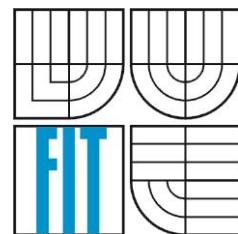


Návrh počítačových systémů

Princip činnosti procesoru

INP 2015
FIT VUT v Brně



Čím se budeme zabývat v INP

- Budou nás zejména zajímat jednoprocесоровé číslicové počítače:
 - funkce počítače
 - struktura – propojení funkčních bloků,
 - organizace – součinnost, chápána dynamicky, časové řízení bloků,
 - realizace – návrh a struktura bloků (např. ALU, řadič, cache, apod.)
 - měření výkonnosti.
- Reprezentace dat, kódy
- Principy a algoritmy základních i složitějších aritmetických operací
- Paměti – typy, paměťová hierarchie, řízení, ...
- Sběrnice a periferní zařízení
- Spolehlivost
- Úvod do paralelních architektur
- Cvičení:
 - obvodová realizace procesoru a jeho bloků
 - Jazyk VHDL
 - FITkit

Počítač (podle Wikipedie)

- Počítač je v informatice elektronické zařízení, které zpracovává data pomocí předem vytvořeného programu. Současný počítač se skládá z hardware, které představuje fyzické části počítače (procesor, klávesnice, monitor atd.) a ze software (operační systém a programy). Počítač je ovládán uživatelem, který poskytuje počítači data ke zpracování prostřednictvím jeho vstupních zařízení a počítač výsledky prezentuje pomocí výstupních zařízení. V současnosti jsou počítače využívány téměř ve všech oborech lidské činnosti.
- Analogový počítač vs číslicový počítač
- Historie – viz např. Wikipedie

První elektronické počítače

- **ENIAC** (Electronic Numerical Integrator and Calculator)
 - první elektronický (elektronkový) počítač na světě
 - postaven na University of Pennsylvania
 - postaven během druhé světové války, ale informace o něm byla zveřejněna až v roce 1946
 - programoval se drátovými propojkami a přepínači, data se zadávala pomocí děrných štítků
 - Příkon: 150 kW; 5000 sčítání/s; plocha 167 m²; hmotnost 27 t
- **EDVAC** (Electronic Discrete Variable Automatic Computer)
 - dokončený na téže univerzitě v roce 1951
 - řízen programem uloženým v paměti (5,5 kB)
 - Jeho koncepce se stala nejrozšířenější a nejznámější počítačovou architekturou. Poněkud neprávem se připisuje americkému matematikovi maďarského původu John von Neumannovi, přestože hlavními osobami projektu byli J. Presper Eckert a John Mauchly.
 - Příkon: 56 kW; 1160 sčítání/s; plocha 45 m²; hmotnost 7,8 t
 - Pracoval 20 hodin denně a průměrně 8 hodin bez poruchy.

Dnešní počítače - příklady

- Běžný notebook
- Lenovo E50-80 Black
- Intel Core i5 5200U Broadwell (2 jádra)
 - 2,2 GHz
 - 15 W
 - 14 nm
 - $1,3 \cdot 10^9$ tranzistorů
- 15,6" LED 1366x768
- RAM 4GB (DDR3L)
- Intel HD Graphics 5500
- HDD 500GB
- cca 15 tis. Kč
- Špičkový počítač na jednom čipu: Intel Xeon Phi
 - 61 jader (Pentium)
 - 244 vláken
 - 16 GB (GDDR5)
 - 1,2 TFLOP max. výkonnost
 - příkon 225 W
- Superpočítač: Salomon v Ostravě
 - 2 PFLOP/s teoretický výpočetní výkon
 - 24192 jader CPU Intel Xeon E5v3 (Haswell-EP) v 1008 uzlech
 - 129 TB RAM
 - 52704 jader v Intel Xeon Phi s 13,8 TB RAM
 - 2 PB diskové kapacity a 3 PB zálohovací páskové kapacity
 - 700 kW
 - 40. nejvýkonnější na světě (16. 9. 2015)
 - www.it4i.cz



Algoritmus

(viz kurz Základy programování)

- **Algoritmus** je přesně definovaná konečná posloupnost příkazů (které jsou vybírány z předem definované konečné množiny elementárních příkazů), jejichž prováděním pro každé přípustné vstupní hodnoty získáme po konečném počtu kroků odpovídající hodnoty výstupní. Intuitivně algoritmem rozumíme postup, který nás dovede k řešení úlohy.

Dvě základní implementace algoritmu

- **aplikacně-specifický číslicový obvod (HW)**
 - výhody: umožňuje optimální využití technických prostředků (optimalizace rychlosti, plochy, spotřeby apod.)
 - nevýhody: obvykle vysoká cena, dlouhá doba návrhu
- **program (SW)** pro procesor
 - výhody: univerzalita/flexibilita, cena
 - nevýhody: výkonnost nemusí být dostatečná

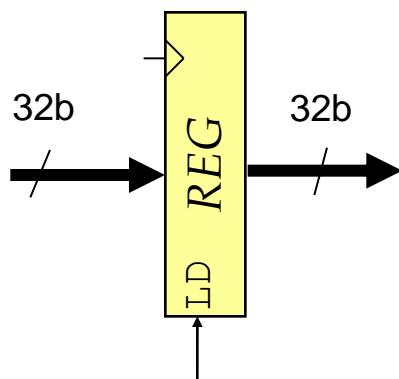
Jak se implementují datové a řídicí struktury?

Datové struktury: SW vs HW

- SW - příklad

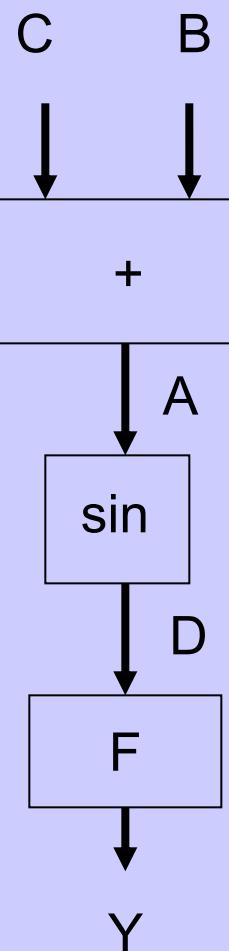
```
int x; // datový typ int je v daném procesoru např. 32b  
x = 34; // přiřazení do proměnné  
y = x; // čtení proměnné
```

- HW - příklad



Sekvence:

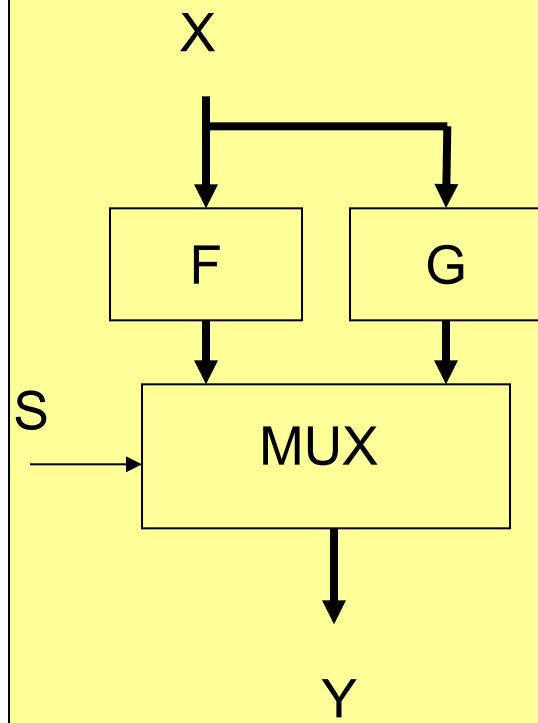
```
A = B + C;
D = sin(A);
Y = F(D);
```



Řídicí struktury: SW vs HW

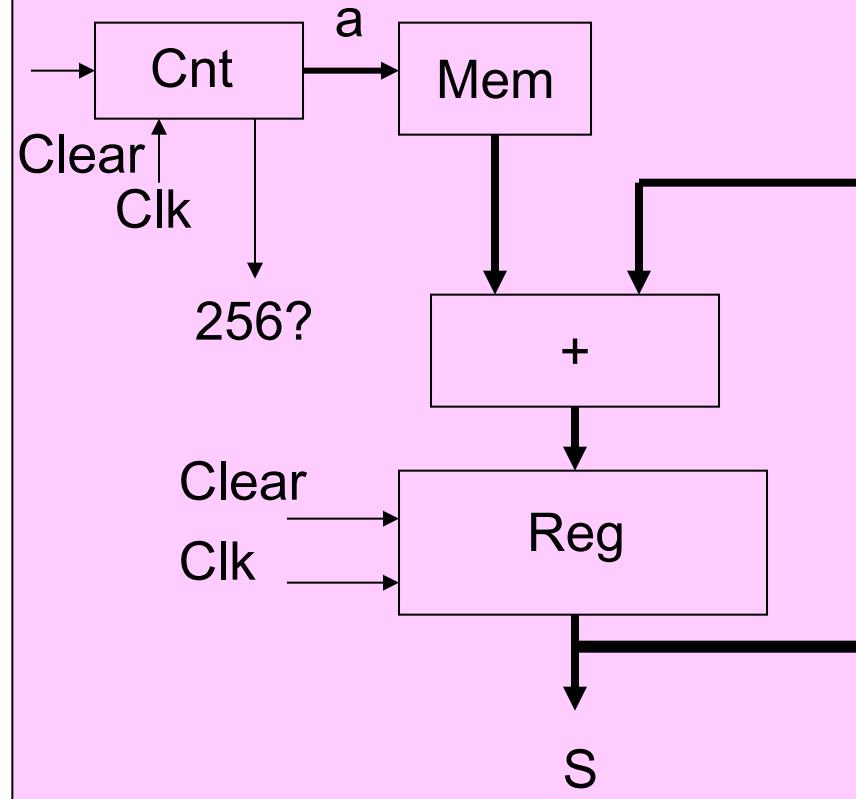
Selekce:

```
If (S == 0)
then Y = F(X)
else Y = G(X);
```



Iterace:

```
a = 0;
s = 0;
while (a < 256) do
{
    s = s + Mem[a];
    a++;
}
```



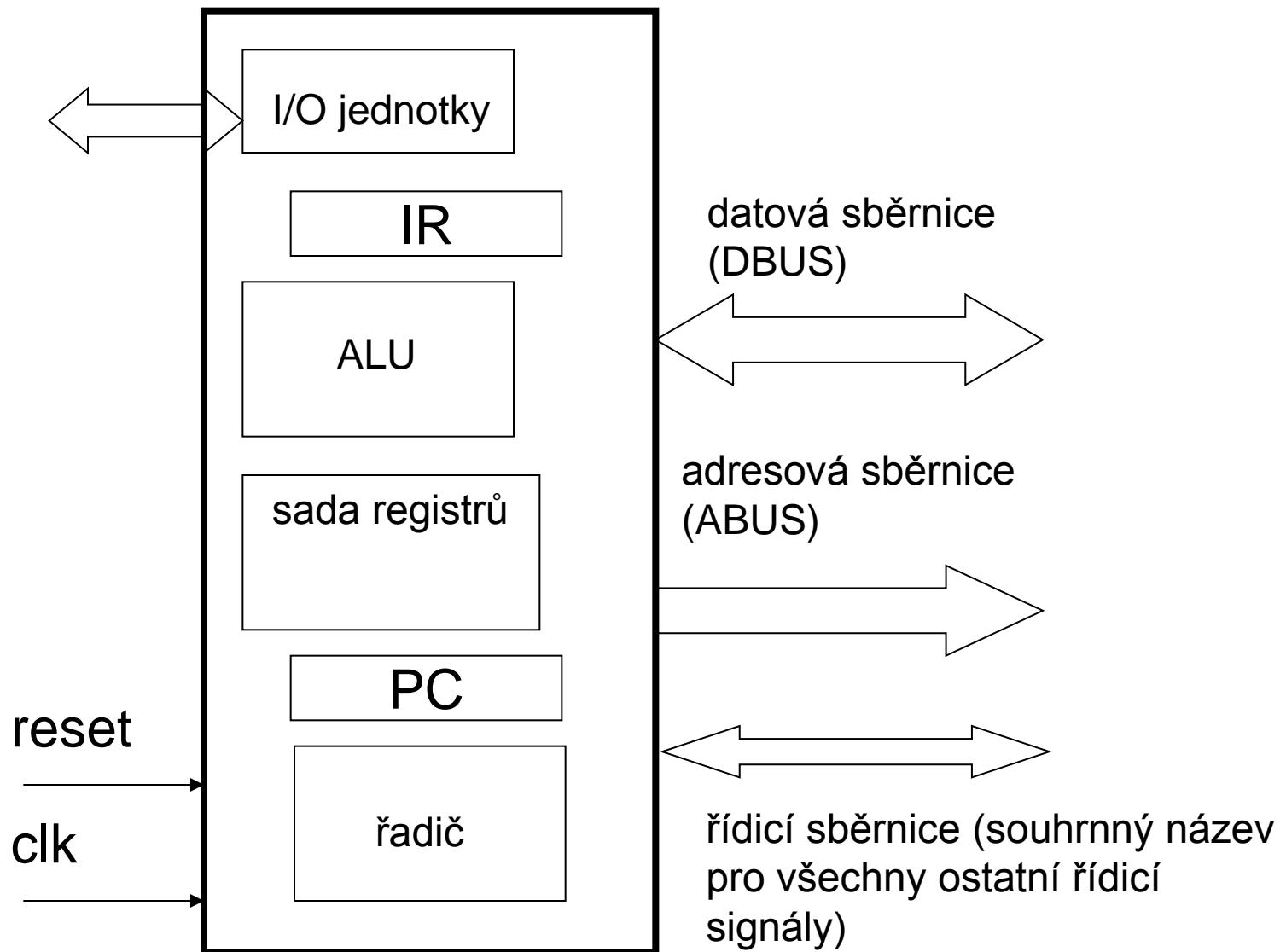
Procesor

- Procesor:
 - obvodová realizace algoritmu, který dokáže vykonávat libovolné **programy** (algoritmy), sestavené z předem definovaných elementárních operací (**instrukcí**) .
 - (obvykle) sekvenční a (obvykle) synchronní číslicový obvod
- Typická činnost:
 - Procesor vykoná **program**, který je uložen od zadané **adresy** v paměti. Program transformuje **data**, uložená na zadané adrese v paměti, na jiná data, jejichž nové umístění musí být rovněž specifikováno.
 - Pokud má procesor **I/O jednotku**, může rovněž při zpracování využívat informace ze svých vstupních portů a produkovat data na výstupní porty.
- Hlavní komponenty
 - **Programový čítač** (PC) určuje adresu, kde se nachází následující instrukce. Po resetu dochází k inicializaci procesoru a mj. je definován obsah PC (např. PC = 0003h).
 - IR – **instrukční registr** – uchovává právě zpracovávanou instrukci
 - **Řadič** řídí činnost procesoru (konečný automat - FSM).
 - **ALU** – aritmeticko-logická jednotka slouží k provádění matematických operací.
 - **I/O jednotka** – umožňuje vstup a výstup dat do/z procesoru (obvykle do jeho registrů) i jinak než pouze skrz paměť.
 - Sada registrů

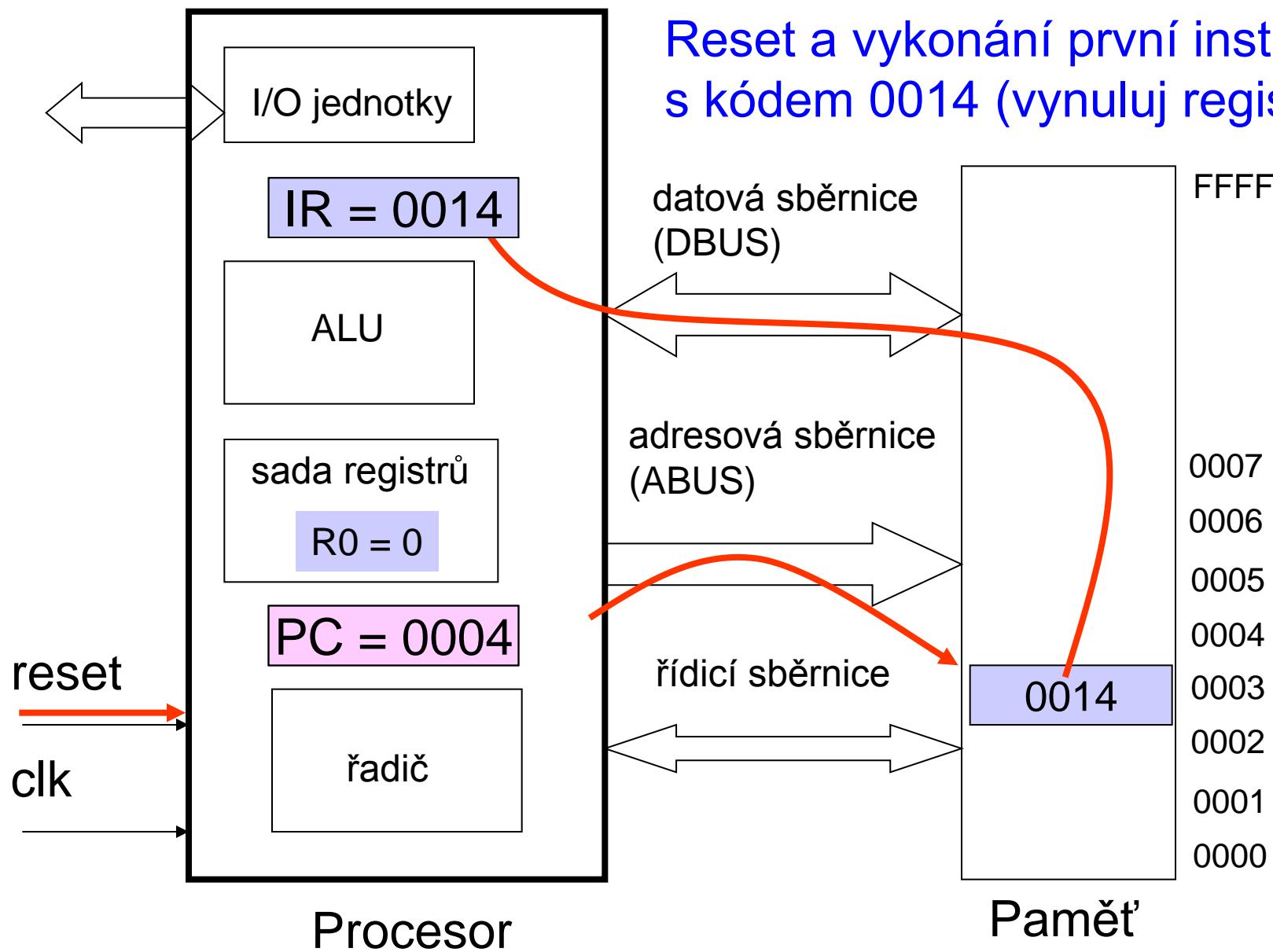
Programy a data

- Programy jsou uloženy v paměti jako posloupnosti instrukcí.
- Program se vykonává sekvenčně (až na výjimky, např. skokové instrukce)
- Data jsou uložena rovněž v paměti.
- Harvardská koncepce procesoru
 - Paměť dat a paměť programu jsou odděleny, tj. je možné současně např. načíst instrukci a zapisovat data.
- Von Neumannova (princetoneská) koncepce procesoru
 - Společná paměť pro data a programy, nelze např. současně načítat instrukci a načítat data.

Procesor a jeho rozhraní



Reset a vykonání první instrukce s kódem 0014 (vynuluj registr R0)



Po resetu se nastaví PC=0003. Adresa určená obsahem PC se vystaví na ABUS. Z této adresy se přečte kód instrukce (0014) a po DBUS se pošle do instrukčního registru IR. Instrukce se dekóduje a vykoná (vynuluje se registr R0) a zvýší se PC o 1.

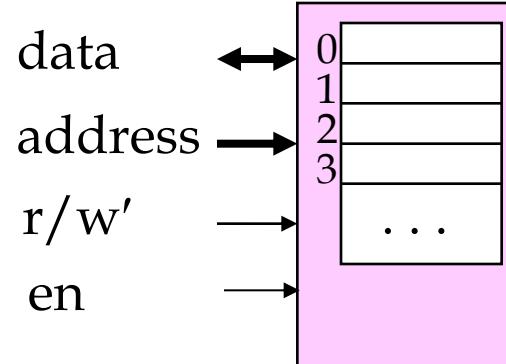
Vykonání instrukce

- Adresa instrukce, která se má vykonat, je uložena v PC.
- Instrukce se vykonává obvykle v několika hodinových taktech.
- Procesor musí zajistit následující:
 - Adresa určená obsahem PC je vystavena na adresovou sběrnici a je iniciováno čtení z paměti.
 - Paměť dodá na datovou sběrnici kód instrukce, která se má vykonat.
 - Instrukce je uložena do IR (instrukčního registru). Tyto tři body se označují jako načtení instrukce – **fetch (F)**.
 - Instrukce je dekódována (tzn., že jsou nastaveny řídicí signály pro komponenty procesoru, které se podílí na vykonání instrukce – je např. vybrána operace sčítání a operandy ALU). Tato fáze se nazývá **instruction decode (ID)**
 - Pokud instrukce vyžaduje pro své vykonání operand z paměti, musí být z příslušné adresy načten tento operand.
 - Instrukce je vykonána (např. provede se sčítání), jsou nastaveny příznaky (např. přetečení). Tato fáze se nazývá **Execute (E)**.
 - Uloží se výsledek do registru, popř. se iniciuje zapsání výsledku do paměti.
 - Určí se nový obsah PC.

Obvodová realizace procesoru

- Při návrhu procesoru se vychází ze **specifikace**, která obsahuje požadavky na instrukční soubor, architekturu, šířku dat a adres, výkonnost, plochu na čipu, spotřebu atd.
- Ukážeme si obvodovou realizaci nejjednoduššího procesoru, cílem bude demonstrovat princip činnosti procesoru.
- Použijeme tzv. **střádačovou architekturu** (tj. všechny instrukce ALU budou pracovat se speciálním registrem – střádačem (akumulátorem))
- Procesor bude připojen k paměti, která bude obsahovat program i data.
- Procesor bude navržen tak, že vytvoříme
 - datovou cestu (reprezentuje tok dat v procesoru) a
 - řadič (který bude implementován jako FSM).
- Než se dostaneme k realizaci procesoru, připomeneme si základní logické obvody – stavební bloky procesoru.

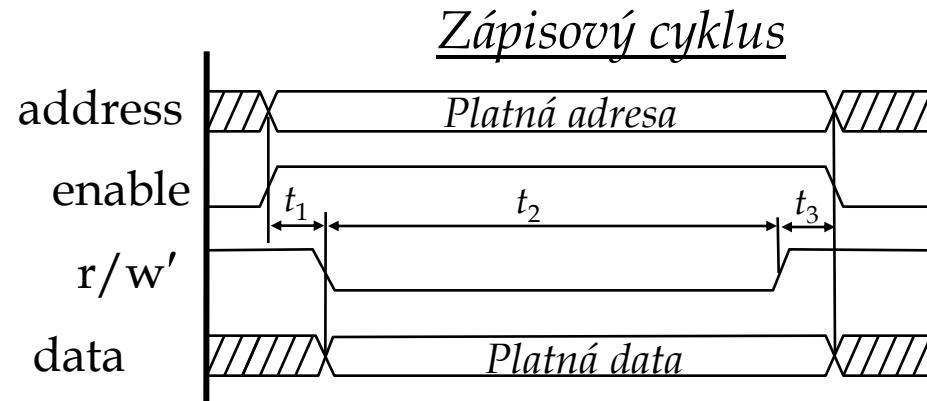
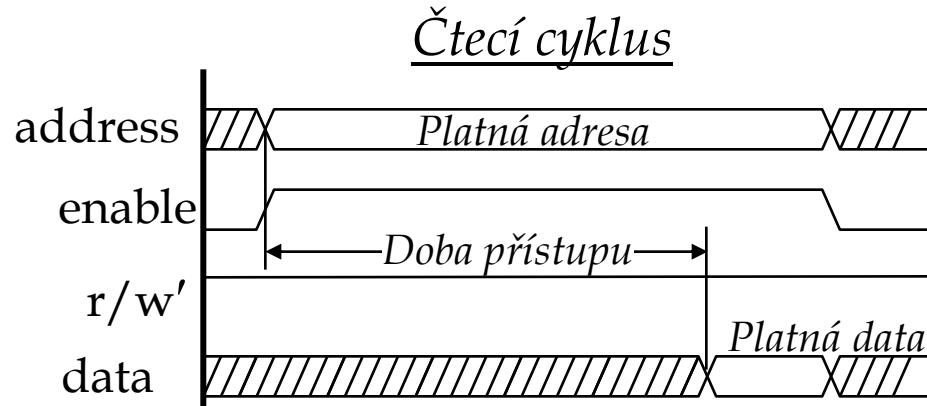
RAM



- RAM = *random access memory*
 - když **en** = 1 a **read/write** = 1, na datovou sběrnici *data* bude přivedena hodnota z paměti uložená na pozici, kterou definuje adresa vystavená na adresové sběrnici *address*
 - když **en** = 1 a **read/write** = 0, hodnota *data* přepíše data uložená v paměti na pozici, kterou definuje *address*
 - en značí signál *enable*, pokud **en** = 0, je paměť ve stavu vysoké impedance
 - nejjednodušší RAM jsou asynchronní
- SRAM vs DRAM
 - organizace, rychlosť, cena, obnova (viz přednáška o pamětech)

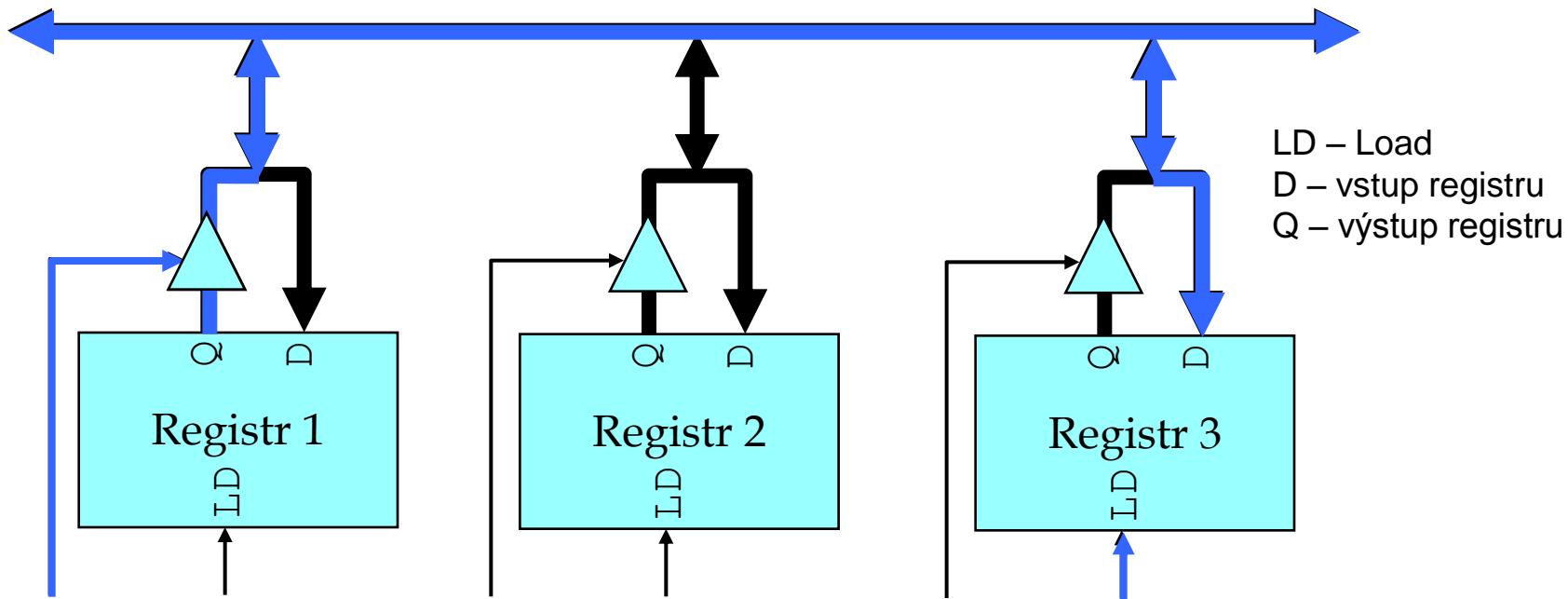
Časování asynchronní RAM

- **Čtecí cyklus**
 - Doba přístupu: doba, za kterou získáme platná data od požadání (nastavení signálů a adresy)
- **Zápisový cyklus**
 - t_1 je min doba pro stabilizaci adresy před aktivací r/w
 - t_2 je min doba, po kterou musí být vstupní data stabilní před deaktivací r/w
 - t_3 je min doba, po kterou musí být adresa platná po deaktivaci r/w
 - Zápisový cyklus: $t_1 + t_2 + t_3$

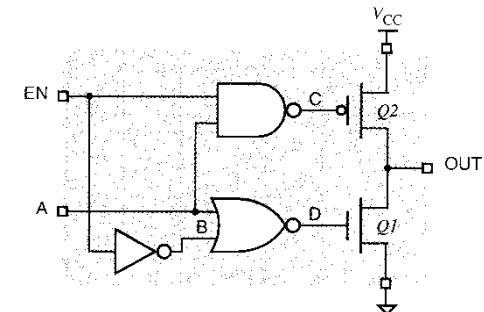


- Obvody, které používají RAM (tj. i náš procesor), musí správné časování zajistit!

Přenos dat s využitím sběrnice

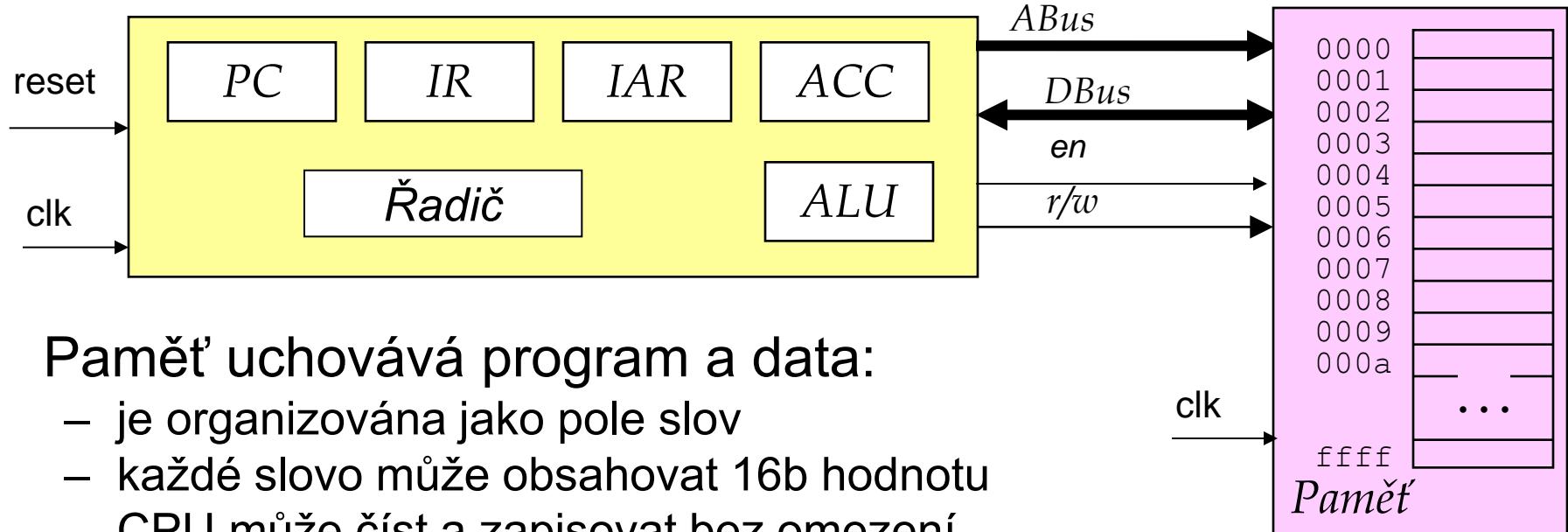


- **Sběrnice** je sdílená skupina vodičů sloužící pro přenos dat mezi několika zdroji/cíli
- Přenos dat zahrnuje:
 - umožnění jednomu zdroji dát data na sběrnici
 - nahrání dat do jednoho nebo několika cílových míst



Realizace 1b třístavového budiče

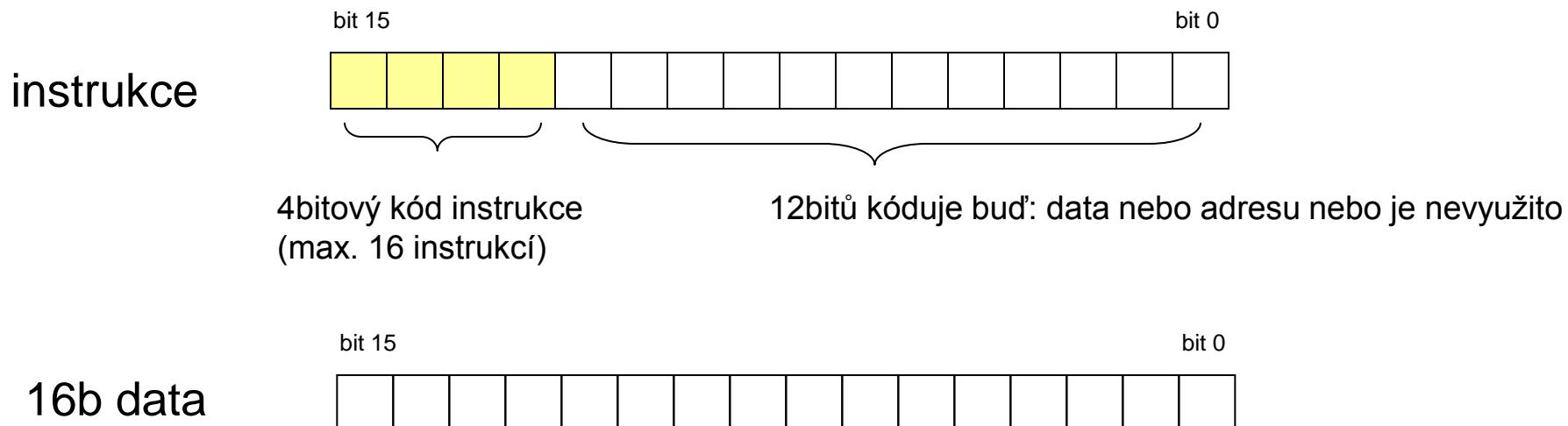
Nejjednodušší počítač



- Paměť uchovává program a data:
 - je organizována jako pole slov
 - každé slovo může obsahovat 16b hodnotu
 - CPU může číst a zapisovat bez omezení
- Zpracování instrukce
 - přečti slovo, jehož adresa je v PC (*Program Counter*) a inkrementuj PC
 - interpretuj tuto hodnotu jako instrukci (dekódování)
 - vykonej instrukci s využitím akumulátoru (ACC) a ALU
 - IAR (Indirect Address Register) – registr využívaný při nepřímém adresování

Instrukce procesoru

- Existuje řada způsobů, jak zvolit instrukce, zakódovat instrukce a zakódovat data.
- Abychom si v našem případě situaci co nejvíce ulehčili, bude procesor pracovat s **16bitovými slovy**, které mohou obsahovat současně jak kód instrukce, tak i data.



Sada instrukcí (1)

Hexa kód:

0000 zastav provádění programu (**halt**)

0001 vytvoř dvojkový doplněk z ACC (**negate**)

$ACC := \neg ACC$

1xxx nahraj do ACC hodnotu **xxx** (**mload**)

pokud je znaménkový bit **xxx** nulový, potom

$ACC := 0xxx$ jinak $ACC := fxxx$

2xxx nahraj do ACC hodnotu z adresy **xxx** (**dload**)

$ACC := M[0xxx]$

3xxx nahraj do ACC hodnotu, která je uložena na adrese,
kterou definuje obsah buňky **xxx** (**iload**)

$ACC := M[M[0xxx]]$

4xxx ulož hodnotu z ACC na adresu **xxx** (**dstore**)

$M[0xxx] := ACC$

Sada instrukcí (2)

Hexa kód:

- 5xxx** ulož hodnotu z ACC na adresu, která je určena hodnotou paměťové buňky s adresou **xxx** (**istore**)
 $M[M[0xxx]] := ACC$
- 6xxx** změň PC na **xxx** (**Branch**)
 $PC := 0xxx$
- 7xxx** změň PC na **xxx** jestliže $ACC = 0$ (**BrZero**)
if $ACC = 0$ then $PC := 0xxx$
- 8xxx** změň PC na **xxx** jestliže $ACC > 0$ (**BrPos**)
if $ACC > 0$ then $PC := 0xxx$
- 9xxx** změň PC na **xxx** jestliže $ACC < 0$ (**BrNeg**)
if $ACC < 0$ then $PC := 0xxx$
- axxx** přičti k ACC obsah paměťové buňky na adrese **xxx** (**Add**)
 $ACC := ACC + M[0xxx]$

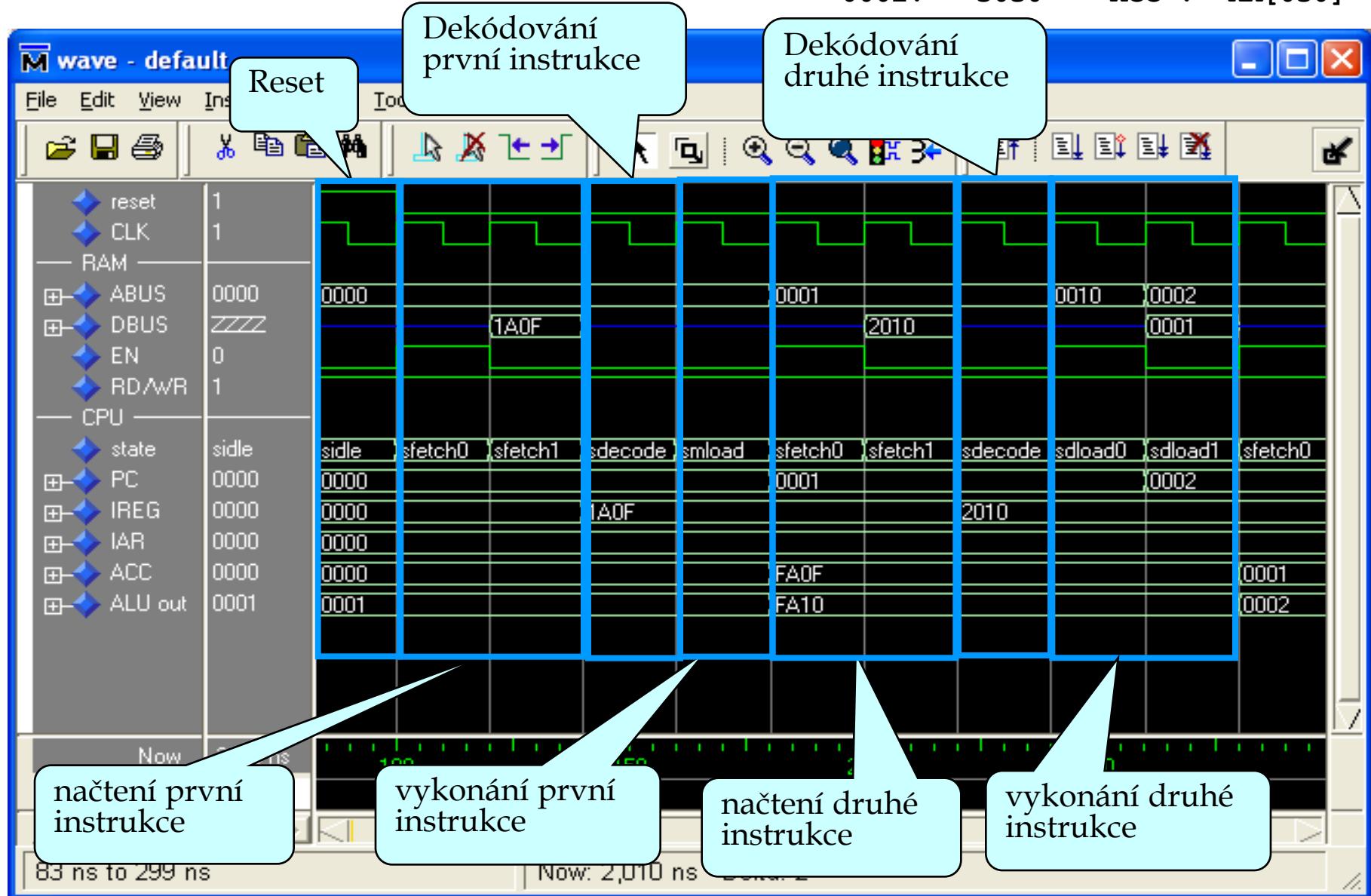
Příklad programu

Sečti hodnoty uložené na adresách 20-2f a zapiš výsledek na adresu 10.

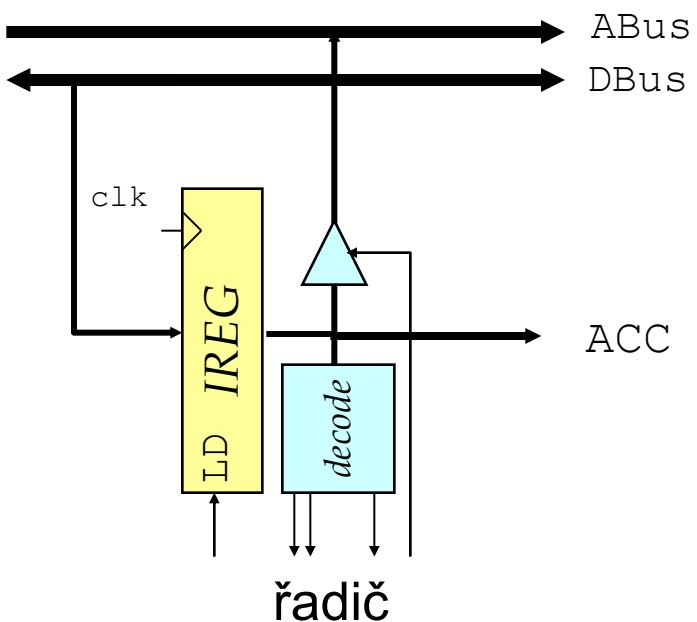
<u>Adresa</u>	<u>Instrukce</u>	<u>Komentář</u>
0000 (start)	1000 (ACC := 0000)	vynuluj součet S
0001	4010 (M[0010] := ACC)	
0002	1020 (ACC := 0020)	inicializuj pointer P
0003	4011 (M[0011] := ACC)	
0004 (loop)	1030 (ACC := 0030)	konec,pokud je P=030
0005	0001 (ACC := -ACC)	
0006	a011 (ACC := ACC+M[0011])	
0007	700f (if 0 goto 000f)	
0008	3011 (ACC := M[M[0011]])	S = S + *P
0009	a010 (ACC := ACC+M[0010])	
000a	4010 (M[0010] := ACC)	
000b	1001 (ACC := 0001)	P = P + 1
000c	a011 (ACC := ACC+M[0011])	
000d	4011 (M[0011] := ACC)	
000e	6004 (goto 0004)	goto loop
000f (end)	0000 (halt)	halt
0010		suma S
0011		pointer P

Př. časování instrukcí

Adresa	Hodnota	Význam
0000:	1A0F	ACC := FA0F
0001:	2010	ACC := M[010]
0002:	3030	ACC := MM[030]

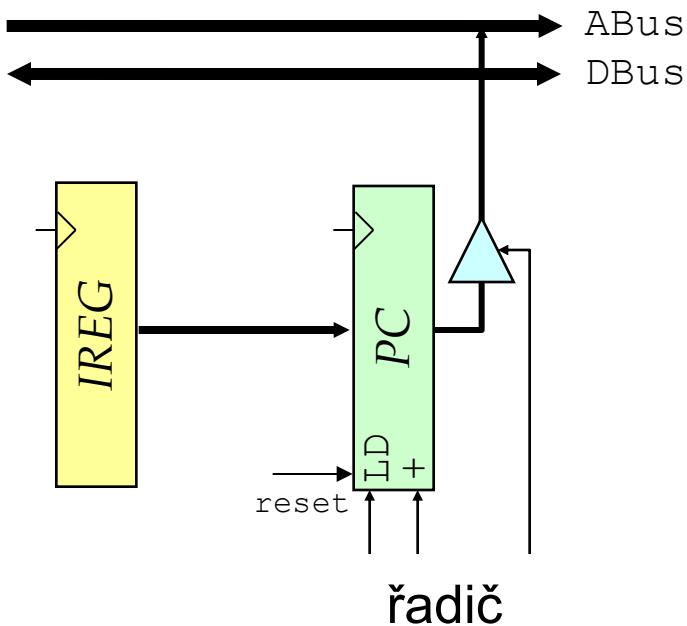


IREG a dekódování instrukce



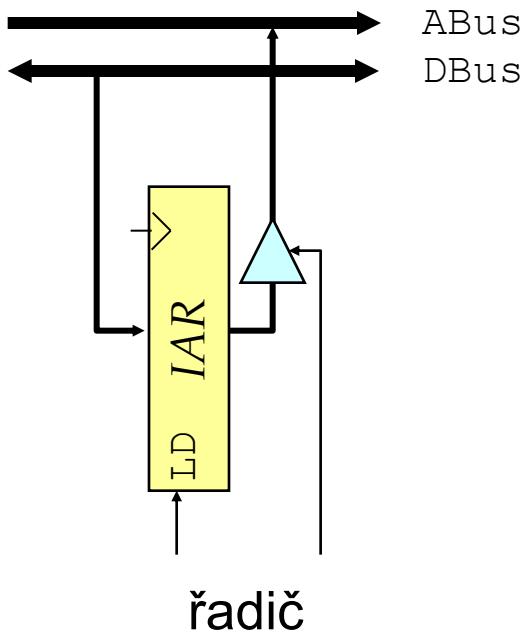
- Řadič vygeneruje signál LD a instrukce, která je na datové sběrnici, bude uložena do IREG.
- Pokud je instrukce v IREG, pak kombinační obvod *decode* nastaví na svém výstupu ty signály pro řadič, které budou v následujících taktech potřeba k vykonání dané instrukce. Např. pokud se jedná o instrukci ADD, potom se nastaví správná operace ALU, nastaví se datová cesta pro operandy apod.
- Vzhledem k tomu, že 16bitová slova přečtená z paměti obsahují jak instrukční kód tak i data, je třeba tato data přivést do střádače ACC.
- Při nepřímém adresování je součástí 16bitového slova, které je uloženo v IREG, i adresa paměti. Tuto adresu je třeba rovněž zpřístupnit na ABus. Za tímto účelem je vložen do schématu třístanovový budič, který je ovládán řadičem. Budič je aktivován jen v tom případě, kdy je třeba dát adresu na ABus, jinak je jeho výstup ve stavu vysoké impedance.

