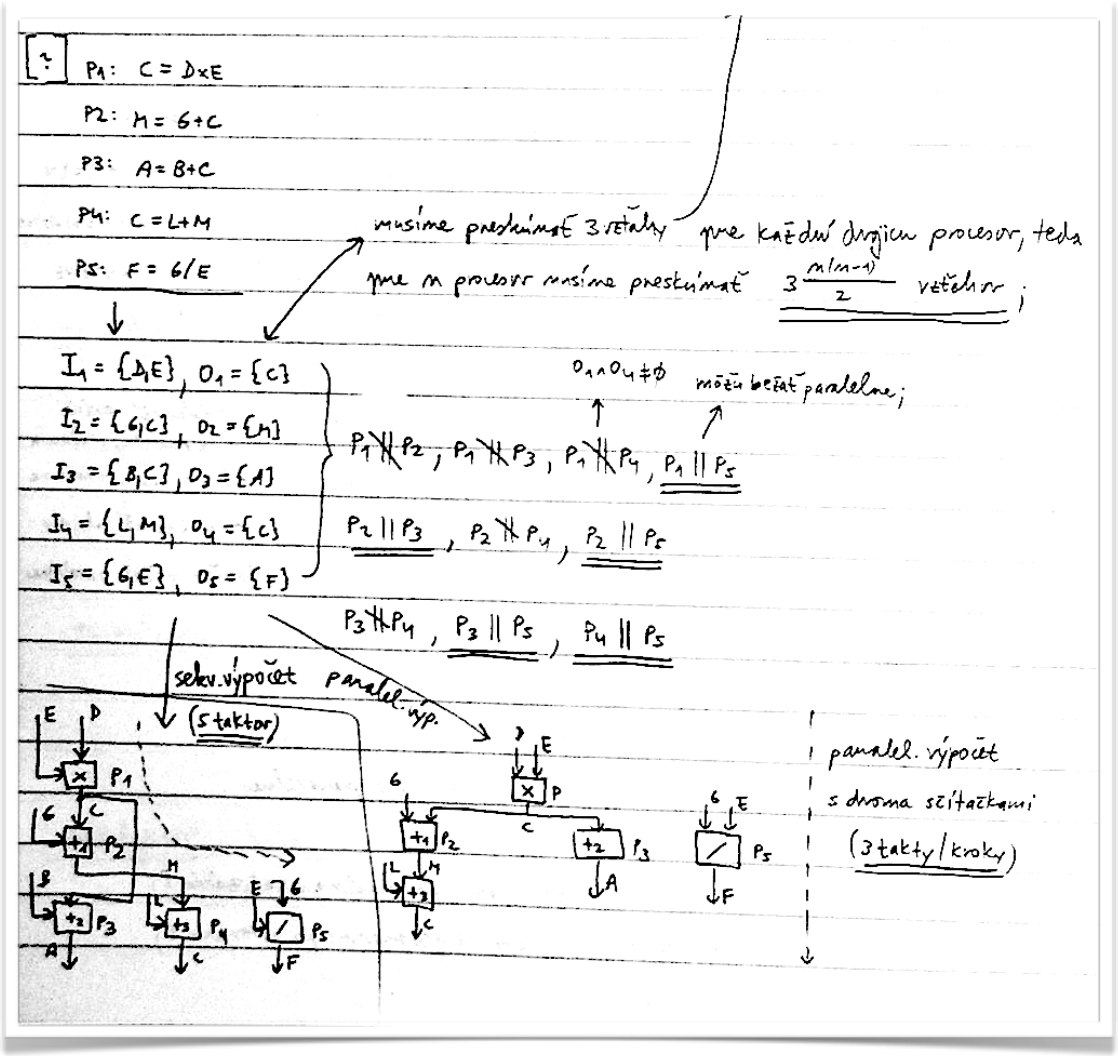
Udávajú čo musí platiť, aby sa výpočty mohli sparalelniť, teda bežať paralelne. Proces je programová entita korešpondujúca programového fragmentu. Proces P:
- vstupná množina I - všetky premenné potrebné na realizáciu P
- výstupná množina O - všetky premenné generované po vykonaní P.

P_1: I_1, O_1
P_2: I_2, O_2

P_1 možno vykonať paralelne, ak sú nezávislé a nevytvárajú nejednoznačný výsledok. Nesmú byť tokovo závislé a nesmú zapisovať do tých istých premenných (výstupná závislosť). ($I_1 \cap O_2 = \emptyset$, $I_2 \cap O_1 = \emptyset$, $O_1 \cap O_2 = \emptyset$) - berštainove podmienky.

Nech máme n procesov P_1, P_2, P_n vieme ich vykonať paralelne ak každé dve z nich môžu bežať paralelne, teda ak pre každé dve z nich platia berštainove podmienky.

Príklad:

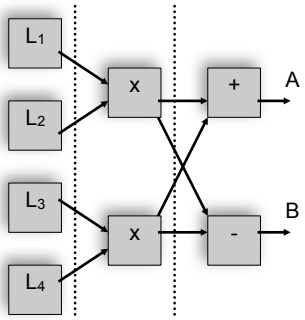


HW versus softvérový paralelizmus:
- SW - definovaný závislosťou dát a závislosťou riadenia
- HW - definovaný architektúrou stroja a následnosťou technických prostriedkov.

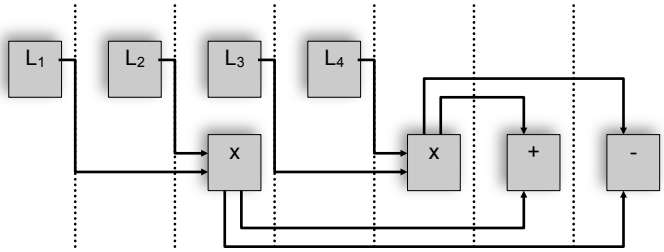
Najdôležitejšie:
- riadiaci paralelizmus - 2 a viac operácii možno vykonať súčasne
- dátový paralelizmus - aspoň isté operácie možno vykonať nad dátami a CPU súčasne, vyšší potenciál na súbežnosť, kód sa ľahšie ladí

Medzi HW a SW paralelizmom potrebujeme súlad. Riešenie:
- podpora prekladača
- redesign.

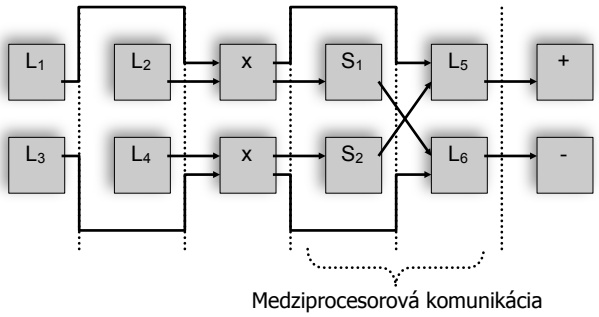
Skúsme uvažovať graf jednoduchého programu pre realizáciu L1*L2 +- L3*L4. Základný paralelizmus je naznačený na obrázku. V prvom takte sa načítajú operandy L1 .. L4 (4 inštrukcie). V druhom takte sa vzájomne operandy vynásobia (2 inštrukcie), nakoniec sa v treťom takte výsledky násobení sčítajú a odčítajú (2 inštrukcie). V tomto prípade teda bola rýchlosť vykonania 8 / 3 = 2.67 inštrukcie/takt.



Teraz uvažujme rovnaký problém, ale pri použití CPU, ktorý dokáže v jednom takte vykonať jednu load a jednu store operáciu. Z obrázku vidíme, že v tomto prípade potrebujeme na realizáciu 8 inštrukcií 7 taktov. Vzniká tu nesúlad medzi možným SW paralelizmom a HW prostriedkami, rýchlosť klesne na 1.14 inštr/takt.



A ešte pre ilustráciu uvažujme počítač s dvomi CPU, na každom sa dá vykonať 1 inštrukcia na jeden takt. Vidíme, že v tomto prípade je ešte stále nesúlad SW paralelizmu a HW, ale rýchlosť je vyššia ako v predchádzajúcom prípade: 2.0 inštrukcie/takt.