Programový čítač



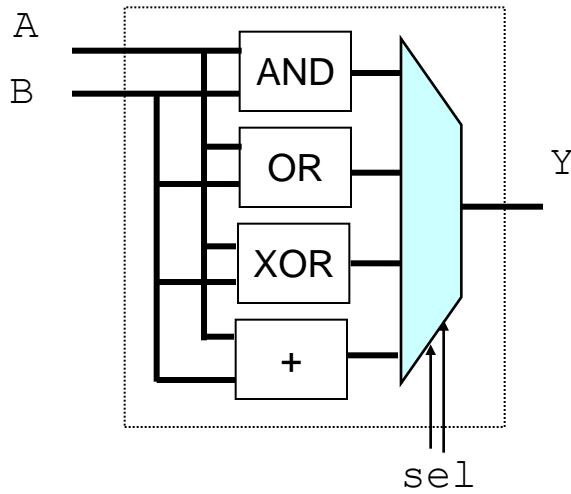
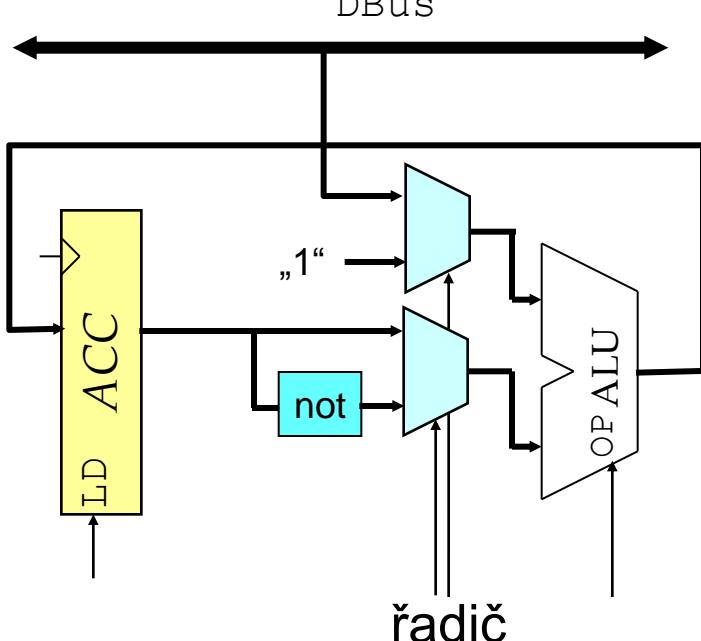
- Při resetu je nastavena počáteční hodnota PC = 0.
- Pokud se nejedná o skok, řadič generuje signál (+), který zvýší hodnotu PC o 1.
- Pokud se jedná o skok, je nová adresa uložena v IREG. Řadič generuje signál LD a načte tuto adresu do PC.
- Řadič zajistí vystavení hodnoty PC na ABus tak, že aktivuje řízení třístavového budiče.

Nepřímé adresování



- Při využití nepřímé adresace je na datovou sběrnici načtena adresa A, se kterou se bude následně pracovat.
- Adresa A je uložena do registru IAR tak, že řadič aktivuje signál LD.
- Pokud je následně třeba adresovat paměťovou buňku s adresou A, aktivuje řadič třístavový budič a tím je adresa A k dispozici na adresové sběrnici.

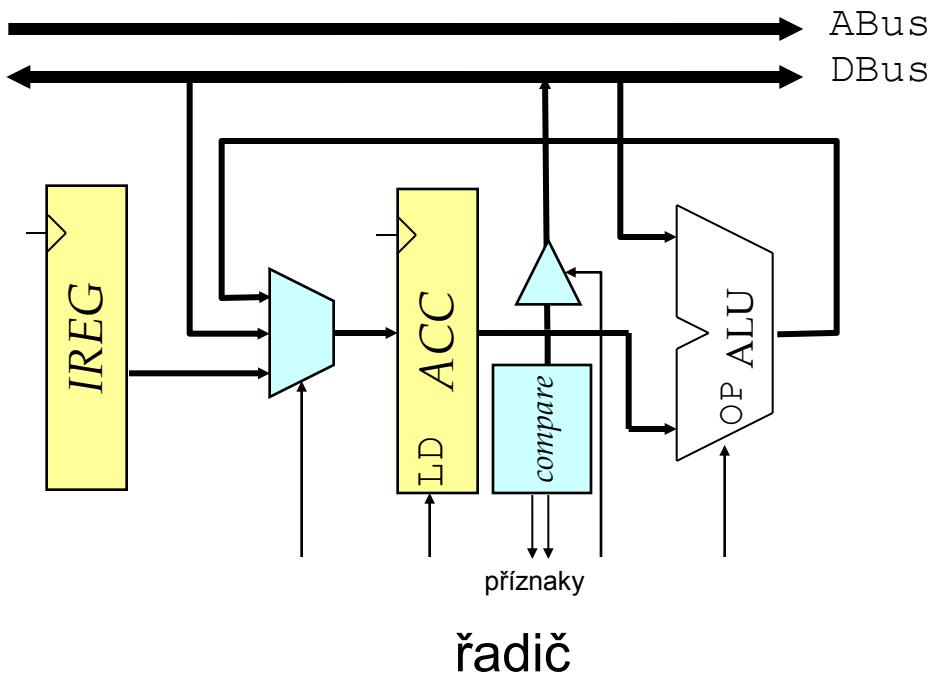
ALU



Princip realizace ALU

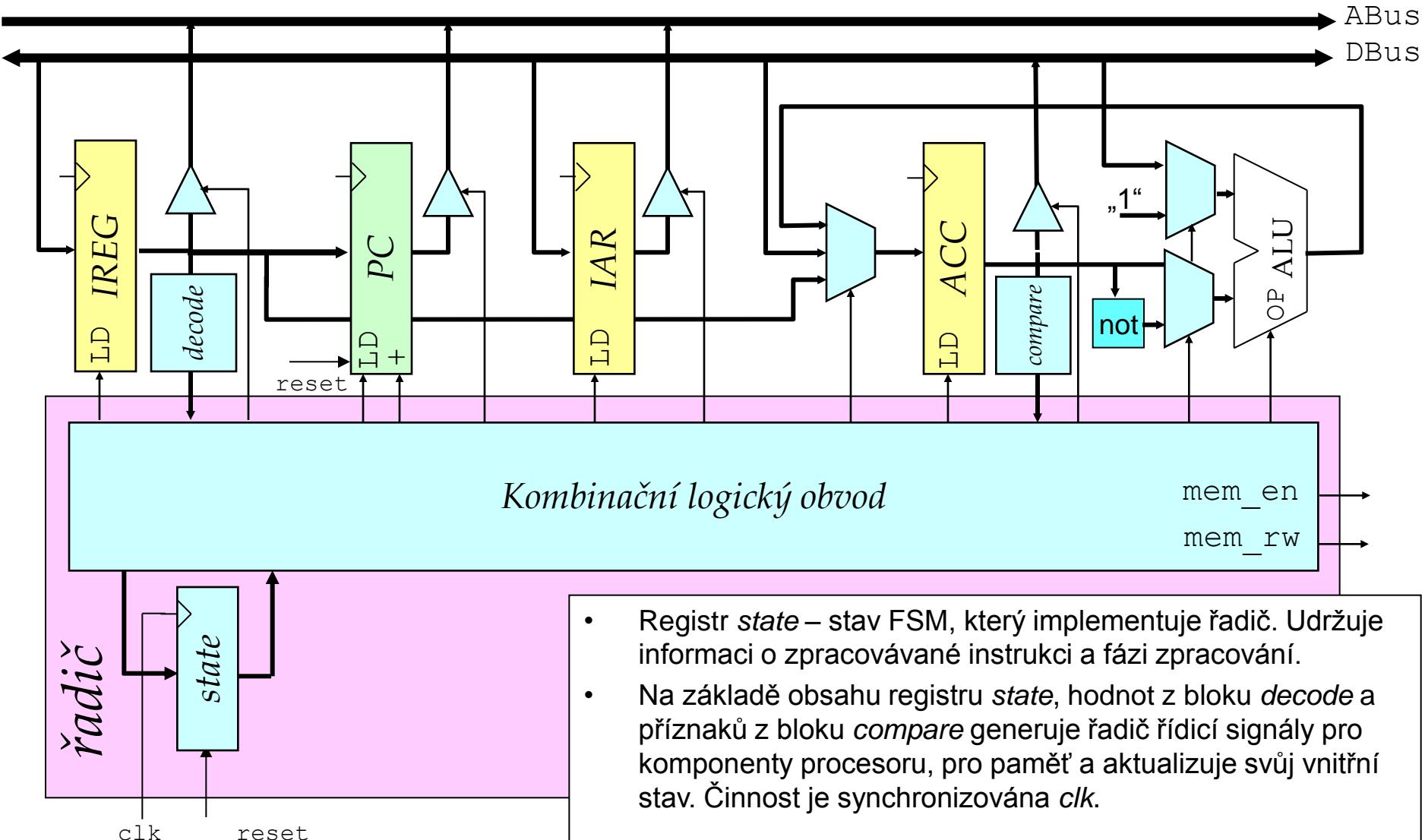
- ALU má dva operandy – střádač a hodnotu přečtenou z datové sběrnice.
- Výstup ALU je uložen do střádače, pokud řadič vygeneruje signál LD.
- Volba funkce ALU se děje pomocí signálu OP, který generuje řadič. Procesor zatím podporuje pouze instrukce sčítání a negace, ale není obtížné repertoár funkcí rozšířit.
- Pokud pracujeme v doplňkovém kódu a potřebujeme vytvořit zápornou hodnotu k obsahu střádače x , potom řadič vygeneruje signál pro multiplexory tak, aby došlo k sečtení negace x a „1“.

Střádač ACC



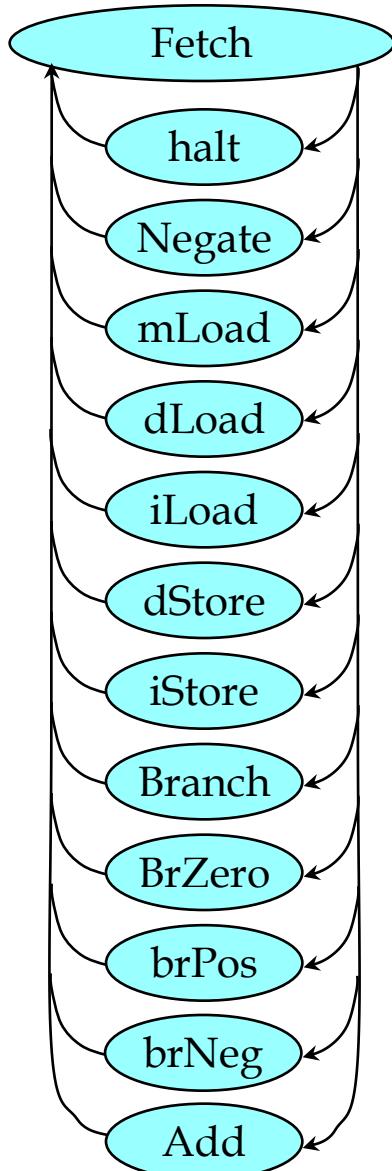
- Do střádače může být uložen buď výsledek operace ALU, hodnota dostupná na datové sběrnici nebo data uložená v IREG. Výběr provádí řadič pomocí řízení multiplexoru.
- Blok *compare* je kombinační obvod, který detekuje speciální stavy ACC – např. uloženou nulu, kladnou nebo zápornou hodnotu. Vytváří tak příznaky, na jejichž základě se rozhoduje řadič.
- Výstup ACC je možné zpřístupnit na datovou sběrnici (např. při instrukci store). Řadič musí aktivovat příslušný třístavový budič.
- Na obrázku nejsou nakresleny multiplexory na vstupu ALU, viz předchozí slide.

Celkové schéma procesoru



- Registr **state** – stav FSM, který implementuje řadič. Udržuje informaci o zpracovávané instrukci a fázi zpracování.
- Na základě obsahu registru **state**, hodnot z bloku **decode** a příznaků z bloku **compare** generuje řadič řídicí signály pro komponenty procesoru, pro paměť a aktualizuje svůj vnitřní stav. Činnost je synchronizována **c/k**.

Instrukční cyklus



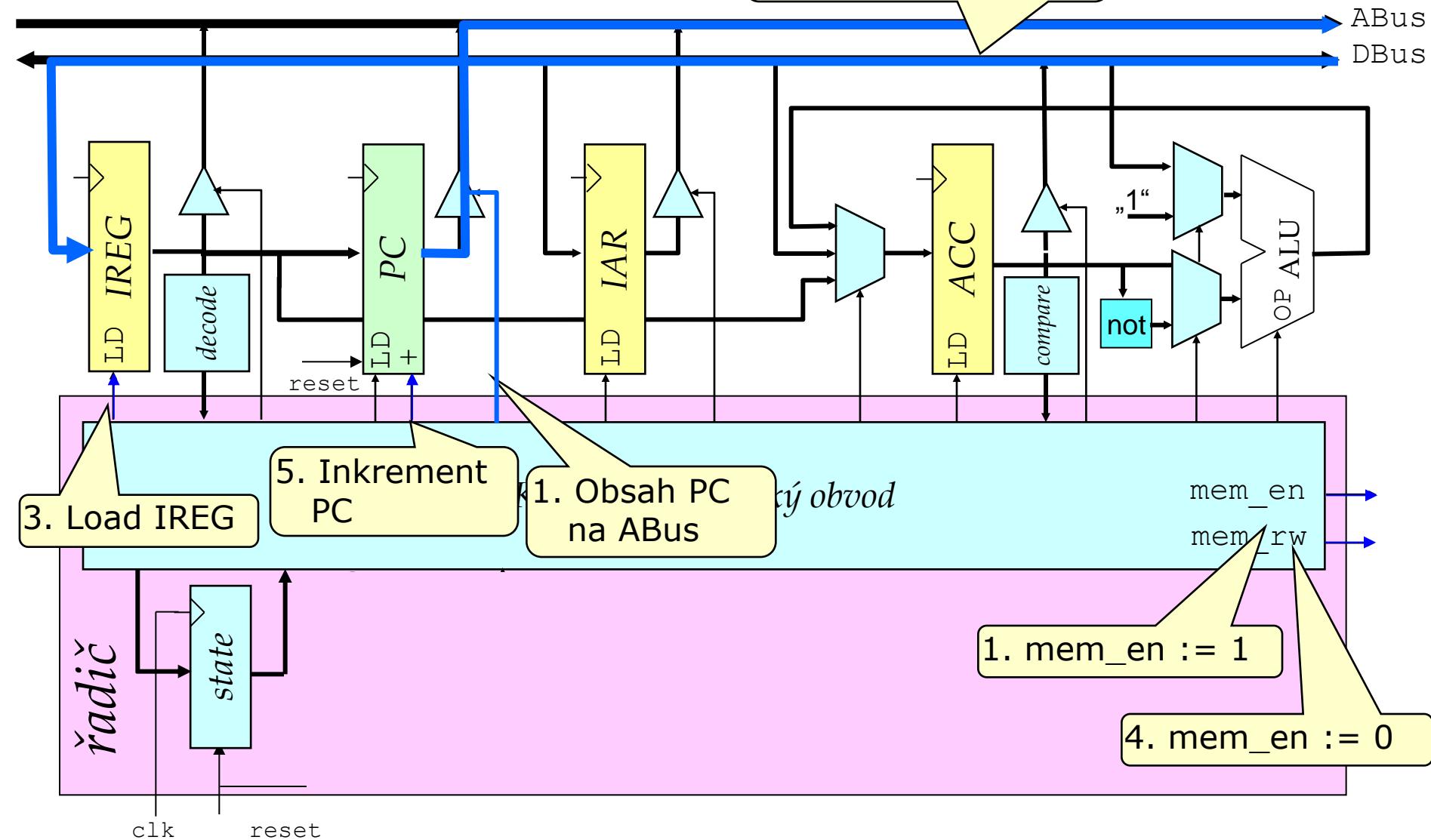
- Načtení instrukce (Instruction fetch)
 - Dle obsahu *PC* je přečtena instrukce z paměti
 - *PC* je inkrementován
- Dekódování instrukce (Instruction decode)
 - podle nejvyšších 4 bitů se určí, co se bude dělat
 - aktivují se příslušné obvody
- Provedení instrukce (Instruction execution)
 - načtení dalších potřebných dat
 - vykonání instrukce
 - zápis do paměti
 - modifikace *PC*, ACC atd.
 - může trvat pro různé instrukce různě dlouho

Vykonání instrukce - příklady

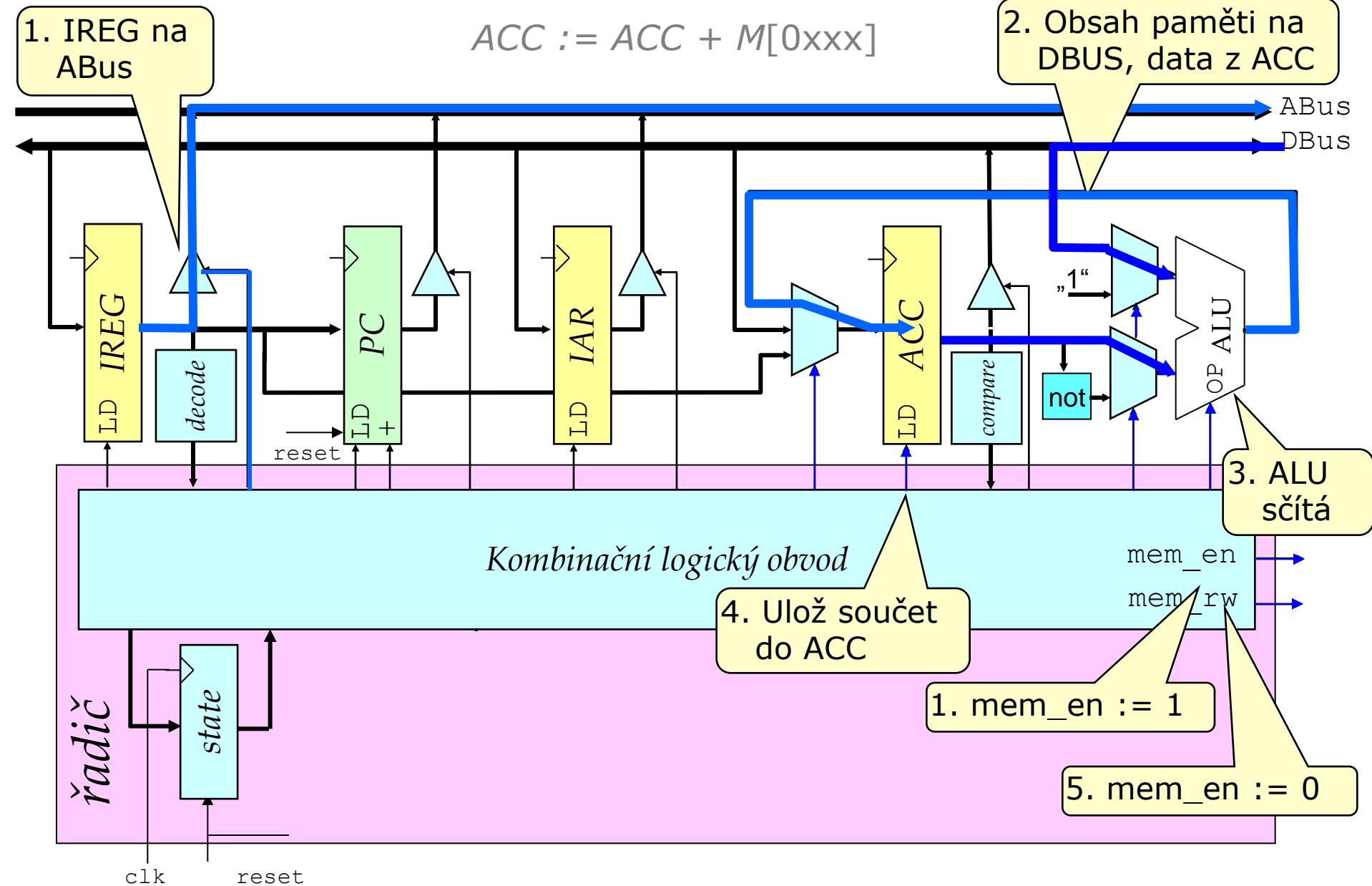
- Přímé čtení
 - přenes data z paměti do ACC, využij 12 nižších bitů jako adresu
 - vyžaduje generování signálů pro paměť a ACC
- Podmíněný skok
 - otestuj zda $ACC=0$ (nebo >0 nebo <0)
 - pokud ano, přenes nižších 12 bitů instrukčního slova do PC
- Nepřímý zápis
 - přenes data z paměti do IAR (Indirect Address Register) s využitím nižších 12 bitů instrukčního slova jako adresy
 - přenes data z ACC do paměti s využitím obsahu IAR jako adresy
 - vyžaduje zaslání hodnoty z IAR na adresovou sběrnici a nastavení signálů pro zápis do paměti

Fetch – načtení instrukce

2. Obsah paměti na DBus



Provedení instrukce ADD

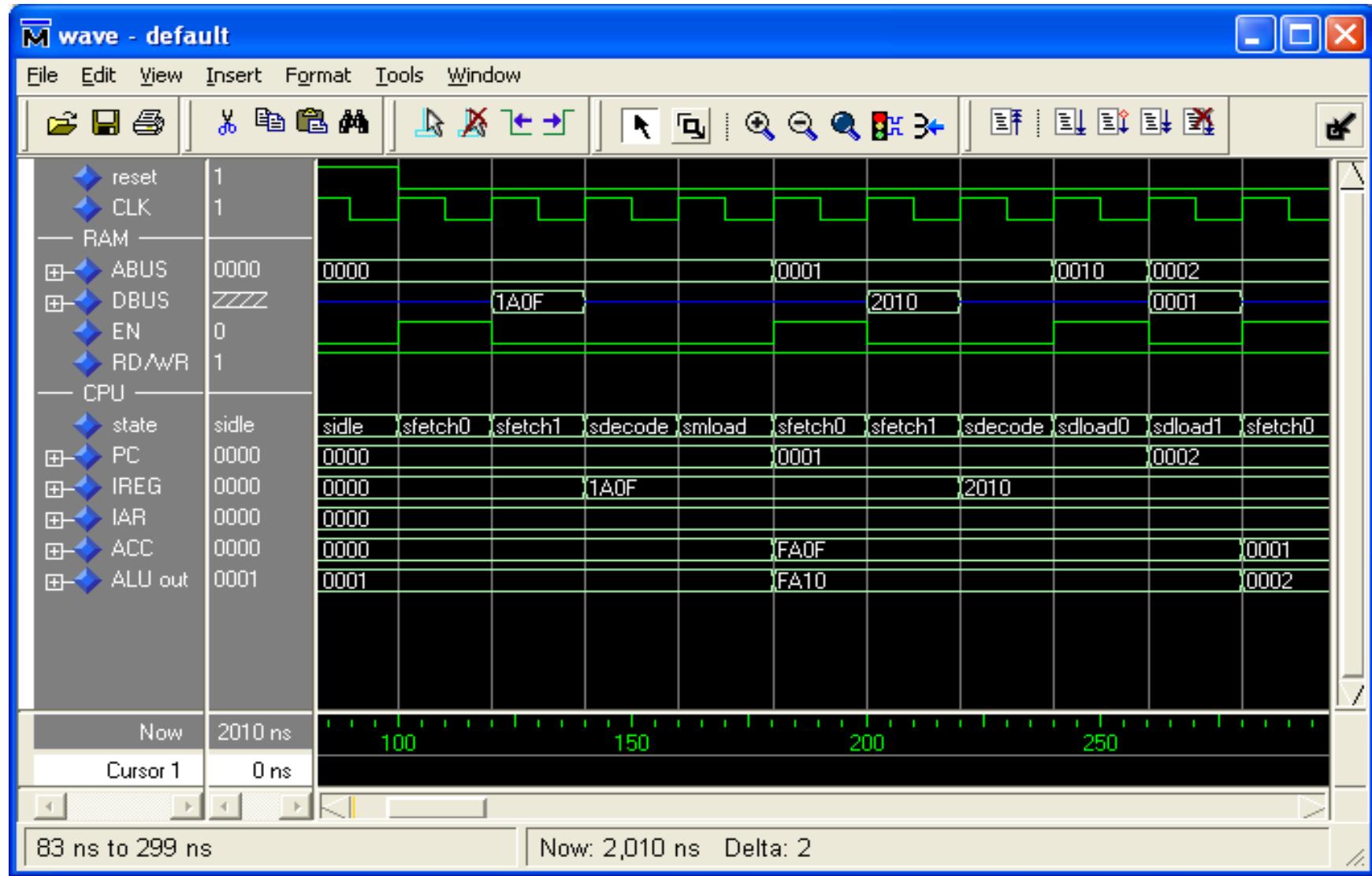


Testovací program

```
ram(0) <= x"1a0f"; -- immediate load
ram(1) <= x"2010"; -- direct load
ram(2) <= x"3030"; -- indirect load
ram(3) <= x"4034"; -- direct store
ram(4) <= x"0001"; -- negate
ram(5) <= x"2034"; -- direct load
ram(6) <= x"0001"; -- negate
ram(7) <= x"5032"; -- indirect store
ram(8) <= x"0001"; -- negate
ram(9) <= x"1fff"; -- immediate load
ram(10) <= x"a008"; -- add
ram(11) <= x"700d"; -- brZero
ram(12) <= x"0000"; -- halt
ram(13) <= x"1400"; -- immediate load
ram(14) <= x"8010"; -- brPos
ram(15) <= x"0000"; -- halt
ram(16) <= x"0001"; -- negate
ram(17) <= x"9013"; -- brNeg
ram(18) <= x"0000"; -- halt
ram(19) <= x"6015"; -- branch
ram(20) <= x"0000"; -- halt
ram(21) <= x"8014"; -- brPos
ram(22) <= x"7014"; -- brZero
ram(23) <= x"0001"; -- negate
ram(24) <= x"9014"; -- brNeg
ram(25) <= x"0000"; -- halt
ram(48) <= x"0031"; -- pointer for iload
ram(49) <= x"5af0"; -- target of iload
ram(50) <= x"0033"; -- pointer for istore
ram(51) <= x"0000"; -- target of istore
ram(52) <= x"f5af"; -- target of dstore
```

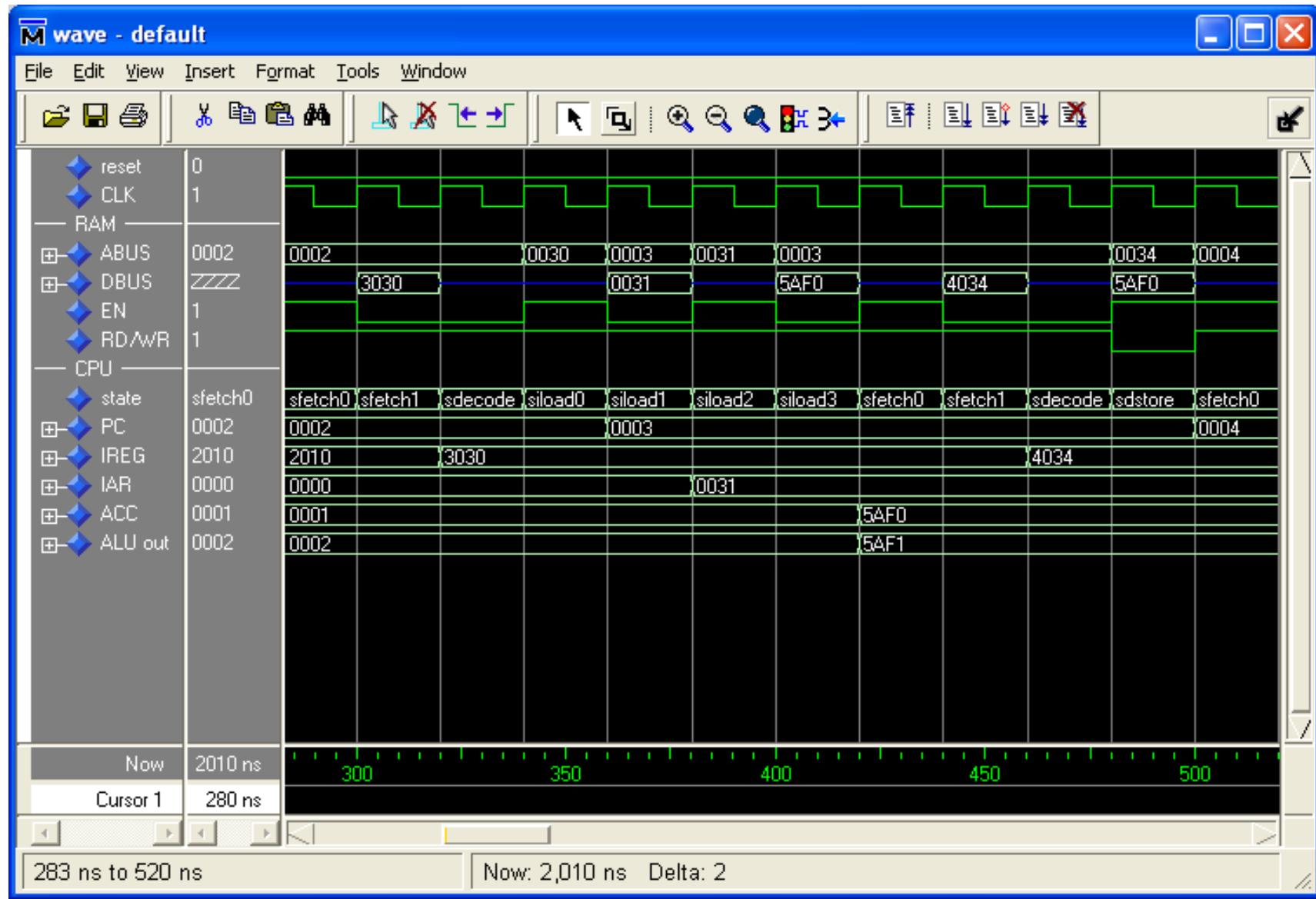
Simulace procesoru (1)

1a0f -- immediate load, 2010 -- direct load



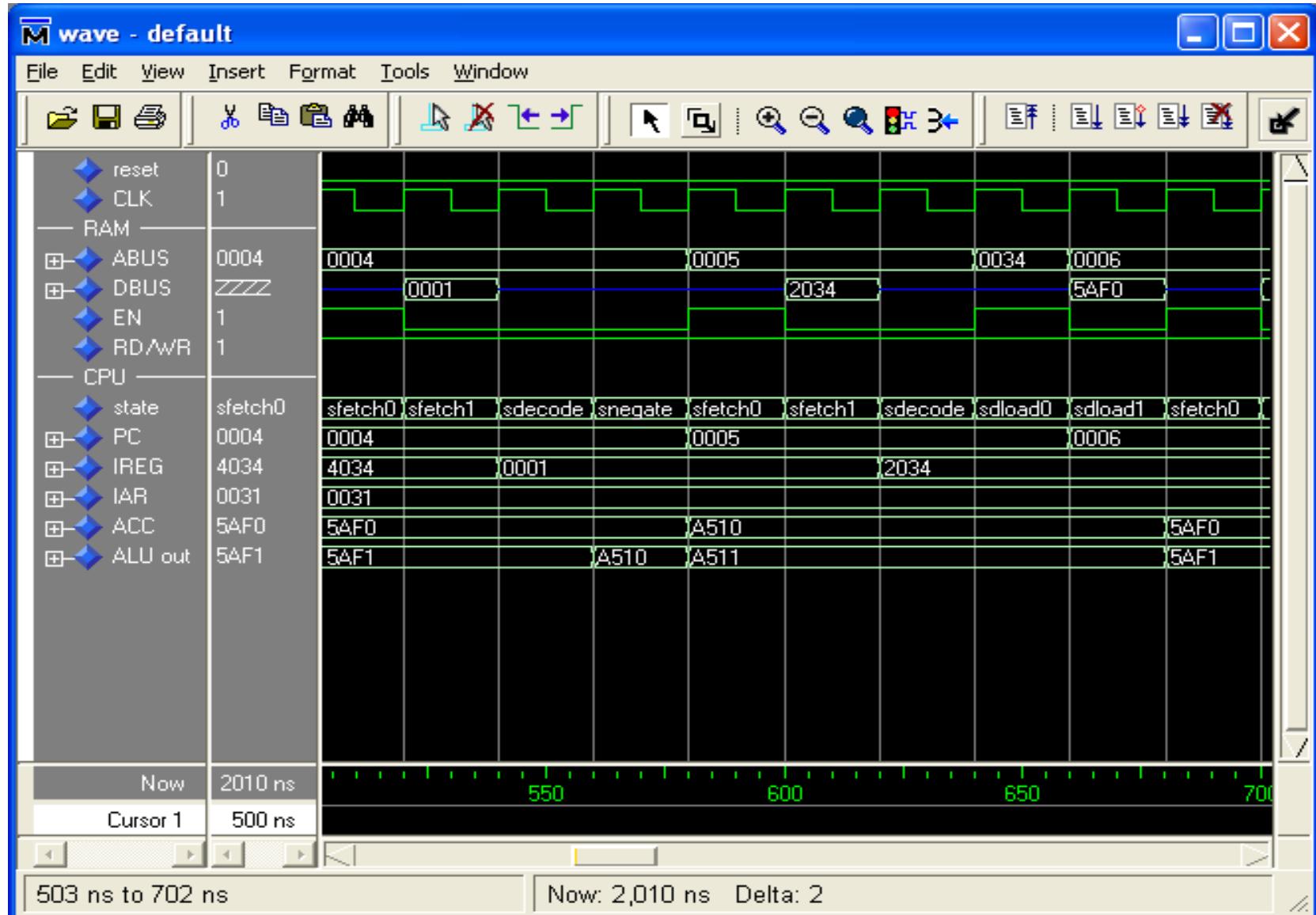
Simulace procesoru (2)

3030 -- indirect load, 4034 -- direct store



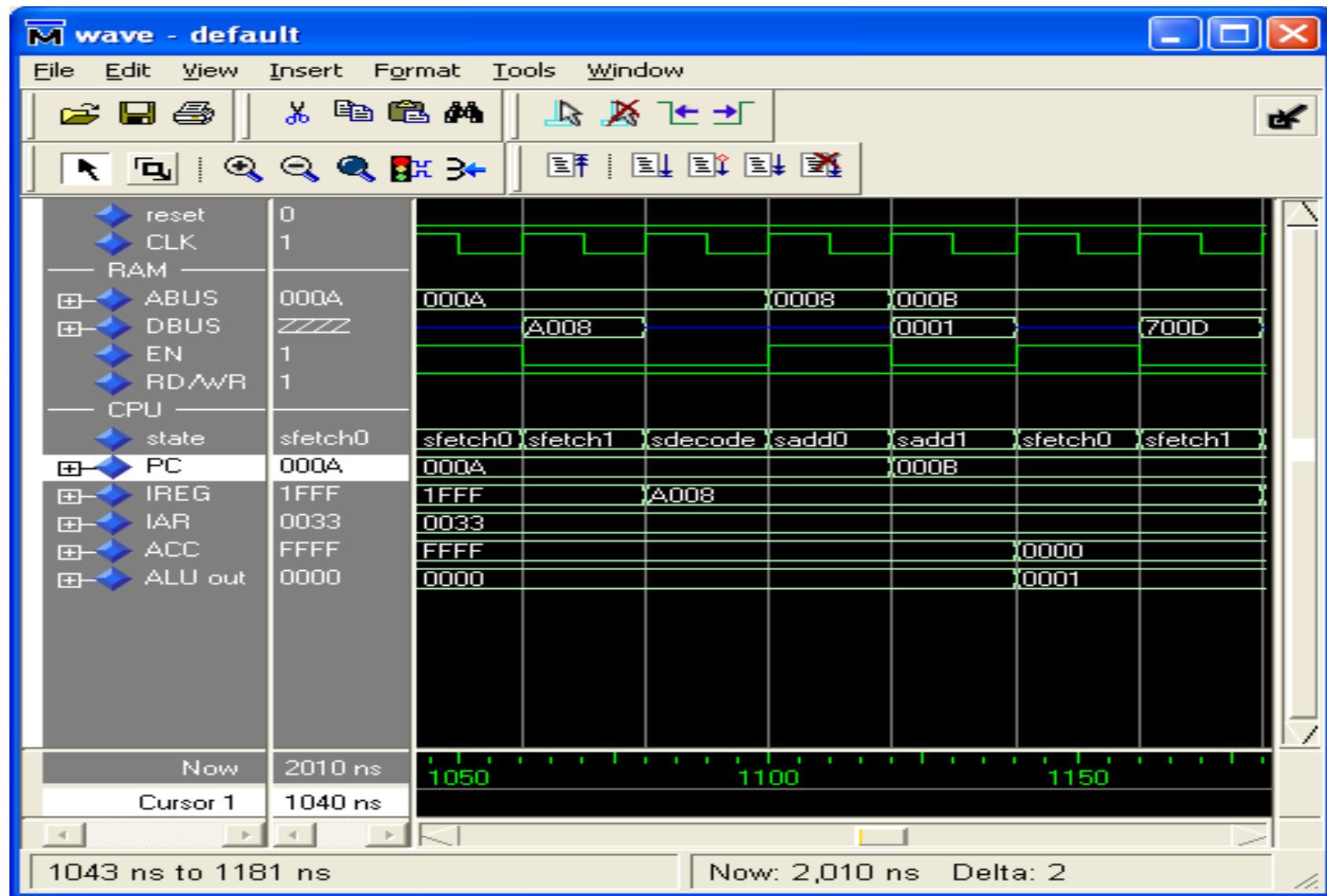
Simulace procesoru (3)

0001 – negate, 2034 -- direct load

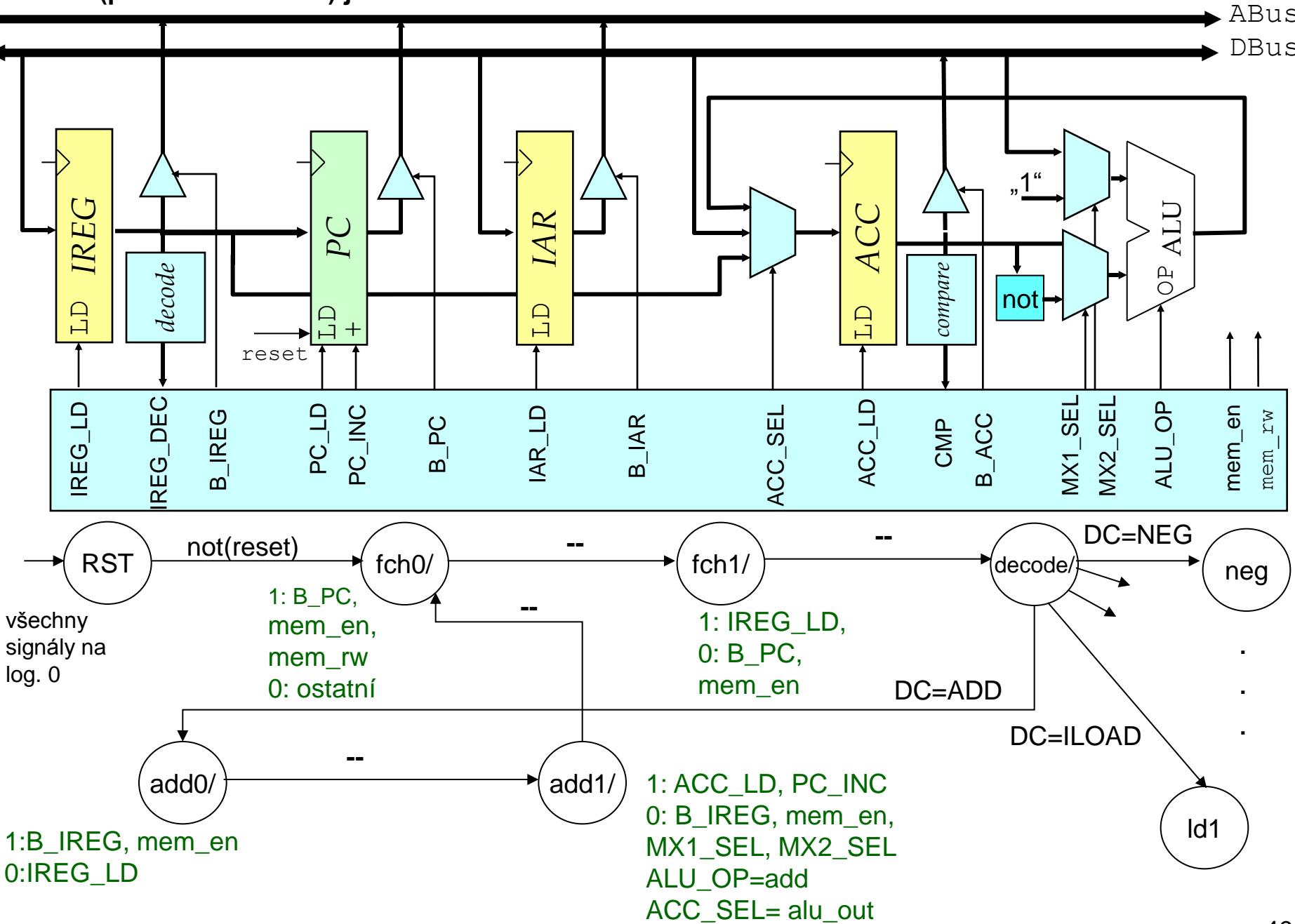


Simulace procesoru (4)

a008 -- add



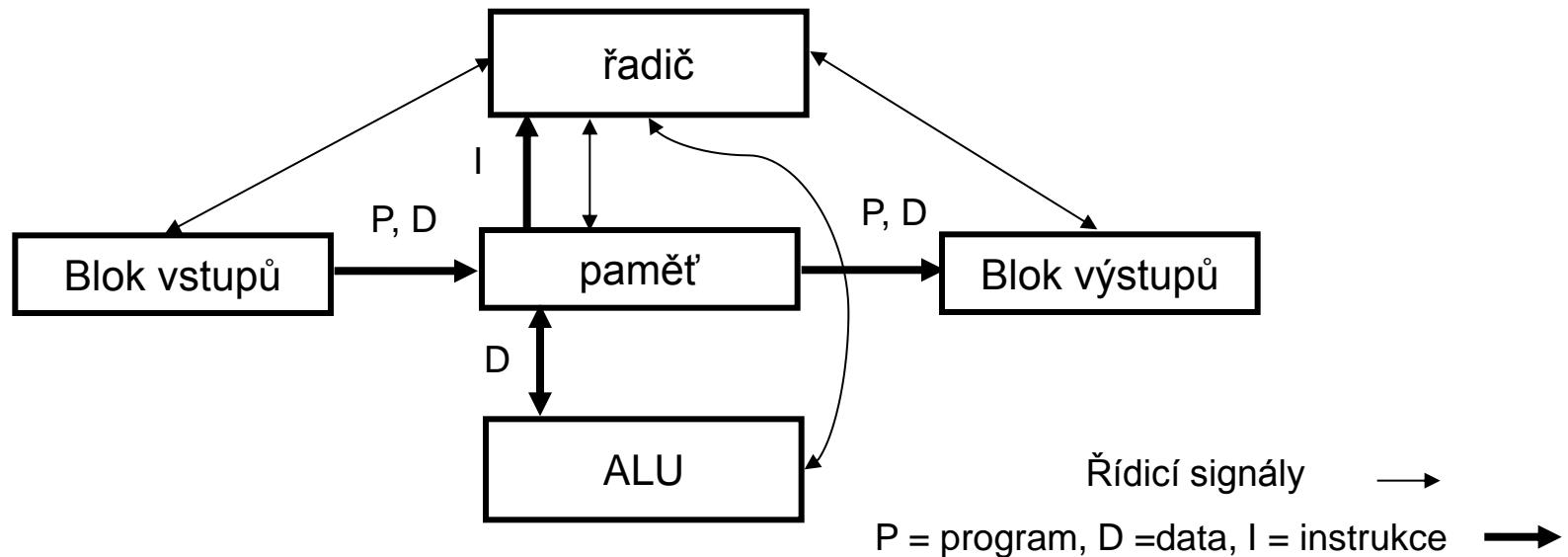
Řadič (pro instrukci ADD) jako Moorův automat.



Řadič – poznámky k obrázku

- Implementován jako Moorův automat (signály, které se (de)aktivují v příslušném stavu, jsou uvedeny vedle každého stavu).
 - Fetch – 2 stavy
 - Decode – 1 stav
 - ADD – 2 stavy
 - Další instrukce: 1-4 cykly (graf příslušných automatů neuveden, k přechodu dojde na základě hodnoty signálu DC)
- Pokud je aktivován signál reset, přejde se z libovolného stavu do stavu RST (příslušné hrany nejsou uvedeny).
- Značka „--“ označuje přechod, který se vždy vykoná.