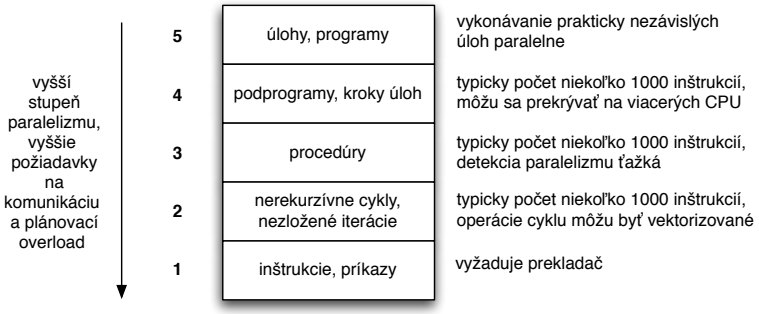
vii. Delenie programu a plánovanie

Treba stanovovať nezávislé časti kódu - granule a alokovať ich na procesoroch - plánovanie granulí. Latencia je doba, ktorá uplynie od vyžiadania údaju až po jeho získanie, miera času potrebná ako komunikačný overhead pri komunikácii dvoch procesov/granulí.

Problémy:
- granule
- plánovanie granulí
- latencia komunikácie.

Granularita je miera množstva výpočtu, ktorá je zahrnutá v procese napr. počet inštrukcií v granule.

Úrovne paralelizmu (granularity):

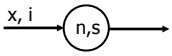


5,4 hrubá granularita
4,3,2 stredná granularita
2,1 jemná granularita

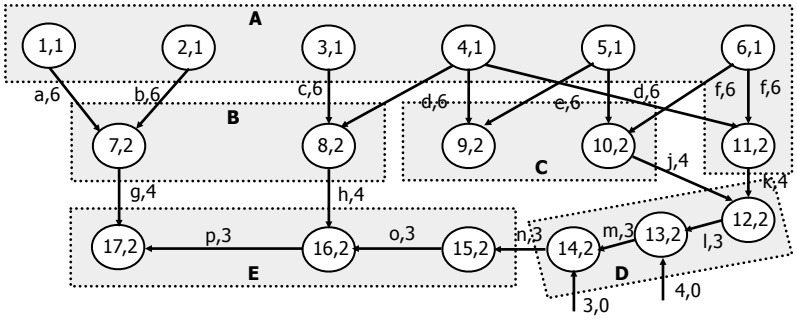
Majme program, príkazy priradenia (1-6) nech trvajú 1 takt, ostatné nech trvajú 2 takty, ostatné oneskorenia sú dané komunikáciou:

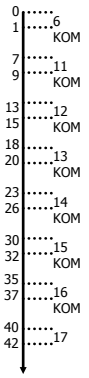
1. a:=1	6. f:=6	11. k:=d.f	16: p:=o.h
2. b:=2	7. g:=a.b	12. l:=j.k	17. q:=p.g
3. c:=3	8. h:=c.d	13. m:=4.l	
4. d:=4	9. i:=d.e	14. n:=3.m	
5. e:=	10. j:=e.f	15. p:=	

Zaveďme označovanie vrcholov v grafe nasledovne:
n ... uzol
s ... oneskorenie (trvanie výpočtu v uzle)
x ... vstup
i ... doba latencie

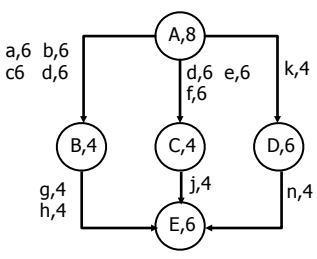


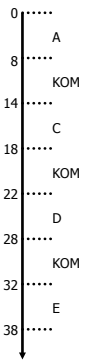
Teraz skúsme vytvoriť model vyššie uvedeného programu pomocou takýchto uzlov:





Toto riešenie je neefektívne, skúsme použiť hrubšie granuly:





[7] ① a := 1 — m := 4 * l ⑬ — → veľkosť granule: počet základ. cyklov CPU + prístup do pamäte;

② b := 2 — n := 3 * m ⑭

③ c := 3 — o := n * i ⑮ — → inštr. ①-⑥ trvajú 1 takt CPU + 6 taktov prístup do pamäti

④ d := 4 — p := o * h ⑯ — → inštr. ⑦-⑰ vyžadujú 2 takty CPU

⑤ e := 5 — q := t * q ⑰ — → ostat. oneskorenia sú dané komunikáciou

⑥ f := 6

⑦ g := a * b

⑧ h := c * d

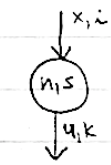
⑨ i := d * e

⑩ j := e * f

⑪ k := d * f

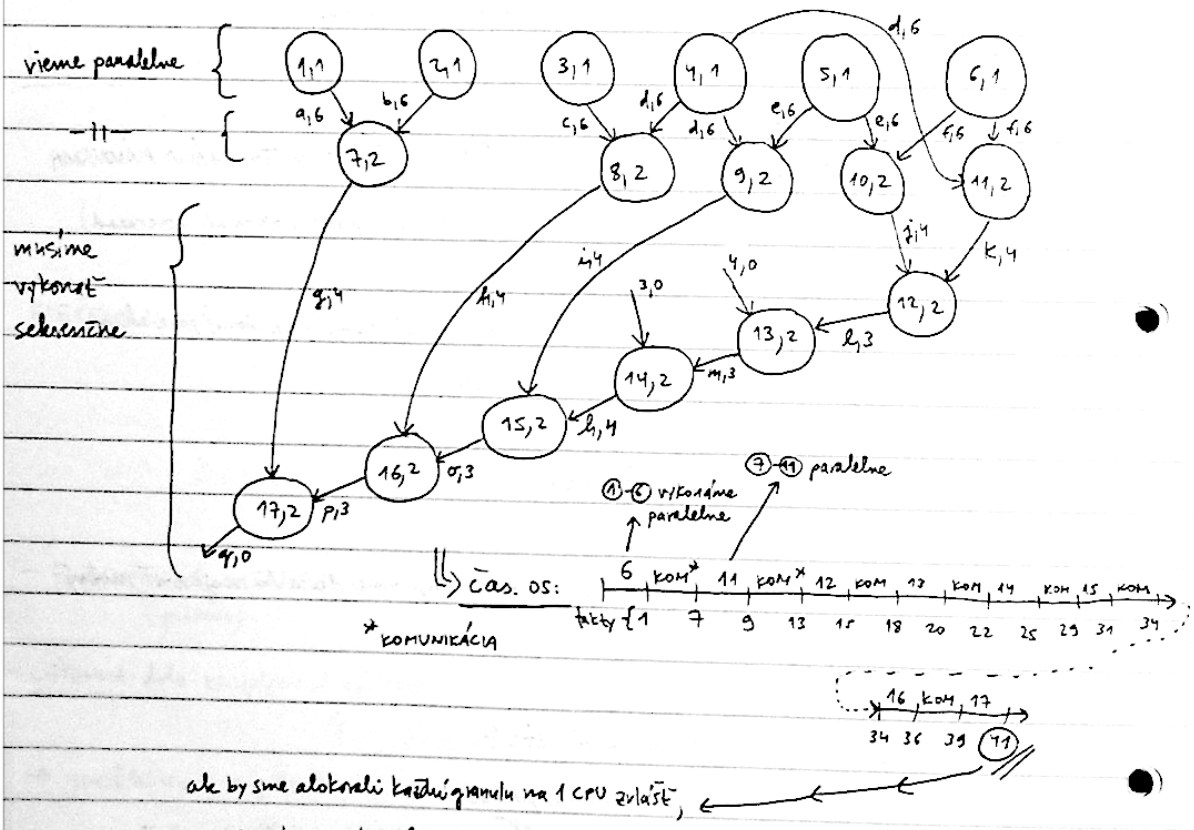
⑫ l := j * k

↳ GRAF PROGRAMU: (uzly sú granule)



n - identif. uzla
s - veľkosť granule (počet taktov)
x - identifikácia vstupu
i,k - oneskorenia
y - identifikácia výstupu

↳ najprv prevedieme graf na jemnej úrovni granularity (m = číslo inštrukcie, každá inš. je granula):



ak by sme alokovali každú granulu na 1 CPU zvlášť, program by sa vykonal za 41 taktov;

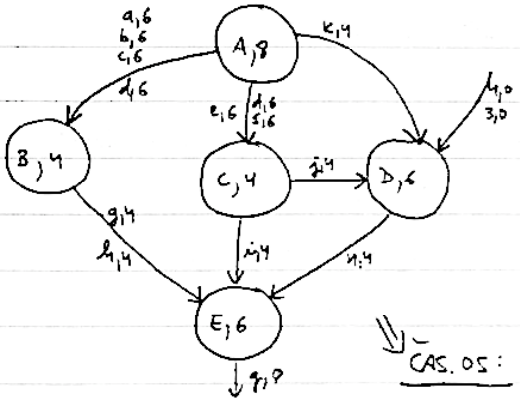
→ môžeme však zdmežiť tieto granule do nových nasledovne: • 1-6,11 = GRANULA A

• 7,8 = -||- B

• 9,10 = -||- C

• 12-14 = -||- D

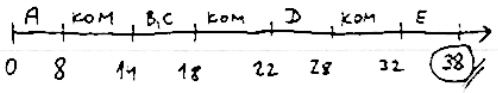
• 15-17 = -||- E



(účelom je redukcia plán. a komunik. overheadu)

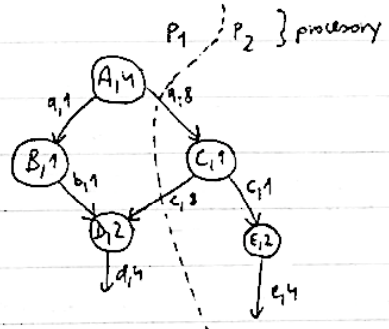
(komunikácia vnútri granúl je zanedbateľná)

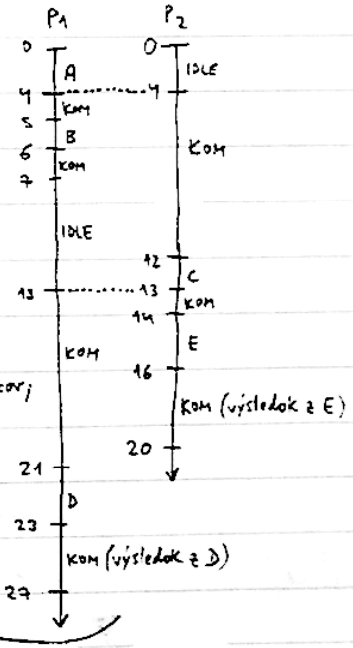
ČAS. OS:



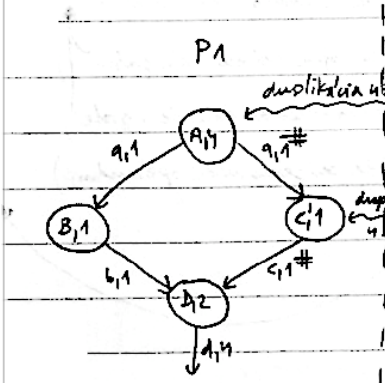
STATICKÉ PLÁNOVANIE V MULTIPROCESORE

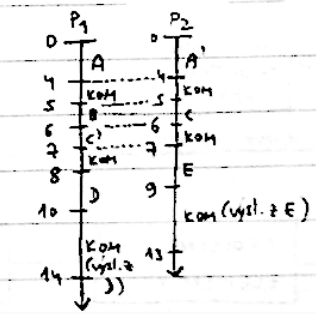
DUPLIKÁCIA UZLA:





počet taktov tam je teraz len 1, lebo premenné (výsledky) sa posielajú vrámci jedného procesora, ateda netrvajú 8 taktov;





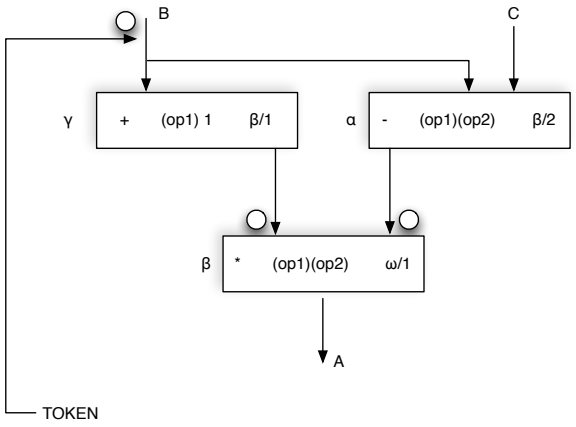
viii. Počítače riadené tokom dát (data flow stroje)

Control flow koncepcia (Von Neumann):
- je riadený tok inštrukcií v poradí, v akom sú zapísané v pamäti
- skokové inštrukcie menia hodnotu registra PC
- inštrukcie ukladajú výsledky do pamäti, resp. registra
- pamäť je dostupná všetkým inštrukciám.

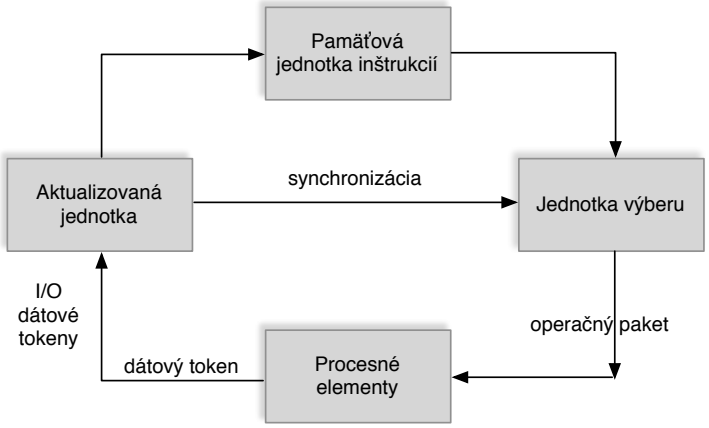
Data flow koncepcia:
- princíp, že inštrukcia sa vykoná vtedy, ak má platné operandy
- namiesto registrov zabezpečuje mechanizmus prenos výsledkov medzi inštrukciami
- každá inštrukcia má preto polia pre operandy
- operandy vstupujúce do inštrukcií sú ich súčasťami spolu s údajom, do ktorej inštrukcie sa zapíše výsledok
- náhradou registra je TOKEN, chodí po operandoch v rámci inštrukcií a kontroluje ich platnosť, zabezpečuje presun hodnôt medzi nimi.

Architektúra data flow:
- statická - jeden token vrámci hrán grafu
- dynamická - viacero tokenov existuje i v grafe, sú série medzi operandmi.

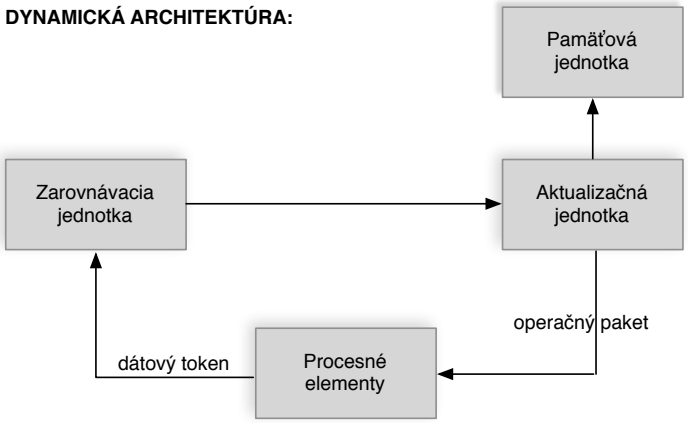
Graf:



STATICKÁ ARCHITEKTÚRA:



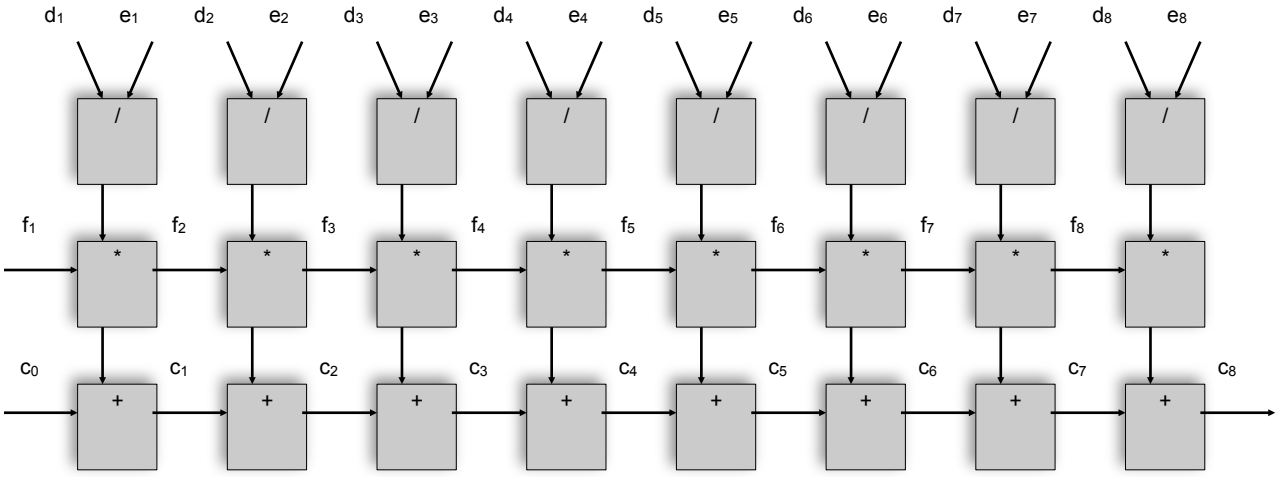
DYNAMICKÁ ARCHITEKTÚRA:



Príklad:

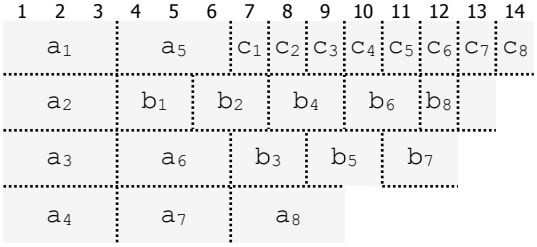
```
input d,e,f
c0=0
for i from 1 to 8 begin
    ai = di / ei
    bi = ai * fi
    ci = bi + ci-1
end
```

Nech sčítanie trvá 1 takt, násobenie 2 takty a delenie 3 takty. Tok programu potom môžeme znázorniť následovnou schémou:



Trvanie na sekvenčnom stroji: 8.1 + 8.2 + 8.3 = 48 taktov.

Na data flow so 4 CPU:



Celkovo trvá výpočet v tomto prípade 14 taktov.

Na 4 procesorovom systéme so spoločnou pamäťou:

Zaveďme následovné:

$s_1 = b_2 + b_1$	$t_1 = b_3 + s_1$	$c_1 = b_1 + c_0$	$c_5 = b_5 + c_4$
$s_2 = b_4 + b_3$	$t_2 = s_1 + s_2$	$c_2 = s_1 + c_0$	$c_6 = s_3 + c_4$
$s_3 = b_6 + b_5$	$t_3 = b_7 + s_3$	$c_3 = t_1 + c_0$	$c_7 = t_3 + c_4$
$s_4 = b_8 + b_7$	$t_4 = s_4 + s_3$	$c_4 = t_2 + c_0$	$c_8 = t_4 + c_4$

Potom tok programu môžeme vyjadriť:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
a_1			a_5			b_1		b_5		s_1	t_1	c_1	c_5
a_2			a_6			b_2		b_6		s_2	t_2	c_2	c_6
a_3			a_7			b_3		b_7		s_3	t_3	c_3	c_7
a_4			a_8			b_4		b_8		s_4	t_4	c_4	c_8

Aj v tomto prípade teda trvá výpočet 14 taktov.

Koncepcie v skratke:

- tok riadenia (control flow)- inštrukcie implicitne vykonávané v poradí ako sú uložené v pamäti, dáta sú voľne dostupné v pamäti
- tok dát (data flow)- inštrukcia sa vykoná, ak má platné operandy, jemný paralelizmus, dáta nie sú voľne dostupné, ale tečú medzi patričnými inštrukciami programu
- tok požiadaviek (demand driven) - výpočet je spúšťaný požiadavkami na výsledok operácie, napr. výpočet a=((b+1)*c) by prebehol ak je požiadavka na premennú a, prípade data flow by prebehol, ak by sme mali stanovené hodnoty premenných b,c.

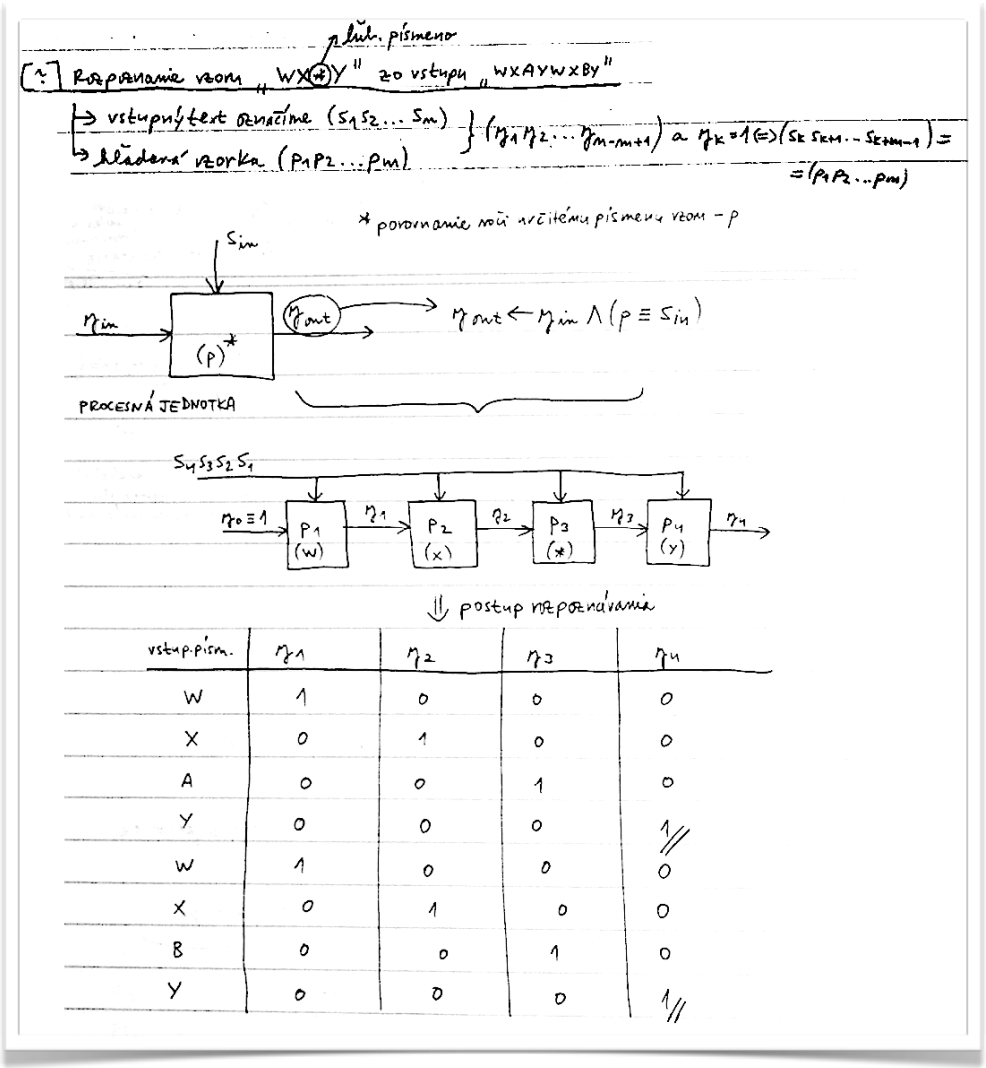
Model stroja	**Tok riadenia**	**Tok dát**	**Tok požiadaviek**
základná definícia	tok riadenia (PC) určuje, ktorá inštrukcia sa vykoná, klasické počítanie	inštrukcie sa vykonajú vtedy, ak majú patné operandy	inštrukcie sa vykonajú iba vtedy, keď sú ich výsledky potrebné
výhody	plné riadenie - programátor má kontrolu nad riadením a vykonávaním programu zložité riadiace a dátové štruktúry sú ľahko implementovateľné komerčne najúspešnejší princíp	vysoký potenciál pre paralelizmus inštrukcií vysoká priepustnosť systému operandy sú chránené pred bočnými efektami	vykonávajú sa iba požadované inštrukcie vysoký stupeň paralelizmu ľahká manipulácia s dátovými štruktúrami