Shrnutí: Von Neumannova architektura počítače



- Obvyklá terminologie
 - **Procesor** = ALU + řadič
 - **Základní jednotka** (CPU – central processing unit) = procesor + paměť
- Počítač je univerzální (= může řešit jakoukoliv algoritmizovatelnou úlohu)
 - má neměnnou strukturu
 - změna chování se dosahuje nahráním programu
- Von Neumannův počítač má tzv. princetonorskou koncepci – ve společné paměti jsou data i programy

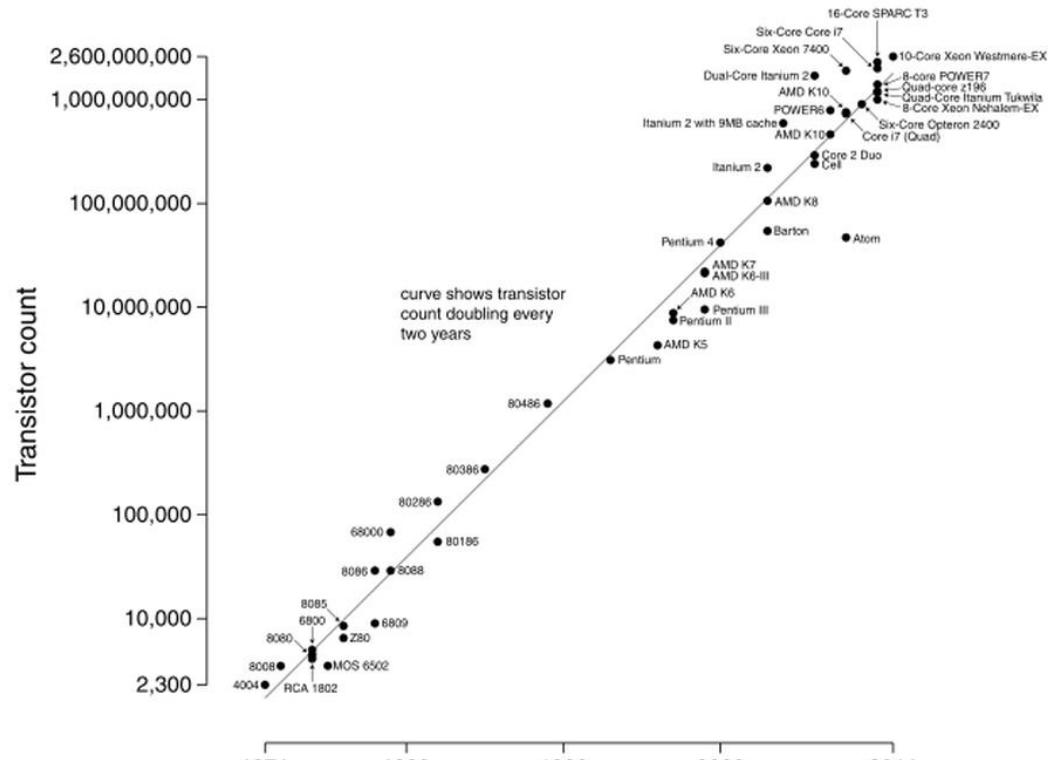
Zdokonalování von Neumannovy koncepce

- Indexregistry (1949), EDSAC, Universita Cambridge
- Jednotka pro operace s pohyblivou řádovou čárkou FPU (Floating Point Unit, 1954), IBM 704, NORC
- Přerušení programu (1956), UNIVAC 1103, Remington Rand
- Univerzální registry (1956), PEGASUS, Ferranti
- Asynchronní činnost vstup-výstupních periferních zařízení V/V PZ (1956), UNIVAC LARC, Remington Rand, založená na vybudování připojovacího kanálu se zapsaným programem
- Nepřímé adresování (1958), IBM 709
- Virtuální paměť (1959), ATLAS, Univ. Manchester
- V 60. letech již k zásadním zdokonalením architektury nedocházelo tak často, rozvoj se ubíral cestou zdokonalování technologie (následující slide), která vedla k zvětšování výkonnosti i kapacit pamětí.
- Za významné jevy 70. let považujeme integrovaný procesor (mikroprocesor) a použití rychlé vyrovnávací paměti cache, viz dále.
- V dalším období docházelo ke zvyšování úrovně paralelismu na různých úrovních: řetězené zpracování instrukcí, současné vydávání instrukcí, vícejádrové systémy ...
- S těmito a dalšími koncepty se v INP ještě setkáme.

Zdokonalování technologie: Moorův zákon

- Gordon Moore (Fairchild Semiconductor) si v r. 1965 všiml, že počet tranzistorů na čipu procesoru se vždy za 18 až 24 měsíců přibližně zdvojnásobí. Platí to dodnes a zdá se, že ještě několik let bude...

Microprocessor Transistor Counts 1971-2011 & Moore's Law



Zdroj: Wikipedia

Flynnova klasifikace

- Flynnova klasifikace je založena na sledování počtu **instrukčních** a **datových** proudů v počítači.
- Jeden (**single**) instrukční resp. datový proud označujeme symboly SI resp. SD, více než jeden proud (**multiple**) označujeme analogicky zkratkami MI resp. MD.

Příklad:

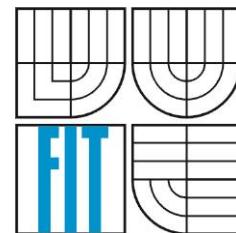
- SISD – von Neumannův počítač
- SIMD – vektorový procesor
- MISD – více počítačů zpracovává stejná data (řídicí počítač Space Shuttle)
- MIMD – multiprocesorový systém

Literatura

- Drábek, V. Výstavba počítačů. Skriptum VUT, 1995
- Turner, J.: Simple processor, kurz CS/EE 260, Washington University in St. Louis, 2003

Reprezentace dat

INP 2015
FIT VUT v Brně



Obsah přednášky

- Kód
- Čísla v pevné řádové čárce (FX)
- Chyby zobrazení (FX)
- Desítková čísla dvojkově kódovaná
- Huffmanův kód
- Čísla v pohyblivé řádové čárce (FP)
- Standard IEEE 754
- Některé vlastnosti FP

Pojem kód a typy kódů

- Kód je vzájemně jednoznačné přiřazení mezi symboly dvou množin.
- Data reprezentujeme pomocí kódů, které můžeme zhruba rozdělit do dvou skupin:
 - kódy pro vnější přenos dat (znaky) (ASCII, UNICODE atd.)
 - kódy pro vnitřní reprezentaci dat (doplňkový kód, BCD atd.),
- přičemž pro čísla zadaná ve formátu s pohyblivou řádovou čárkou (FP) se používají jiné kódy než pro čísla s pevnou řádovou čárkou (FX).

Typy kódů pro znaky dle Wikipedie

Early telecommunications	ASCII · ISO/IEC 646 · ISO/IEC 6937 · T.61 · BCD (6-bit) · Baudot code · Morse code · Chinese telegraph code
ISO/IEC 8859	-1 · -2 · -3 · -4 · -5 · -6 · -7 · -8 · -9 · -10 · -11 · -12 · -13 · -14 · -15 · -16
Bibliographic use	ANSEL · ISO 5426 / 5426-2 / 5427 / 5428 / 6438 / 6861 / 6862 / 10585 / 10586 / 10754 / 11822 · MARC-8
National standards	ArmSCII · CNS 11643 · GOST 10859 · GB 2312 · HKSCS · ISCII · JIS X 0201 · JIS X 0208 · JIS X 0212 · JIS X 0213 · KPS 9566 · KS X 1001 · PASCII · TIS-620 · TSCII · VISCI · YUSCI
EUC	CN · JP · KR · TW
ISO/IEC 2022	CN · JP · KR · CCCII
MacOS codepages ("scripts")	Arabic · CentralEurRoman · ChineseSimp / EUC-CN · ChineseTrad / Big5 · Croatian · Cyrillic · Devanagari · Dingbats · Farsi · Greek · Gujarati · Gurmukhi · Hebrew · Icelandic · Japanese / ShiftJIS · Korean / EUC-KR · Roman · Romanian · Symbol · Thai / TIS-620 · Turkish · Ukrainian
DOS codepages	437 · 667 · 668 · 720 · 737 · 770 · 773 · 775 · 790 · 819 · 850 · 851 · 852 · 853 · 854 · 855 · 857 · 858 · 860 · 861 · 862 · 863 · 864 · 865 · 866 · 867 · 868 · 869 · 872 · 895 · 912 · 915 · 932 · 991 · Kamenický · Mazovia · MIK · Iran System
Windows codepages	874 / TIS-620 · 932 / Shift JIS · 936 / GBK · 949 / EUC-KR · 950 / Big5 · 1250 · 1251 · 1252 · 1253 · 1254 · 1255 · 1256 · 1257 · 1258 · 28604 · 54936 / GB18030
EBCDIC codepages	37 / 1140 · 273 / 1141 · 277 / 1142 · 278 / 1143 · 280 / 1144 · 284 / 1145 · 285 / 1146 · 297 / 1147 · 420 / 16804 · 424 / 12712 · 500 / 1148 · 838 / 1160 · 871 / 1149 · 875 / 9067 · 930 / 1390 · 933 / 1364 · 937 / 1371 · 935 / 1388 · 939 / 1399 · 1025 / 1154 · 1026 / 1155 · 1047 / 924 · 1112 / 1156 · 1122 / 1157 · 1123 / 1158 · 1130 / 1164 · JEF · KEIS
Platform specific	ATASCII · CDC display code · DEC-MCS · DEC Radix-50 · ELWRO-Junior · Fieldata · GSM 03.38 · HP roman8 · PETSCII · TI calculator character sets · WISCII · ZX Spectrum character set
Unicode / ISO/IEC 10646	UTF-8 · UTF-16/UCS-2 · UTF-32/UCS-4 · UTF-7 · UTF-1 · UTF-EBCDIC · GB 18030 · SCSU · BOCU-1
Miscellaneous codepages	APL · Cork · HZ · IBM code page 1133 · KOI8 · TRON
Related topics	control character (C0 C1) · CCSID · Character encodings in HTML · charset detection · Han unification · ISO 6429 / IEC 6429 / ANSI X3.64 · mojibake

Číselné soustavy

Číselné soustavy dělíme na **polyadické** a **nepolyadické**.

Polyadická číselná soustava (angl. radix number system, číselná soustava se základem) má definovaný základ z (base, radix), $z \geq 2$.

V polyadické soustavě s jedním základem definujeme **z -adické číslice** a_i , $0 \leq a_i < z$, nejvyšší číslice $a_{max} = z - 1$. Obrazem čísla A

$$A = \sum_{i=-m}^{n-1} a_i z^i$$

je k -tice $a_{n-1} a_{n-2} \dots a_1 a_0, a_{-1} a_{-2} \dots a_{-m}$. Váhový součet z -adických číslic a_i , kde váhy jsou příslušné mocniny základu z , můžeme nazvat *polyáda*.

Používané základy jsou 2, 10, 16; z hlediska teorie kódování informace je optimální základ $e = 2,71828182\dots$, tedy nejbližší je základ 3.

Nepolyadické soustavy

Příkladem nepolyadické soustavy je soustava římských číslic

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

Tato soustava je pro počítání nevhodná.

Použitelná nepolyadická soustava je **soustava zbytkových tříd** (RNS - Residue Number System), označovaná jako **kód zbytkových tříd** KZT. Soustava je definovaná pomocí uspořádané k -tice vzájemně různých **prvočísel** základů z_0, \dots, z_{k-1} . Obrazem čísla A je uspořádaná k -tice celých čísel $a_0, a_1, a_2, \dots, a_{k-1}$, pro která platí

$$a_i = A \bmod z_i$$

	2	3	5
0	0	0	0
1	1	1	1
2	0	2	2
3	1	0	3
4	0	1	4
5	1	2	0
6	0	0	1
7	1	1	2
8	0	2	3
9	1	0	4
10	0	1	0
11	1	2	1
12	0	0	2
13	1	1	3
	...		

Soustava zbytkových tříd

Př. Máme zadány základy 2, 3, 5. Zbytkové třídy pak jsou:

$$\begin{array}{ccc} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & \\ 3 & & \\ 4 & & \end{array}$$

Např. číslo 5 pak vyjádříme trojicí zbytků po dělení zadanými základy, tedy (1 2 0). Jednoznačně lze vyjádřit pouze číslo A , pro které platí

$$A < \prod_i z_i$$

pro všechna i , tedy pouze číslo, které je menší než tzv. perioda, v našem příkladě je to $2*3*5 = 30$.

Soustava zbytkových tříd umožňuje rychlé operace sčítání, odčítání a násobení, protože se neuplatňují přenosy mezi jednotlivými stupni.

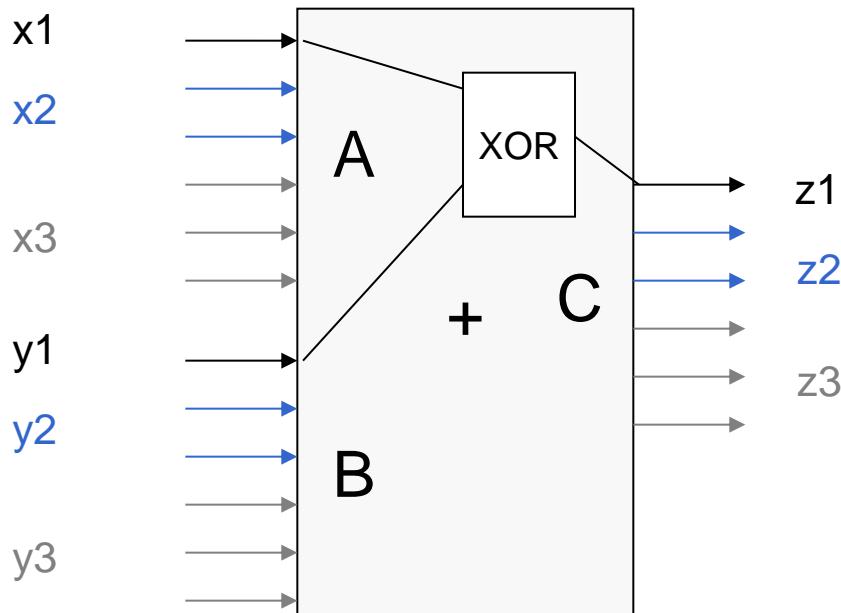
Dělení není jednoznačně definovaná operace, rovněž porovnávání velikosti čísel je prakticky obtížné.

Doba převodu do a zpět ze soustavy KZT může často pohltit časovou úsporu získanou na rychlých aritmetických operacích.

Rychlých operací v KZT se používá ve speciálních případech, např. v kryptografii (RSA provádí operace na 2048 bitech).

Realizace sčítáčky v KZT(2|3|5)

$$(z_1|z_2|z_3)_{KZT(2|3|5)} = (x_1|x_2|x_3)_{KZT(2|3|5)} + (y_1|y_2|y_3)_{KZT(2|3|5)}$$



Výpočet z_2 (operandy i výsledek na 2b)

x2	y2	z2
ab	cd	
00	00	00
00	01	01
00	10	10
01	00	01
01	01	10
01	10	00
10	00	10
10	01	00
10	10	01

$$u = f_1(a,b,c,d)$$

$$v = f_2(a,b,c,d)$$

Výpočet z_1 (operandy i výsledek 1b)

x1	y1	z1
0	0	0
0	1	1
1	0	1
1	1	0

Výpočty z_1 , z_2 a z_3 jsou
vzájemně nezávislé =>
rychlé!

Výpočet z_3 (operandy i výsledek na 3b)
obdobně jako z_2

$$r = f_1(e,f,g,h,i,j)$$

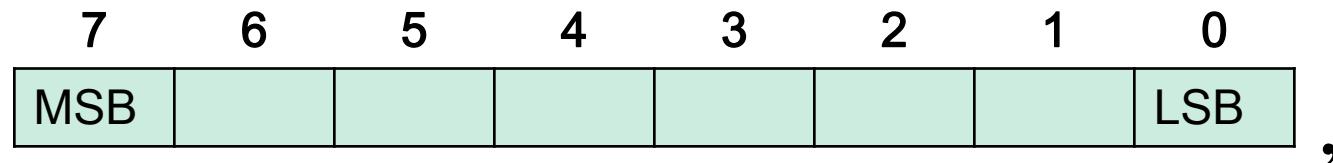
$$s = f_2(e,f,g,h,i,j)$$

$$t = f_3(e,f,g,h,i,j)$$

Opakování: Základní kódy

Př. Obrazy +7 a -7 na 8 bitech včetně znaménka

	+7	-7
Přímý kód se znaménkem	0000 0111	1000 0111
Inverzní kód (1- doplněk)	0000 0111	1111 1000
Dvojkový doplňkový kód	0000 0111	1111 1001
Kód se sudým posunutím (128)	1000 0111	0111 1001
Kód s lichým posunutím (127)	1000 0110	0111 1000



znaménko

řádová čárka

Význam kódových kombinací (8 bitů)

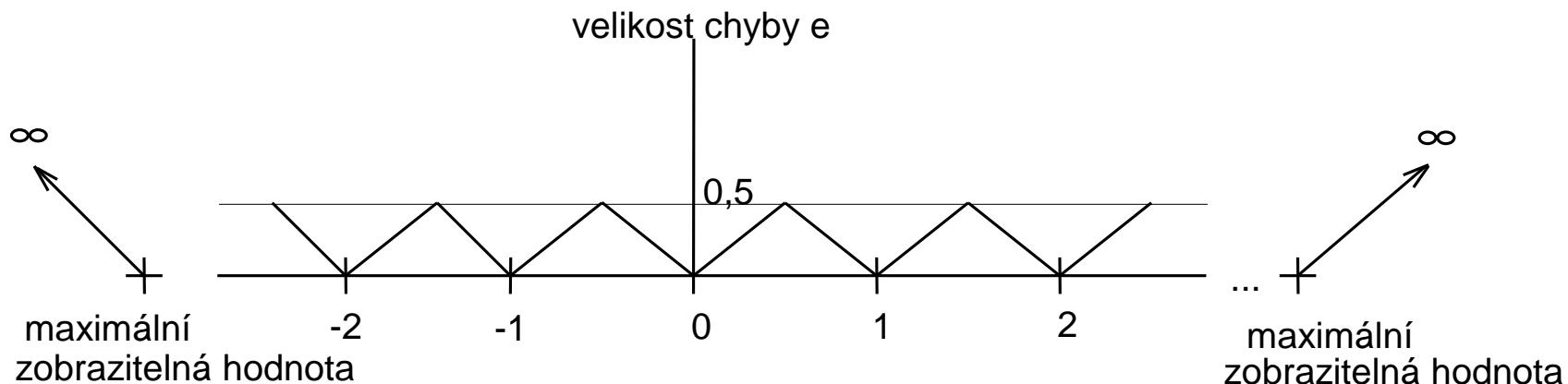
76543210	Význam v kódu				
	Přímý se zn.	Inverzní	Doplňkový	Se sud. pos.	S lich. pos.
00000000	0	0	0	-128	-127
00000001	1	1	1	-127	-126
00000010	2	2	2	-126	-125
...
01111110				-2	-1
01111111	127	127	127	-1	0
10000000	-0	-127	-128	0	+1
10000001	-1	-126	-127	+1	+2
...
11111110	-126	-1	-2	126	127
11111111	-127	-0	-1	127	128

Chyby zobrazení čísla FX

- Zobrazené číslo FX je zatíženo třemi typy chyb:
 - **chyba měření**: vzniká při pořizování čísla vlivem chyby metody měření
 - **chyba stupnice** (scaling): číselná soustava nemůže na konečném počtu míst vyjádřit přesně všechny hodnoty
 - **chyba zanedbáním** (truncation = odseknutí) a zaokrouhlením (rounding)

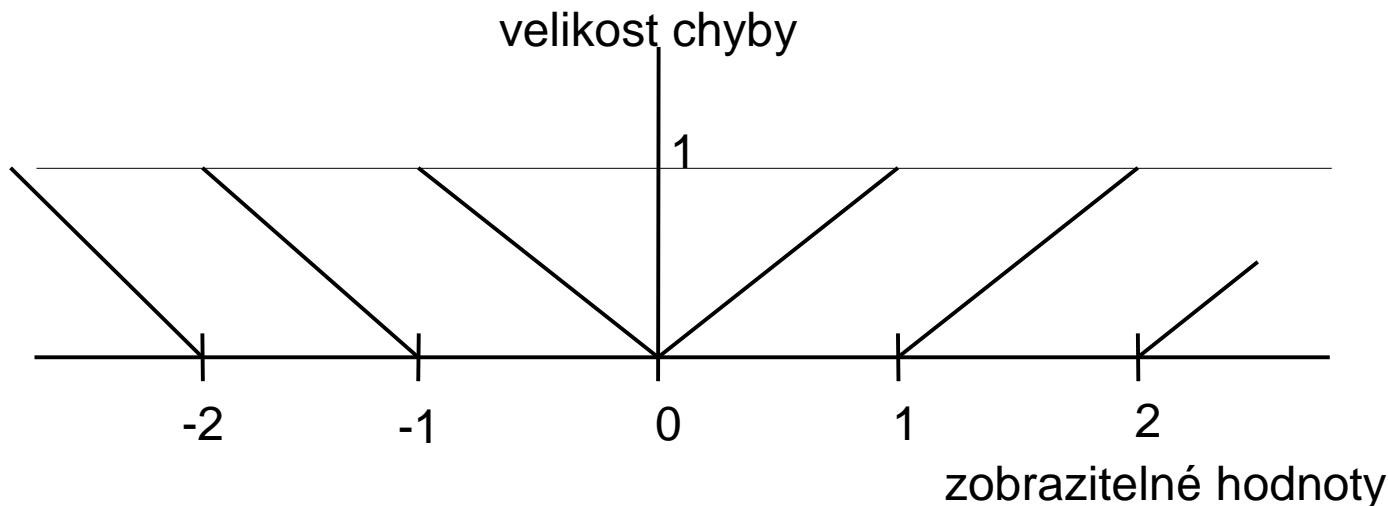
Chyba stupnice

- Na obrázku je průběh funkce chyby stupnice pro celá čísla s pevnou řádovou čárkou.
- Vidíme, že jsou přesně vyjádřena pouze celá čísla $0, 1, 2, 3 \dots$, kdy chyba = 0. Např. obraz čísla $1,5$ má maximální velikost chyby, a to $0,5$.
- Průběh funkce zobrazující velikost chyby je lineární mezi body celých čísel a čísla $0,5, 1,5, \dots$ atd. Od obrazu největšího (a nejmenšího) zobrazitelného čísla začíná chyba lineárně růst nad všechny meze.



Chyba zanedbáním a zaokrouhlením

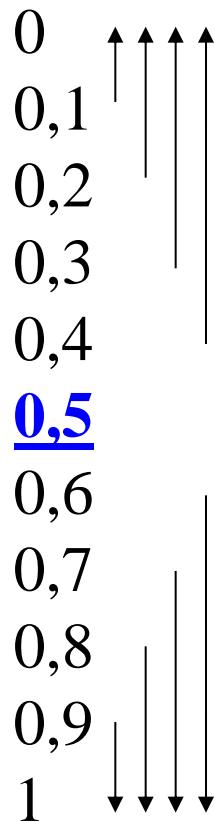
- Vidíme, že roste chyba zanedbáním a zaokrouhlením v intervalu $<0,1)$ od nuly do 1, a obdobně v dalším intervalu $<1,2)$, atd.



Pozn: Způsob zaokrouhlování je věcí konvence: Kč za zboží, body v kurzu

Statistické zaokrouhlení

Čísla "přesně uprostřed" zaokrouhuje jednou nahoru a jednou dolů. To se obvykle v praxi dělá zaokrouhlením na sudé (např. v normě IEEE), nebo na liché číslo.



Př. Zaokrouhlení k **sudému** číslu:

$$1,\underline{3}5 \rightarrow 1,4$$

$$1,\underline{4}5 \rightarrow 1,4$$

Desítková čísla dvojkově kódovaná

- Člověk pracuje s desítkovými čísly, kdežto nejpřirozenější vnitřní reprezentace v počítači je dvojková. Z toho vyplývá nutnost převodu čísel v obou směrech. Doba převodu však není zanedbatelná a proto se v počítačích často používá rovněž **aritmetika desítková**, která pracuje s desítkovými číslicemi kódovanými binárně.
- Označení **BCD** je vyhrazené pro jedený kód, přestože toto označení je obecně použitelné pro celou skupinu desítkových dvojkově vyjádřených kódů:

Číslice	BCD	ASCII	n + 3	2 z 5
0	0000	0011 0000	0011	11000
1	0001	0011 0001	0100	00011
2	0010	0011 0010	0101	00101
3	0011	0011 0011	0110	00110
4	0100	0011 0100	0111	01001
5	0101	0011 0101	1000	01010
6	0110	0011 0110	1001	01100
7	0111	0011 0111	1010	10001
8	1000	0011 1000	1011	10010
9	1001	0011 1001	1100	10100

Sčítání v kódu BCD

- Pro návrh desítkové aritmetiky (příslušných obvodů) je třeba zjistit aritmetické vlastnosti uvedených kódů.
- Analýzou sčítání dvou číslic v BCD zjistíme, že je-li binární součet **větší než 9**, je pro návrat do kódu BCD nutná **korekce**, a to přičtení konstanty **6** (binárně 0110).
- Nevýhoda BCD: Neúspornost, složitější HW

$$\begin{array}{r} 2 \ 0010 \\ +3 \ 0011 \\ \hline 5 \ 0101 \end{array}$$

$$\begin{array}{r} 5 \ 0101 \\ +6 \ 0110 \\ \hline ? \ 1011 \\ +K_1 \ 0110 = 6 \\ \hline 1 \ 0001 = 11_{dec} \end{array}$$

$$\begin{array}{r} 9 \ 1001 \\ +9 \ 1001 \\ \hline 1 \ 0010 = 12 \\ +K_1 \ 0110 \\ \hline 1 \ 1000 = 18_{dec} \end{array}$$

Počítání v kódu 2 z 5

- Kód 2 z 5 je neváhový kód, který kóduje informaci nadbytečným množstvím bitů, je tedy redundantní.
- Redundance se projevuje příznivě schopností kódu detekovat jednobitové chyby (viz dále).
- Jeho aritmetické vlastnosti jsou však natolik nepříznivé, že použití binární sčítáčky je prakticky nemožné. Je proto třeba navrhnout speciální sčítáčku, pracující v tomto kódu.
- Často se realizuje **tabulkou v paměti**.
- Adresu tvoří všechny kombinace hodnot vstupních operandů a obsah je hodnota výsledku včetně případného přenosu.
- Jaké je využití paměťové kapacity: adresových bitů je $5 + 5 = 10$, paměťových míst je tedy $2^{10} = 1024$. Využitých paměťových míst je $10 \times 10 = 100$. Využití paměti je $100:1024 = 9,76\%$.

Adresa ROM	Data ROM
...	...
00101 00110	01010
A=2 B=3	C=5
...	...

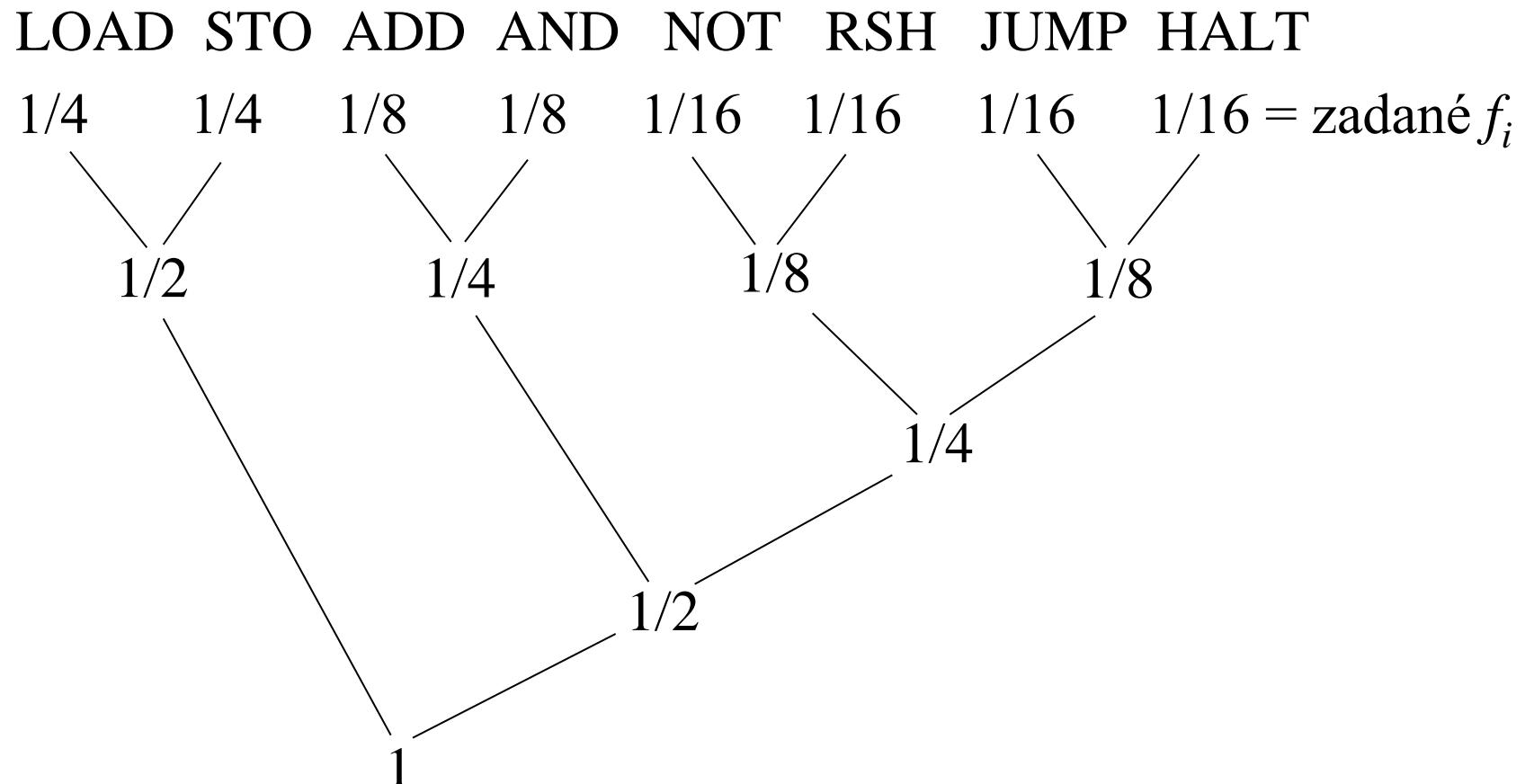
Huffmanův kód

- Jakým způsobem zakódovat znaky abecedy tak, aby častěji se vyskytující znaky byly zakódovány pomocí kratší binární sekvence?
 - Př. Morseovka, JPEG, ZIP, ...
- Huffmanovo kódování umožňuje optimálně vyřešit tento problém. Vychází ze známých frekvencí jednotlivých kódových značek. Pokud je četnost výskytu značek neznámá, musí se odhadnout.
- Huffmanovo kódování patří mezi kódy s proměnnou délkou (VLC – Variable Length Coding)
- Příklad: Máme zadán hypotetický instrukční soubor a frekvence výskytu jednotlivých instrukcí. Jak zakódovat častěji se vyskytující instrukce kratším kódem a zřídka se vyskytující instrukce delším kódem?

frekvence výskytu f_i

– LOAD	1/4
– STORE	1/4
– ADD	1/8
– AND	1/8
– NOT	1/16
– SHIFTR	1/16
– JUMP	1/16
– HALT	1/16

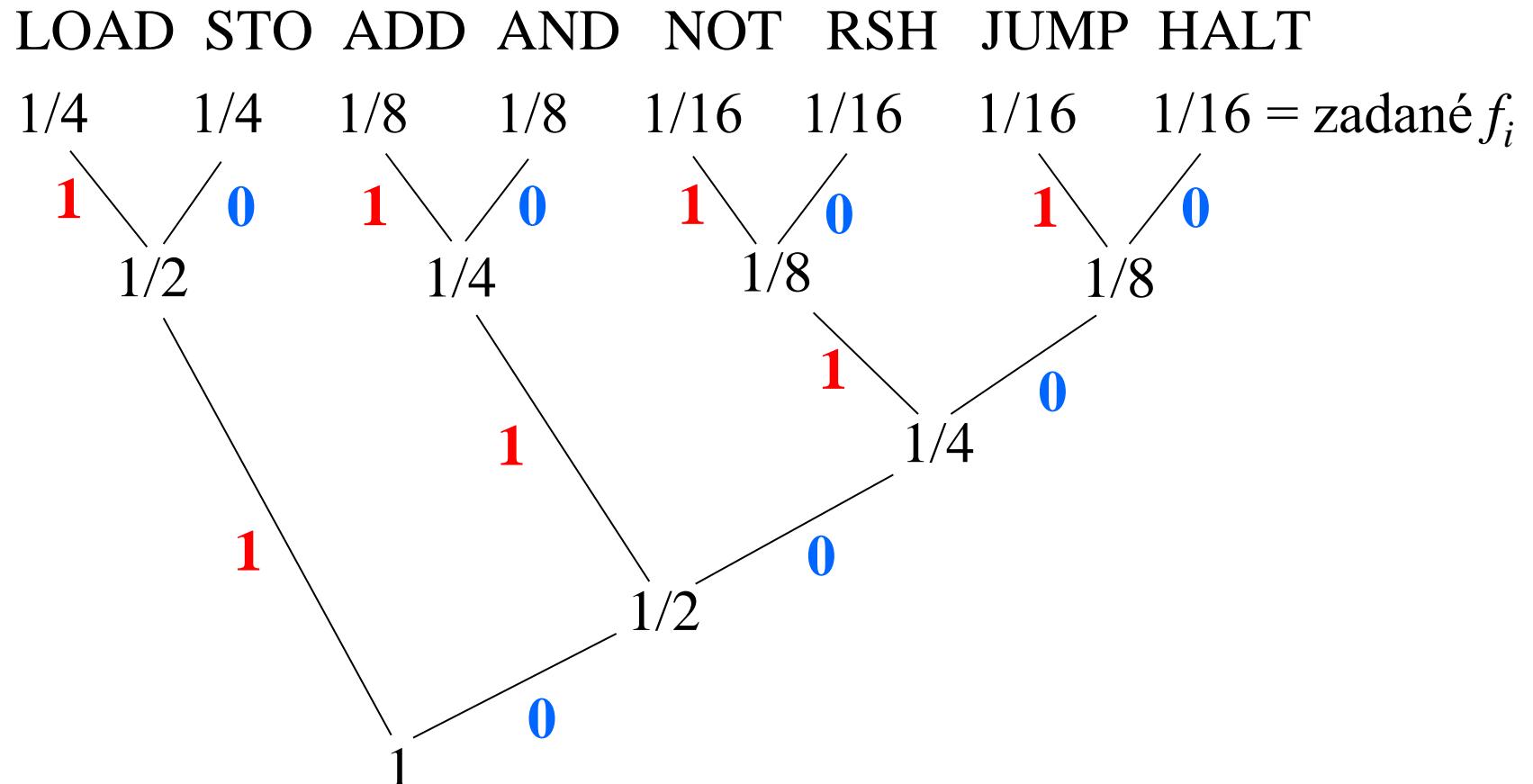
Př. Huffmanův kód



Huffmanův kód – postup

- Konstrukce stromu: Najdeme dvojici operačních znaků, jejichž součet pravděpodobnosti je nejmenší. Tyto dva znaky se nahradí společným uzlem v grafu s pravděpodobností výskytu danou součtem pravděpodobností. Dostali jsme tak skupinu znaků s počtem znaků o jedničku menším než dřív. Na této nové skupině opět hledáme dvojici s nejmenším součtem pravděpodobností. Naznačený postup opakujeme tak dlouho, až spojíme poslední dvojici do jednoho kořenového uzlu s pravděpodobností výskytu rovnou jedné.
- Kódování: Vycházíme z kořenového uzlu. Systematicky ohodnotíme hrany stromu (např. hrany vedoucí do uzlu s menším ohodnocením budou 0, jinak 1). Postup opakujeme tak dlouho, až označíme všechny hrany. Kód jednotlivých znaků zjistíme tak, že procházíme pro každý znak celou cestu od kořenového uzlu do příslušného listového uzlu a zaznamenáváme si popis hran, kterými procházíme.

Př. Huffmanův kód



Př. Huffmanův kód – zakódování

	kód	délka l_i	
• LOAD	11	2	
• STORE	10	2	
• ADD	011	3	
• AND	010	3	
• NOT	0011	4	
• SHIFTR	0010	4	
• JUMP	0001	4	
• HALT	0000	4	

Huffmanův kód je
prefixový.

Příklad 1: Dekódujte posloupnost 00110010111110011111010.

Příklad 2: Kolik bitů by bylo potřeba pro zakódování posloupnosti instrukcí z příkladu 1 pomocí standardního binárního kódování?

Míry kódů s proměnnou délkou

- Střední aritmetická délka (celkem je N značek) [bit]:
$$l_{ar} = \frac{1}{N} \sum_{i=1}^N l_i$$
- Střední dynamická délka [bit]:
$$l_{dyn} = \sum_{i=1}^N l_i f_i$$
- Teoreticky optimální délka [bit]:
$$l_{opt} = -\sum_{i=1}^N f_i \log_2 f_i$$
- Redundance kódu:
$$R = \frac{l_{dyn} - l_{opt}}{l_{dyn}}$$
- Pro náš příklad: $l_{ar} = 3,25; l_{dyn} = l_{opt} = 2,75; R = 0$
V tomto případě je sestrojený kód optimální!

Čísla v pevné vs. pohyblivé řádové čárce

- Pevná řádová čárka – FX

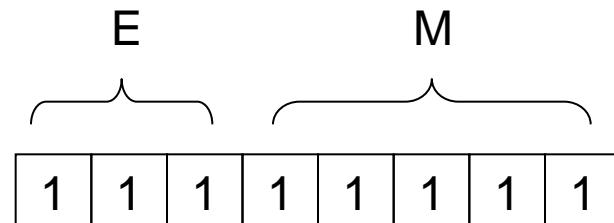
- bez „desetinné části“ (8 bitů)
 - Přímý kód: 0 až 255
 - Doplňkový kód: -128 až 127
 - aj.
- s „desetinnou částí“ (8 bitů)
 - Př. Přímý kód: 0 až 63,75
 - aj.
- Čísla jsou na číselné ose rozložena rovnoměrně.

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

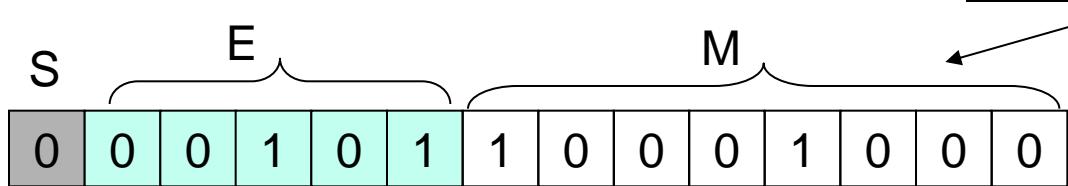
- Pohyblivá řádová čárka – FP

- $X = (-1)^S \cdot M \cdot B^E$
 - M je mantisa,
 - S je znaménko,
 - B je základ,
 - E je exponent
- Čísla nejsou na číselné ose rozložena rovnoměrně, což umožňuje zvýšit přesnost (více bitů M) nebo rozsah (více bitů E) oproti FX.

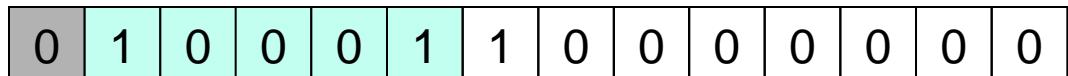


Příklad FP na 14 bitech

- E: 5 bitů, M: 8 bitů, S: 1 bit, B = 2
- Interpretace: $y = (-1)^S \cdot 0,M \times 2^E$
- Př. $17_{10} = 10001 \times 2^0 = 1000,1 \times 2^1 = 100,01 \times 2^2 = 10,001 \times 2^3 = 1,0001 \times 2^4 = \underline{0,10001 \times 2^5}$



- Př. $65536 = 2^{16} = 0,1 \times 2^{17}$ – se na 14 bitů FX v přímém kódu nevejde, ale v FP to lze



Příklad FP na 14 bitech

Problém 1: malá čísla nelze přesně zobrazit

- Potřebujeme záporný exponent
 - Řešení 1: Přidat znaménkový bit k exponentu – nepoužívá se
 - Řešení 2: Posunout exponent – používá se, je potom jednoduší obvodová realizace porovnání čísel v FP
- Skutečný_exponent = hodnota_pole_exponentu – BIAS
 - tj. exponent uložen v kódu s lichým nebo sudým posunutím
- Použijeme BIAS = 16 (polovina 2^5)
- Př. $17_{10} = 0,10001 \times 2^5$ (protože $16 + 5 = 21$)

0	1	0	1	0	1	1	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Př. $0,25_{10} = 0,1 \times 2^{-1}$ ($15 - 16 = -1$)

0	0	1	1	1	1	1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---

Příklad FP na 14 bitech

Problém 2: zobrazení čísel není unikátní

0	1	0	1	0	1	1	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---

0	1	0	1	1	0	0	1	0	0	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Př. $17_{10} = 0,10001 \times 2^5 = 0,010001 \times 2^6$
- Unikátnost podpoříme zavedením normalizované mantisy
 - nejlevější bit musí být 1
- Explicitní jednička – v nejlevějším bitu vždy musí být 1

0	0	1	1	0	0	1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---

 $= 0,1 \times 2^{-4} = 0,03125$
- Implicitní jednička
 - protože víme, že v nejlevějším bitu mantisy musí být vždy jednička, není nutné ji v mantise reprezentovat, ale stále ji uvažujeme
 - výhoda: získáme jeden bit rozlišení navíc
- Problém: Pokud se zavede normalizace, musí být nula ošetřena zvláštním způsobem

Příklad FP na 14 bitech

Problém 3: chyby zobrazení

- Rozsah zobrazení je $-0,1111111 \times 2^{15}$ až $+0,1111111 \times 2^{15}$.
- Nejmenší kladné číslo (pokud neuvažujeme normalizaci):
 $0,00000001 \times 2^{-16}$
- tj. například 2^{-39} nebo 2^{128} nelze zobrazit
- Není ale možné ani dostatečně přesně zobrazit např. 128,5.
- $128,5_{10} = 10000000,1$ je na 9 bitů, nejnižší bit se musí zanedbat nebo zaokrouhlit, vzniká chyba: $(128,5-128)/128,5 \sim 0,39\%$.
- Chyba se při použití výsledku v dalších operacích zvyšuje a zvyšuje.

Standard pro FP: IEEE 754

- Standard IEEE 754 z roku 1985, poprvé implementován v koprocesorech I 8087.
- IEEE 754-2008 rozšiřuje IEEE 754-1985; převzaly ho také ISO/IEC/IEEE 60559:2011
- Kromě definice B, M, E definuje standard další výjimečné situace.
 - Nečíselný výsledek, označený zkratkou NaN - Not a Number. Tento výsledek se ohlásí např. při výpočtu odmocniny z -1.
 - Definice nekonečna, které vznikne podílem 1/0. S tím souvisí definice aritmetiky na nekonečných hodnotách $+\infty$, $-\infty$, $1/\infty$, $\arctan(\infty) = \pi/2$, $\arccos(-1) = \pi$.
- decimal – přesně emuluje desítkové zaokrouhlování (účetnictví...)

IEEE 754-2008	IEEE 754-1985	bitů	základ	znaménko	exponent	mantisa	pozn.
binary16	-	16b	2	1b	5b	10+1b ^(*)	poloviční přesnost, "Half"
binary32	single	32b	2	1b	8b	23+1b	základní přesnost
binary64	double	64b	2	1b	11b	52+1b	dvojitá přesnost
-	extended	80b	2	1b	?	?	dvojitá rozšířená přesnost
binary128	-	128b	2	1b	15b	112+1b	čtyřnásobná přesnost
decimal32 ^(x)	-	32b	10	1b	-95 až +96	7 číslic	základní přesnost
decimal64 ^(x)	-	64b	10	1b	-383 až +384	16 číslic	dvojitá přesnost
decimal128 ^(x)	-	128b	10	1b	-6143 až +6144	34 číslic	čtyřnásobná přesnost

IEEE 754: Vybrané formáty čísel

	rozsah	přesnost mantisy	zlomková část f	
krátké reálné	$10^{\pm 38}$	24 bitů	S E7...E0 F1...F23	F0 je implicitní
dlouhé reálné	$10^{\pm 308}$	53 bitů	S E10...E0 F1...F52	F0 je implicitní

Číslo N se získá z hodnoty E uvedené v poli exponentu a z hodnoty z pole mantisy (zlomková část) podle vzorce

$$N = (-1)^S (2^{E - BIAS}) (F_0, F_1 \dots F_{23}, \text{ nebo } F_{52}), \text{ kde}$$

$$BIAS = 127 \text{ nebo } 1023.$$

Mantisa je vyjádřena přímým kódem se znaménkem, exponent kódem s lichým posunutím. Pozor, v poli exponentu je uvedeno číslo zvětšené o hodnotu $BIAS$. Rozlišujeme tedy pojmy **pole exponentu**, což je posunutý exponent, a **exponent**, resp. neposunutý exponent.

Zlomková část f udává číslo menší než 1. Mantisu však získáme součtem $1 + f$, což můžeme zapsat $1.f$.

IEEE 754: Příklad

Příklad.

Jaké číslo je zaznamenáno na 32 bitech v jednoduché přesnosti?

1 1000 0001 0100 0000 0000 0000 0000 000

v poli exponentu je číslo 129

exponent je tedy $129 - 127 = 2$

zlomková část $f = ,01_2 = ,25$

mantisa je tedy 1,25

Jde tedy o číslo $-1,25 \cdot 2^2 = -5$

IEEE 754: Výjimečné hodnoty v „single precision“

Povolené hodnoty exponentu čísel leží v intervalu $<-126, +127>$, po posunu $+127$ získáme povolené hodnoty v poli exponentu $<1, 254>$. Je-li tedy v poli exponentu 0, nebo 255, jde o hodnoty vyhrazené pro speciální účely:

pole exponentu	Zlomková část	Význam
255	0	$\pm \infty$
255	$\neq 0$	NaN – je jich mnoho
0	0	0
0	$\neq 0$	subnormalizované číslo - nenaplnění

Subnormalizované (denormalizované) číslo: nepočítá se se skrytou 1 a exponent je chápán jako -126.

IEEE 754: Příklady

X	Reprezentace X v IEEE 754 – single precision
1,0	0 0111111 000000000000000000000000000
2,0	0 1000000 000000000000000000000000000
19,5	0 1000011 001110000000000000000000000
-3,75	1 1000000 111000000000000000000000000
0 (spec.)	0 0000000 000000000000000000000000000
+/- nekonečno	0/1 1111111 000000000000000000000000000
NaN	0/1 1111111 cokoliv nenulového
Denormalizované číslo	0/1 0000000 cokoliv nenulového

IEEE 754: Rozsah „single precision“

- MAX = $2^{127} \times 1,11111111111111111111111$
 - je to normalizované číslo $\sim 3,4 \times 10^{38}$
- MIN = $2^{-126} \times 0,00000000000000000000000000000001$
 - je to denormalizované číslo $\sim 1,4 \times 10^{-45}$
- Čtyři intervaly nelze reprezentovat
 - Záporná čísla menší než $-\text{MAX}$ (negative overflow)
 - Záporná čísla větší než $-\text{MIN}$ (negative underflow)
 - Kladná čísla menší $+\text{MIN}$ (positive underflow)
 - Kladná čísla větší než $+\text{MAX}$ (positive overflow)

IEEE 754: Zaokrouhlování

- K zaokrouhlování dochází v případě, že dané číslo nelze přesně vyjádřit.
 - Např. při násobení v desítkové soustavě máme výsledek operace $2,1 \times 0,5 = 1,05$ zaokrouhlit na 2 významové číslice. Je věcí konvence, zda za výsledek prohlásíme 1,1, nebo 1,0. Oba výsledky jsou zatíženy stejně velkou chybou.
- Norma IEEE zaokrouhuje na číslo, jehož nejnižší číslice je **sudá** (ve dvojkové soustavě). Zaokrouhlovací procedura je definovaná pro 4 případy:
 - Zaokrouhlení k nejbližšímu číslu
 - Zaokrouhlení k nule
 - Zaokrouhlení k $+\infty$
 - Zaokrouhlení k $-\infty$

Absolutní chyba zobrazení

- Maximální (absolutní) chyba zobrazení **Err** čísel FP (resp. vzdálenost zobrazitelných bodů) závisí na počtu číslic v mantise a na intervalu definovaném v exponentu:

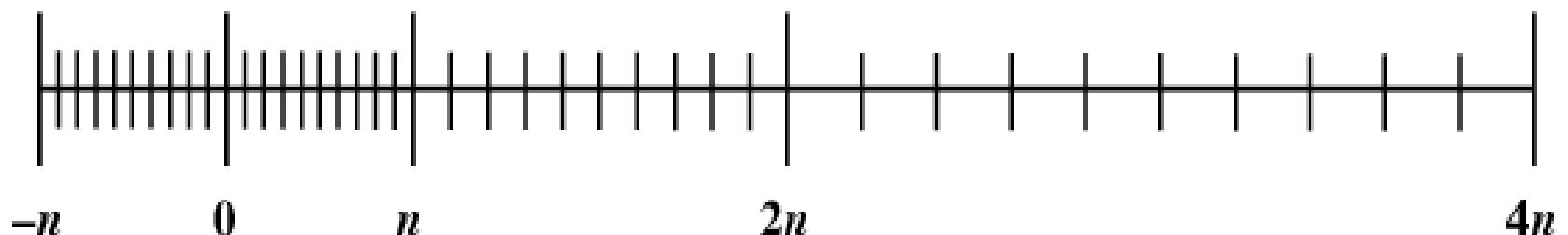
$$\begin{aligned} \text{Err} &= \frac{\text{délka intervalu stupnice pro jistou hodnotu exponentu}}{\text{počet možných číselných kombinací v mantise}} = \\ &= \frac{\text{základ}^{\text{exponent}} - \text{základ}^{\text{exponent} - 1}}{\text{základ}^{\text{počet číslic v mantise}}} \end{aligned}$$

- Př. Vrat'me se k motivačnímu příkladu E: 5 bitů (v přímém kódu), M: 8 bitů (0,M a nenormalizovaná)
- Pro $E = 00101$ a $M = 10001000$ platí $0,10001 \times 2^5 = 17$.
 - $\text{Err} = (2^5 - 2^4) / 2^8 = (32 - 16) / 256 = 0,0625$
- Pro $E = 00110$ a $M = 01000100$ platí $0,010001 \times 2^6 = 17$.
 - $\text{Err} = (2^6 - 2^5) / 2^8 = (64 - 32) / 256 = 0,125$
- Zvýšení exponentu o 1 vede na dvojnásobnou chybu Err.

0	0	0	1	0	1	1	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---

Přesnost čísel FP

- Čísla s pohyblivou řádovou čárkou mají různou velikost intervalů pro každou konkrétní hodnotu exponentu (obrázek), přičemž pro každý interval je odlišná množina bodů s konstantním rozestupem (bodů je stejný počet). Rozestup mezi reprezentovatelnými body a velikost největší chyby pro jistý exponent je základ-krát větší než chyba v předchozím intervalu.
- Vidíme, že není pravda, že čísla FP jsou "přesnější", než čísla s pevnou řádovou čárkou. Je-li počet bitů pro číslo FX a počet bitů pro číslo FP stejný, může být reprezentace některých čísel FX přesnější, než jejich vyjádření ve FP, protože u FP je pro mantisu určena jen část z celkového počtu bitů a druhá část je určena pro exponent.



Problémy při sčítání čísel FP - asociativita

FP sčítání **není asociativní**, tedy platí $x + (y + z)$ není vždy rovno $(x + y) + z$

Příklad:

Máme tři čísla $x = -1,5 \cdot 10^{38}$, $y = 1,5 \cdot 10^{38}$, $z = 1,0$.

$$\begin{aligned} \text{Pak } x + (y + z) &= -1,5 \cdot 10^{38} + (1,5 \cdot 10^{38} + 1,0) = \\ &= -1,5 \cdot 10^{38} + 1,5 \cdot 10^{38} = 0 \end{aligned}$$

$$\begin{aligned} \text{Naopak } (x + y) + z &= (-1,5 \cdot 10^{38} + 1,5 \cdot 10^{38}) + 1,0 = \\ &= 0,0 + 1,0 = 1,0 \end{aligned}$$

Důvodem tohoto chování je omezená přesnost vyjádření na daném počtu bitů a approximace skutečných hodnot přibližnými.

Problémy při sčítání čísel FP – zaokrouhlování

Uvažme sčítání s 5ti významovými číslicemi v HW
(pro jednoduchost v desítkové soustavě)

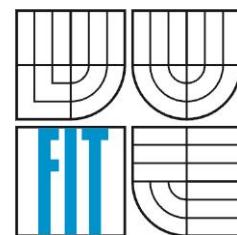
$$\begin{array}{r} 4,5674 \\ +0,0002500\textcolor{red}{1} \\ \hline 4,5676500\textcolor{red}{1} \end{array} \quad \begin{array}{l} .10^0 \\ \\ \text{zaokrouhlíme na } 4,5677 \end{array}$$

Nestačí při výpočtu přidat další jednu nebo dvě významové pozice, musí se zapamatovat i **jednička** na konci čísla, která rozhodne o směru zaokrouhlení. Ve dvojkové soustavě se zapamatuje takovýto nenulový bit při posuvu vpravo v záhytném klopném obvodu *s* (**sticky bit**).

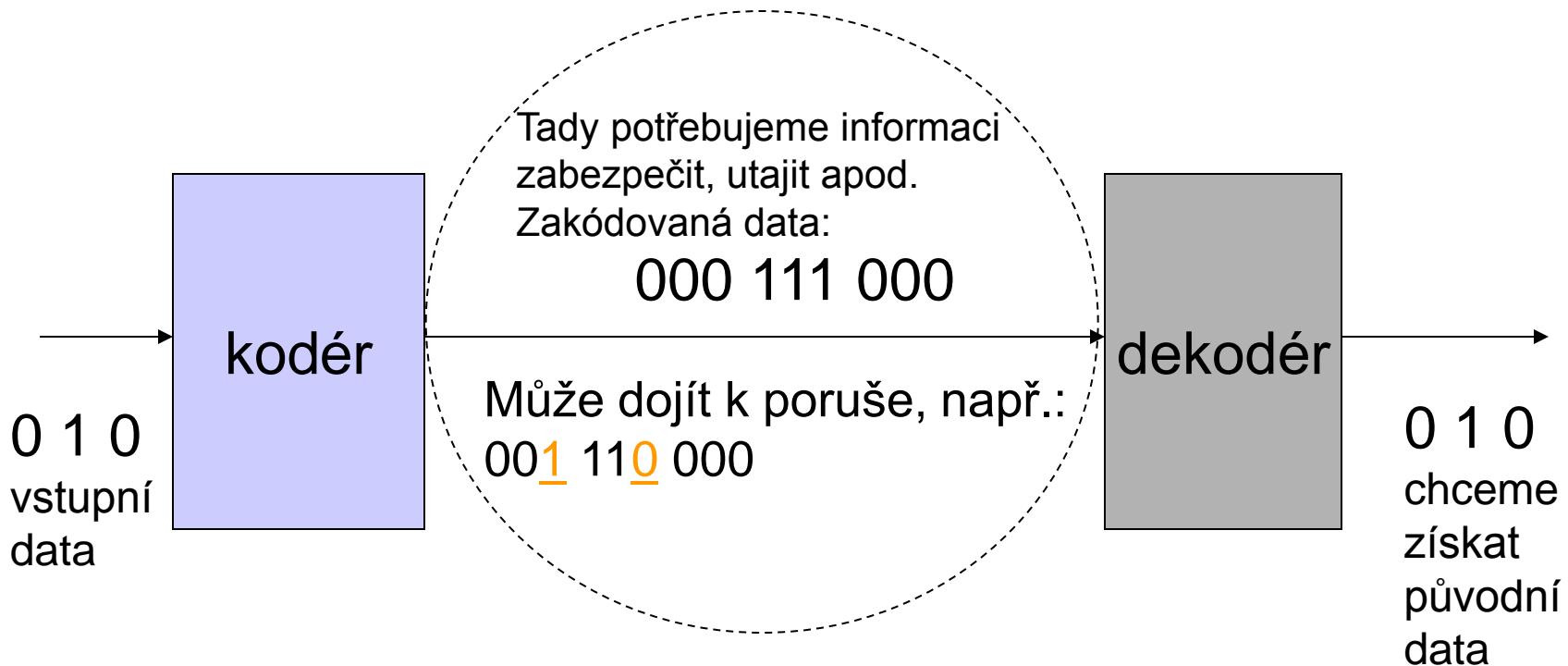
* Konec kapitoly *

Kódy pro detekci a opravu chyb

INP 2015
FIT VUT v Brně



Princip kódování



- Předpokládejme kódovací předpis, např.:
 $0 \rightarrow 000, 1 \rightarrow 111$
- Uvedené kódování může být použito pro přenos dat bud' mezi jednotkami nebo celými systémy (například počítači).
- Zabezpečení informace je založeno na vhodném využití **redundance**.

Základní kódy pro detekci a opravu chyb

- Parita
- Ztrojení
- Hammingův kód (7,4)
- Rozšířený Hammingův kód

Paritní kód

Nejjednodušší kód **detekující** jednu chybu (SED – Single Error Detection) dostaneme doplněním **paritního bitu**, např. na *sudou* paritu.

0110 1010 0

1000 0000 1

1111 1111 0

...

Popsané uspořádání se nazývá **paritní kód**. Kombinace se zvolenou sudou (tedy správnou) paritou se označují jako **kódové**, kombinace s chybnou (lichou) paritou jako **nekódové**. Kontrola správnosti dat se zjišťuje *kontrolou parity*.

Hammingova vzdálenost

Hammingova vzdálenost (kódových složek) je definovaná jako nejmenší počet bitů, v nichž se dvojice kódových kombinací liší, zjištěný pro všechny dvojice.

Příklad sudého paritního kódu:

x_2	x_1	x_0	p
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Minimální vzdálenost, zjištěná u každé dvojice kódových slov, je **Hammingova vzdálenost kódu** d_H .

U paritního kódu je to $d_H = 2$.

Ztrojení (kód typu SEC - Single-Error Correction)

Ztrojením jednoho bitu dostaneme dvě kódové kombinace, a to 000, 111, a 6 nekódových kombinací.

kódové kombinace

0 0 0

1 1 1

nekódové kombinace

0 0 1

0 1 0

0 1 1

1 0 0

1 0 1

1 1 0

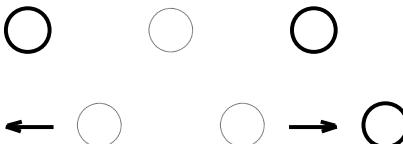
Za předpokladu jediné chyby (jednobitové) je možno určit, ze které kódové kombinace daná nekódová kombinace vznikla. Dostali jsme tak kód opravující jednoduché chyby (Single-Error Correction - SEC). Jeho Hammingova vzdálenost je 3.

Hammingova vzdálenost a schopnost kódu detekovat opravovat chyby

Pro opravu x -násobné chyby musí být Hammingova vzdálenost

$$d_H \geq 2x + 1.$$

Obrázek: Kroužky nakreslené plnou čarou představují kódové kombinace, tečkované kroužky znamenají nekódové kombinace. Mezi sousedními kroužky v jednom řádku je Hammingova vzdálenost rovna jedné. Je zřejmé, že pro kód s Hammingovou vzdáleností $d_H = 2$ nemůžeme rozhodnout, ze které kódové kombinace vzniklo vlivem jednobitové chyby nekódové slovo, a nedokážeme tedy chybu opravit. Opravu jednobitové chyby můžeme provést až u kódu se vzdáleností $d_H = 3$.

$d =$		
	1	
	SED	2
	SEC	3
	SEC - DED	4
	DEC	5

Hammingův kód (n, k)

- n – délka kódového slova (v bitech)
- k – počet informačních bitů
- m – počet kontrolních bitů
- $n = 2^m - 1$
- $n = m + k$
- Př. HK(7, 4), HK(15, 11), ...
- Nejznámější SEC je HK(7, 4)

Hammingův kód (7,4) – kódování

i =

2^6	2^5	2^4	2^3	2^2	2^1	2^0
7	6	5	4	3	2	1
I_7	I_6	I_5	C_4	I_3	C_2	C_1
X		X		X		X
X	X			X	X	
X	X	X	X			

generující matice

$$C_1 = I_3 \otimes I_5 \otimes I_7$$
$$C_2 = I_3 \otimes I_6 \otimes I_7$$
$$C_4 = I_5 \otimes I_6 \otimes I_7$$

generující rovnice
(\otimes znamená XOR)

Podle hodnoty zavedeného indexu i se rozhodne o funkci příslušného bitu: je-li i mocnina dvojky, je bit **kontrolní** (C), v ostatních případech je bit **informační** (I). Rozmístění symbolů **X** v generující matici je popsáno generujícími rovnicemi. Je tak definován způsob doplňování hodnot kontrolních bitů, tedy jistým způsobem vypočítávaných paritních bitů.

Tvar kódové značky: $I_7 I_6 I_5 C_4 I_3 C_2 C_1$

Hammingův kód (7,4) - dekódování

Přičteme-li operací XOR k oběma stranám generujících rovnic pořadě C_1 , C_2 , C_4 , dostaneme tzv. **kontrolní rovnice**

$$C_1 \oplus I_3 \oplus I_5 \oplus I_7 = 0 = S_1$$

$$C_2 \oplus I_3 \oplus I_6 \oplus I_7 = 0 = S_2$$

$$C_4 \oplus I_5 \oplus I_6 \oplus I_7 = 0 = S_4$$

Výpočtem kontrolních rovnic pro kódová (správná) slova dostaneme nuly. Pro nekódová slova, která vzniknou jednobitovou chybou z kódových slov, vyjdou výpočtem kontrolních rovnic nenulové hodnoty S_4 , S_2 , S_1 , zvané syndrom chyby. Syndrom jednoduché chyby udává binárně hodnotu indexu i bitu s chybou. Chybu pak můžeme opravit změnou hodnoty takto zjištěného bitu na hodnotu opačnou.

Pro dvojnásobnou chybu však mechanizmus selhává a syndrom chyby udává nesprávnou polohu chyby. Je to způsobeno tím, že takto definovaný kód je SEC, nikoli však DED. Proto je nutno doplnit definici kódu tak, aby kód získal vlastnost DED, získáme rozšířený Hammingův kód.

Rozšířený Hammingův kód (SEC, DED)

I ₇	I ₆	I ₅	C ₄	I ₃	C ₂	C ₁	C ₀
X		X		X		X	
X	X			X	X		
X	X	X	X				
X	X	X	X	X	X	X	X

Do kódu se doplní **kontrolní bit C₀** (normální paritní bit),
popsaný generující rovnicí

$$C_0 = C_1 \oplus C_2 \oplus I_3 \oplus C_4 \oplus I_5 \oplus I_6 \oplus I_7$$

a kontrolní rovnicí

$$S_0 = C_0 \oplus C_1 \oplus C_2 \oplus I_3 \oplus C_4 \oplus I_5 \oplus I_6 \oplus I_7$$

Rozšířený Hammingův kód

$$d_H = 4$$

čtyři informační bity, čtyři kontrolní bity.

Linearita kódu:

součet (pomocí xor) dvou kódových slov
vytvoří opět platné kódové slovo.

I_7	I_6	I_5	C_4	I_3	C_2	C_1	C_0
x		x		x		x	
x	x			x	x		
x	x	x	x				
x	x	x	x	x	x	x	x
0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	0	1	1	1	1	0	0
0	1	0	1	0	1	0	1
0	1	0	1	1	0	1	0
0	1	1	0	0	1	1	0
0	1	1	0	1	0	0	1
1	0	0	1	0	1	1	0
1	0	0	1	1	0	0	1
1	0	1	0	0	1	0	1
1	0	1	0	1	0	1	0
1	1	0	0	0	0	1	1
1	1	0	0	1	1	0	0
1	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1

Rozšířený Hammingův kód - syndrom

Definujeme **syndrom chyby**

$$S = S_1 \vee S_2 \vee S_4 \quad (\text{tj. OR})$$

Pomocí hodnot S, S_0 dostaneme klasifikaci chyb:

S	S_0	význam
0	0	bez chyby
0	1	neopravitelná chyba, např. porucha hlídace kódu
1	0	neopravitelná 2-, 4- atd. násobná chyba
1	1	opravitelná chyba

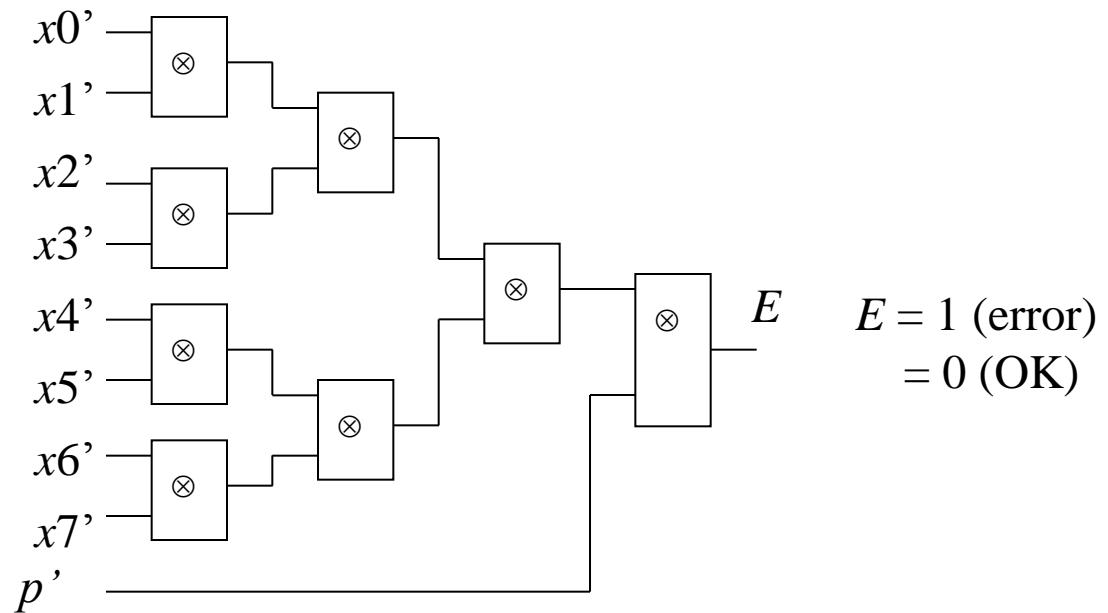
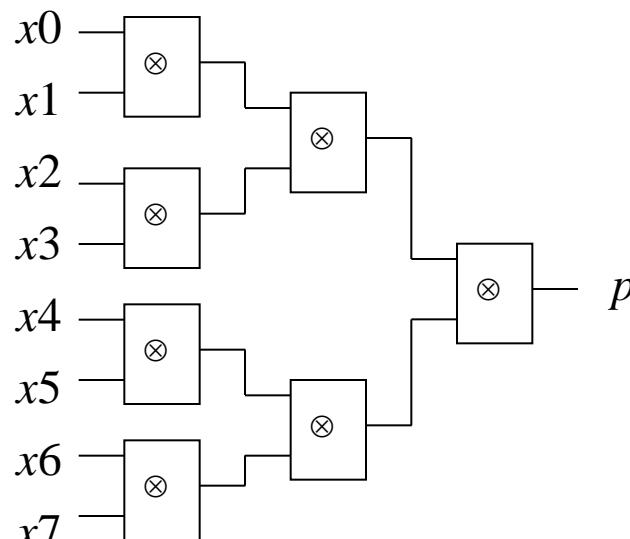
Základní typ *jednoduché chyby* se projeví nenulovým syndromem a chybou parity. V takovém případě se provede oprava.

Stejně se však projeví i *trojnásobná chyba* a další chyby s *lichou násobností*.

Dvojitá chyba (a další *chyby se sudou násobnosti*) se projeví nenulovým syndromem a správnou paritou. Oprava není možná.

Zvláštním případem je hlášení s nulovým syndromem a chybnou paritou. Jde buďto o případ *vícenásobné chyby*, nebo o *poruchu hlídace parity*. V obou případech *se oprava chyby nedá provést*.

Generování a kontrola parity (8 bitů)

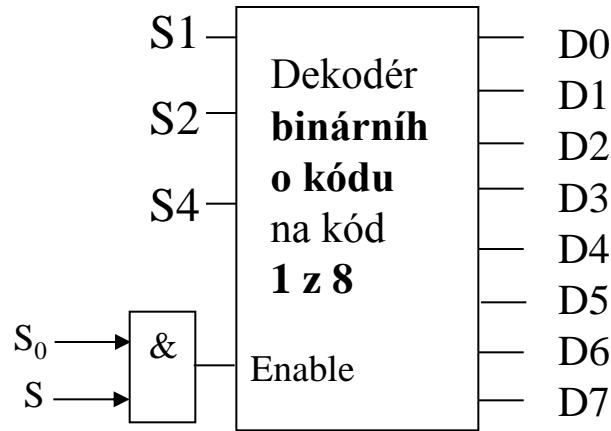


V kodéru: $p = x_0 \otimes x_1 \otimes x_2 \otimes x_3 \otimes x_4 \otimes x_5 \otimes x_6 \otimes x_7$

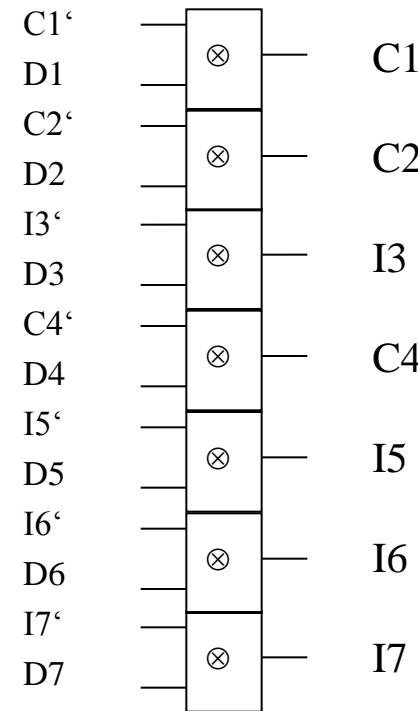
V dekodéru: $E = x'_0 \otimes x'_1 \otimes x'_2 \otimes x'_3 \otimes x'_4 \otimes x'_5 \otimes x'_6 \otimes x'_7 \otimes p'$

Pozn.: Apostrof označuje přijatý symbol, který v důsledku poruchy nemusí být stejný jako vyslaný symbol.

Oprava chyb pomocí syndromu Hammingova kódu



Dekodér syndromu



Korektor

Redundance kódu a CNC

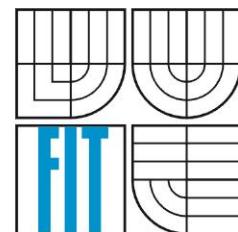
- *Redundance kódu* R je procentuální vyjádření počtu *přidaných (kontrolních)* bitů C k původnímu počtu *informačních (datových)* bitů /

$$R = C / I$$

- Redundance 8-bitového kódu s přidaným paritním bitem je $R_{parity8} = 1/8 = 0,125 = 12,5\%$
- Redundance ztrojeného kódu je 200%.
- Dále se můžeme setkat s parametrem, označeným zkratkou *CNC* – *Code to Noncode ratio*. Je to poměr počtu kódových a nekódových slov, tedy kódových a nekódových binárních kombinací z celkového množství binárních kombinací dané délky.
- Pro paritní kód je poměr CNC 1:1, tedy 1, pro ztrojený kód je poměr CNC 2:6, tedy 0,33.
- **Oázka:** Jaká je hodnota redundance a CNC pro jednoduchý a rozšířený Hammingův kód s délkou n informačních bitů?

Základy počítačových architektur

INP 2015
FIT VUT v Brně



Obsah

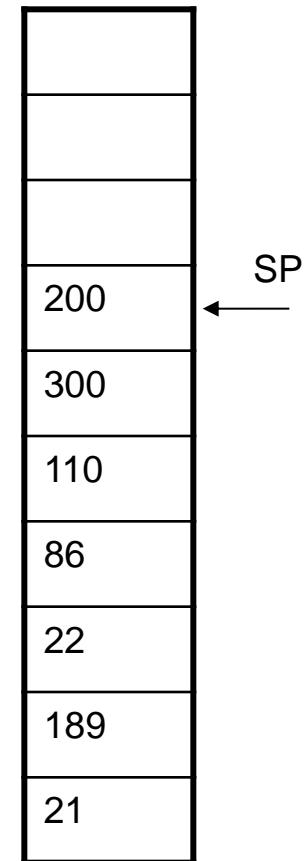
- Typy architektur instrukčních souborů
- Datové objekty
- Adresové módy
- Kódování instrukcí
- CISC vs. RISC

Typy architektur instrukčních souborů (ISA) (resp. procesorů)

- Zásobníková (stack)
- Střadačová (accumulator)
- Registrová (GPR – General Purpose Register)
- Smíšená
 - kombinace předchozích
 - dlouhodobě nejosvědčenější koncepce

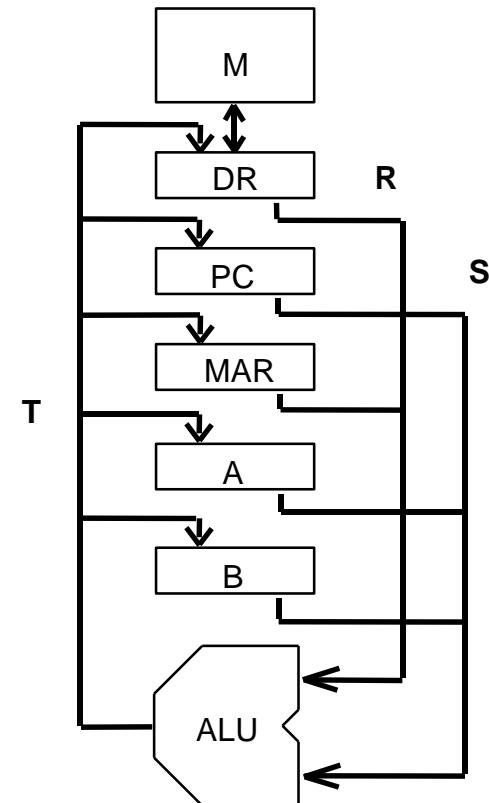
Zásobníkové počítače

- Dnes asi jen teoretická varianta.
- Programátor nemá k dispozici střadač ani registry – veškeré operace se provádějí se zásobníkem.
- Zásobník je implementován buď v paměti nebo pomocí registrů nebo kombinací registrů a paměti.
- Program a data mají v paměti oddělené segmenty, jejichž meze určují specializované registry.
- **Stack pointer** (SP) – ukazatel na vrchol zásobníku (nejdůležitější registr)
- Typické instrukce
 - **PUSH m_a**: přečti slovo z paměti s adresou m_a a ulož ho na vrchol zásobníku
 - **POP m_a**: vyber slovo z vrcholu zásobníku a ulož ho do paměti na adresu m_a
- Aritmetické instrukce (ADD, MUL apod.) nemají operandy.
 - ADD: vybere z vrcholu zásobníku dvě slova, sečte je a výsledek uloží na vrchol zásobníku
- Výhody: jednoduchá architektura
- Nevýhody: zásobník je úzké místo (lze pracovat jen s jeho vrcholem)
- Př. HP 3000

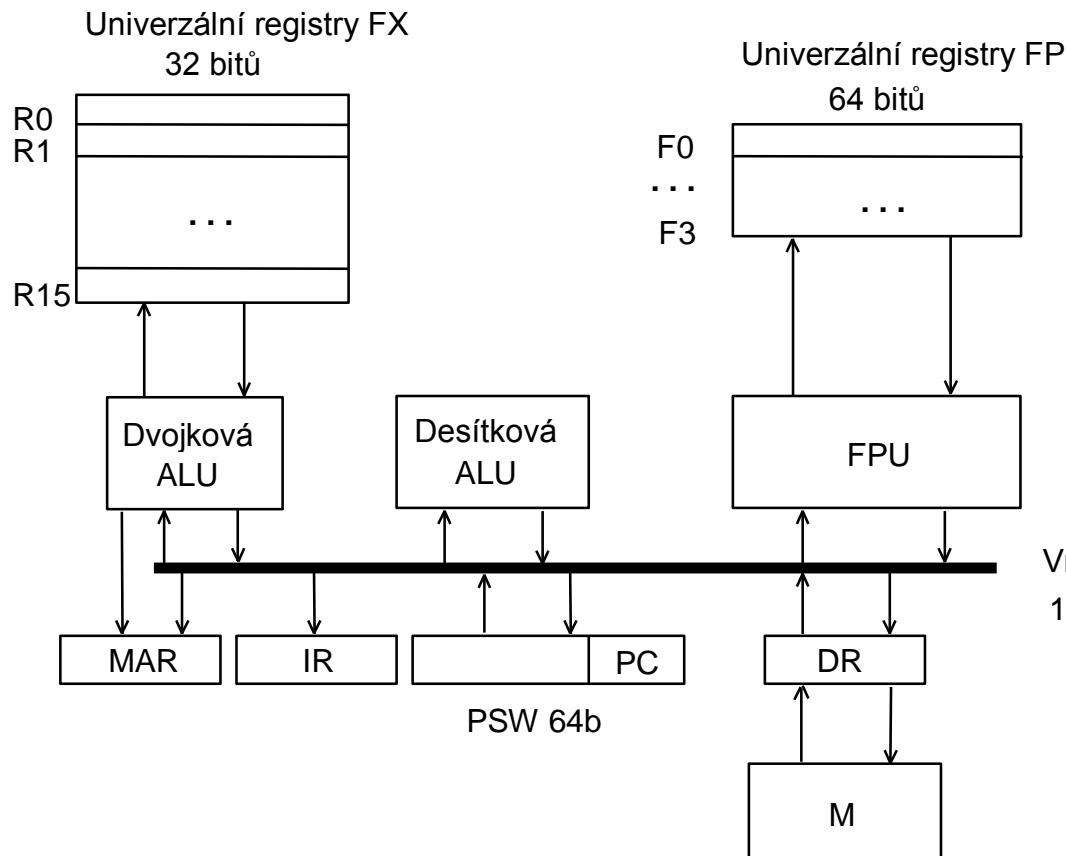


Střadačové počítače

- Existuje jeden, popř. dva, významné registry – **střadače** – které využívají prakticky všechny instrukce.
- Typická aritmetická instrukce má **jeden operand implicitní** (střadač) a výsledek je uložen opět do střadače.
 - Př. ADA a_m: k obsahu střadače A je přičten obsah paměťové buňky na adresu a_m, výsledek je uložen do střadače
- Data čtená z, popř. zapisovaná do, paměti směřují vždy do/z střadače.
 - Př. LDB a_m: Do střadače B je vložen obsah paměťové buňky na adresu a_m
- Nevýhody: střadač je úzké místo; často se komunikuje s pamětí
- DR - datový registr, přes který přecházejí instrukce a data čtená nebo zapisovaná do paměti.
- MAR - registr adresy paměti, obdoba IAR z první přednášky.



Registrové počítače



Př. IBM 360/370

- Programátor má k dispozici sadu registrů.
- Př. IBM 360/370 (1965)
 - 3 speciální ALU
 - FX
 - FP
 - Operace s proměnnou délkou, zahrnující desítkovou aritmetiku a operace s řetězci znaků.
 - dvě sady nezávisle adresovatelných registrů
 - 16 FX registrů
 - 4 FP registry
 - 64b registr pro Program Status Word (PSW) – popisuje stav procesoru (příznaky, PC, maska přerušení...)

Srovnání v úloze $M[C] = M[A] + M[B]$ (A, B i C jsou adresy)

- Zásobníkový stroj
- Střadačový stroj
- Registrové stroje:
 - registr – paměť (R-M): Memory Reference (v libovolné instrukci)
 - registr – registr (Load-Store): přístup k paměti pouze v instrukcích Load a Store
 - memory – memory (historie – LGP, kalkulačka M3T)

Zásobníkový	Střadačový	Registrový (R-M)	Registrový (L-S)
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R1, B	Load R2, B
Add	Store C	Store C, R1	Add R3, R1, R2
Pop C			Store C, R3

Výhody a nevýhody základních architektur

- Při hodnocení se používá těchto tří kritérií:
 - jak vyhovuje struktura potřebám kompilátoru,
 - jak vhodná je struktura z hlediska implementace,
 - jak dlouhý vyjde program ve srovnání s ostatními koncepcemi.

Architektura	Výhody	Nevýhody
Zásobníková	Jednoduché vyčíslování výrazů (polštá notace), díky krátkým instrukcím je výsledný strojový kód hustý.	Přístup k zásobníku není libovolný, proto je obtížné vytvořit efektivní kód. Zásobník je úzké místo architektury.
Střadačová	Minimalizuje se počet vnitřních stavů počítače, instrukce jsou krátké.	Střadač je pouze dočasná paměť, zatížení paměti M je vysoké.
Registrová	Nejobecnější model pro generování kódu (tzv. tříadresový kód).	Všechny operandy musejí být pojmenovány, což vede na dlouhé instrukce.

Architektury s univerzálními registry

- Vyčíslování výrazů je pružnější než při použití střadačů nebo zásobníku.
 - Např. vyčíslování výrazu $(A \times B) - (C \times D) - (E \times F)$ může částečně proběhnout v libovolném pořadí, kdežto u zásobníkové koncepce musí proběhnout zleva doprava.
- Ještě důležitější je použití registrů pro ukládání proměnných!
 - Jsou-li proměnné umístěny v registrech, provoz paměti se sníží a program se provede rychleji.
 - Viz sečtení pole čísel na procesoru z 1. přednášky.
- Otázkou zůstává, kolik (nespecializovaných) registrů by mělo v procesoru být.

Datové objekty

- Podle délky rozlišujeme:

– byte	B	1B
– půlslovo	HW (Halfword)	2B
– slovo	W (Word)	4B
– dvojslovo	DW (Doubleword)	8B
– čtyřslovo	QW (Quadword)	16B

Uspořádání bytů ve slově

- Adresa slova

LittleEndian

0	3	2	1	0
4	7	6	5	4

- Adresa slova

BigEndian

0	0	1	2	3
4	4	5	6	7

- Příklady:

- Little Endian: DEC PDP/11, VAX, Intel 80x86, Atmel AVR, 8051

- Big Endian: IBM 360/370, Motorola 68000

- Bi-Endian (Mixed Endian) – podpora obou způsobů

- ARM versions 3+, PowerPC, Alpha, SPARC V9, IA-64

Př. IA-64 Instrukce Load Little-endians (Intel Itanium, 64b)

Paměť

Adresa 7 0

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h

Registry

63

0

LD1 [1]=>

0	0	0	0	0	0	0	b
---	---	---	---	---	---	---	---

LD2 [2]=>

0	0	0	0	0	0	d	c
---	---	---	---	---	---	---	---

LD4 [4]=>

0	0	0	0	h	g	f	e
---	---	---	---	---	---	---	---

LD8 [0]=>

h	g	f	e	d	c	b	a
---	---	---	---	---	---	---	---

Př. IA-64 Instrukce Load Big-endians

Paměť

Adresa 7 0

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h

Registry

63

0

LD1 [1]=>

0	0	0	0	0	0	0	b
---	---	---	---	---	---	---	---

LD2 [2]=>

0	0	0	0	0	0	c	d
---	---	---	---	---	---	---	---

LD4 [4]=>

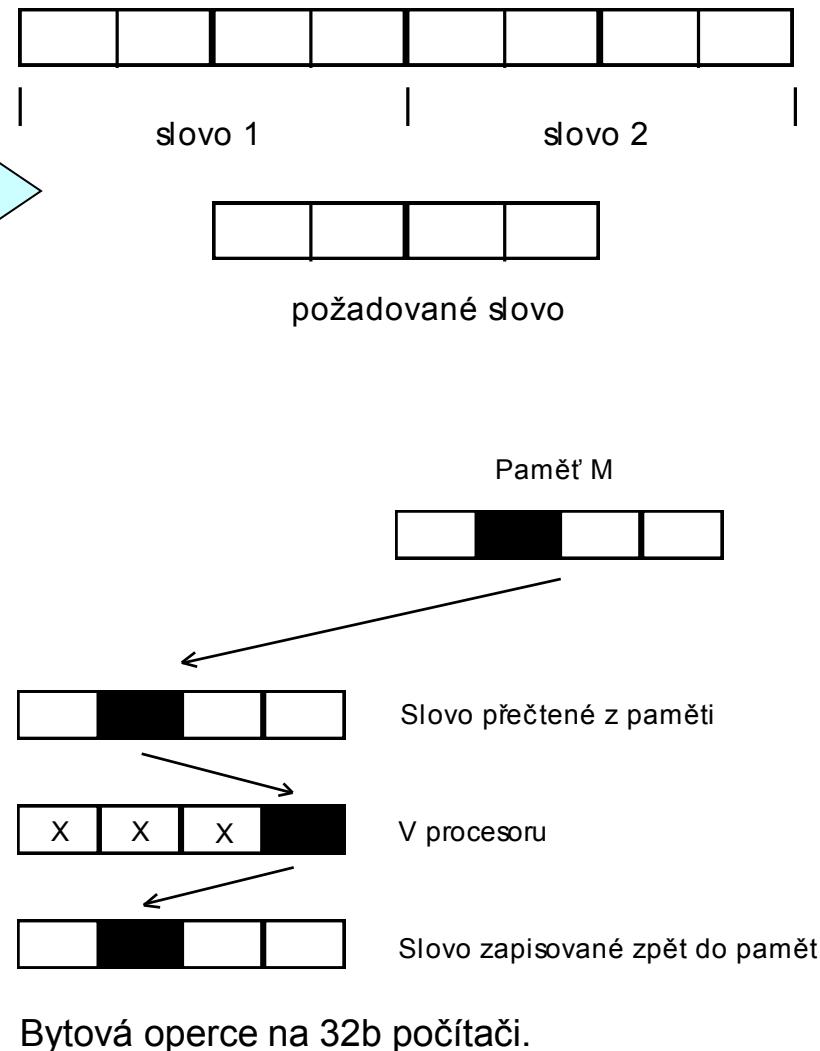
0	0	0	0	e	f	g	h
---	---	---	---	---	---	---	---

LD8 [0]=>

a	b	c	d	e	f	g	h
---	---	---	---	---	---	---	---

Nezarovnaný přístup k paměti

- U některých počítačů musí být přístup k objektům větším než jeden byte zarovnán.
- Je zřejmé, že na získání nezarovnaného slova musí paměť provést **dva cykly** a z každého slova se bere jen jedna polovina.
- I když jsou data zarovnaná, podpora bytových a půlslovních přístupů vyžaduje **zarovnávací obvod**, který požadovaný byte nebo půlslovo zarovnává vregistrech.
- Taková potřeba se objevila u procesorů, které z důvodů kompatibility zachovávají instrukční soubor a jeho operace, např. u 32-bitového procesoru se zachovávají bytové operace. Při práci s jedním bytem se nesmí ostatní tři byty slova změnit .



Adresové módy (1)

- U registrových strojů může adresový mód určovat konstantu, registr, nebo paměťové místo. Používá-li se paměťové místo, nazývá se skutečná adresa paměti specifikovaná adresovým módem efektivní adresa. V tabulce je přehled adresových módů, které se vyskytují u hlavních typů počítačů.

Adresový mód	Příklad instrukce	Význam	Kdy se použije
Registr	Add R4,R3	R4:=R4 + R3	Když je hodnota v registru.
Bezprostřední, literál	Add R4,#3	R4:=R4 + 3	Pro konstanty, u některých procesorů jde o dva různé adresové módy.
Bázový, s posunem	Add R4,100(R1)	R4:=R4+M[100+R1]	Adresování lokálních proměnných.
Přes registr, nepřímý	Add R4,(R1)	R4:=R4+M[R1]	Přístup s ukazatelem nebo s výpočtem adresy.

Adresové módy (2)

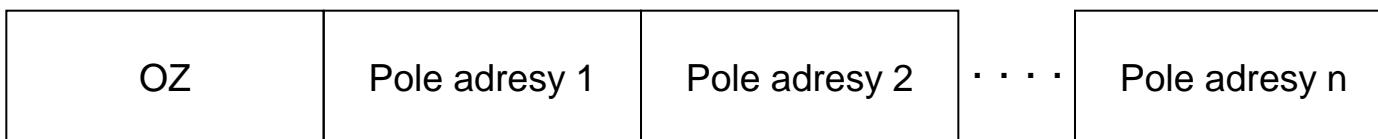
Indexovaný	Add R3,(R1+R2)	$R3 := R3 + M[R1+R2]$	Při adresování v poli: R1 je báze pole, R2 je index.
Přímý, absolutní	ADD R1,(1001)	$R1 := R1 + M[1001]$	Pro přístup k pevně umístěným datům; konstanta může být značně velká.
Nepřímý přes paměť	Add R1,@(R3)	$R1 := R1 + M[M[R3]]$	V R3 je adresa ukazatele.
Autoinkrement	Add R1,(R2)+	$R1 := R1 + M[R2]$ $R2 := R2 + d$	Užitečný pro průchod polem v cyklu. R2 ukazuje na začátek pole a při každém použití se zvětšuje o d.
Autodekrement	Add R1,-(R2)	$R2 := R2 - d$ $R1 := R1 + M[R2]$	Obdoba autoinkrementu.
Indexovaný s měřítkem	Add R1,100(R2)[R3]	$R1 :=$ $R1 + M[100 + R2 + R3 * d]$	K indexování pole. Velikost kroku je proměnná.