Model stroja	Tok riadenia	Tok dát	Tok požiadaviek
nevýhody	malá efektivita ťažkosti pri programovaní ťažkosti pri zamedzení runtime chýb	nutný vysoký riadiaci overhead ťažkosti pri manipulácii s dátovými štruktúrami časové straty na čakanie pre nepotrebné argumenty	nepodporuje zdieľanie objektov s meniacimi sa lokálnymi stavmi je potrebný čas na šírenie požiadavkových tokenov

ix. Systolické siete

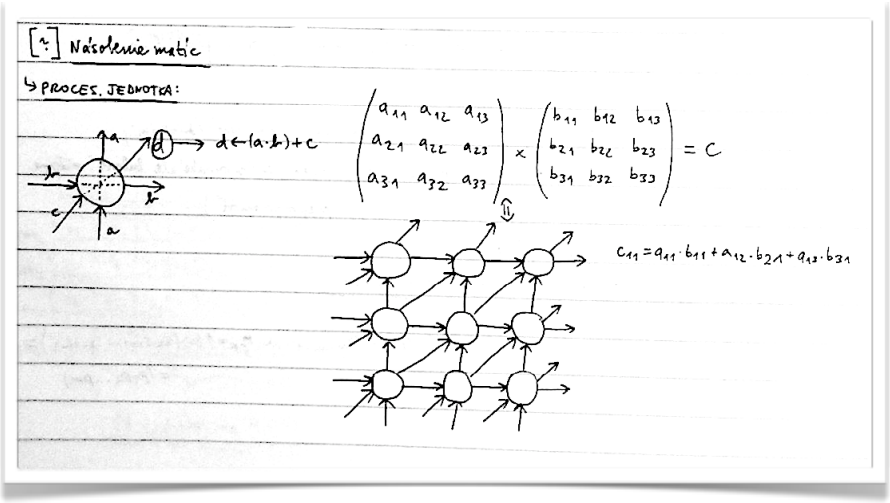
Algoritmus priamo realizujú svojou štruktúrou. Môžu byť lineárne alebo dvoj dimenzionálne pole identických procesorových jednotiek. Každá z nich komunikuje iba so susednými jednotkami, I/O komunikáciu zabezpečujú iba hraničné jednotky. Takéto sa nazývajú čisté systolické siete. Nevyhnutnými súčasťami našich životov sa však stali modifikované systolické siete, ktoré obsahujú globálnu komunikáciu medzi jednotkami/bunkami a programovateľné bunky.

Príklad:



vstup. písm.	η_1	η_2	η_3	η_4
W	1	0	0	0
X	0	1	0	0
A	0	0	1	0
Y	0	0	0	1 //
W	1	0	0	0
X	0	1	0	0
B	0	0	1	0
Y	0	0	0	1 //

Príklad:



x. Architektúra prepojovacích systémov (sietí)

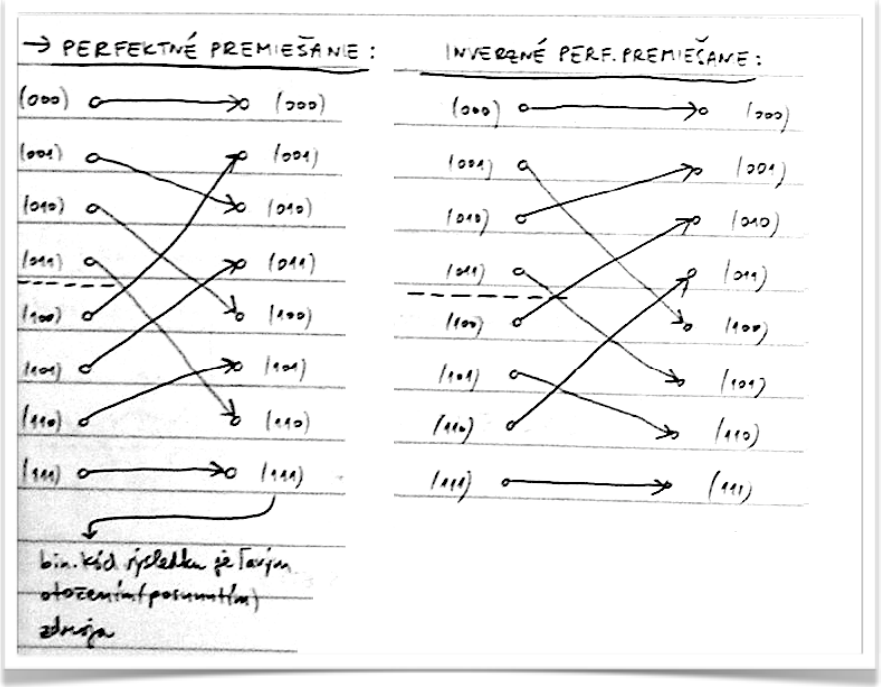
Pojmy:

- veľkosť siete - počet uzlov v sieti,
- stupeň siete - počet buniek vstupujúcich/vystupujúcich do/z uzla,
- priemer siete - maximum z najkratších ciest medzi ľub. uzlami (čím väčší, tým lepšie komunikačné schopnosti siete)
- šírka rezu - ak rozdelíme sieť na dve polovice, tak šírka rezu je počet hrán na polovici, je to i indikátor maximálnej priepustnosti siete.

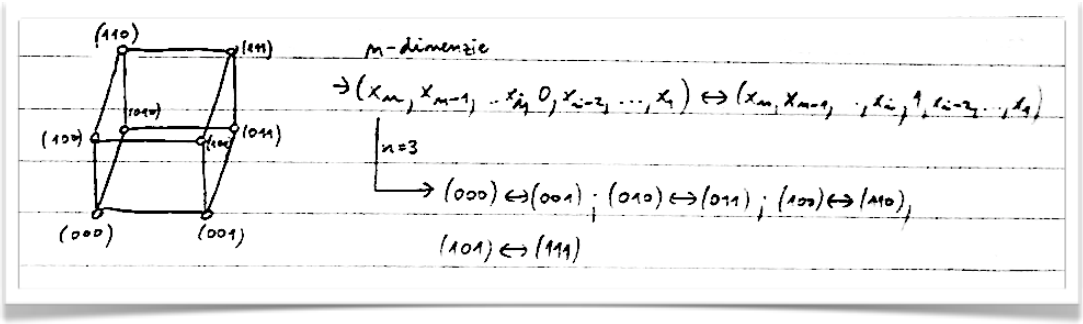
Počet rôznych smerovaní dát je n!. Takúto vlastnosť má však len krížový prepínač.

Smerovacie funkcie:

- perfektné premiešanie: a perfect shuffle is used between groups of elements. To visualize this, picture a deck of only 8 cards. Cut exactly in two, one stack will have cards 1-4, the other 5-8. If then shuffled to perfection and counting from 0, the new order will be 0-4-1-5-2-6-3-7. Mathematically, this places the nth card as follows: PS(n) = 2 * n for n < N/2, PS(n) = 2 * n - N + 1 for n >= N/2

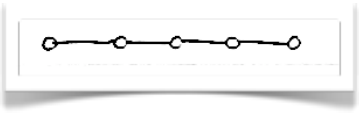


- kubické smerovacie funkcie:



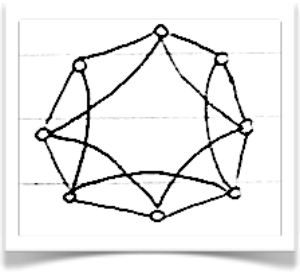
Siete so statickým prepojením:
Prepojenia sa nedajú meniť, alebo prepínať, vhodné pre stroje kde sú známe komunikačné vzory.

Lineárne pole:

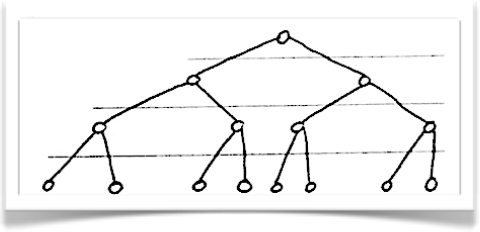


Kruh: získame z lineárneho pola prepojením začiatočného a koncového uzla, kruh je jednosmerný alebo obojsmerný.

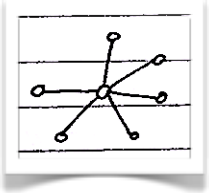
Premostený kruh: má lepšie vlastnosti ako kruh (vyšší stupeň uzlov, ale nižší priemer siete).



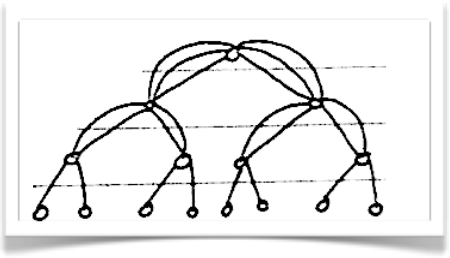
Strom: úplný vyvážený binárny strom, max. stupeň uzla = 3, priemer siete je 2(k-1), kde k je hĺbka stromu, počet uzlov $N=2^k-1$.