Adresové prostory

- Velikost adresového prostoru je daná vyhrazeným počtem bitů v adrese.
- Rozlišujeme
 - paměťový adresový prostor
 - V/V adresový prostor – určuje možné adresy připojených periferních zařízení
 - další adresové prostory
 - adresový řídicí prostor (přerušovací) – pro stavové informace počítače
 - registrový
 - zásobníkový
- Zopakujte si pojmy z Operačních systémů:
 - logický adresový prostor
 - fyzický adresový prostor
 - virtuální paměť
 - stránkování
 - segmentování
 - překlad adresy
 - stránka a rámec
 - TLB

Kódování instrukcí

- definuje způsob zakódování operačních znaků (OZ; opcode)
- předepisuje velikost jednotlivých polí pro adresy registrů a pro adresy paměti
- Pozor na zarovnaný přístup do paměti!
- Obecný tvar instrukce:



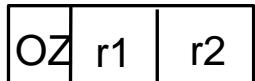
Kódování instrukcí – délka instrukce

- **Pevná délka instrukce**
 - Neúsporné, ale výhodné, pokud je použito řetězené zpracování instrukcí
- **Proměnná délka instrukce (nejčastější)**
 - Obtížnější dekódování, ale dovoluje zkrátit program
 - Typické formáty instrukce s proměnnou délkou
 - OZ např. NOP, HALT
 - OZ + adresa např. LOAD adr (střadač je implicitní)
 - OZ + 2 adresy např. LOAD R1, adr
 - OZ + 3 adresy např. ADD R1, R2, R3
- **Rozšiřující se kód** – kompromisní přístup (př. pro 16-bit. instrukce)

0000	a1	a2	a3	}	15 instrukcí s 3 adresami
...					
1110	a1	a2	a3	}	14 instrukcí s 2 adresami
1111	0000	a1	a2		
...					
1111	1101	a1	a2	}	31 instrukcí s 1 adresou
1111	1110	0000	a1		
...					
1111	1111	1110	a1	}	16 instrukcí bez adresového pole
1111	1111	1111	0000		
...					
1111	1111	1111	1111		

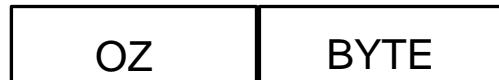
Př. Formáty instrukcí I 8080

1B



r1, r2 jsou tříbitové adresy registru. Např. jednobyтовá instrukce s hexadecimálním kódem 78 znamená MOV A, B, což je přesun obsahu registru B do A

2B



Např. zápis 3E BYTE znamená instrukci MVI, tedy přesun bezprostředně následujícího operandu BYTE do registru A

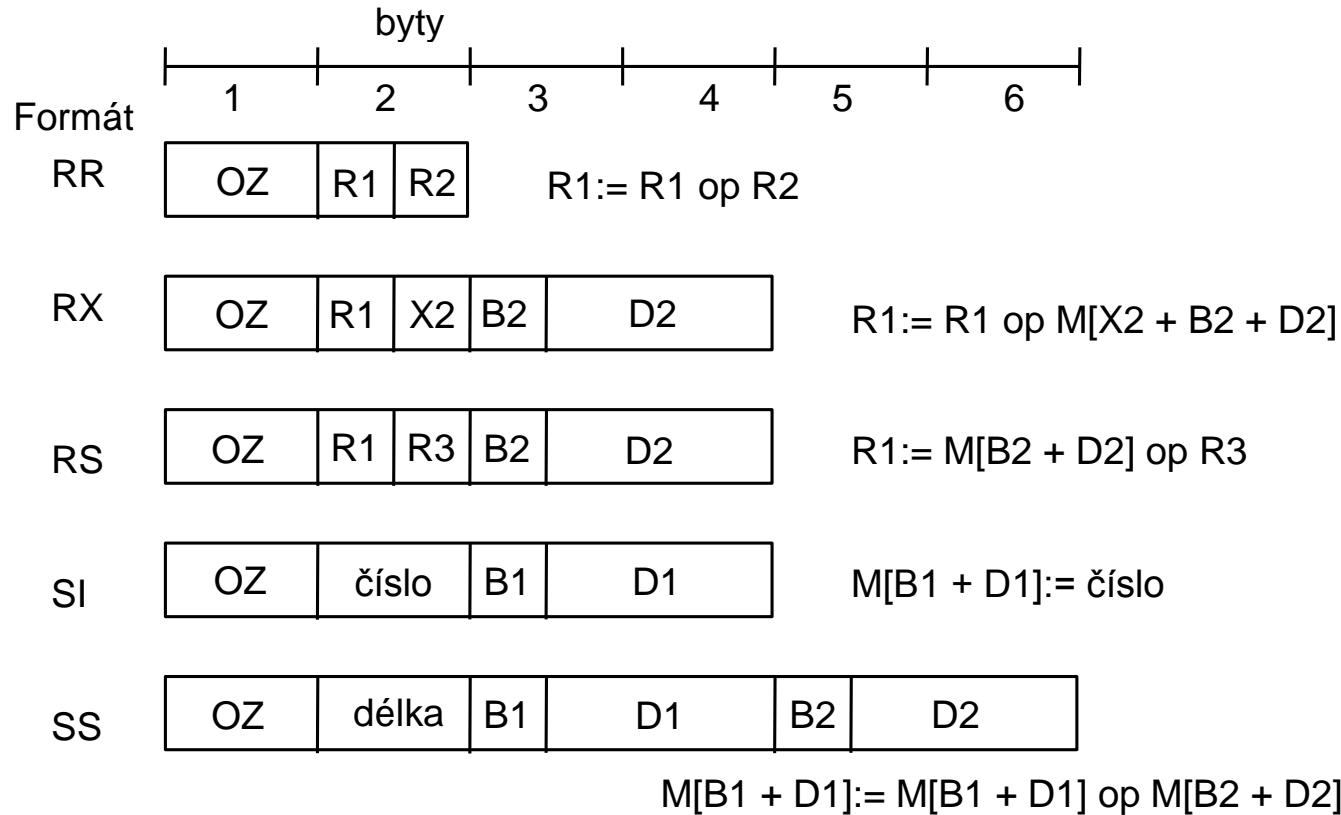
3B



Např. zápis C3 ADRL ADR znamená instrukci JMP ADR ADRL, tedy nepodmíněný skok na adresu ADR ADRL

- Adresa paměti je 16-bitová, což znamená, že velikost adresového paměťového prostoru je $2^{16} = 64$ K, adresované místo má šířku 1 B.
- Vstup-výstupní instrukce mají formát 2B, v části BYTE je číslo V/V jednotky, kterých tedy může být 256. Říkáme, že adresový prostor V/V má 256 míst.

Př. Formáty instrukcí IBM/360



Pozn.: RR: Registr - Registr

RX: Registr - Indexovaná adresa

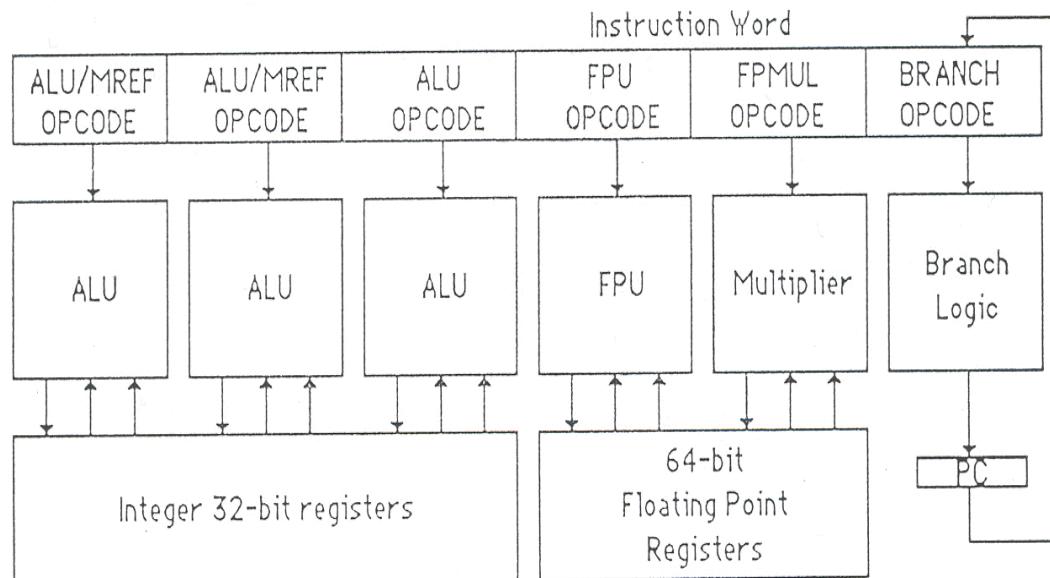
RS: Registr - Paměť (Storage)

SI: Paměť - Bezprostřední (Immediate) operand

SS: Paměť - Paměť

VLIW – Very Long Instruction Word

- architektura procesoru s paralelně pracujícími vícenásobnými jednotkami.
- v jednom instrukčním slově může být zakódováno několik operací s pevnou řádovou čárkou, s pohyblivou řádovou čárkou i instrukce čtení nebo zápisu do paměti
- Př.



Složitost instrukčního souboru

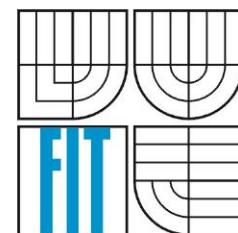
- Podle složitosti instrukcí v instrukčním souboru dělíme procesory na
 - **CISC** - Complex Instruction Set Computers, tedy procesory se složitým instrukčním souborem a
 - **RISC** - Reduced Instruction Set Computers, tedy procesory s jednoduchým instrukčním souborem.
- Nejstarší procesory měly málo instrukcí, které byly značně jednoduché.
- Kolem roku 1960 měly procesory kolem 100 typů instrukcí, které se dále modifikovaly použitým adresovým módem a datovým typem operandů - CISC.
- Architektura CISC se vyvíjela postupně, přidáváním dalších a dalších, stále složitějších instrukcí, které podporovaly vyšší programovací jazyky a principiálně vyplňovaly tzv. **sémantickou mezeru** mezi strojovým kódem počítače a příkazy vyšších programovacích jazyků.
- Ukázalo se však, že tyto složité instrukce jsou **využívány jen zřídka**.
- Vznikla otázka, zda není možné využít plochu na čipu, kterou zabírá jejich obvodová podpora, užitečněji.

RISC – Reduced Instruction Set Computer

- IBM 801 (1973)
- Relativně málo typů instrukcí a adresových módů (složité instrukce nahrazeny podprogramy sestavenými z jednoduchých instrukcí).
- Pevné a snadno dekódovatelné formáty instrukcí.
- Snaha o CPI = 1 (Cycles per Instruction)
- Řadič procesoru je kvůli rychlosti obvodový, tedy není mikroprogramový.
- Přístup k paměti je pouze v jednoduchých instrukcích Load a Store.
- Pro generování cílového kódu se využívá optimalizujících kompilátorů.
- Jde tedy o koncepci procesoru R-R, přičemž registrů je vyšší počet (deset a více).
- Pro jednoduché úlohy, např. v pevné řádové čárce FX, poběží program „RISC“ rychleji. Bude-li však vysoké procento operací FP, poběží rychleji program „CISC“ (toto nastává u vědeckotechnických výpočtů).
- Nejúspěšnější procesory pragmaticky kombinují principy RISC i CISC.
- Podrobnější výklad RISC bude v kapitole o řetězeném zpracování.

Řetězené zpracování

INP 2015
FIT VUT v Brně



Techniky urychlování výpočtu v HW

- Lze realizovat speciální kódování dle potřeby dané úlohy
 - Příklad: kód zbytkových tříd
- Lze zvolit „optimální“ počet bitů pro danou úlohu
 - Příklad: 13-bitová ALU, pokud 13 bitů stačí pro danou úlohu (a ne 16 nebo 32 bitů)
- Lze realizovat speciální výpočetní jednotky dle potřeby dané úlohy
 - Příklad: FFT, která není v běžném procesoru
- Paralelní zpracování (násobné výpočetní jednotky)
- Zřetězené zpracování – jinak též *pipelining*, překládané též jako *proudové zpracování*.
 - tato přednáška

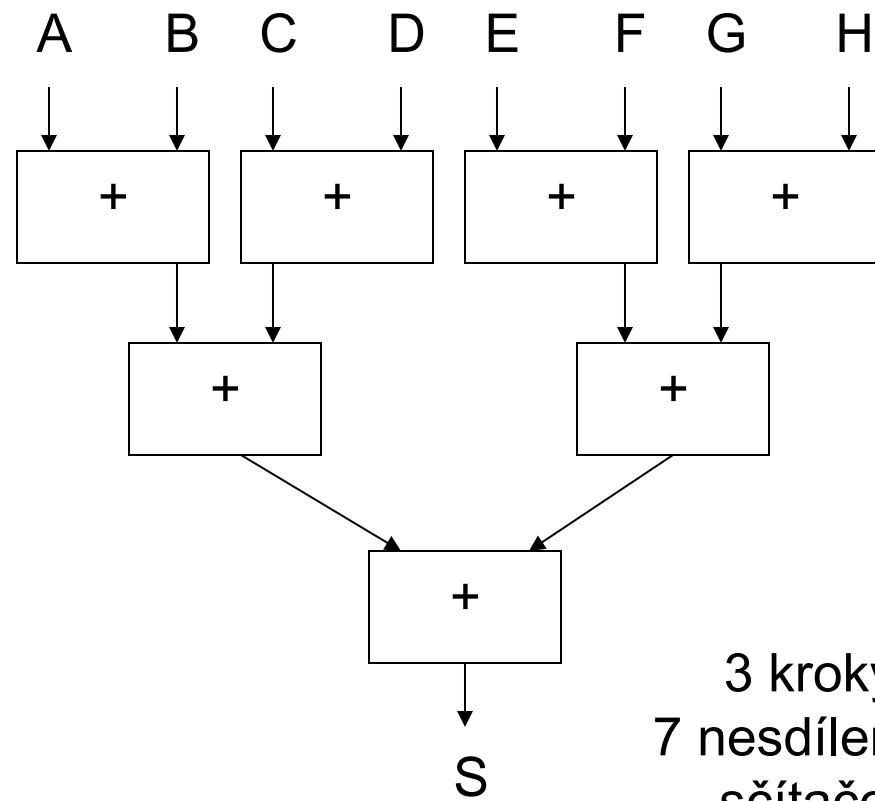
Paralelní zpracování

Př. $S = A + B + C + D + E + F + G + H$

SW: sekvenčně

```
S = A + B;  
S = S + C;  
S = S + D;  
S = S + E;  
S = S + F;  
S = S + G;  
S = S + H;
```

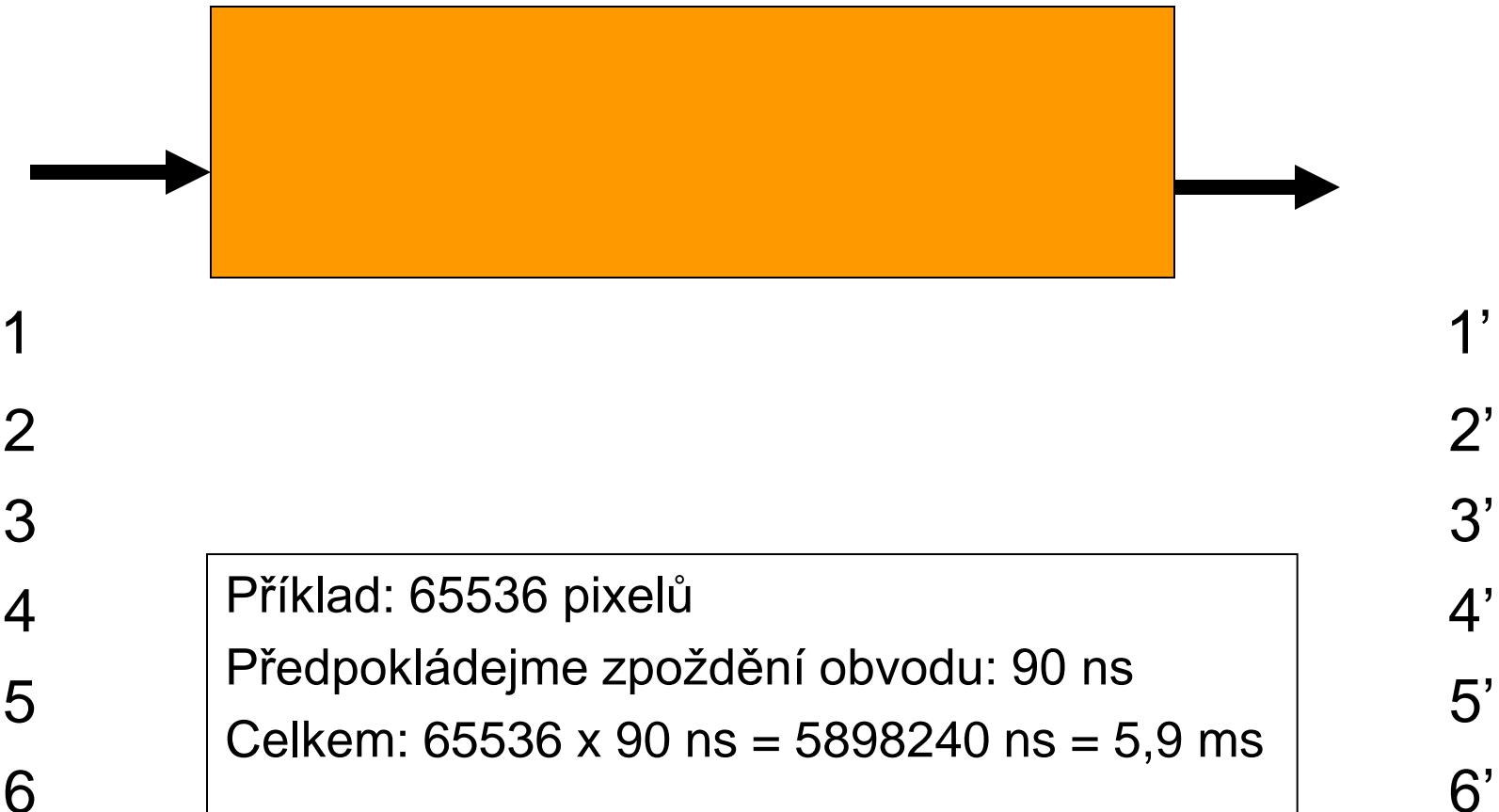
7 kroků,
1 sdílená
sčítačka



3 kroky,
7 nesdílených
sčítaček

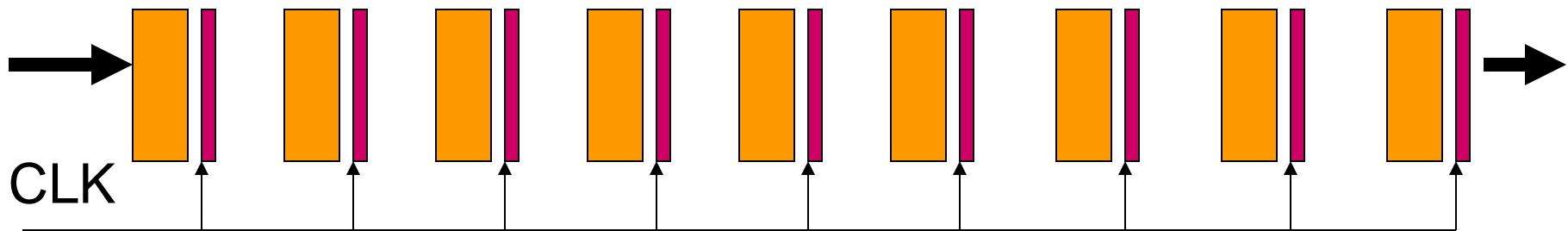
Obvod bez zřetězení

(Př. obvod=filtr, vstup=pixel, výstup=upravený pixel)



Zřetězený filtr:

rozděl původní obvod, přidej registry



1.PIXEL									
2	1								
3	2	1							
4	3	2	1						
5	4	3	2	1					
6	5	4	3	2	1				
7	6	5	4	3	2	1			
8	7	6	5	4	3	2	1		
9	8	7	6	5	4	3	2	1	
10	9	8	7	6	5	4	3	2	
11	10	9	8	7	6	5	4	3	
12	11	10	9	8	7	6	5	4	

9 pixelů je zpracováváno současně \Rightarrow zrychlení 9x oproti předchozímu řešení

Zřetězený filtr – 9 stupňů

- Příklad: 65536 pixelů
- Předpokládejme zpoždění stupně: 10 ns
- 1. pixel bude zpracován za $9 \times 10 = 90$ ns
- 2., 3. až 65536. pixel – každý za 10 ns
- Celkem: $90\text{ ns} + 65535 \times 10\text{ ns} = 655440\text{ ns} = 0.655440\text{ ms}$
- **Zrychlení: 8.9989 krát**
 - není to 9x, protože se musí naplnit zřetězená linka
 - při výpočtu jsme neuvažovali zpoždění registru

Zrychlení použitím řetězení:

vytvoříme **k** stupňů oddělených registry

- Počet vstupů, které zpracováváme: N
- Počet stupňů: k
- Zpoždění stupně: t
- Zpoždění registru: d
- **Zrychlení:**
$$\frac{N \cdot k \cdot t}{(k + N - 1)^* (t + d)}$$

Kdy má smysl zavést řetězené zpracování?

- Pokud je N dostatečně velké!
- Pokud je zpoždění stupňů přibližně stejné.

Jaký je optimální počet stupňů?

Jaká je maximální frekvence, pokud je zpoždění stupňů různé?

Zřetězení + duplikace jednotek \Rightarrow další urychlení

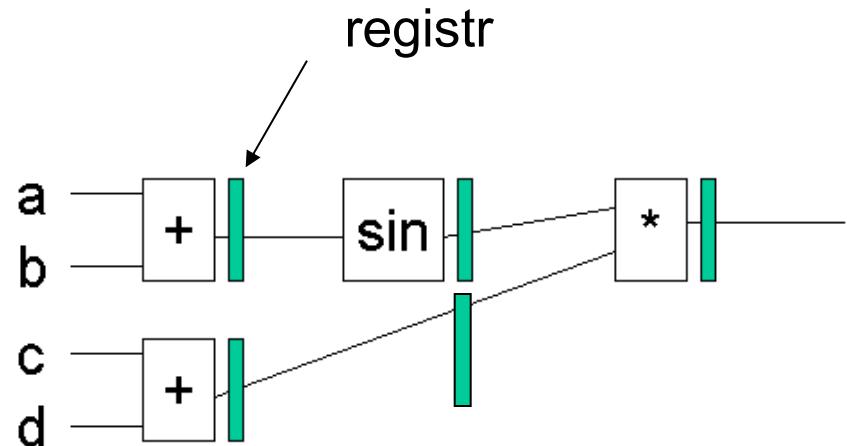
SW:

Vypočtěte

$F[i] = (c[i] + d[i]) * \sin(a[i] + b[i])$
pro $i=1..100$

```
for (i=1; i<=100; i++)  
{  
    pom = a[i]+b[i];  
    pom1 = sin(pom);  
    pom2 = c[i]+d[i];  
    F[i] = pom1 * pom2;  
}
```

Celkem: $100 \times 4 = 400$ kroků
(a to neuvažujeme test podmínky
v cyklu)



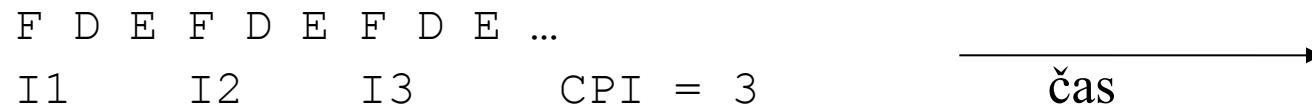
První výsledek za 3 kroky.
2. až 100. výsledek za 1 krok.
Celkem 102 kroků
Zrychlení **3,92 krát.**

Řetězené zpracování instrukcí v procesorech

- Princip zřetězení se značně překrývá s principy procesorů typu RISC.
- Základní myšlenka: V procesorech CISC používali složité strojové instrukce ($CPI >> 1$) pouze špičkoví programátoři, ale standardní rutiny kompilátoru je nepoužívaly.
- Výhodnější by bylo implementovat pouze jednoduché, ale rychlé instrukce
 - Dojde ke zrychlení zpracování instrukcí a úspoře plochy na čipu.
 - Chybějící složité instrukce jsou nahrazeny podprogramy sestavenými z jednoduchých instrukcí.
- Cílem je dosáhnout parametru instrukčního souboru $CPI = 1$.
- To lze zajistit pouze trikem: Překrýváním cyklů F, D a E.
 - Pozn.: U jednoduchého procesoru má instrukce CPI minimálně 3.
- Pozn: **CPI – Cycles Per Instruction** (počet cyklů nutných pro vykonání instrukce). Uvádí se u každé instrukce a také průměrná hodnota pro celý instrukční soubor.

Cykly procesoru

- von Neumannův procesor 1. generace



- Řetězený procesor, základní, teoretický typ, ideální případ bez prostojů

F D E	Instr. 1
F D E	Instr. 2
F D E	Instr. 3
F D E	Instr. 4
F D E	Instr. 5
...	

CPI = 1

- Řetězený procesor – reálná verze, ideální případ bez prostojů

F D E M W	Instr. 1
F D E M W	Instr. 2
F D E M W	Instr. 3
F D E M W	Instr. 4
F D E M W	Instr. 5
...	

CPI ≥ 1 ?

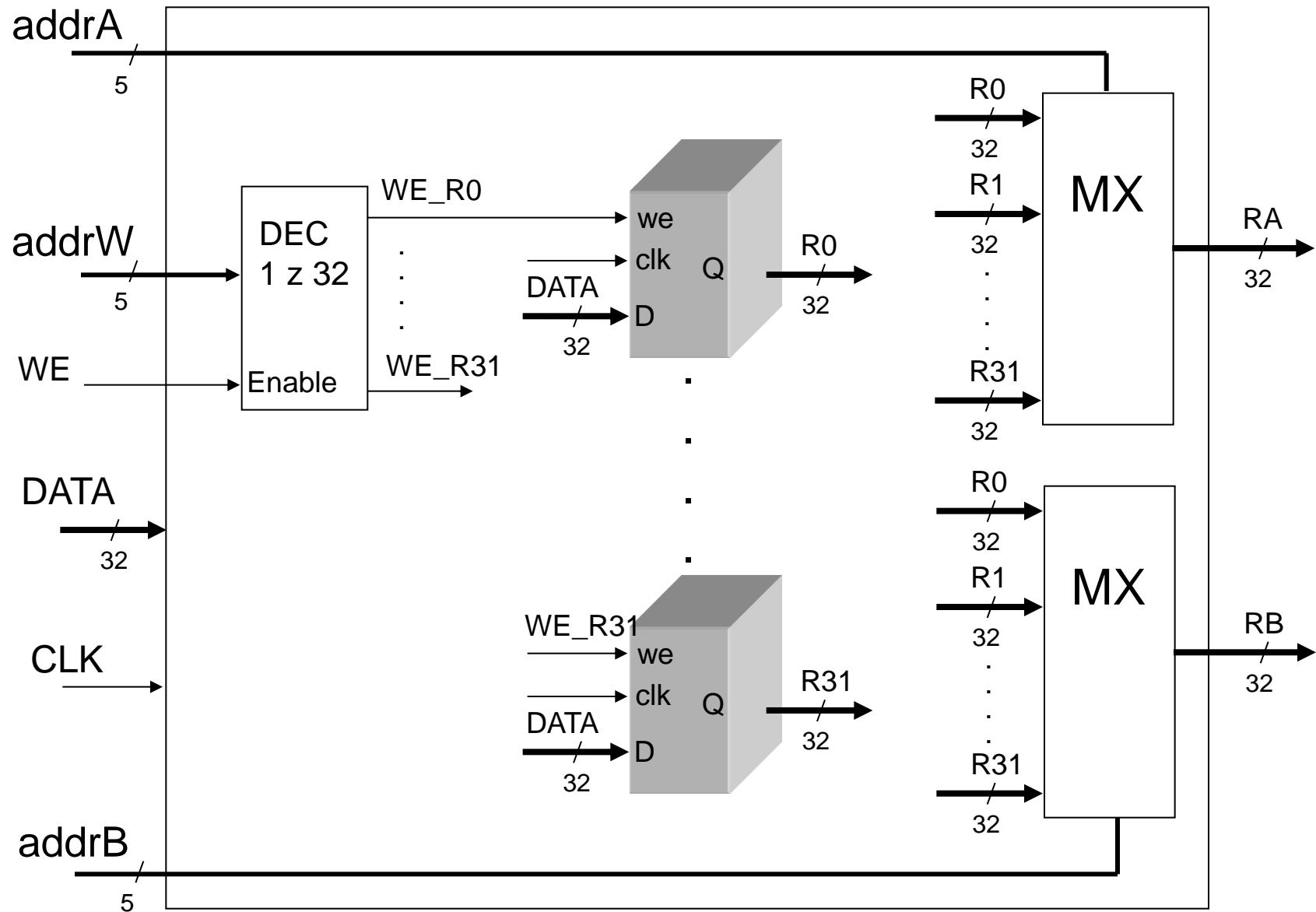
M – memory access; W – write back

Obecné rysy architektur RISC

- Snaha o CPI = 1 (v ideálním případě)
- Všechny instrukce mají stejnou složitost
- Přístup k paměti mají pouze dvě instrukce, LOAD a STORE, ostatní pracují s registry.
- Registrová architektura (sada registrů)
- Rychlý obvodový řadič
- Oddělená paměť (cache) instrukcí a dat

Opakování: Sada registrů (Register File)

Př. 32 x 32b registr



Popis cyklů reálného řetězeného procesoru

F – instruction fetch: $IR \leftarrow M[PC]$, $NPC \leftarrow PC + 4$ (načtení instrukce, zvýšení PC)

D – instruction decode + register fetch:

$A \leftarrow R_i$, $B \leftarrow R_j$ (výběr operandů ze sady registrů)

$Imm \leftarrow IRadr$ (extrakce adresy z IR)

E – provedení + cyklus výpočtu efektivní adresy

MRef. : $ALUoutput \leftarrow A + Imm$

I R-R : $ALUoutput \leftarrow A \text{ op } B$

...

I Branch : $ALUoutput \leftarrow PC + Imm$ (výpočet nové adresy)

Cond \leftarrow nastavení příznaku

M – přístup k paměti + cyklus dokončení skoku

MRef. : $RegDat \leftarrow M[ALUoutput]$, nebo

$M[ALUoutput] \leftarrow B$

Branch : if (cond) $PC \leftarrow ALUoutput$ else $PC \leftarrow NPC$

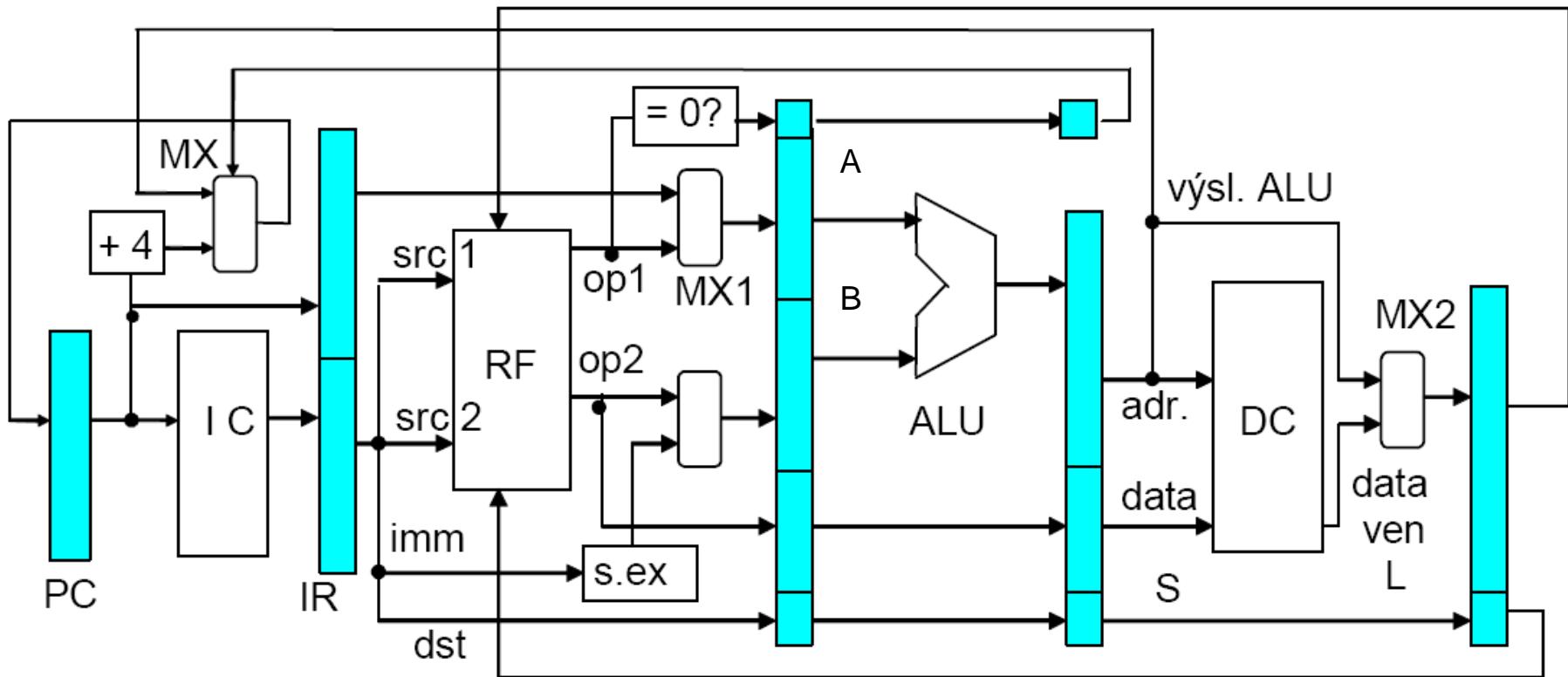
W – uložení výsledků

I R-R : $Regs[IRadr] \leftarrow ALUoutput$

I R-Imm: $Regs[IRadr] \leftarrow ALUoutput$

I Load: $Regs[IRadr] \leftarrow Reg\ Dat$

DLX procesor – výukový RISC procesor



MX: multiplexor, **DC:** cache dat, **= 0?** : detektor nuly (Cond), **IC:** cache instrukcí, **ALU:** aritmeticko-logická jednotka, **PC:** čítač instrukcí, **IR:** registr instrukce, **RF:** soubor registrů (register file), **s.ex:** jednotka rozšíření znaménka (sign extension), **dst:** adresa cílového registru, **src1** a **src2:** adresy zdrojových registrů, **imm:** přímý operand, **L/S:** load/store

Paměť (M) je rozdělena na paměť instrukcí (IC) a paměť dat (DC).

DLX – poznámky

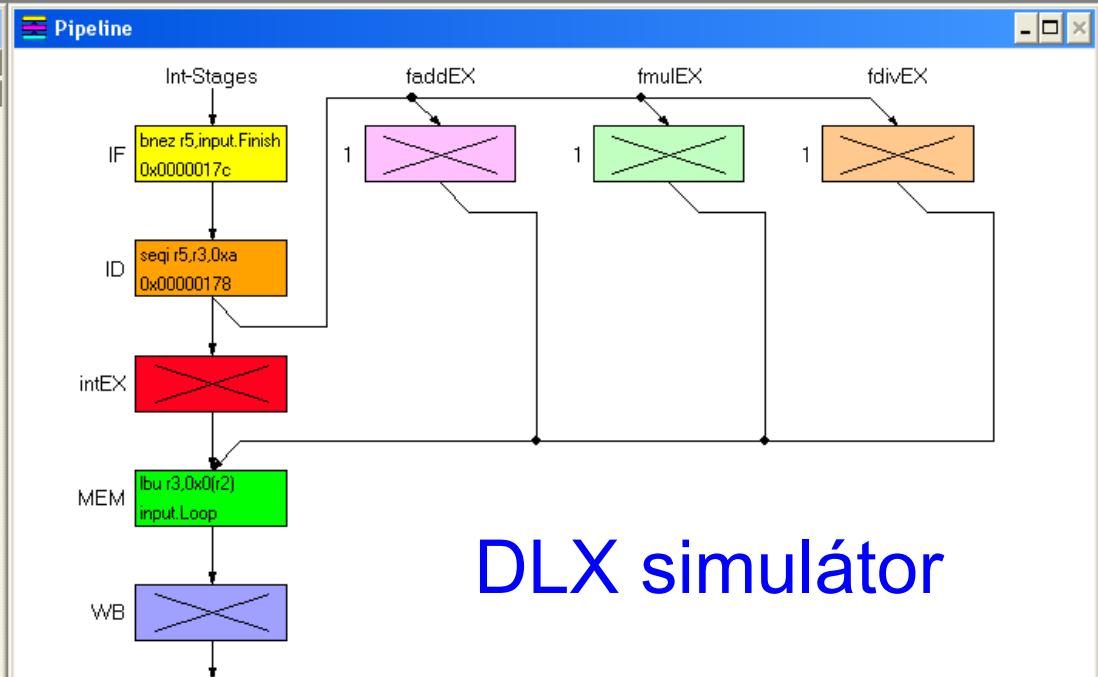
- V každém taktu je celý stav zpracování instrukce (kontext) obsažen úplně a výhradně v obvodech jen jednoho stupně.
- Proto se např. index cílového registru dst (5 bitů) kopíruje z jednoho stupně do druhého, než se nakonec použije jako adresa výsledku v souboru registrů. Např. proto se také kopíruje obsah registru (op2) určený k zápisu do paměti.
- Uvedená vlastnost linky znamená, že ostatní stupně jsou volné a dají se použít pro paralelní zpracování dalších nezávislých instrukcí.
- U podmíněného skoku se adresa získá buď inkrementací (+ 4 byty u 32-bitové instrukce) nebo se stávající adresa modifikuje přičtením hodnoty získané z instrukce (26 bitů) sčítáčkou v ALU.
- Přímý operand instrukce (imm) má šířku 16 bitů a v jednotce rozšíření znaménka (s.ex) se z něj vytvoří 32-bitový operand.
- Při čtení nebo zápisu do paměti D-cache může být adresa v jednom z registrů a může se k ní přičíst odstup (imm) uvedený v instrukci (16 bitů). Výpočet adresy se provádí v ALU.

Code

```

0x00000128      0x04401006      multd f2,f2,f0
0x0000012c      0x04040005      subd f0,f0,f4
0x00000130      0x0bffffec     j factLoop
0x00000134      0xbc02102c     sd PrintValue(r0),r2
0x00000138      0x200e1028     addi r14,r0,0x1028
0x0000013c      0x44000005     trap 0x5
0x00000140      0x44000000     trap 0x0
0x00000144      0xac021094     sw SaveR2(r0),r2
0x00000148      0xac031098     sw SaveR3(r0),r3
0x0000014c      0xac04109c     sw SaveR4(r0),r4
0x00000150      0xac0510a0     sw SaveR5(r0),r5
0x00000154      0xac011090     sw input.PrintPar(r0),r1
0x00000158      0x200e1090     addi r14,r0,0x1090
0x0000015c      0x44000005     trap 0x5
0x00000160      0x200e1084     addi r14,r0,0x1084
0x00000164      0x44000003     trap 0x3
0x00000168      0x20021034     addi r2,r0,0x1034
0x0000016c      0x20010000     addi r1,r0,0x0
0x00000170      0x2004000a     addi r4,r0,0xa
inputLoop    0x90430000 MEM   lbu r3,0x0(r2)
0x00000178      0x6065000a ID    seqi r5,3,0xa
0x0000017c      0x14a00014 IF   bnez r5,input.Finish
0x00000180      0x28630030     subi r3,r3,0x30
0x00000184      0x00240819     multu r1,r1,r4
0x00000188      0x00230820     addi r1,r1,r3
0x0000018c      0x20420001     addi r2,r2,0x1
0x00000190      0x0bffffe0     j inputLoop
0x00000194      0x8c021094     lw r2,SaveR2(r0)
0x00000198      0x8c031098     lw r3,SaveR3(r0)
0x0000019c      0x8c04109c     lw r4,SaveR4(r0)
0x000001a0      0x8c0510a0     lw r5,SaveR5(r0)
0x000001a4      0x4be00000    jr r31
0x000001a8      0x00000000     nop
0x000001a9      0x00000000

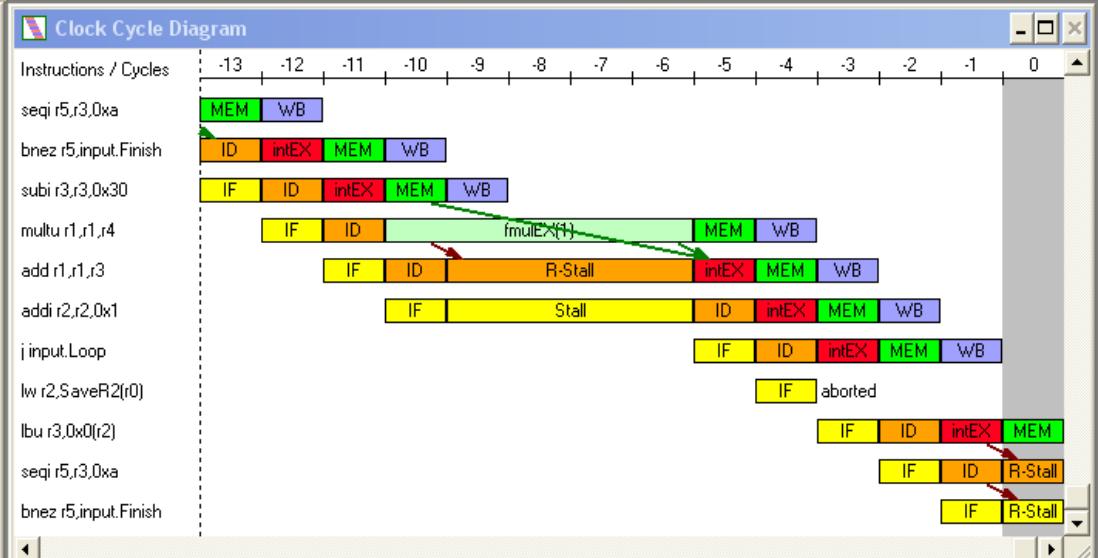
```



DLX simulátor

Register

PC= 0x00000180	R10= 0x00000000	F4= 0 F30= 0
IMAR= 0x0000017c	R11= 0x00000000	F5= 0 F31= 0
IR= 0x14a00014	R12= 0x00000000	F6= 0 D0= 0
A= 0x00000007	R13= 0x00000000	F7= 0 D2= 0
AHI= 0x00000000	R14= 0x00001084	F8= 0 D4= 0
B= 0x00000000	R15= 0x00000000	F9= 0 D6= 0
BHI= 0x00000000	R16= 0x00000000	F10= 0 D8= 0
BTA= 0x00000000	R17= 0x00000000	F11= 0 D10= 0
ALU= 0x00000000	R18= 0x00000000	F12= 0 D12= 0
ALUHII=0x00000000	R19= 0x00000000	F13= 0 D14= 0
FPSR= 0x00000000	R20= 0x00000000	F14= 0 D16= 0
DMAR= 0x00001035	R21= 0x00000000	F15= 0 D18= 0
SDR= 0x00000000	R22= 0x00000000	F16= 0 D20= 0
SDRHII=0x00000000	R23= 0x00000000	F17= 0 D22= 0
LDR= 0x0000000a	R24= 0x00000000	F18= 0 D24= 0
LDRHII=0x00000000	R25= 0x00000000	F19= 0 D26= 0
RO= 0x00000000	R26= 0x00000000	F20= 0 D28= 0
R1= 0x00000007	R27= 0x00000000	F21= 0 D30= 0
R2= 0x00001035	R28= 0x00000000	F22= 0
R3= 0x00000007	R29= 0x00000000	F23= 0
R4= 0x0000000a	R30= 0x00000000	F24= 0
R5= 0x00000000	R31= 0x00000108	F25= 0
R6= 0x00000000	FO= 0	F26= 0
R7= 0x00000000	F1= 0	F27= 0
R8= 0x00000000	F2= 0	F28= 0
R9= 0x00000000	F3= 0	F29= 0



Konflikty (hazardy) u řetězeného zpracování v procesorech, které mohou vést ke zpomalení linky

1. Strukturální – obvodová struktura neumožňuje současné provedení určitých akcí – např. současné čtení dvou hodnot z paměti nebo současné provedení dvou sčítání, pokud má procesor jednu ALU
2. Datové – když jsou zapotřebí data z předcházející instrukce, která není dokončena.
3. Řídicí – když skoková instrukce mění obsah PC, nebo jiné.

ad 1) F D E M W

F D E M W

F D E M W

F D E M W

požadavek na současné čtení instrukce a čtení operandu z paměti

Řešení: rozdělit paměť na paměť instrukcí a paměť dat

Obecné řešení: přidání výpočetních jednotek

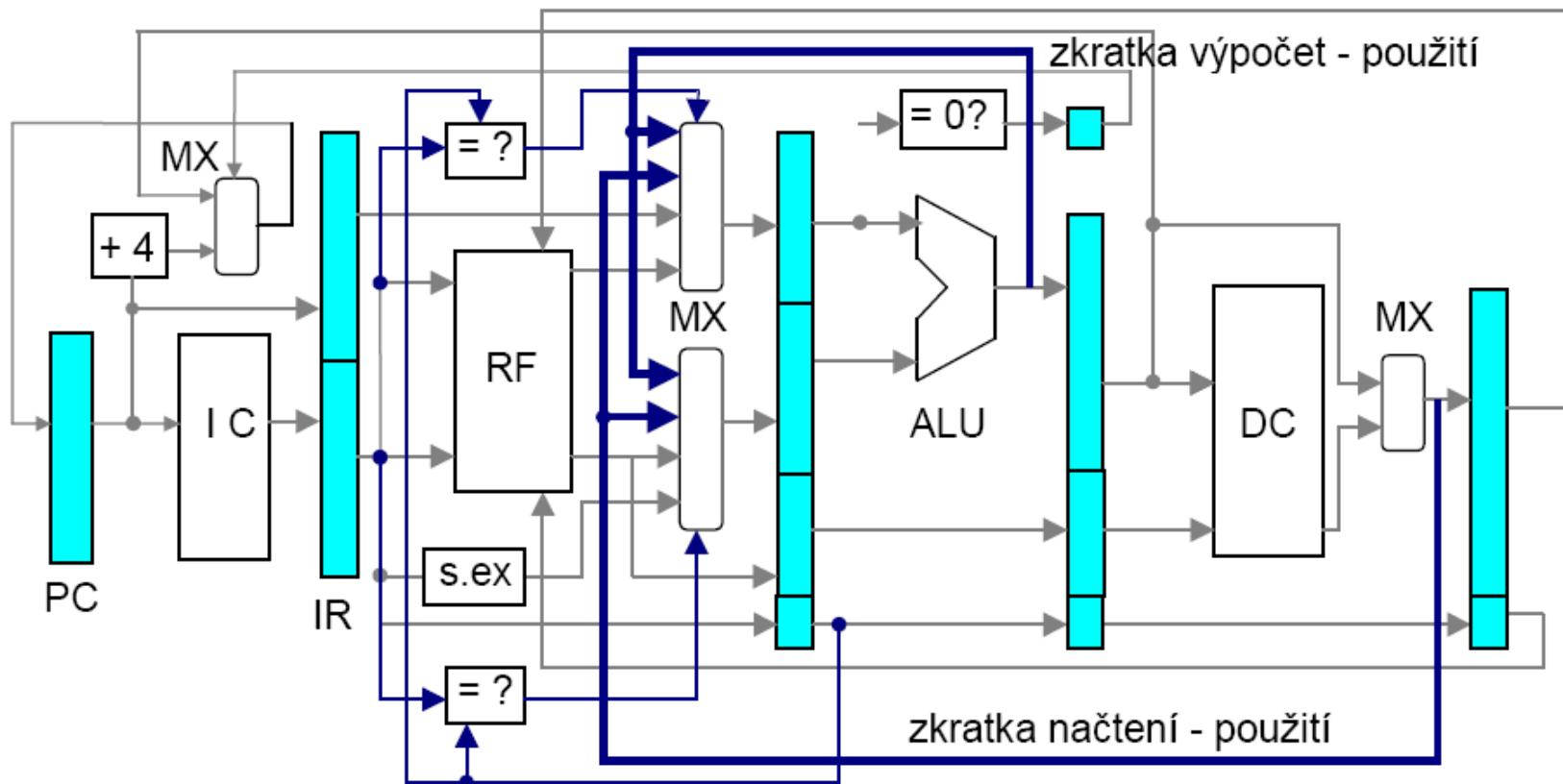
Datové konflikty

	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

Výsledek instrukce ADD bude k dispozici v 5. taktu.
Instrukce SUB, AND a OR tak provedou chybný výpočet.