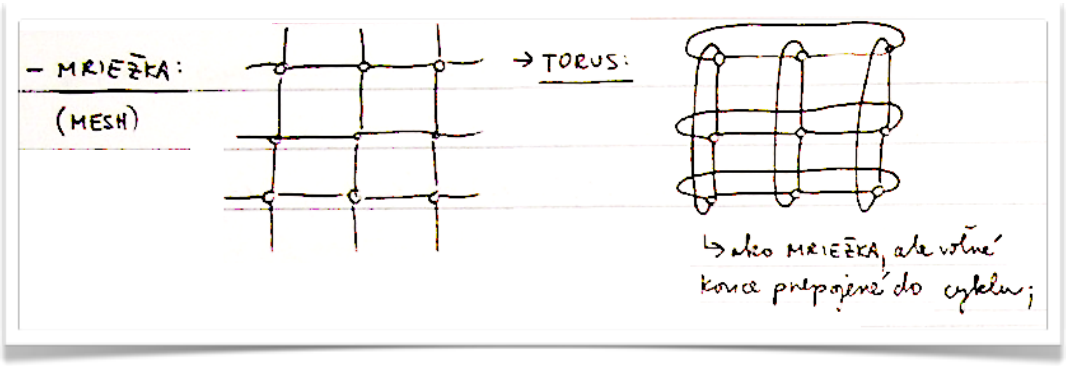
Hviezda:



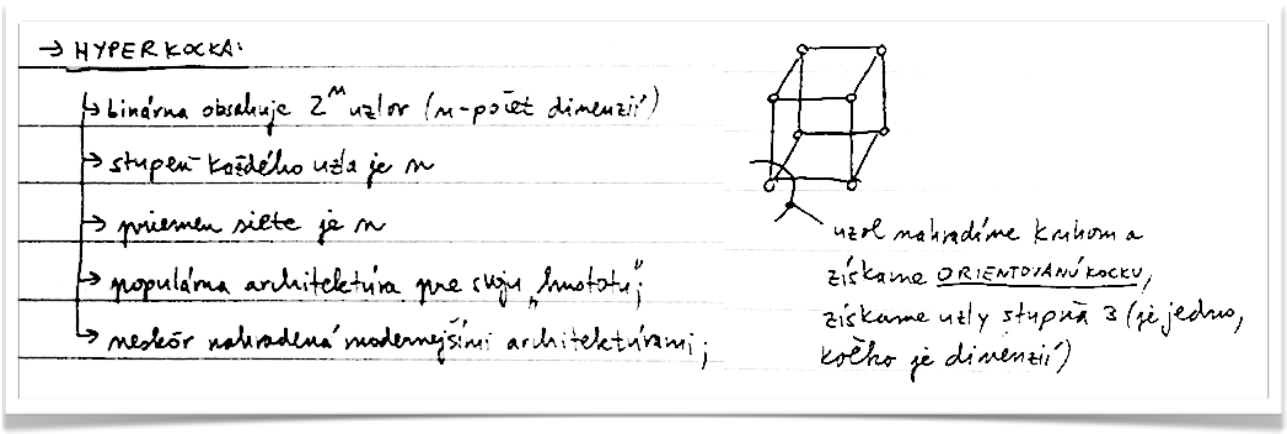
Tučný strom:



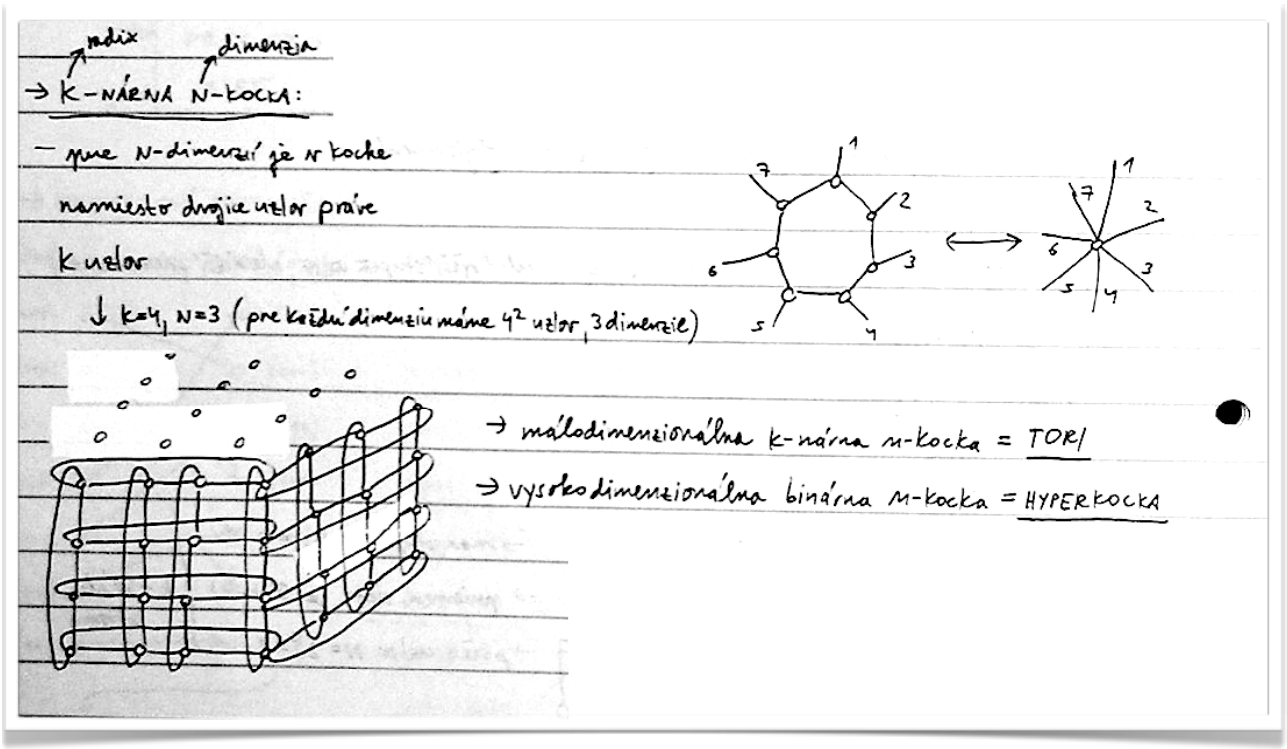
Mriežka:



Hyperkocka:



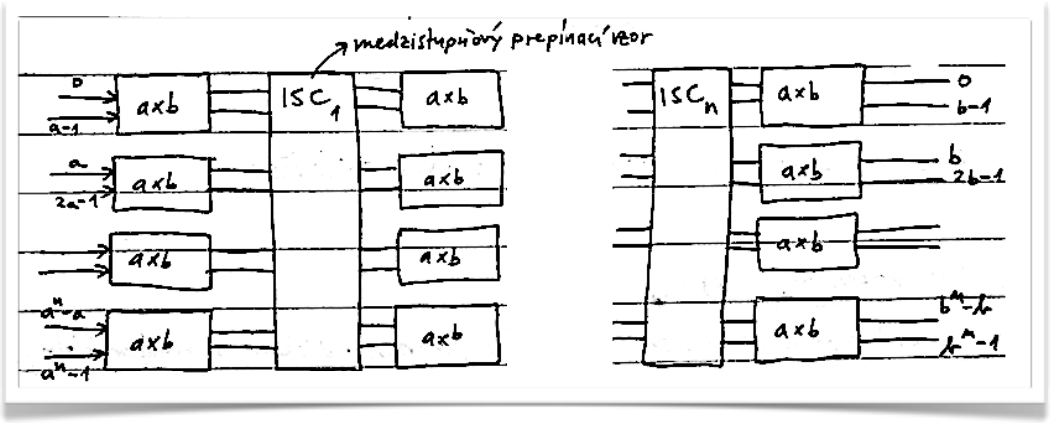
K-nárna N-kocka:



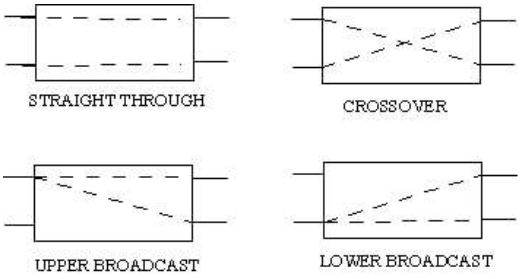
Typ siete	Stupeň uzla	Priemer siete	Počet prepojení	Šírka rezu	Symetria	Veľkosť siete
lineárne pole	2 (konc. 1)	n-1	n-1	1	nie	n-uzlov
kruh	2	n/2	n	2	áno	n-uzlov
úplne prepojenie	n-1	1	n(n-1)/2	$(n/2)^2$	áno	n-uzlov
binárny strom	3,2,1 listy	2(h-1)	n-1	1	nie	$h=\log_0 n$ h je výška stromu
hyperkocka	n	n	n*n/2	n/2	áno	$n=\log_2 n$
k-nárna n-kocka	2n	n*k/2	n*N	$2k^{n-1}$	áno	$n=k^n$

Pri aplikácii HW prepínania je parameter stupňa uzla nepodstatný, malý stupeň uzlov je výhodný pre škálovateľné aplikácie.

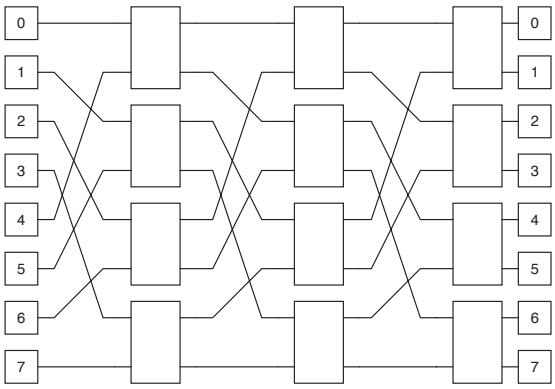
Siete s dynamickým prepínaním:
Vhodné pre stroje, kde nie sú dopredu dohodnuté komunikačné vzory. Elementami sú krížové prepínače:



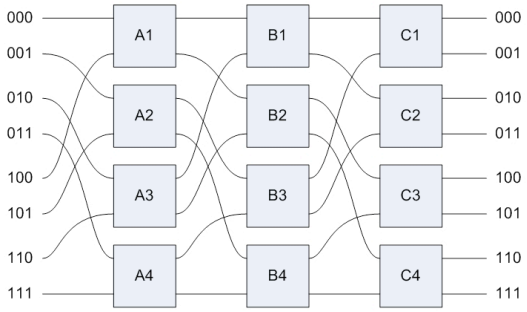
Módy prepínačov:



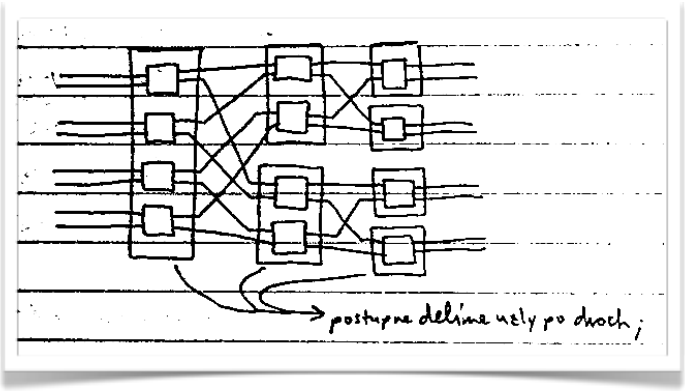
Sieť Omega:



- prepínače 2x2
- ISC - perfektné premiešanie vstupov aj výstupov
- počet stupňov je $\log_2 n$, n je počet vstupov a výstupov
- postup: postupne hneď rotujeme bity destinácie zľava, pričom ak tam je 0 tak prepneme hore, ak tam je 1 tak prepneme dole (Worm-hole routing)



Sieť baseline:

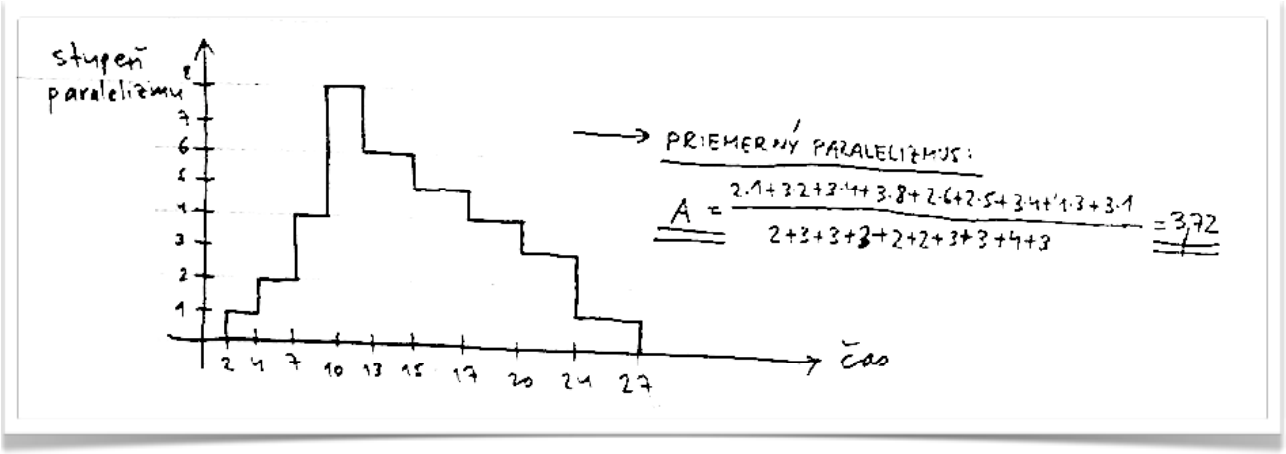


- základné prepojiteľné prvky - 2x2
- počet stupňov je $\log_2 n$, n je počet vstupov a výstupov
- rekurzívne delíme prvky po dvoch

Charakteristika siete	Zbernice	Viacstupňová sieť	Krížový prepínač
minimálna latencia na presun jednotky dát	konšt.	$\sigma(\log_k n)$ počet stupňov	konšt.
priepustnosť pre procesor	$\sigma(w/n)$ w -veľkosť prenášaných dát n - počet procesorov	$\sigma(w) \rightarrow \sigma(nw)$	$\sigma(w)$
zložitosť drátovania	$\sigma(w)$	$\sigma(nw*\log_k n)$	$\sigma(n^2 w)$
konektivita a smerovacie schopnosti	one-to-one	niektorá permutácia	niektorá permutácia

xi. Metrika a meranie výkonnosti paralelných počítačov

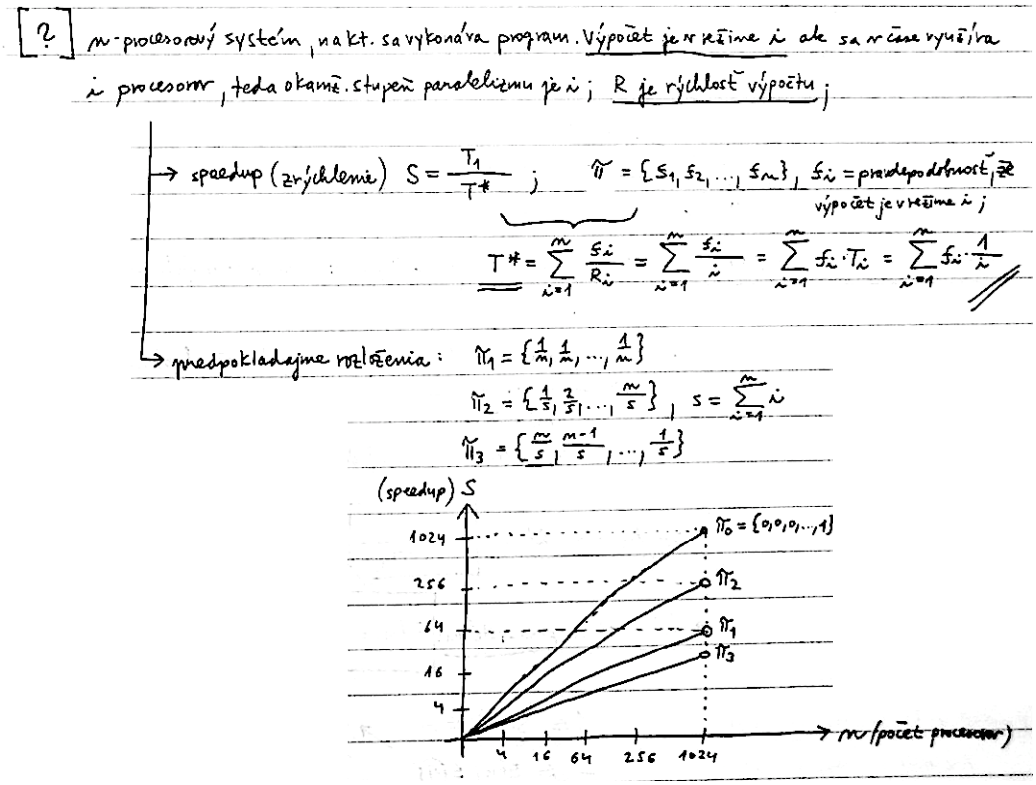
Profil paralelizmu v programoch na grafe. Stupeň paralelizmu - koľko procesorov v momente využíva program, diskrétne nezáporné číslo, ak ho rozvinieme v čase, dostaneme profil paralelizmu v programe.