Forwarding (bypassing)

Poskytnutí mezivýsledku dřív než bude zapsán do registru. Je to umožněno přidáním speciálních datových cest a rozšířením multiplexorů – viz příklady na obrázku. Pomáhá řešit datové konflikty.



Kdy Forwarding nefunguje?

		1	2	3	4	5	6	7	8
LW	R1, a	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

LW (load word) má data až na konci fáze MEM, ale SUB je potřebuje na začátku EX.

Je nutné přidat obvod (pipeline interlock), který hazard detekuje a pozastaví linku (**fáze stall**), aby nebyla porušena logika programu. Někdy stačí, když překladač „přeskládá“ instrukce.

		1	2	3	4	5	6	7	8	9
LW	R1, a	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

Klasifikace datových konfliktů

Mějme instrukce A a B, a platí, že A předchází B.

RAW – Read After Write

B se pokouší číst zdroj před tím, než A do něj zapsala. B přečte starou hodnotu. Řešení: forwarding.

WAW – Write After Write

B se pokouší zapsat operand dřív, než je proveden zápis instrukcí A. Zápis je tak proveden v chybném pořadí. V DLX nenastane, protože je povoleno zapisovat jen ve fázi WB.

WAR – Write After Read

B se pokouší zapsat operand dřív než je přečten A. A tak získá novější hodnotu, což je chyba. Nenastane v DLX.

RAR – není hazard :-)

Řídicí konflikty

ad 3) BC F D E M W // BC...podmíněný skok
BC+1 F - - F D E M W
BC+2 F D E M W
BC+3 F D E M W
atd. linka pozastavena na 3 takty, pokud se má skočit.

Nevím, kam skočit! Adresa bude známa až po fázi M na řádku BC.
Je třeba co nejrychleji zjistit adresu následující instrukce.

Řešení:

1. **Zpožděný skok**, tedy vložit jiné, užitečné instrukce, což znamená přeskládat instrukce programu (kompilátor).
2. Nelze-li takové užitečné instrukce najít, vložit NOPy.
3. Zavést specializované obvody (**prediktor skoku** a **paměť cílové adresy skoku** - BTB).

Zpožděný skok

Původní kód:

```
JC L1  
XOR R5, R5, R3  
JMP 12  
L1: AND R5, R5, R3  
L2: LD R6, adr1  
     LD R7, adr2  
     MOV R8, R7  
     ST R1
```

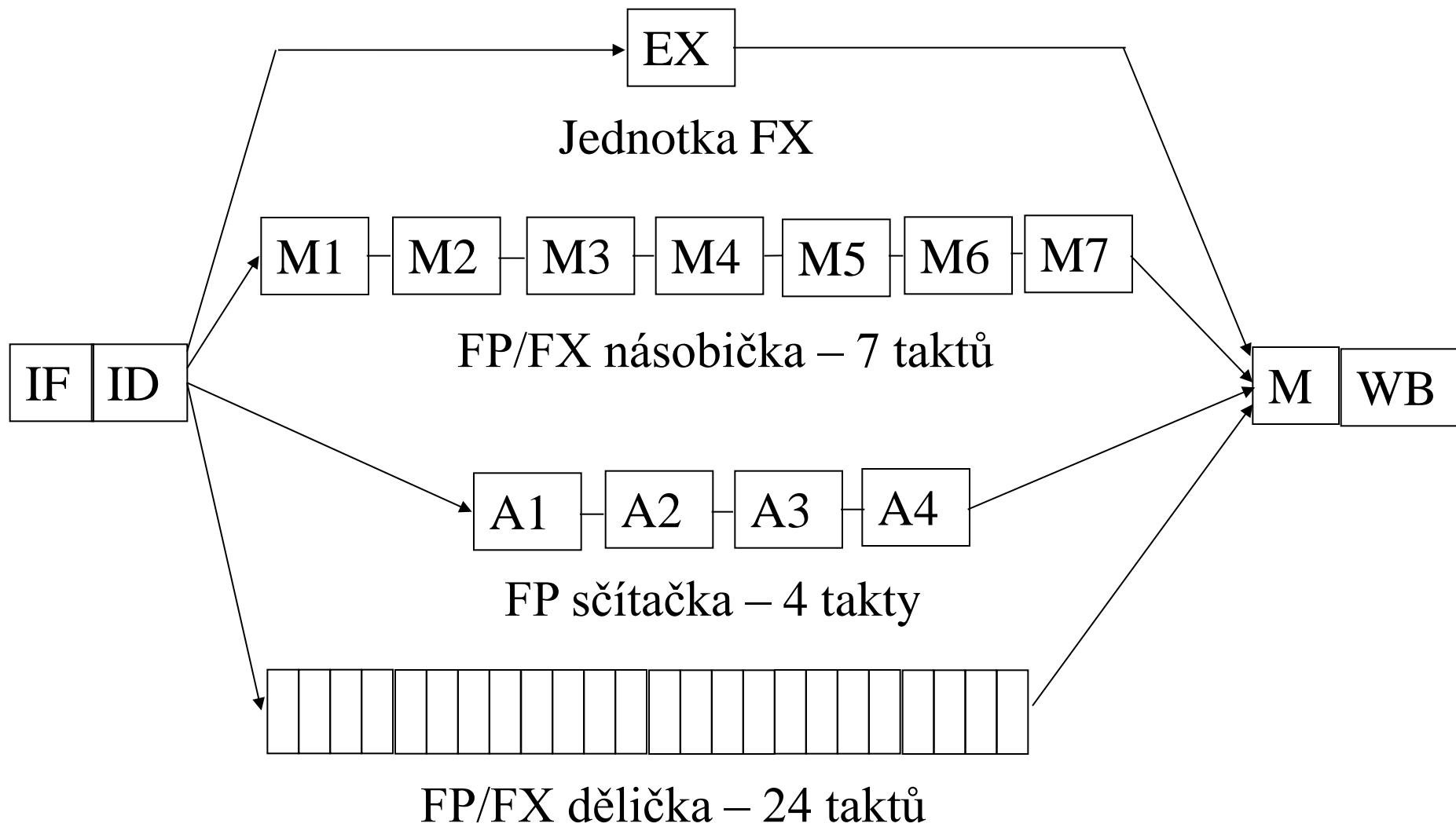
Optimalizovaný kód:

```
JC L1  
LD R6, adr1  
LD R7, adr2  
MOV R8, R7  
XOR R5, R5, R3  
JMP 12  
L1: AND R5, R5, R3  
L2: ST R1
```

Jaká bude následující adresa po JC?

Tři vyznačené instrukce nejsou datově závislé na okolním kódu a musí se vždy vykonat. Je tedy možné je přesunout a tím získat čas na zjištění následující adresy po JC. Nedoje k pozastavení linky.

Rozšíření pipeline: FX a FP operace



Příklad a potenciální problém

Mějme tento úsek programu: (.D znamená dvojnásobnou přesnost)

DIV.D F0, F2, F4

ADD.D F10, F10, F8

SUB.D F12, F12, F14

Na první pohled zde nejsou problémy, protože mezi instrukcemi nejsou datové závislosti. Ovšem podle předcházející řetězené struktury budou instrukce ADD a SUB dokončeny dříve, než dříve zahájená instrukce DIV (out-of-order completion).

Pokud však nastane v době dokončování DIV výjimka, instrukce ADD a SUB se již nesmějí provést - tak jak by se stalo *na klasické neřetězené výpočetní struktuře*.

Řešení: nutno zavést koncept precizního přerušení (viz pokročilý kurz o procesorech)

Poznámky k zřetězenému zpracování

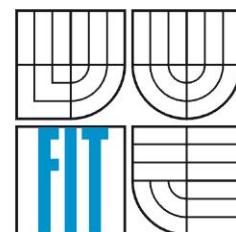
- Zřetězené zpracování přináší urychlení výpočtu nejen v procesorech, ale i jiných číslicových obvodech (např. pro zpracování obrazu, bioinformatických dat apod.).
- Pokud použijeme zřetězené zpracování v procesoru, musíme dodat řadu podpůrných obvodů a řešit řadu nových problémů.
- Podrobnější analýza problémů zřetězeného zpracování v procesorech bude provedena v magisterském kurzu Architektura procesorů.
- Kromě řetězení se používají i další koncepty, které vedly ke vzniku nových kategorií procesorů:
 - superskalární architektury
 - VLIW procesory
 - vektorové procesory
 - multivláknové procesory
 - atd.

Literatura

- Drábek, V. Výstavba počítačů. Skriptum VUT, 1995
- Dvořák, V., Drábek, V.: Architektura procesorů. Studijní opora. FIT VUT v Brně 2006
- Win DLX simulator
 - <http://electro.fisica.unlp.edu.ar/arq/downloads/Software/WinDLX/windlx.html>
- Prabhu G. M.: Computer architecture tutorial
 - <http://www.cs.iastate.edu/~prabhu/Tutorial/title.html>

Operace ALU

INP 2015
FIT VUT v Brně



Princip ALU (FX)

Požadavky - funkční:

Logické operace

Sčítání (v doplňkovém kódu)

Posuvy/rotace

Násobení

Dělení

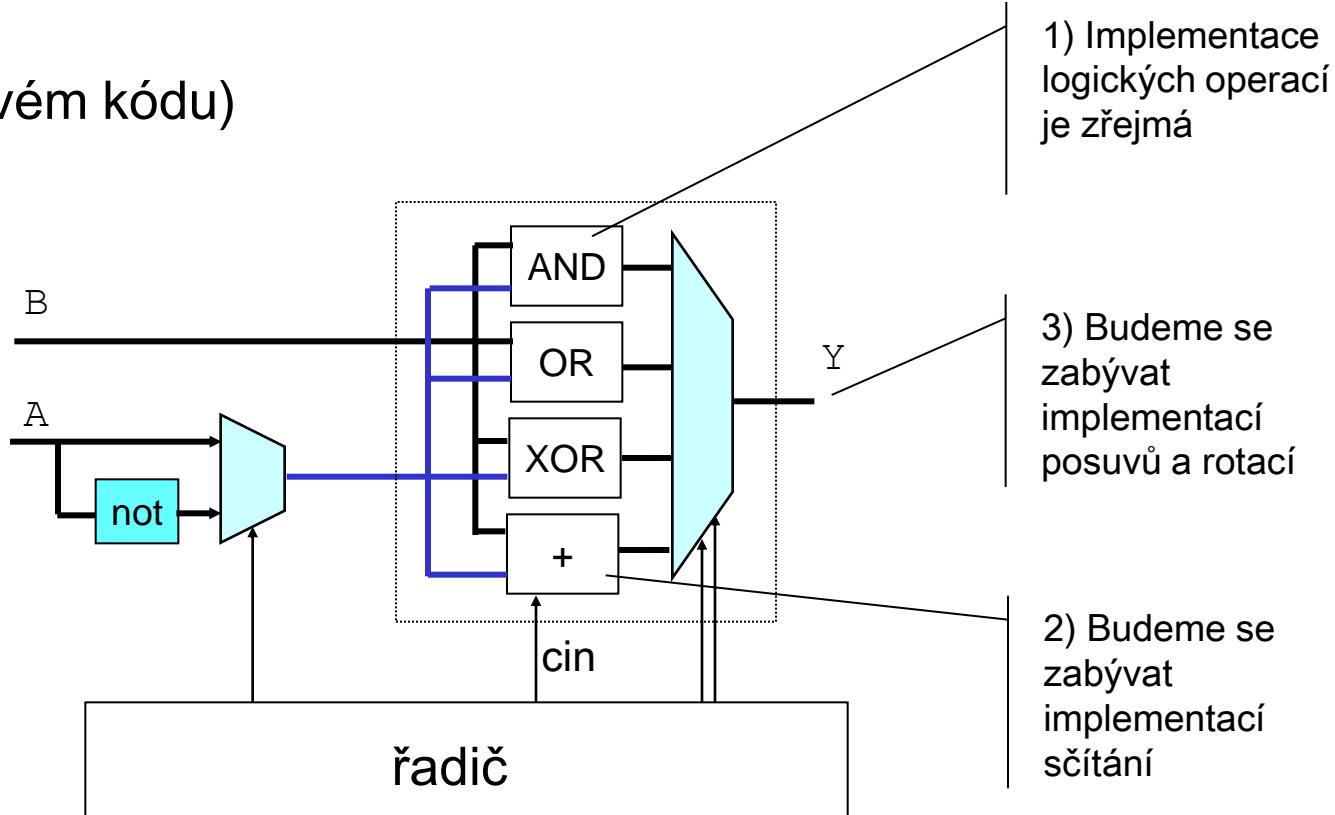
Další požadavky:

zpoždění

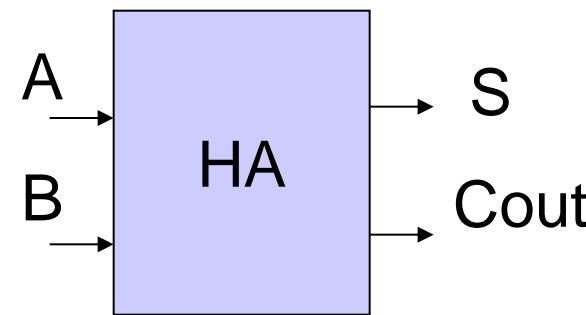
plocha

příkon

atd.



Poloviční sčítačka



S:

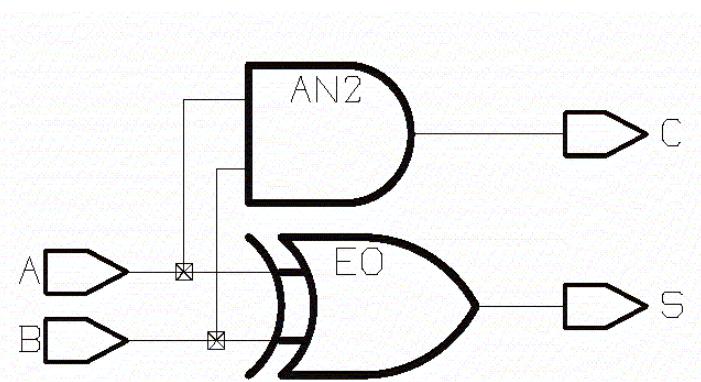
	A
0	1
1	0

B

C:

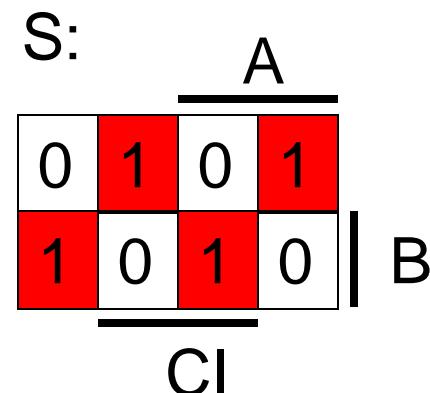
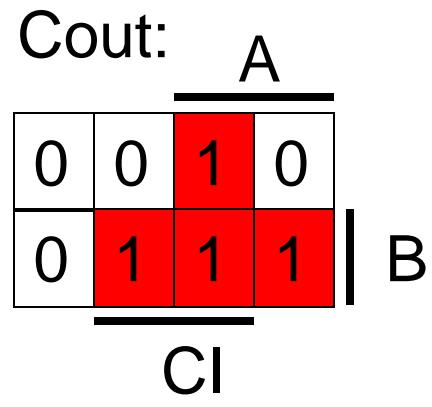
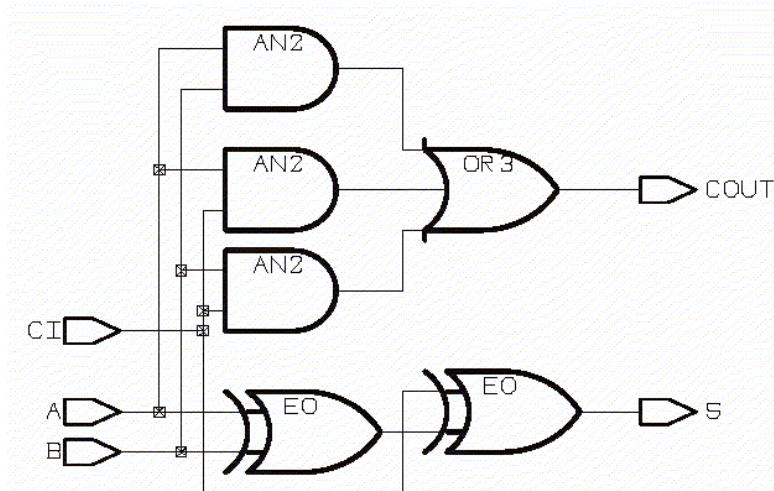
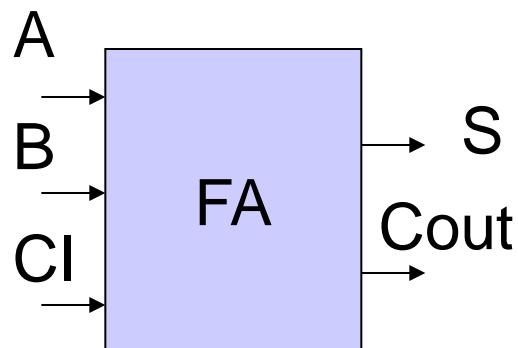
	A
0	0
0	1

B



Úplná sčítačka

A	B	CI	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



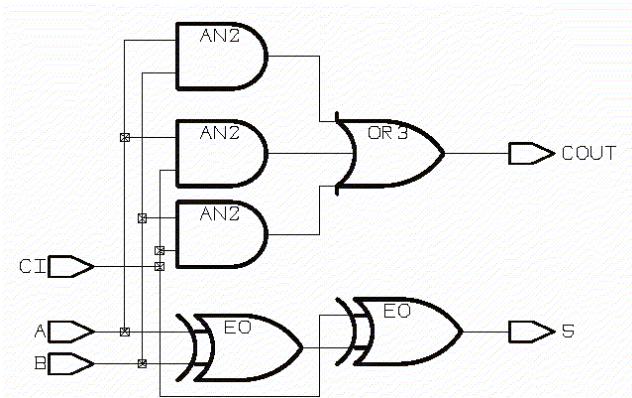
Úplná sčítačka – úrovně popisu

1. Úroveň chování

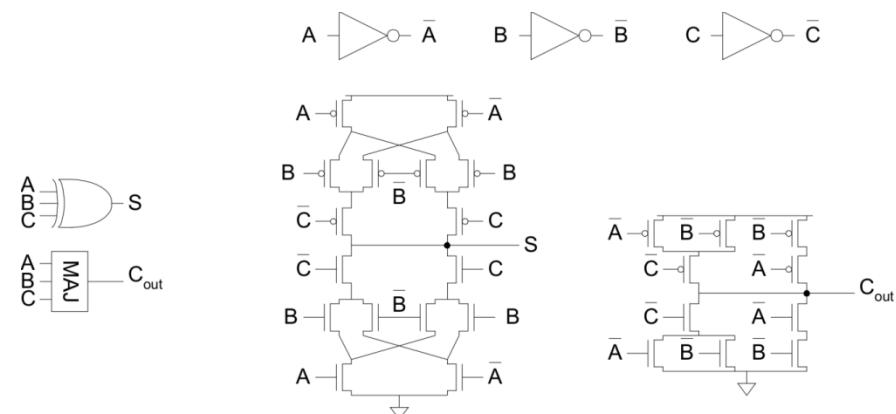
$$S \leq A \text{ xor } B \text{ xor } CI;$$

$$COUT \leq (A \text{ and } B) \text{ or } (A \text{ and } CI) \text{ or } (B \text{ and } CI);$$

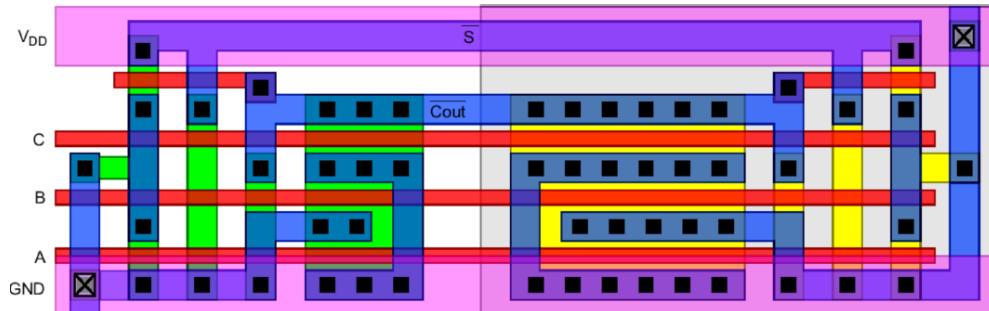
2. Úroveň hradel



3. Úroveň tranzistorů



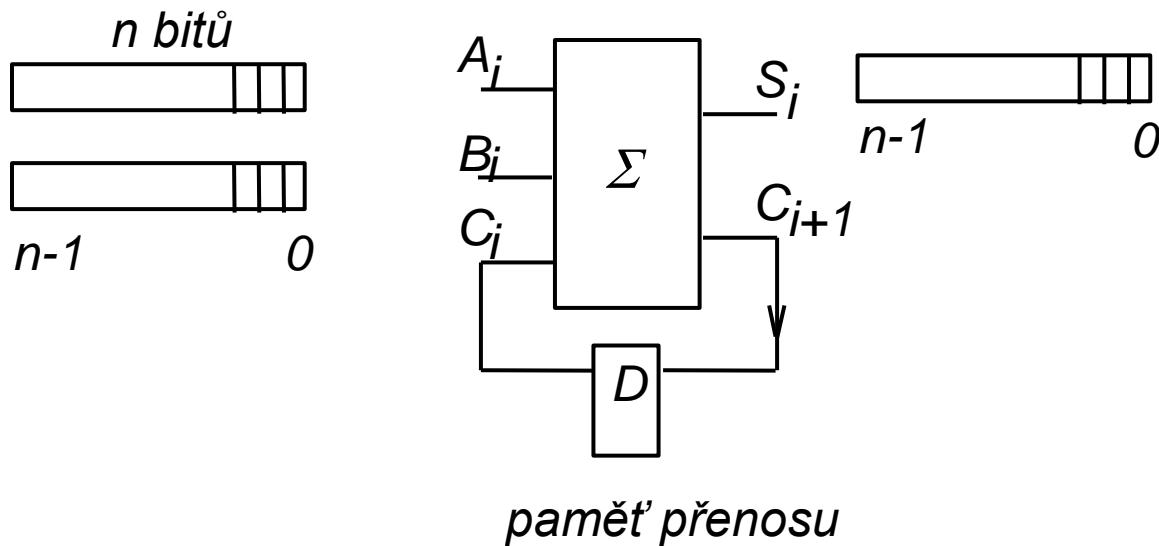
4. Úroveň fyzická (layout)



Cf Weste, Harris: CMOS VLSI Design, 4th ed, 2010

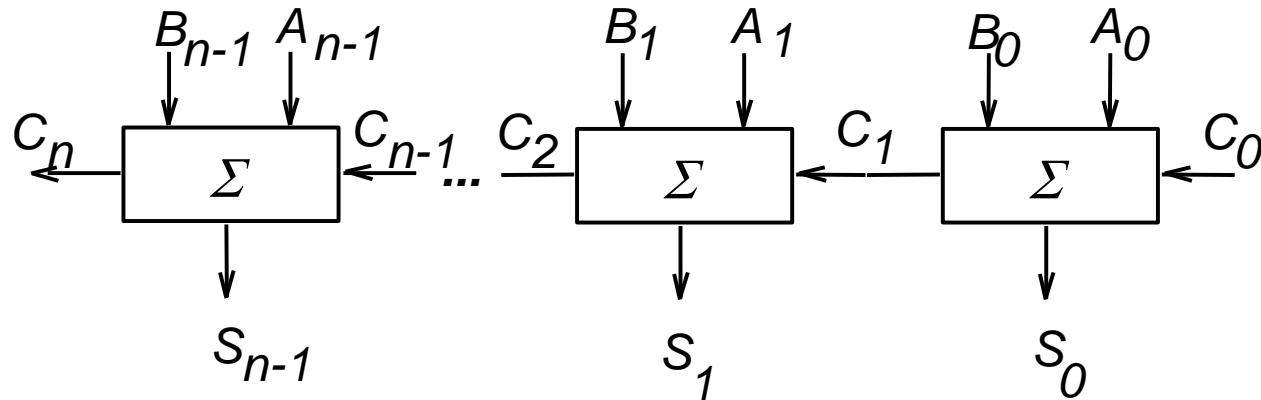
Čím nižší úroveň popisu, tím přesnější znalost parametrů obvodu (plocha, zpoždění, příkon ...)

Sériová sčítačka



- Jde o jednabitovou sčítačku použitou pro **sériové sčítání** dvou n -bitových čísel, připravených ve vstupních registrech. Na výstupu S_i se postupně objevují bity součtu počínaje nejnižším bitem 0, a výstup přenosu C_{i+1} se zachycuje na dobu jednoho taktu (T_c) v klopném obvodu D (Carry Save). (Předpokládáme, že jde o synchronní sčítačku s taktem T_c).

Paralelní sčítáčka s postupným přenosem



- Objeví-li se u první sčítáčky výstup přenosu za dobu 2Δ , kde Δ je přenosové zpoždění jednoho logického členu, na výstupu druhé sčítáčky je to již 4Δ , atd. Výstup přenosu se u poslední sčítáčky objeví za dobu $2n\Delta$.
- Pro prakticky používané šířky sčítáček 32, 64 a 128 bitů je tato doba nepřijatelně dlouhá.
- Je proto snaha navrhovat sčítáčky s rychlým přenosem.
- Hledáme kompromis mezi zpožděním a počtem hradel (plochou).

Rozšířená sčítáčka

C _i	A _i	B _i	S _i	P _i	G _i	C _{i+1}
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	0	0	1	1
1	0	0	1	0	0	0
1	0	1	0	1	0	1
1	1	0	0	1	0	1
1	1	1	1	0	1	1

Rozšířená sčítáčka:

Zavedeme dva pomocné výstupy:

P_i - propagate carry („1“, když přenos sčítáckou prochází, A_i≠B_i)

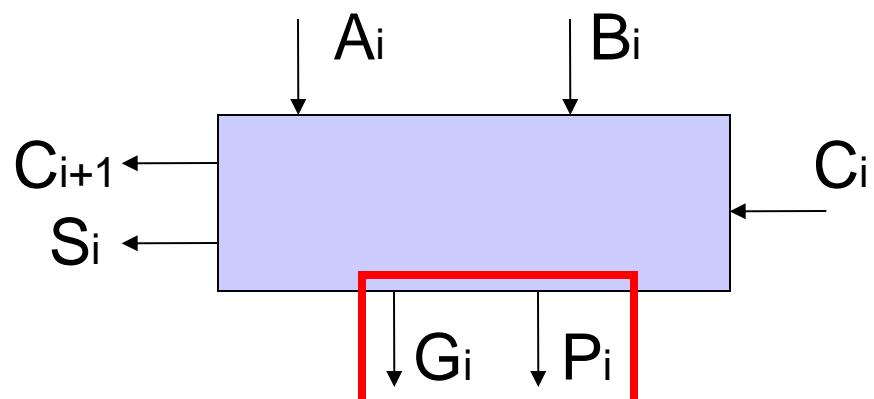
G_i - generate carry (vznik přenosu bez ohledu na hodnotu C_i)

$$S_i = A_i \oplus B_i \oplus C_i \quad (\text{zpoždění: } 2\Delta)$$

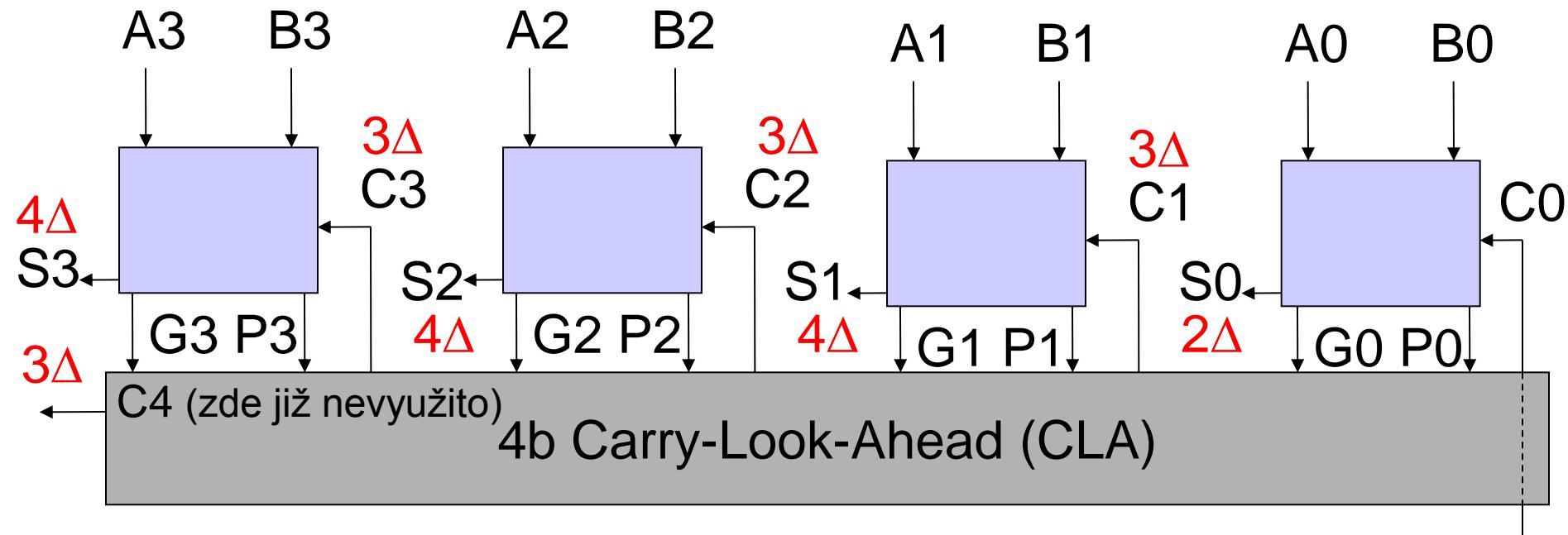
$$P_i = A_i \oplus B_i \quad (\Delta)$$

$$G_i = A_i \cdot B_i \quad (\Delta)$$

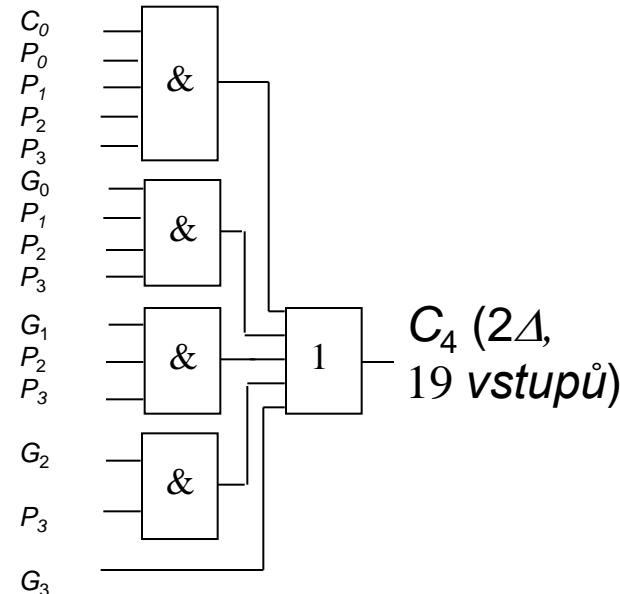
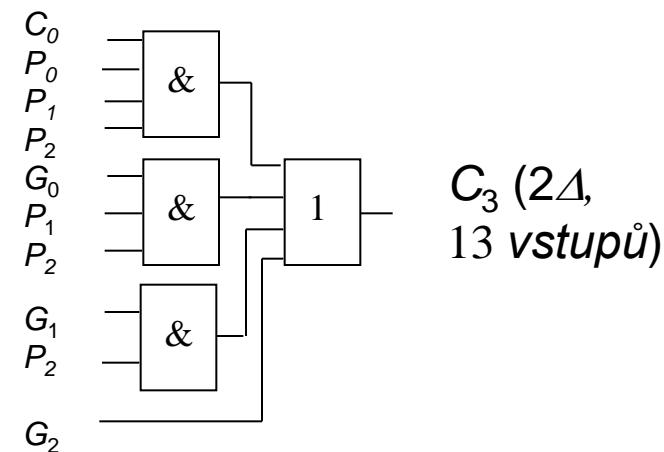
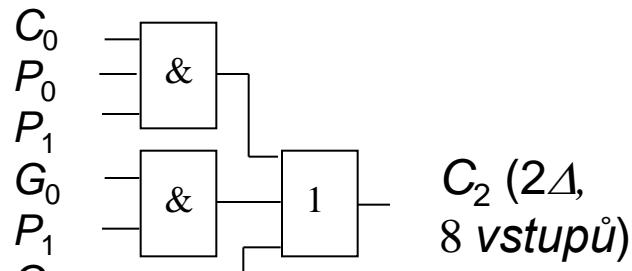
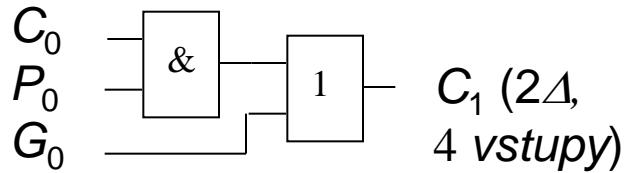
$$C_{i+1} = P_i \cdot C_i + G_i \quad (3\Delta)$$



4b sčítačka s CLA – zpoždění 4Δ



Logický obvod CLA

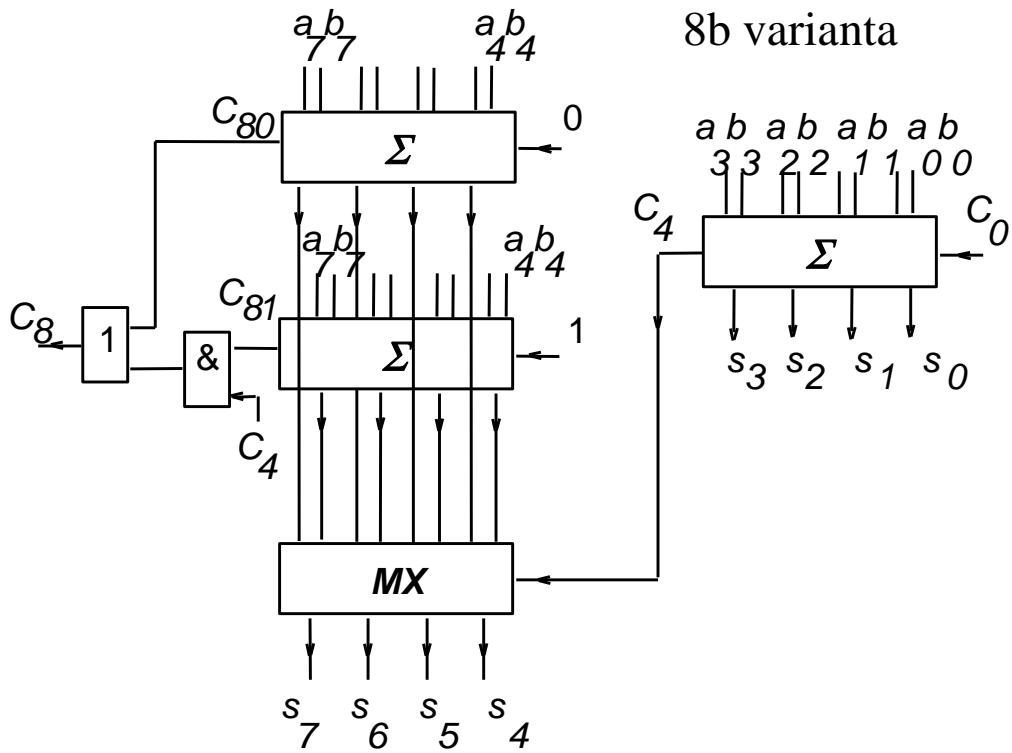


Určete obecný matematický výraz pro složitost (tedy pro „počet vstupů“ u jednotlivých funkcí C_i) obvodu CLA.

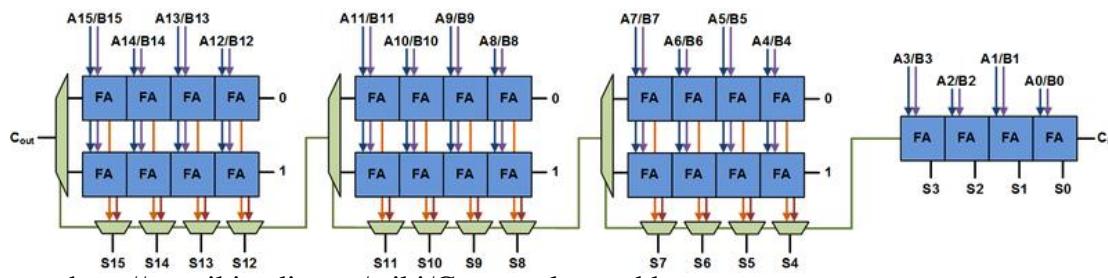
n-bitová sčítačka s CLA – konst. zpoždění 4Δ

- Cesty pro šíření postupného přenosu jsou zrušeny a na vstupy přenosu všech sčítaček se přivádějí příslušné výstupy generátoru přenosu (obvodu CLA).
- Funkce P_i , G_i se tvoří se zpožděním Δ , v čase 3Δ jsou k dispozici všechny rychlé přenosy, a součet je tedy vytvořen v čase 4Δ .
- Popsané uspořádání n-bitové sčítačky je nejrychlejší možné řešení.
- Složitost (zejména počet vstupů u log. členů) dvoustupňového generátoru přenosu však roste pro rostoucí šířku sčítačky s druhou mocninou šířky. Pro šířky 32 a 64 bitů je toto řešení již technologicky nepřijatelné.
- Byla proto navržena řešení umožňující za cenu nárůstu zpoždění zmenšit potřebnou plochu na čipu:
 - stromový generátor přenosu s CLA – např. pomocí několika 4b obvodů CLA uspořádaných do stromu
 - výběr přenosu – viz příklad
 - přeskakování přenosu atd.

Sčítačka s výběrem přenosu



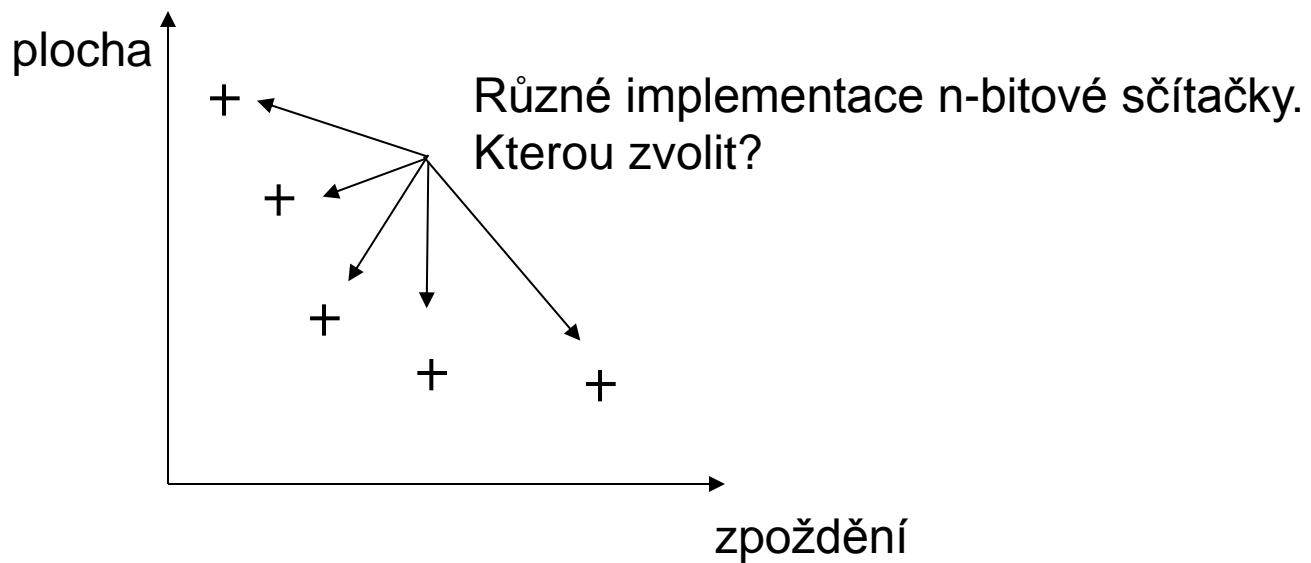
- 8-bitová sčítačka je rozdělena na dvě 4-bitové sčítačky.
- Horní sčítačka je zdvojená, přičemž jedna má vstup přenosu 0, a druhá 1.
- Obě připraví výsledek, a 4-vstupový multiplexor pak vybere jednu z nich podle hodnoty přenosu C4.
- Obvodové řešení se tedy prodražilo o více než 50% vzhledem ke sčítačce s postupným přenosem.



- Pro větší šířky sčítaček se postupuje obdobně; každý blok je zdvojen a je přidán výběrový multiplexor.

Hodnocení složitosti logických obvodů

- Jde o nalezení **kompromisního řešení** mezi **cenou** a **výkonností**. V poslední době se navíc hledá kompromis s příkonem.
- **Cena** – popíše se např. součtem počtu vstupů všech použitých logických členů, součtem počtu logických členů, plochou na čipu apod.
- **Výkonnost** – popíše se hodnotou nejdelšího přenosového zpoždění daného obvodu (které následně určuje f_{max}).



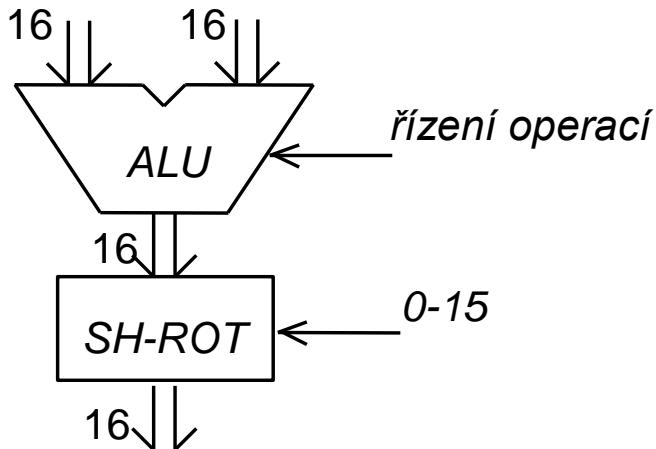
Operační rychlosť a plocha rôznych typů sčítaček šířky n

Typ	čas	plocha
Postupný prenos	$O(n)$	$O(n)$
2 – stupňový CLA	4	$O(n^2)$
Stromový CLA k -nární	$O(\log_k n)$	$O(n \log_k n)$
Přeskakování prenosu	$O(\sqrt{n})$	$O(n)$
Výběr prenosu	$O(\sqrt{n})$	$O(n)$

kde k je počet dílčích prenosov, ktoré se skladají v jednom uzlu stromu.

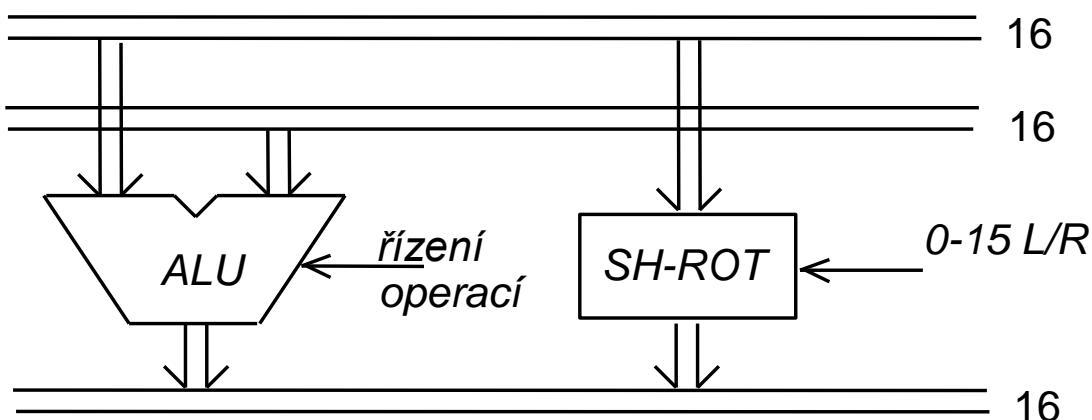
Další operace ALU – posuvy a rotace

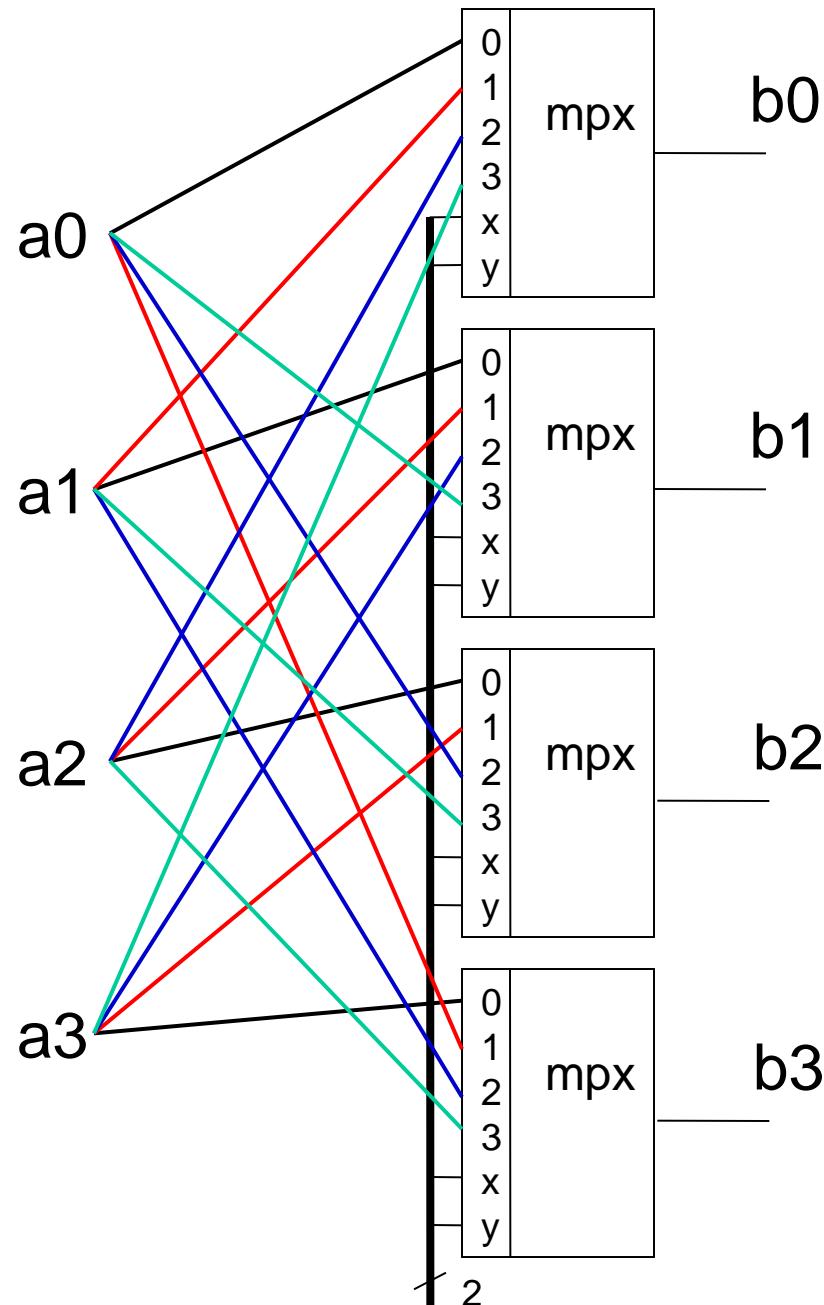
a)



- Blok posuvů a rotací je umístěn buď za výstupem ALU, nebo paralelně k bloku ALU.
- **Řídicí vstupy**: směr posuvu, jeho typ (logický nebo aritmetický), a počet bitů, o které se posouvá.
- U každého operačního bloku se musí pamatovat na operaci **identity**.
- Implementace: válcový posouvač (Barrel Shifter) – pomocí multiplexorů, ne posuvný registr!!

b)





4b válcový posouvač pro rotace vpravo
(4 x 4-vst. MUX, zpoždění 2Δ)

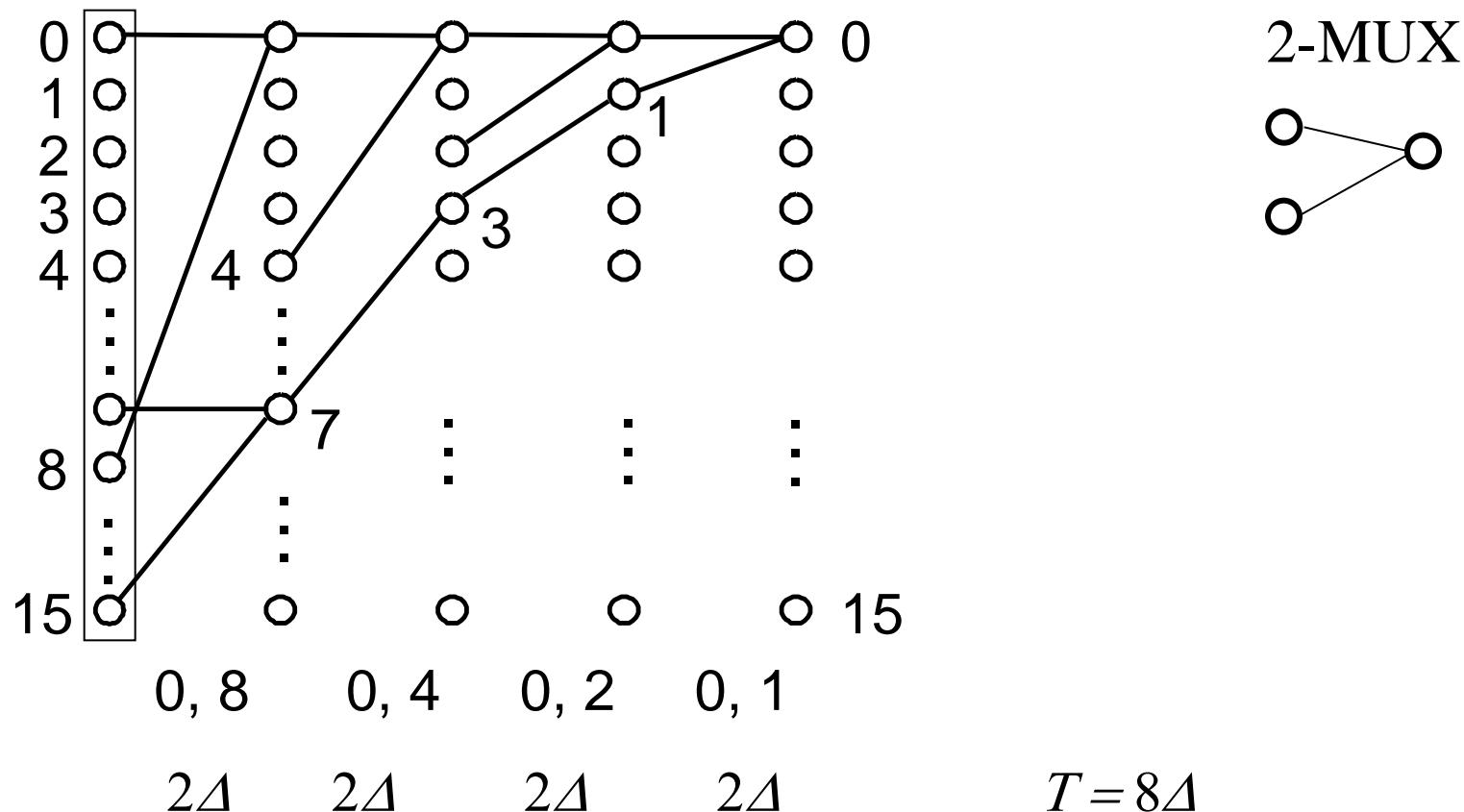
b_3	b_2	b_1	b_0	
↓	↓	↓	↓	
a_3	a_2	a_1	a_0	0 bit
a_0	a_3	a_2	a_1	1 bit
a_1	a_0	a_3	a_2	2 bit
a_2	a_1	a_0	a_3	3 bit

Možnosti realizace 16b válcového posouvače

(16) - 16 x 16-vstupový multiplexor: drahé řešení, zpoždění 2Δ

(2, 2, 2, 2) - 64 x 2-MUX, zpoždění 8Δ , levnější řešení, viz obrázek

(2, 4, 2) – cenu a zpoždění odvodte jako domácí úkol

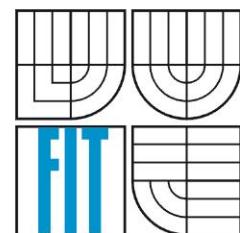


Poznámky k obrázku

- Při optimalizaci vnitřní struktury válcového posouvače se kromě základních kritérií, což je cena (počet logických členů příp. počet vstupů logických členů) a přenosové zpoždění, používá pro každou technologii ještě povolené vstupní a výstupní větvení, příp. ještě další kritéria. Může se pak dospět ke struktuře, která používá v jednotlivých stupních **multiplexory s rozdílným počtem vstupů**.
- Je vhodné pak popisovat struktury symbolicky. Jednostupňové uspořádání s 16-vstupovými multiplexory se zapíše (16), čtyřstupňové uspořádání se 2-vstupovými multiplexory jako (2,2,2,2), se 4-vstupovými MUX (4,4), smíšená struktura např. (2,4,2) atd. Přenosové zpoždění válcového posouvače se strukturou (2,2,2,2) je 8 jednotkových zpoždění Δ .
- Poznámka: Logická struktura multiplexoru je dvoustupňová, s přenosovým zpožděním 2Δ .

Násobení

INP 2015
FIT VUT v Brně



Násobení a násobičky

- Při násobení čísel v dvojkové soustavě můžeme násobit absolutní hodnoty čísel a pak doplnit do výsledku znaménko, anebo raději násobit přímo čísla se znaménkem.
- Vlastní násobení může být prováděno sekvenčně (postupně), anebo kombinačně (v jednom kroku). Rozlišujeme proto
 - sekvenční násobičky
 - kombinační násobičky
- Podle typu operandu rozlišujeme násobičky pracující v
 - pevné řádové čárce a
 - plovoucí řádové čárce
- Cílem implementace je získat součin co nejrychleji, za co nejnižší cenu (počet hradel, plocha čipu), popř. s co nejnižším příkonem.
- Pro pevnou řádovou čárku si ukážeme:
 - Princip násobení
 - Sekvenční násobičky
 - Kombinační násobičky
 - Princip urychlení – Boothovo překódování, Wallaceův strom

Princip násobení (bez znaménka)

Mějme dvě čísla: N-bitový násobitel $x_{N-1}x_{N-2}\dots x_0$ a M-bitový násobenec $y_{M-1}y_{M-2}\dots y_0$. Součin P potom bude:

$$P = \left(\sum_{j=0}^{M-1} y_j 2^j \right) \left(\sum_{i=0}^{N-1} x_i 2^i \right) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$

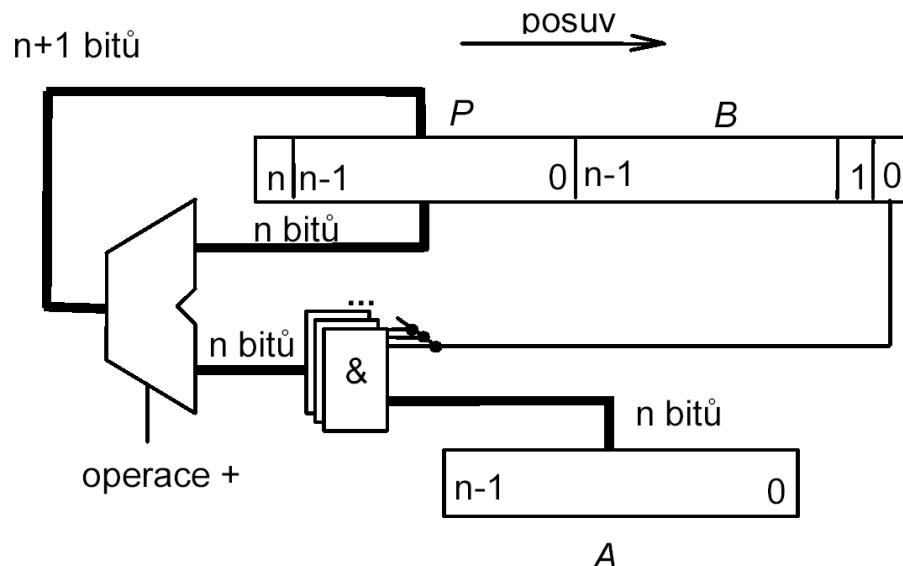
	y_5	y_4	y_3	y_2	y_1	y_0	Násobenec (multiplicand)
	x_5	x_4	x_3	x_2	x_1	x_0	Násobitel (multiplier)
	$x_0 y_5$	$x_0 y_4$	$x_0 y_3$	$x_0 y_2$	$x_0 y_1$	$x_0 y_0$	
	$x_1 y_5$	$x_1 y_4$	$x_1 y_3$	$x_1 y_2$	$x_1 y_1$	$x_1 y_0$	
	$x_2 y_5$	$x_2 y_4$	$x_2 y_3$	$x_2 y_2$	$x_2 y_1$	$x_2 y_0$	
	$x_3 y_5$	$x_3 y_4$	$x_3 y_3$	$x_3 y_2$	$x_3 y_1$	$x_3 y_0$	
	$x_4 y_5$	$x_4 y_4$	$x_4 y_3$	$x_4 y_2$	$x_4 y_1$	$x_4 y_0$	
	$x_5 y_5$	$x_5 y_4$	$x_5 y_3$	$x_5 y_2$	$x_5 y_1$	$x_5 y_0$	
p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4
							p_3
							p_2
							p_1
							p_0

částečné
součiny
(partial
products)

Součin (product)

Výsledek je na 12 bitech, obecně na $M+N$ bitech.

Sekvenční násobička



Př. $n=4, 1111 \times 1011 = 10100101$
 $(\Rightarrow \text{označuje posun vpravo})$

Inicializace:

A: 1111

PB: 00000 1011

Krok 1:

$$\begin{array}{r} \text{PB: } 00000 \underline{1011} \\ +1111 \\ \hline \end{array}$$

$$\text{PB: } 01111 \underline{1011}$$

$$\Rightarrow 00111 \underline{1101}$$

Krok 3:

$$\begin{array}{r} \text{PB: } 01011 \underline{0110} \\ +0000 \\ \hline \end{array}$$

$$\text{PB: } 01011 \underline{0110}$$

$$\Rightarrow 00101 \underline{1011}$$

Krok 2:

$$\begin{array}{r} \text{PB: } 00111 \underline{1101} \\ +1111 \\ \hline \end{array}$$

$$\text{PB: } 10110 \underline{1101}$$

$$\Rightarrow 01011 \underline{0110}$$

Krok 4:

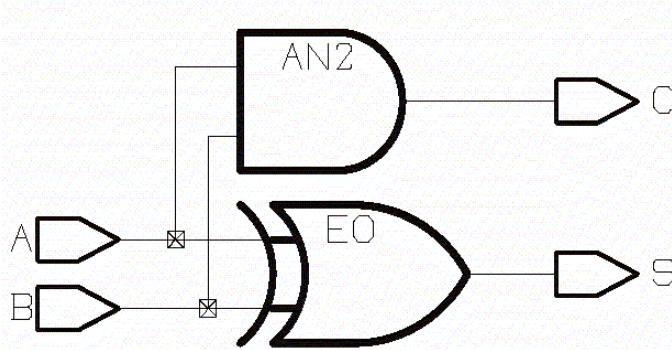
$$\begin{array}{r} \text{PB: } 00101 \underline{1011} \\ +1111 \\ \hline \end{array}$$

$$\text{PB: } 10100 \underline{1011}$$

$$\Rightarrow \underline{01010 \ 0101}$$

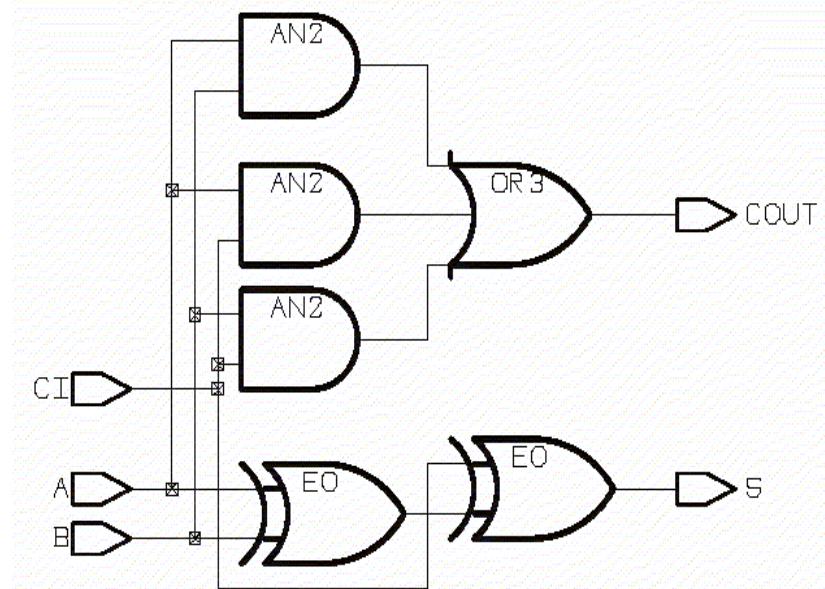
- U sekvenčního násobení čísel A x B se do dolní poloviny spojených registrů PB připraví násobitel B, do horní poloviny nuly, násobenec do registru A. Postup násobení je následující:
- (1) Nejnižším bitem registru PB se vynásobí násobenec A a přičte se k horní části registru PB, kde se udržuje průběžný součet dílčích součinů.
- (2) Obsah registrů PB se posune o jeden bit vpravo.
- Kroky 1, 2 provedeme celkem n-krát.
- Výhoda: nízký počet hradel. Nevýhoda: nízká rychlosť, n taktů pro násobení

Obvodová realizace násobení je založena na sčítačkách



Poloviční sčítačka:

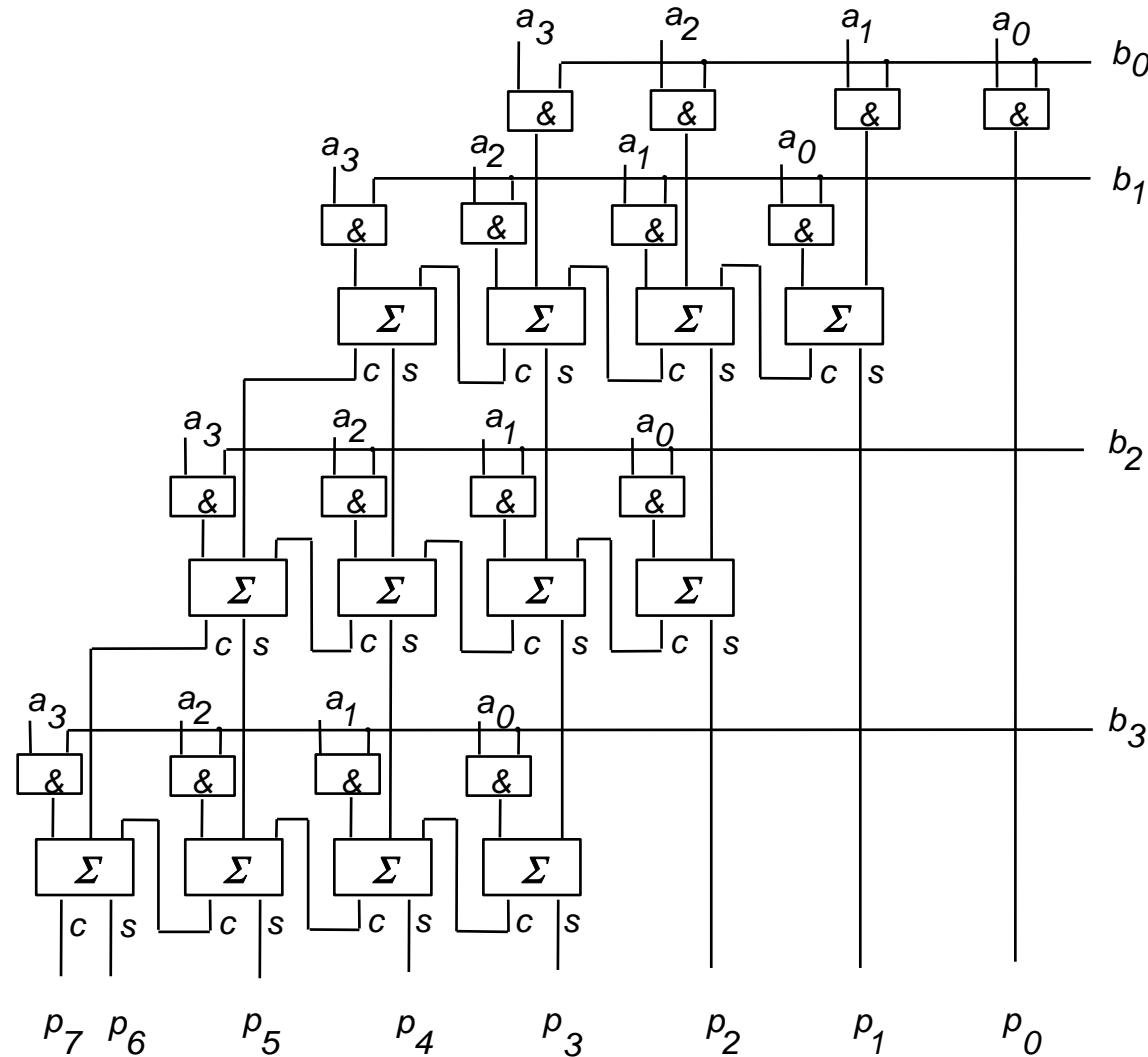
Zpoždění: 1 logický člen



Úplná sčítačka:

Zpoždění: 2 logické členy

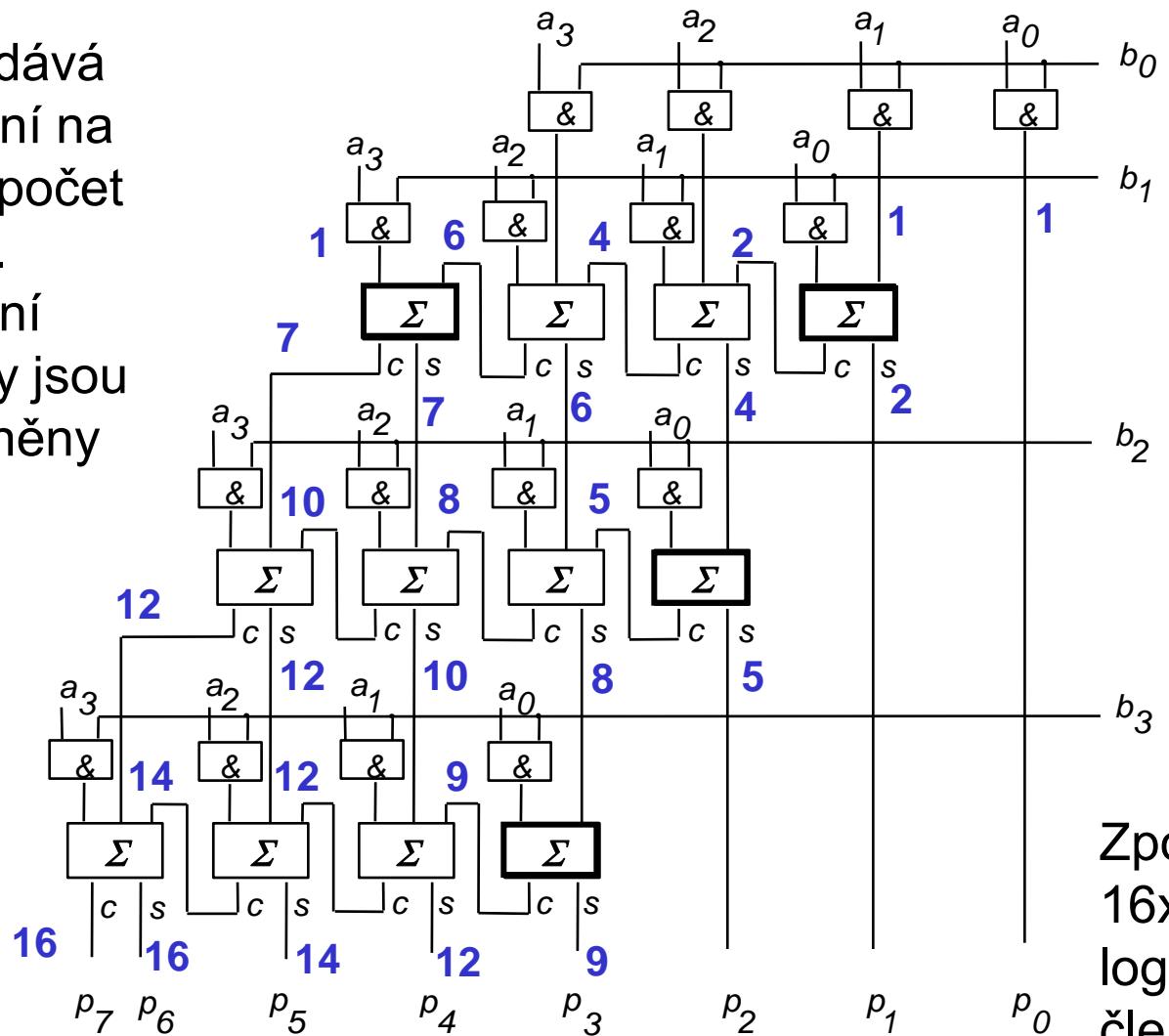
Kombinační násobička (v jednom kroku)



Kombinační násobička – odvození zpoždění

Číslo udává zpoždění na vodiči (počet hradel).

Poloviční sčítačky jsou znázorněny tučně.



Zpoždění:
16x zpoždění
logického
členu.

Popis ke kombinační násobičce

Jednotlivé dílčí součiny, tedy 1- a 0-násobky násobence, se tvoří řadou hradel a sčítají se na čtyřbitových sčítačkách s postupným přenosem.

Celkový **počet hradel** pro násobičku $n \times n$ bitů je $O(n^2)$, celkový **počet jednobitových sčítaček** je $(n-1) \times n$.

Zpoždění sčítačky určíme nalezením **nejdelší cesty** v kombinační síti. Je-li přenosové zpoždění hradla T a sčítačky $2T$, má **nejdelší cesta přenosové zpoždění $16T$ (nebo 8Δ , $\Delta = 2T$)**.

Toto zpoždění se s rostoucí délkou operandů dále zvětšuje. Je proto snaha nalézt uspořádání násobičky s rychlejší funkcí. Nejdříve se však zaměřme na násobení čísel se znaménkem.

Násobení čísel se znaménkem v doplňkovém kódu

$$P = (-y_{M-1} 2^{M-1} + \sum_{j=0}^{M-2} y_j 2^j) \cdot (-x_{N-1} 2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i)$$

Příklad pro $M=N=6$

$$\begin{aligned}
 P = & \left[\sum_{i=0}^{N-2} \sum_{j=0}^{M-2} x_i y_j 2^{i+j} \right] \\
 & + \\
 & x_{N-1} y_{M-1} 2^{M+N-2} \\
 & + \\
 & - \sum_{i=0}^{N-2} x_i y_{M-1} 2^{i+M-1} \\
 & + \\
 & - \sum_{j=0}^{M-2} x_{N-1} y_j 2^{j+N-1}
 \end{aligned}$$

y_5	y_4	y_3	y_2	y_1	y_0
x_5	x_4	x_3	x_2	x_1	x_0
	$x_0 y_4$	$x_0 y_3$	$x_0 y_2$	$x_0 y_1$	$x_0 y_0$
	$x_1 y_4$	$x_1 y_3$	$x_1 y_2$	$x_1 y_1$	$x_1 y_0$
	$x_2 y_4$	$x_2 y_3$	$x_2 y_2$	$x_2 y_1$	$x_2 y_0$
	$x_3 y_4$	$x_3 y_3$	$x_3 y_2$	$x_3 y_1$	$x_3 y_0$
	$x_4 y_4$	$x_4 y_3$	$x_4 y_2$	$x_4 y_1$	$x_4 y_0$

Záporný člen:
negace, přičtení 1

		$x_5 y_5$										
1	1	$\overline{x_4 y_5}$	$\overline{x_3 y_5}$	$\overline{x_2 y_5}$	$\overline{x_1 y_5}$	$\overline{x_0 y_5}$	1	1	1	1	1	1
1	1	$\overline{x_5 y_4}$	$\overline{x_5 y_3}$	$\overline{x_5 y_2}$	$\overline{x_5 y_1}$	$\overline{x_5 y_0}$	1	1	1	1	1	1

Výsledek je na 12 bitech, obecně na $M+N$ bitech.

Př. Násobení čísel se znaménkem - prakticky

(-4) x (-14) na 5 bitech v doplňkovém kódu. Protože je výsledek na 10 bitech, musíme důsledně rozširovat znaménko na 10 bitů – týká se záporných čísel.

$$\begin{array}{r} S \qquad S \\ 1111111100 \times 1111110010 \\ \hline & 00000 \\ & 1111111100 \\ & 00000 \\ & 00000 \\ & 1111111100 \\ \hline & 1111111100 \\ & 1111111100 \\ & 1111111100 \\ & 1111111100 \\ & 1111111100 \\ \hline & 0000111000 \\ & S \end{array}$$

Násobení čísel se znaménkem - praxe

Co s tím? Důsledná práce se znaménkem:

- Výsledek násobení dvou čísel na n bitech bude na $2n$ bitech
- Tedy i vstupní operandy musí být správně zakódovány na $2n$ bitech
- Princip šíření hodnoty znaménkového bitu doleva - z toho vyplývají dva problémy:
 - **U dílčích součinů se objeví levostranné jedničky**
 - **Objeví se další dílčí součiny**
- Počet částečných součinů a šíření znaménkového bitu (1) lze redukovat pomocí Boothova algoritmu.
- Počet úrovní nutných pro sečtení částečných součinů lze redukovat pomocí Wallaceova stromu.

Princip Boothova překódování

Překódováním násobitele potenciálně velmi usnadníme násobení, protože zredukujeme počet částečných součinů a zbavíme se levostanných jedniček.

$$\begin{array}{r} \mathbf{A*31} & \mathbf{A} & = & \mathbf{A*32-A} & \mathbf{A} & \text{násobenec} \\ \hline & 11111 & & & 10000-1 & \text{násobitel} \\ & A & & & -A & \\ & A & & A & & \\ & A & & & & \\ & A & & & & \\ \hline & 4 \text{ součty} & & & 1 \text{ součet} & \end{array}$$

Konvenční přístup

Boothova metoda

Boothovo překódování s radixem 2

Základem Boothova algoritmu je překódování násobitele do soustavy **relativních číslic**. Máme-li násobit číslem 31, tedy 00011111, dostaneme stejný výsledek, vynásobíme-li číslem (32 - 1), což zapíšeme v soustavě relativních číslic, která používá číslice 0, +1, -1, jako:

$$\begin{array}{r} 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1 \\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ -1 \end{array}$$

Tento princip aplikujeme na dvojkové číslo opakovaně pro všechny skupiny jedniček, přičemž za skupinu jedniček považujeme i jedinou jedničku.

Příklad:

$$\begin{array}{r} 0\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1 \\ 1\ 0\ -1\ 0\ 0\ 1\ -1\ 1\ 0\ -1\ 1\ 0\ 0\ 0\ -1\ 1\ 0\ 0\ 0\ -1 \end{array}$$

Odtud můžeme sestavit překódovací tabulku. Kromě překódovaného bitu se řídíme ještě hodnotou **sousedního bitu vpravo**. Dostáváme tak základní Boothovo překódování, zvané též překódování s radixem 2.

Boothovo překódování s radixem 2

Překódovaná číslice	Sousední bit vpravo	Boothův kód
0	0	0
0	1	1
1	0	-1
1	1	0

Všimneme si překódování záporného násobitele, např. -6 na 5 bitech včetně znaménka. Toto číslo 11010 se překóduje na 0-11-10.

Rozšíříme-li zobrazení čísla -6 na 10 bitů, tedy na 1111111010, pak v Boothově překódování se to projeví přidáním pěti levostranných nul, tedy dostaneme 00000-11-10.

Potom vznikají **nulové částečné součiny**, které usnadňují výsledné sčítání.

Příklad

13 x (-6) na 5 bitech vč. znaménka. Překódování násobitele ponecháme pouze na 5 bitech, protože levostranné nuly vyvolají vznik nulových a tedy bezvýznamných dílčích součinů.

13: 01101 6: 0 0 1 1 0

-13: 10011 -6: 1 1 0 1 0

Překódování násobitele 0-1 1-1 0

0 x 13 0000000000

-1 x 13 111110011

1 x 13 00001101

-1 x 13 1110011

0 x 13 000000

1110110010

S

-0001001110 tj. -78

Boothovo překódování s radixem 4

Je tedy odstraněn nepříznivý efekt záporného násobitele, zůstává však nutnost šíření znaménka u záporných dílčích součinů. Dalším požadavkem pro urychlení násobení je snížení počtu dílčích součinů. Oba tyto požadavky se řeší dalšími modifikacemi Boothova násobení.

Z jednoduchého Boothova překódování je odvozeno překódování "2 bity najednou", neboli *Boothovo překódování s radixem 4*.

1	1	0	1	0	1	0	1	původní číslo rozdělené do dvojic
2	1	2	1	2	1	2	1	váhy
0	-1	1	-1	0	-1	0	1	Boothovo překódování po jednom bitu
-1	1	2	-2	1				překódování "2 bity najednou"

Boothovo překódování – obecný počet bitů

Boothovo překódování lze definovat pro skupiny bitů libovolné velikosti. Obecný postup je takový: V každé skupině zopakujeme nejvyšší bit, a přičteme k nejnižšímu bitu nejvyšší bit ze skupiny vpravo. Výsledné číslo pak považujeme za relativní číslici v doplňkovém kódu.

Příklad pro radix 4 (tj. 2 bity najednou):

00	00	11	01	11	10	01	00	10	10		původní číslo
000	000	111	001	111	110	001	000	110	110		rozšíření znam. bitu
0	1	0	1	1	0	0	1	1	0		přičtení bitu zprava
<hr/>											
000	001	111	010	000	110	001	001	111	110		doplňkový. kód rel. číslice
0	+1	-1	2	0	-2	+1	1	-1	-2		překódování s radixem 4

Z tohoto příkladu můžeme odvodit tabulku pro Boothovo překódování s radixem 4.

Boothovo překódování s radixem 4

Překódovaná skupina	Bit vpravo	Relativní číslice
00	0	0
00	1	+1
01	0	+1
01	1	+2
10	0	-2
10	1	-1
11	0	-1
11	1	0

Příklad – Boothovo překódování s radixem 4

$13 \times (-6)$:

$$\begin{array}{r} 13 \\ -13 \end{array} \quad \begin{array}{r} 01101 \\ 10011 \end{array} \quad \begin{array}{r} 6: 00110 \\ -6: 11010 \end{array}$$

Boothovo překódování

$$\begin{array}{r} 0-11-10 \\ 0-1 -2 \end{array}$$

potřebujeme sudý počet bitů,
rozšíříme znaménko na 111010
po jednom bitu
po dvou bitech

S

$$\begin{array}{l} -2x13 \\ -1x13 \\ 0x13 \end{array} \quad \begin{array}{r} 1111100110 \\ 11110011 \\ 000000 \end{array} \quad \begin{array}{l} \text{tj. } -13 \text{ posunutá vlevo o 1 bit} \\ \text{s krokem 2 bity} \end{array}$$

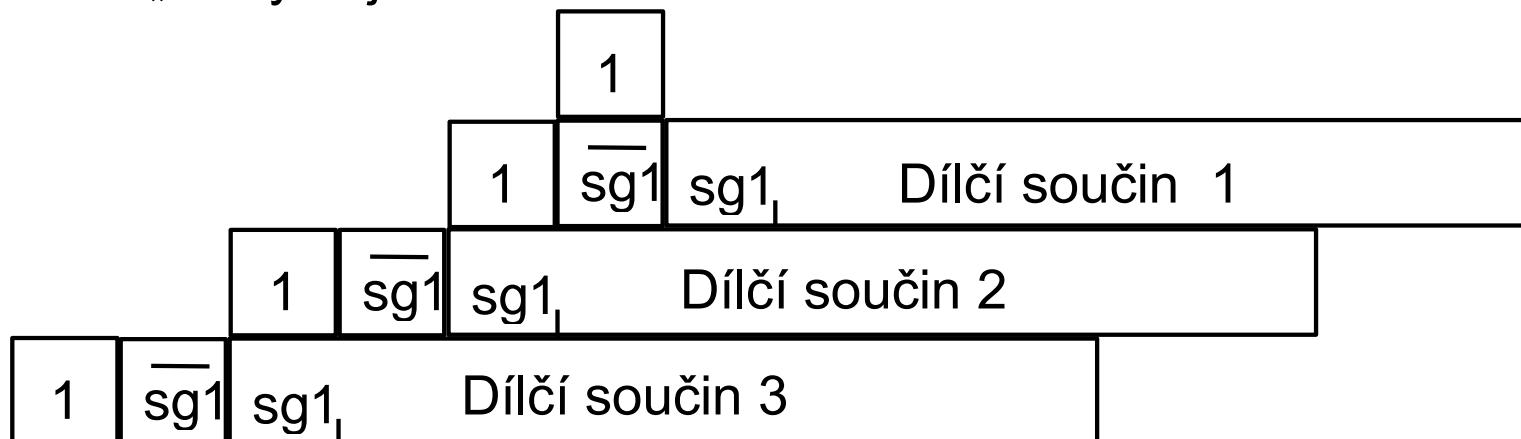
1110110010 **-78**

S

Komentář k příkladu $13 \times (-6)$

V prvním dílčím součinu se nám posunulo znaménko o jeden bit doleva. Tomu musíme přizpůsobit všechny ostatní dílčí součiny, tedy zapisovat je na 6 bitech. Znaménko výsledku očekáváme ovšem v 10. bitu.

Šíření znaménka vlevo odstraňuje metoda vroubení. Místo šíření znaménka se ke každému dílčímu součinu připíše zleva negace znaménkového bitu a jednička, a nad znaménkový bit prvního dílčího součinu se napíše též jednička. Pozor, tento postup funguje pouze pro metodu „2 bity najednou“!

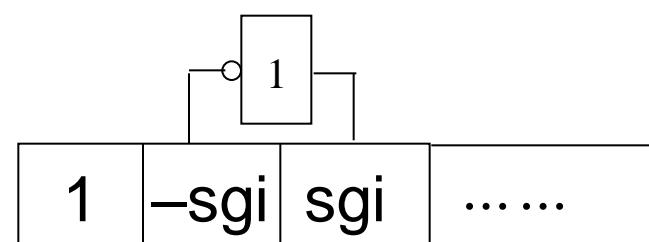


Př. $13 \times (-6)$ s vroubením znaménka, 2 bity najednou

$$\begin{array}{r} 001101 \\ 0 -1 -2 \\ \hline \end{array}$$

$$\begin{array}{r} 1 \\ 10100110 \\ 10110011 \\ 11000000 \\ \hline 111110110010 \\ S \end{array}$$

Obvodová realizace vroubení znaménka:



Boothovo překódování s radixem 8 (po 3 bitech)

Tabulka odvozena stejným způsobem jako pro radix 4.

Používá 9 relativních číslic.

Překódovat	Rel. číslice
000 0	0
000 1	+1
001 0	+1
001 1	+2
010 0	+2
010 1	+3
011 0	+3
011 1	+4
100 0	-4
100 1	-3
101 0	-3
101 1	-2
110 0	-2
110 1	-1
111 0	-1
111 1	0

Jak urychlit kombinační násobičku?

Urychlit sečtení částečných součinů zavedením „uchování přenosů“

Příklad: Sečtěte $6 + 7 + 12 + 8$

$$\begin{array}{r} 110 \quad (6) \\ 111 \quad (7) \\ \hline 1101 \quad (13) \\ + 1100 \quad (12) \\ \hline 11001 \quad (25) \\ + 01000 \quad (8) \\ \hline 100001 \quad (33) \end{array}$$

Konvenční sčítání:

Sečtou se první dva operandy, potom se k průběžnému výsledku přičítají další operandy. V rámci sčítání dvou operandů se použije sčítáčka s postupným přenosem složená z úplných sčítáček.

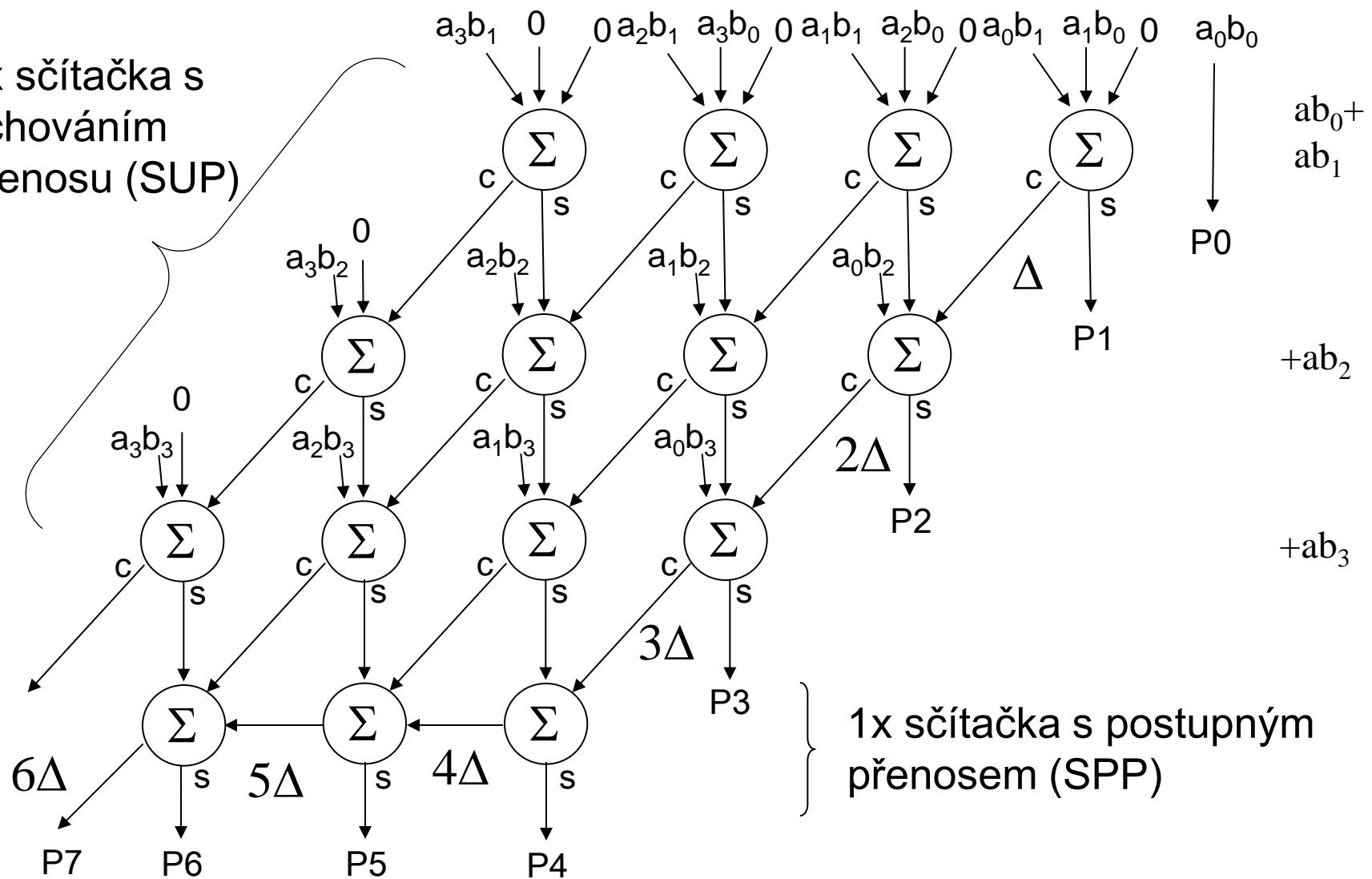
$$\begin{array}{r} 110 \quad (6) \\ 111 \quad (7) \\ \hline 001 \quad \text{součet bez přenosů (S1)} \\ 110 \quad \text{uchování přenosu (C1)} \\ + 1100 \quad (12) \\ \hline 0001 \quad S1+C1+12 \text{ bez přenosů (S2)} \\ 1100 \quad \text{uchování přenosu (C2)} \\ + 01000 \quad (8) \\ \hline 100001 \quad \text{součet S2+C2+8 s přenosem} \end{array}$$

Sčítání s uchováním přenosu:

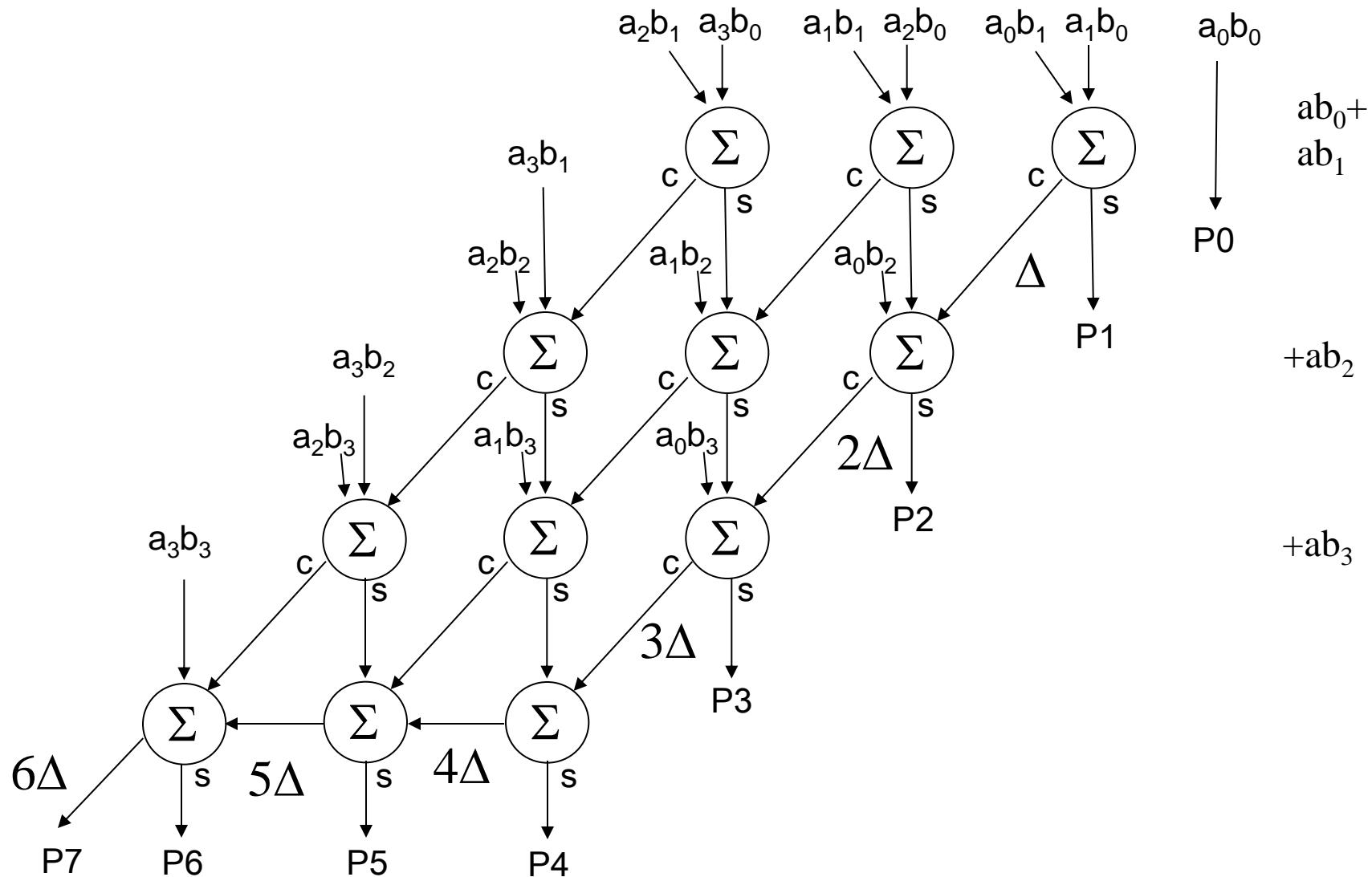
Sčítá se bez uvážení přenosů (tj. rychle). Avšak přenosy se uchovávají v registru a přičítají (pomocí úplných sčítáček) k průběžnému součtu a dalšímu operandu v následujícím kroku (opět bez přenosu). Přičtení posledního operandu se provede pomocí sčítáčky s postupným přenosem.

4b kombinační násobička s uchováním přenosu

3x sčítačka s
uchováním
přenosu (SUP)



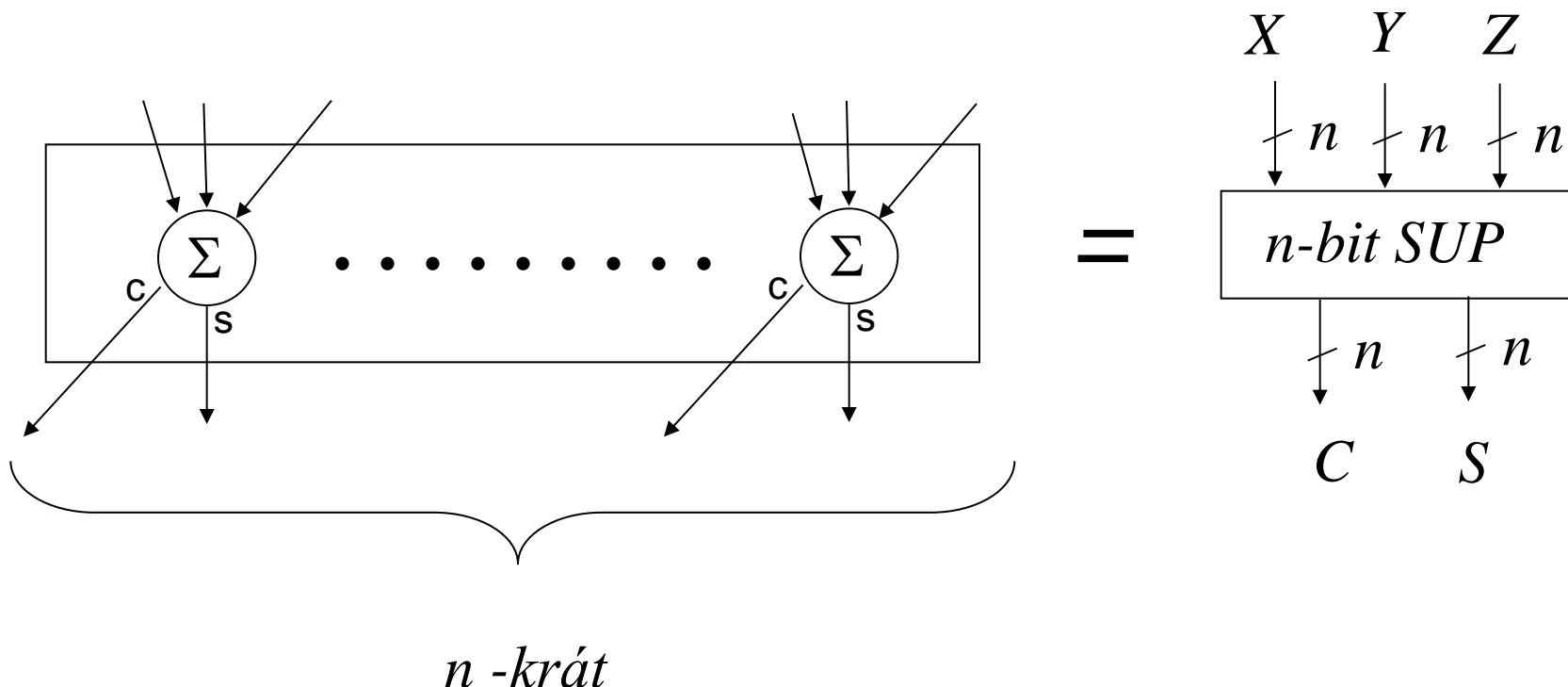
Optimalizovaná 4b kombinační násobička s uchováním přenosu



Komentář k násobičce s uchováním přenosů

Ze tří řad sčítacího stromu jsme odstranili sčítačky s postupným přenosem. Sčítačky v jedné řadě se označují jako **sčítačka s uchováním přenosu** (SUP, angl. CSA - Carry-Save Adder). V každém stupni se sčítačka v nejvyšším řádu redukovala na hradlo. Museli jsme však nakonec jednu řadu sčítaček s postupným přenosem (SPP) přidat. Nicméně toto uspořádání činnost násobičky obecně zrychluje.

Základem n-bitové násobičky s uchováním přenosu je n-bitová SUP:



Zobecněné využití sčítáčky

Sčítáčka s uchováním přenosu dovoluje chápat sčítání poněkud odlišným způsobem. Víme, že jednobitová sčítáčka má tři vstupy a dva výstupy. Vzhledem k tomu, že vstupy pro přenos všech jednobitových sčítáček v prvním stupni jsou nevyužité, může se na ně přivést třetí dílčí součin.

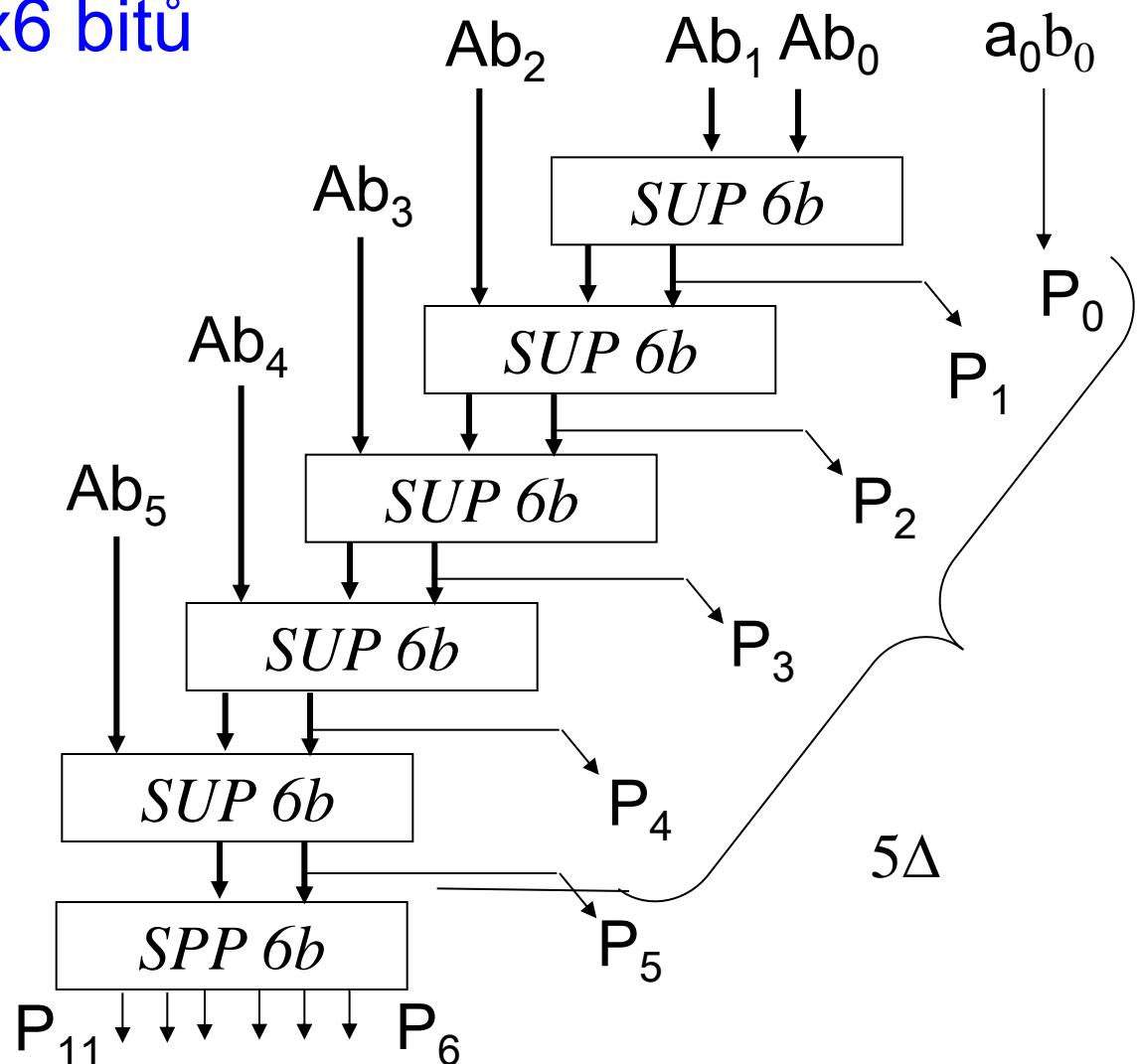
Na vstupech má tedy sčítáčka tři binární vektory délky n , ovšem vzájemně posunuté do správné polohy. To se projeví tak, že v každém stupni je plný počet sčítáček (n). Součet tří dílčích součinů je dán výstupním vektorem součtu a výstupním vektorem přenosů. Poloha vektoru součtu je vzhledem ke konečnému výsledku správná, vektor přenosů se posouvá o jeden bit do vyššího řádu.

Takováto zobecněná **optimalizovaná sčítáčka** je známá též pod označením **pseudosčítáčka**, protože nedává ještě definitivní součet vstupních vektorů.

Blokové schéma násobičky s uchováním přenosu 6x6 bitů

$$A = a_5 a_4 a_3 a_2 a_1 a_0$$

$$B = b_5 b_4 b_3 b_2 b_1 b_0$$

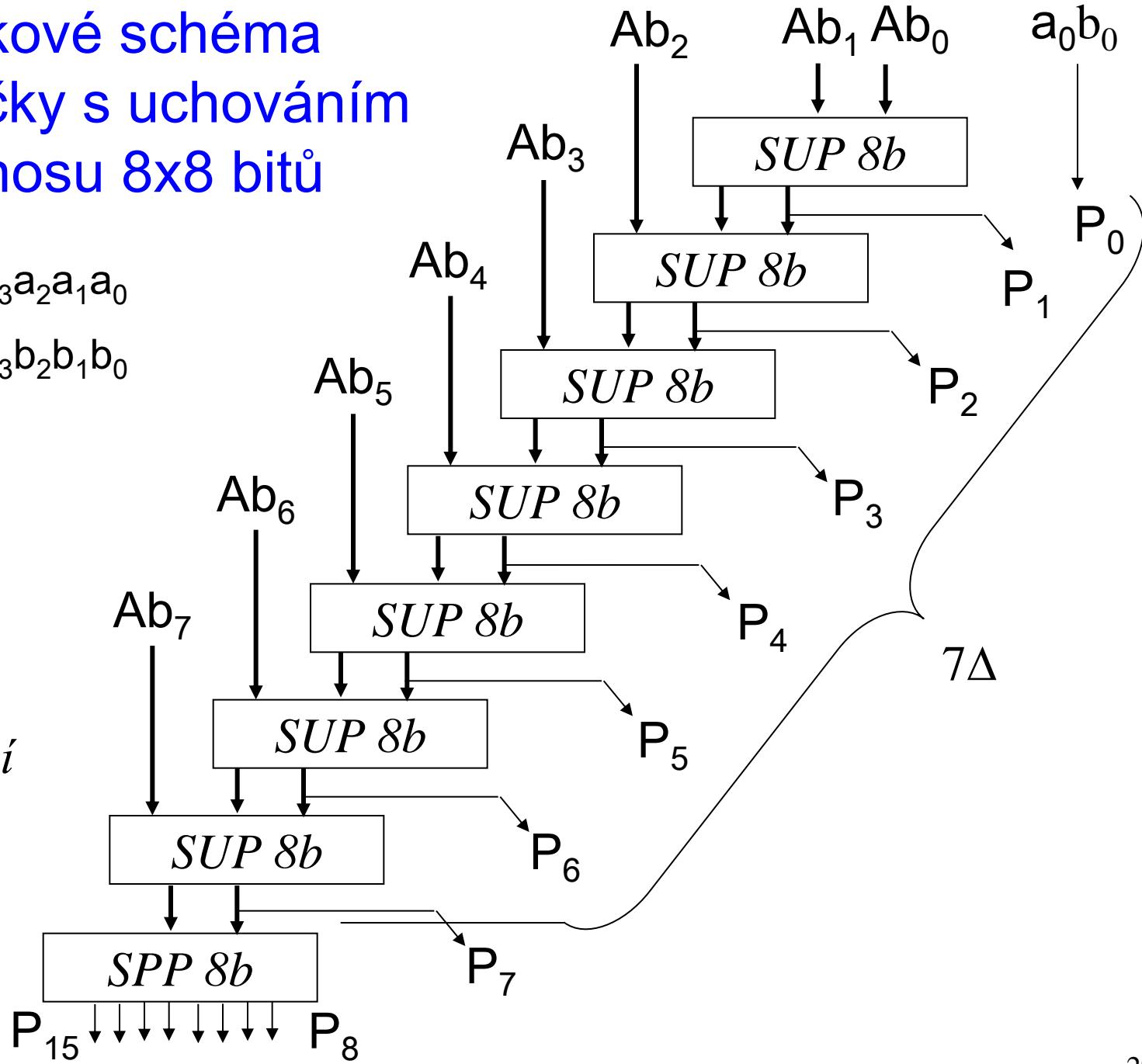


Blokové schéma násobičky s uchováním přenosu 8x8 bitů

$$A = a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0$$

$$B = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

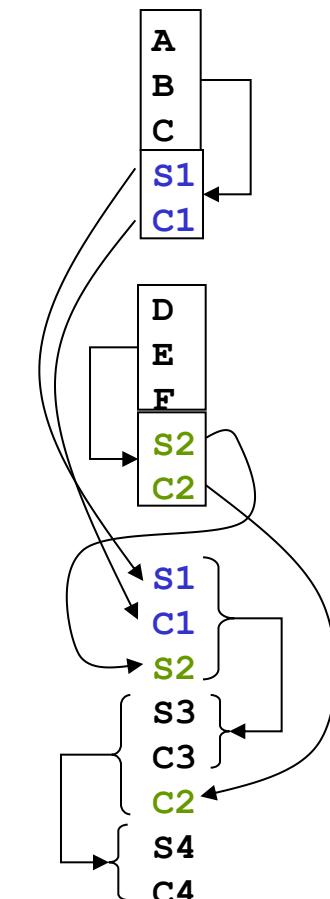
Nebylo by možné zredukovat počet sčítání a tím i zpoždění?



Zrychlené sčítání částečných součinů (A, B, C, D, E, F) při násobení s uchováním přenosu M x Q (6b x 6b)

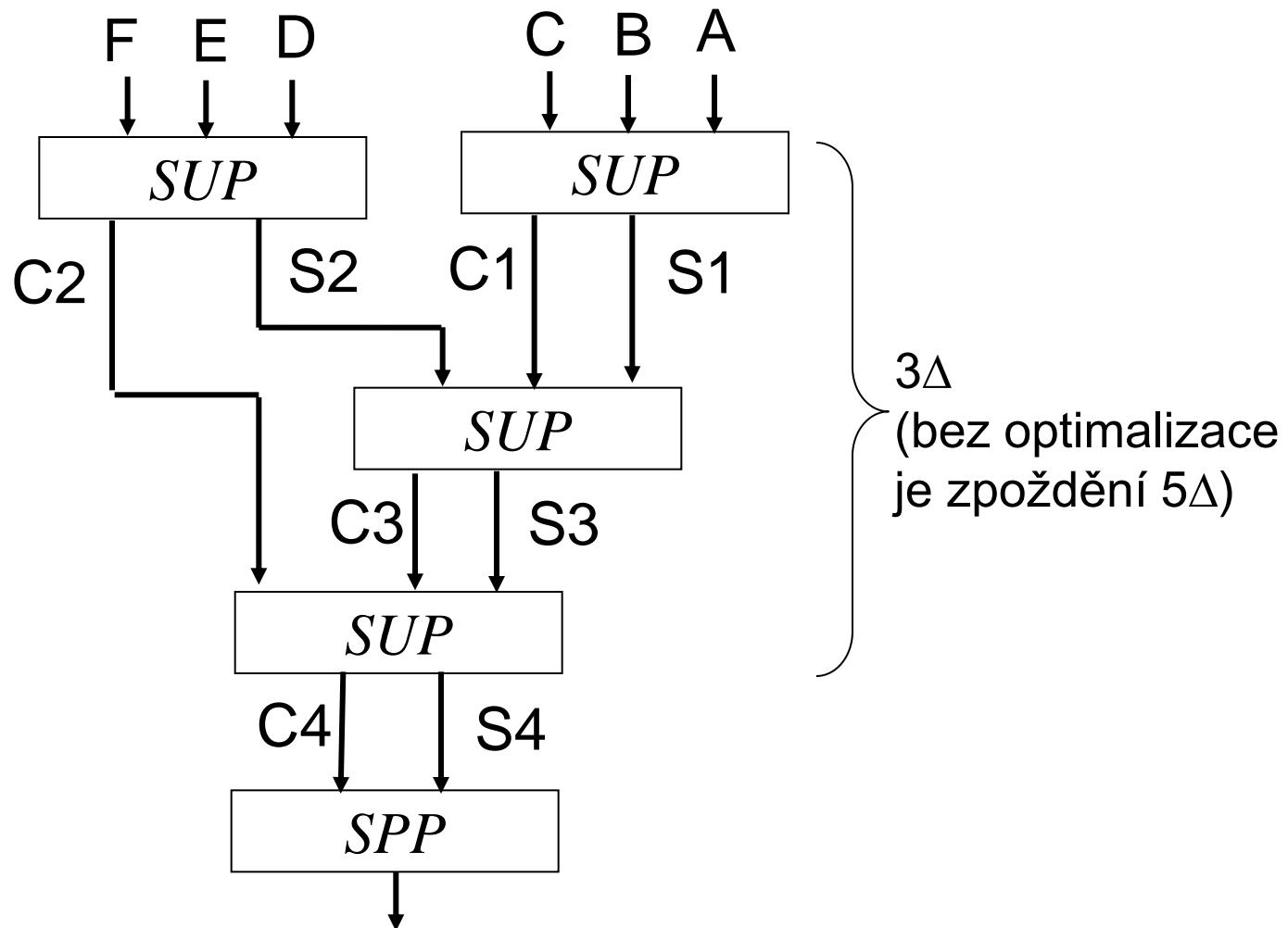
Nejdříve jsou sečteny částečné součiny A, B a C, výsledek je v S1 a C1. Potom jsou sečteny částečné součiny D, E a F, výsledek je v S2 a C2. Následuje součet S1, C1 a S2, výsledek je v S3 a C3. Nakonec jsou sečteny S3, C3 a C2, výsledek je v S4 a C4. Sčítáčkou s postupným přenosem jsou nakonec sečteny S4 a C4. Kromě posledního sčítání jsou všechna ostatní sčítání s uchováním přenosu.

$$\begin{array}{r}
 & 1 & 0 & 1 & 1 & 0 & 1 & M \\
 \times & 1 & 1 & 1 & 1 & 1 & 1 & Q \\
 \hline
 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & \underline{1} & 0 & 1 & 1 & 0 & 1 \\
 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
 \hline
 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & \underline{1} & 0 & 1 & 1 & 0 & 1 \\
 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
 \hline
 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
 & \underline{1} & 1 & 0 & 0 & 0 & 1 & 1 \\
 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
 & \underline{0} & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
 & + & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
 \hline
 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1
 \end{array}$$



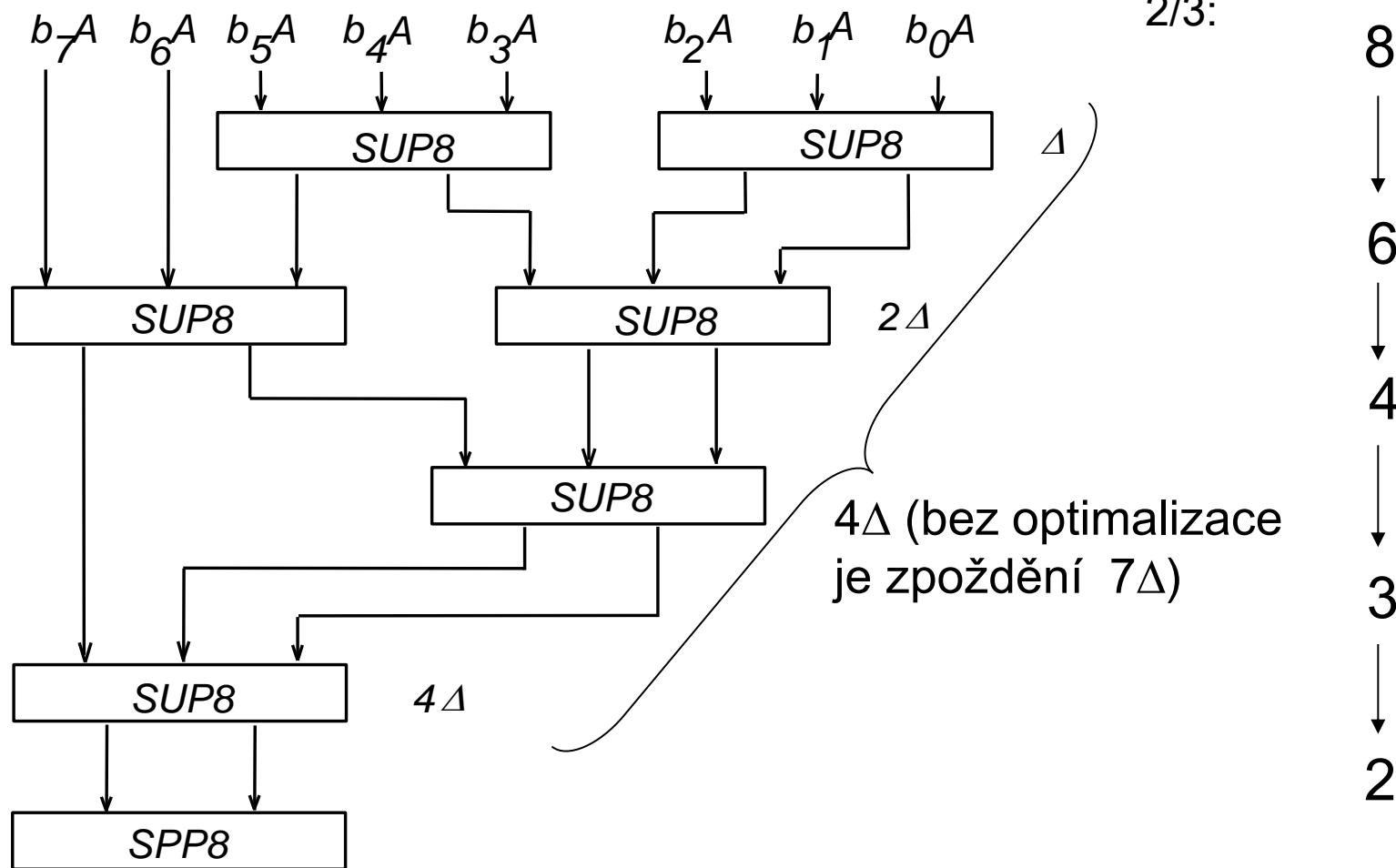
Součin

Zrychlené sčítání částečných součinů při násobení s uchováním přenosu 6×6 bitů – Wallaceův strom

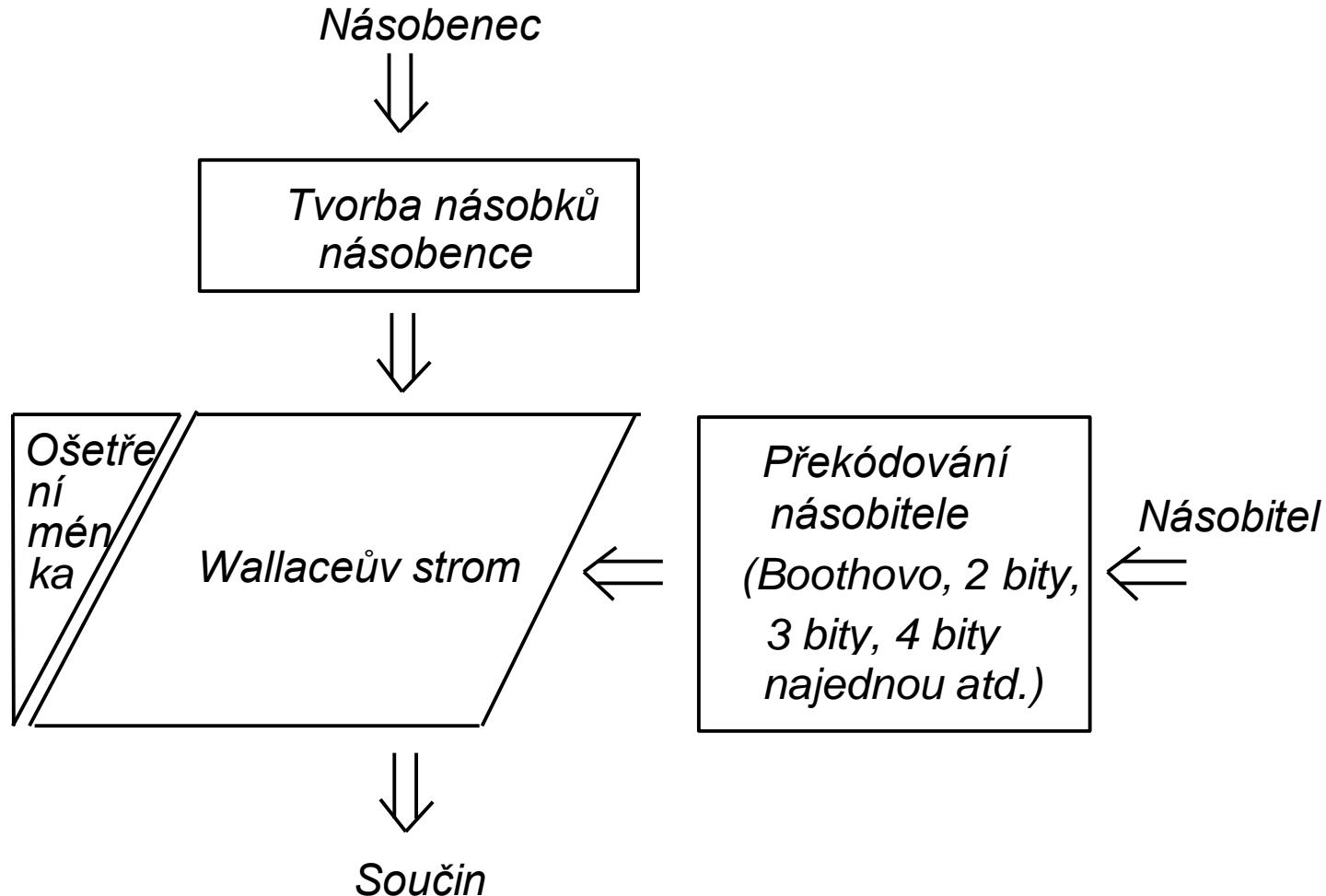


Wallaceův strom pro násobičku 8x8 bitů

V každém stupni (SUP) je redukován počet operandů na 2/3:



Shrnutí: Kombinační násobička čísel se znaménkem

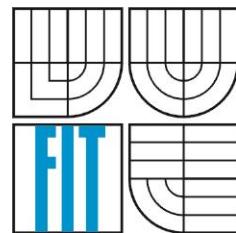


Použitá literatura

- Drábek, V.: Výstavba počítačů. Skriptum VUT, 1995
- Weste, N. H. E., Harris, D.: CMOS VLSI Design, 3. vydání, Addison Wesley, 2005
- Hamacher, C., Vranesic, Z., Zaky, S.: Computer Organization. McGraw-Hill, 2001

Sekvenční dělení

INP 2015
FIT VUT v Brně



Dělení čísel s pevnou řádovou čárkou

Budeme zabývat dělením čísel s pevnou řádovou čárkou **bez znaménka**.
Pro jednotlivé činitele operace dělení zavedeme symboly

D	dělenec
d	dělitel
Q	podíl
q_i	i -tý bit podílu
R_i	i -tý (průběžný) zbytek

Máme vypočítat Q , R tak, aby byla splněna rovnice

$$D = Q \cdot d + R, \quad 0 \leq |R| < d.$$

Pro d , Q , R máme k dispozici n bitů, pro D vyhradíme $2n$ bitů.

Nejdříve si ukážeme dělení čísel bez znamének, resp. dělení jejich absolutních hodnot.

Nejdříve posuneme d o n bitů doleva.

Příklad 2293 : 51 (dekadicky vs binárně, n = 6)

$$\underline{2293} : 51 = 0$$

-00

$$\underline{229}_3 : 51 = 04$$

-204

$$\underline{0253} : 51 = 044$$

- 204

49 (zbytek)

V kroku i se pokoušíme odečíst od průběžného zbytku R_i posunutý dělitel $2^{n-i} d$

$$\begin{array}{r} \underline{100011}110101 \\ -000000 \end{array} : \begin{array}{r} 110011000000 \\ = 0 \end{array} \quad (i = 0)$$

$$\begin{array}{r} \underline{1000111}10101 \\ -110011 \end{array} : \begin{array}{r} 1100110000 \\ = 01 \end{array}$$

$$\begin{array}{r} \underline{0101001}0101 \\ -000000 \end{array} : \begin{array}{r} 1100110000 \\ = 010 \end{array}$$

$$\begin{array}{r} \underline{01010010}101 \\ -110011 \end{array} : \begin{array}{r} 110011000 \\ = 0101 \end{array}$$

$$\begin{array}{r} \underline{111111}01 \\ -110011 \end{array} : \begin{array}{r} 11001100 \\ = 01011 \end{array}$$

$$\begin{array}{r} \underline{00110001} \\ -000000 \end{array} : \begin{array}{r} 1100110 \\ = 010110 \end{array}$$

$$\begin{array}{r} \underline{00110001} \\ -000000 \end{array} : \begin{array}{r} 110011 \\ = 0101100 \quad (44) \end{array}$$

110001 (49, zbytek)

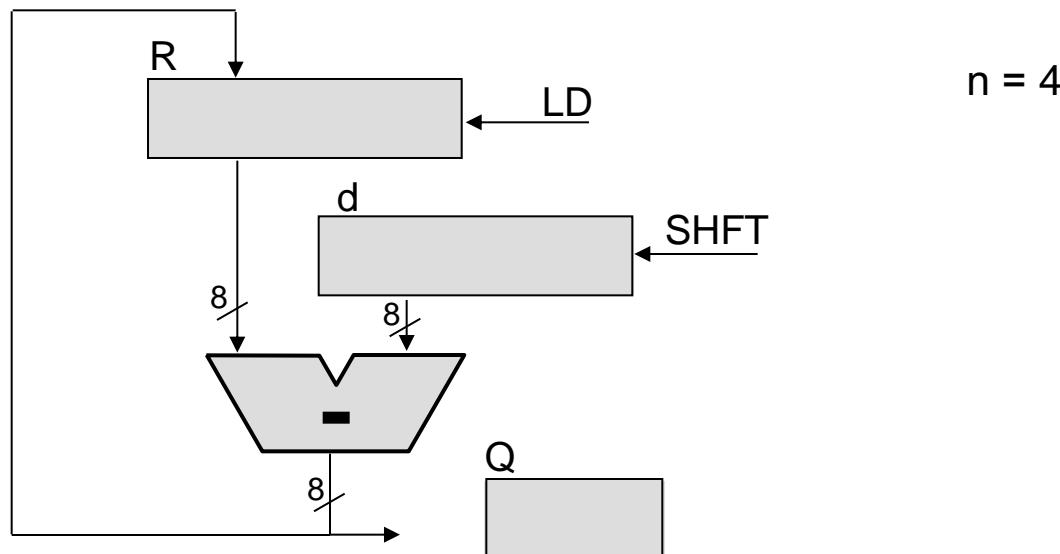
Postup dělení a HW realizace

Při rozhodování o hodnotě bitu podílu q_{n-i} jsme postupovali podle vztahů:
je-li $2^{n-i} d$ menší než nebo rovno R_i , pak $q_{n-i} = 1$,
je-li $2^{n-i} d$ větší než R_i , pak $q_{n-i} = 0$.

$$Q = q_n \dots q_0$$

Nový zbytek se vypočte:

$$R_{i+1} := R_i - q_{n-i} 2^{n-i} d$$



Modifikovaný postup – d je v pevné poloze

HW realizace z předchozího obrázku vyžaduje uchovávat hodnotu dělitele na $2n$ bitech a používat $2n$ bitovou odčítačku / sčítacíku => zbytečně drahé.

V praxi se posouvá dílčí/průběžný zbytek vlevo a d je v pevné poloze, takže

$$R_{i+1} := \mathbf{2}R_i - q_{n-i} d$$

Dělení – modifikovaný postup (38 : 5)

(Praxe: posuv R_i vlevo, d v pevné poloze)

$$\underline{D=100110} \quad \underline{d=101}$$

$$Q = q_3 q_2 q_1 q_0$$

$$= \textcolor{red}{0111}$$

$$R_{i+1} = 2R_i - q_{n-i}d$$

$$\begin{array}{r} i=0 & 100110 \\ & -000 \\ \hline & 100110 \end{array}$$

$$\begin{array}{l} 2R_0 = D \\ q_3 d \\ R_1 \end{array}$$

$$\Rightarrow Q = \textcolor{red}{0}$$

$$\begin{array}{r} i=1 & 100110x \\ & \underline{-101000} \\ \hline & 10010x \end{array}$$

$$\begin{array}{l} 2R_1 \\ q_2 d \\ R_2 \end{array}$$

$$\Rightarrow Q = \textcolor{red}{01}$$

$$\begin{array}{r} i=2 & 10010xx \\ & \underline{-101000} \\ \hline & 1000xx \end{array}$$

$$\begin{array}{l} 2R_2 \\ q_1 d \\ R_3 \end{array}$$

$$\Rightarrow Q = \textcolor{red}{011}$$

$$\begin{array}{r} i=3 & 1000xxx \\ & \underline{-101000} \\ \hline & 011xxx \end{array}$$

$$\begin{array}{l} 2R_3 \\ q_0 d \\ R_4 = 2^4 R \Rightarrow R = 2^{-4} R_4 \end{array}$$

Pravidlo pro určení q_{n-i}

Pravidlo pro určení q_{n-i} :

když d je větší než $2R_i$, pak $q_{n-i} = 0$

jinak $q_{n-i} = 1.$

V praxi se porovnávání čísel založené na použití komparátorů nepoužívá. Odečtení se provede vždy (zkusmo), tedy

když $2R_i - d$ je menší než 0, pak $q_{n-i} = 0$

jinak $q_{n-i} = 1.$

Dva postupy dělení

a) Je-li $q_{n-i} = 0$, pak je výsledek "zkušebního" odečtení $R'_{i+1} = 2R_i - d$. Správný zbytek má ale být $R_{i+1} = 2R_i$. Správný zbytek dostaneme opravou, přičtením $+d$, což nazýváme **restaurací nezáporného zbytku (návrat k nezápornému zbytku)**, tedy $R_{i+1} := R'_{i+1} + d$. Je-li pravděpodobnost výskytu jedničky v podílu Q rovna 1/2, potřebujeme pro n odečtení v průměru ještě $n/2$ krát přičítat.

b) Postup **bez restaurace nezáporného zbytku (bez návratu k nezápornému zbytku)**:

Restaurací provádíme operaci $R_i := R'_i + d$. V dalším kroku po odečtení d dostaneme

$$R_{i+1} := 2R_i - d \quad (1)$$

nebo po přičtení d

$$R_{i+1} := 2R_i + 2d - d = 2R_i + d \quad (2)$$

Vidíme, že můžeme postupovat podle rovnic (1), (2):

když $q_{n-i} = 1$ (R_i větší než 0), použijeme vztah (1),

když $q_{n-i} = 0$, použijeme vztah (2).

Postup je úspornější, protože v každém kroku pak jen přičítáme d , nebo odčítáme d , nikdy neprovádíme obě operace. Při dělení bez restaurace tedy provedeme n aritmetických operací (+ nebo -), pro dělení s restaurací $3/2 n$ operací.

$$30=00011110, \ 7=0111, \ -7=1001$$

00011110

+1001 -d
10101110 <0 => c4 = 0

0101110x posuv <-

+0111 +d
1100110x <0 => c3 = 0

100110xx posuv

+0111 +d
000010xx >0 => c2 = 1

00010xxx posuv

+1001 -d
10100xxx <0 => c1 = 0

0100xxxx posuv

+0111 +d
1011xxxx <0 => c0 = 0

+0111 +d (korekce na kladný zbytek)
0010xxxx zbytek 2

Příklad: $30:7=4$,
zb 2
bez návratu k
nezápornému zbytku

Dělení SRT

Dělení čísel se znaménkem se po dlouhou dobu převádělo na dělení absolutních hodnot a dodatečné určení znaménka výsledku. Tuto nedokonalost odstranil **algoritmus SRT** autorů **Sweeneyho, Robertsona a Tochera** (1958). Prováděné operace se pro každý krok určují "zejména" podle nejvyšších bitů průběžného zbytku R_i .

Pro demonstraci principu uvedeme základní metodu, kdy se operace provádí **podle 3 bitů R_i** .

R_i	d>0 bit podílu	d>0 operace	d<0 bit podílu	d<0 operace
000 111	0	posuv vlevo	0	posuv vlevo
001 010 011	1	-d, posuv vlevo	-1	+d, posuv vlevo
101 110 100	-1	+d, posuv vlevo	1	-d, posuv vlevo

$$49=00110001, \quad 7=0111, \quad -7=1001$$

$$-49=11001111 \quad (d>0)$$

-1 **11001111**

$$\begin{array}{r} 0111 \\ \hline +d \end{array}$$

00111111 <-

+1 **0111111x**

$$\begin{array}{r} 1001 \\ \hline -d \end{array}$$

0000111x

0 **000111xx** <-

+1 **00111xxx** <-

$$\begin{array}{r} 1001 \\ \hline -d \end{array}$$

11001xxxx <-

-1 **1001xxxx**

$$\begin{array}{r} 0111 \\ \hline +d \end{array}$$

0000xxxx

zbytek 0

Uplatníme váhy $2^n \dots 2^0$

$Q = -1 \ 1 \ 0 \ 1 \ -1$, tj.

$$-16+8+0+2-1 = -7$$

Příklad:

-49:7=-7, zb. 0
SRT

Ri	d>0 bit podílu	d>0 operace	d<0 bit podílu	d<0 operace
000 111	0	posuv vlevo	0	posuv vlevo
001 010 011	1	-d, posuv vlevo	-1	+d, posuv vlevo
101 110 100	-1	+d, posuv vlevo	1	-d, posuv vlevo

Pozn. V případě záporného zbytku je zapotřebí provést korekci na kladný zbytek a **zvýšit** (pro d<0), popř. **snížit** (pro d>0) Q o 1.

Reálný algoritmus dělení SRT

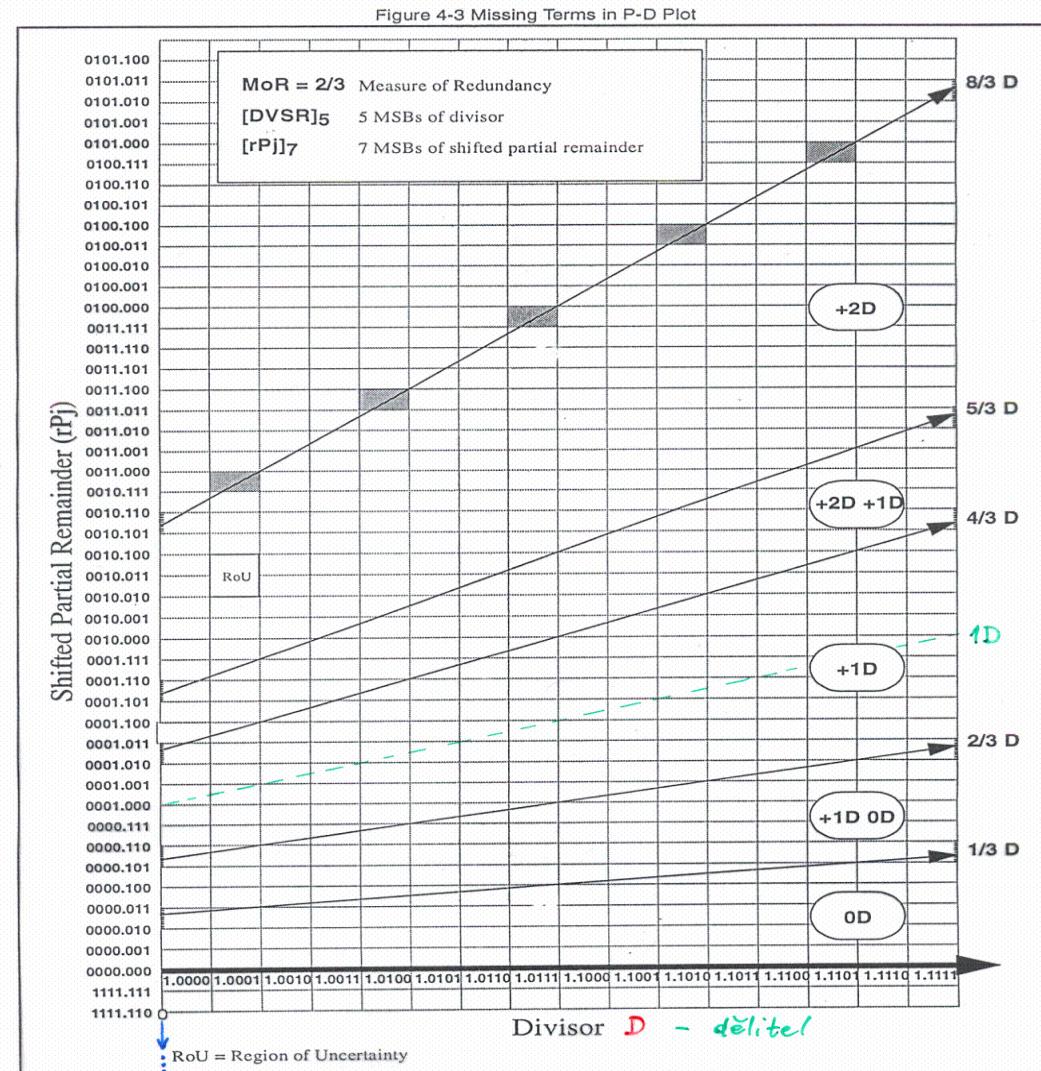
Praktické realizace postupu SRT určují hodnotu číslice podílu podle více bitů průběžného (okamžitého) zbytku a podle více bitů dělitele.

Dále pracují s kódováním „několik bitů najednou“.

Například v Pentiu se určují číslice podílu **podle 7 bitů průběžného zbytku a podle 5 bitů hodnoty dělitele**. Používá se radix 4.

Chyba dělení u Pentia (listopad 1994, ztráta \$475M)

index má $7+5=12$ bitů
 (7-bitový odhad podle průběžného zbytku + ---)