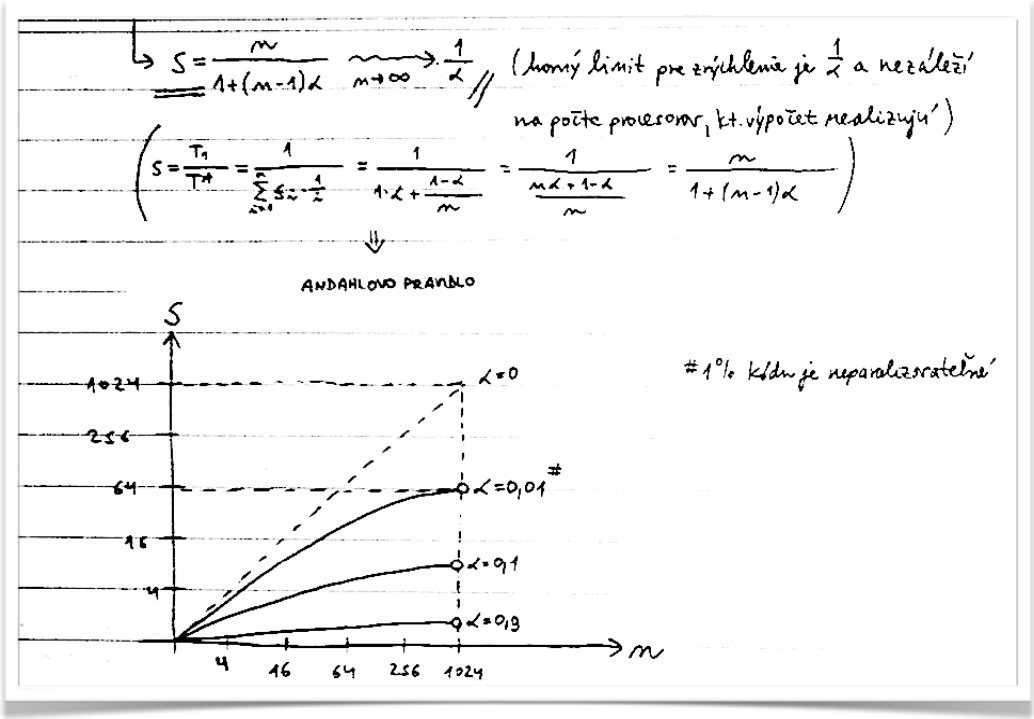
Skutočný paralelizmus:

- optimistický odhad - 500-3500 aritmetických operácii súčasne pre vedecko technické výpočty, faktor 90 pre VLIW architektúru, teda aplikácie využívajú až 90 procesných jednotiek
- pesimistický odhad - S-7 inštrukcií vykonávaných súčasne ak sú k dispozícii všetky procesné jednotky, ktoré program potrebuje, dobre vyváženým HW sa dá dosiahnuť trvalo 2,0-5,8 inštrukcií súčasne (superskalárny stroj).

Príklad:

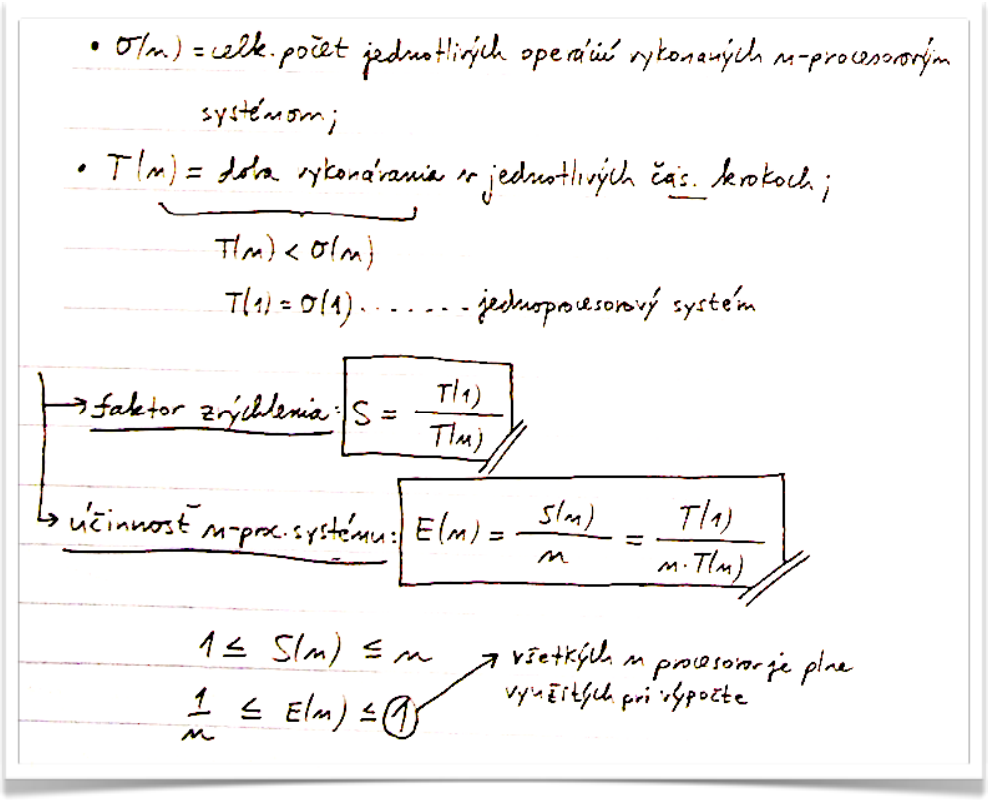


Príklad: analyzujme prípad, keď π={α,0,0,..,0,1-α}, teda program má neparalelizovateľnú časť, ktorá sa bude vykonávať s pravdepodobnosťou α:



Amdáhlovo pravidlo: akonáhle máme v programe neparalelizovateľnú časť, tak výkonnosť je výrazne obmedzená a nezáleží na počte procesorov.

Účinnosť počítačového systému:

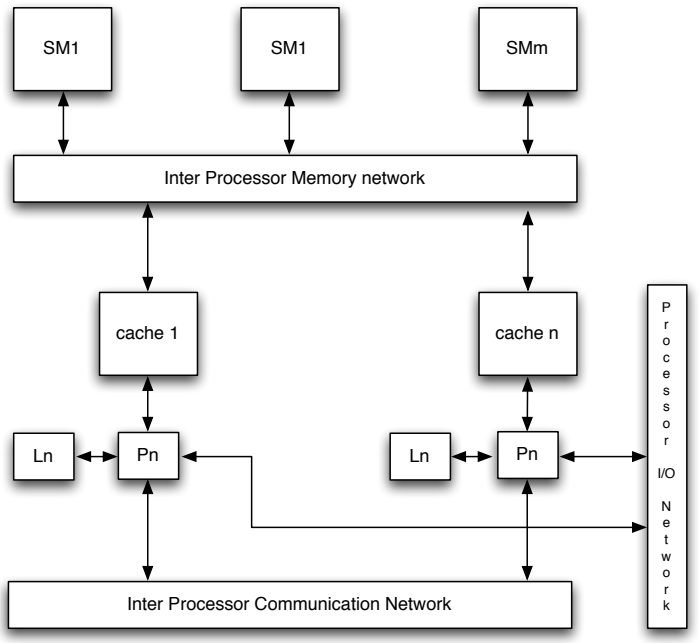


xii. Symbolické procesory

Atribút	Charakteristika
reprezentácia znalostí	zoznamy, tabuľky, objekty
bežné operácie	prehľadávanie, triedenie, porovnávanie obrazcov, filtrovanie, rozdeľovanie, tranzitívne uzávery, unifikácia, výber textu, množinové operácie, vyvodzovanie
požiadavky na pamäť	rozsiahla pamäť s rôznymi prístupovými vzormi, adresovanie často asociatívne, lokalita referencií nemusí platiť
typy komunikácií	prenos správ sa mení vo veľkosti aj v cieloch, granularita aj formát správ sa mení s aplikáciou
vlastnosti používaných algoritmov	nedeterministické algoritmy, možné paralelné a distribuované spracovanie, dátové závisloti môžu byť globálne a nepravidelné v typovej granularite
architektonické črty	paralená aktualizácia rozsiahlych báz znalostí, dynamický loadbalancing, HW podpora čistenia pamäti, architektúra zásobníkového stroja

xiii.Koherencia dát v multiprocesore so spoločnou pamäťou

Všeobecná schéma:



Abstrakcia: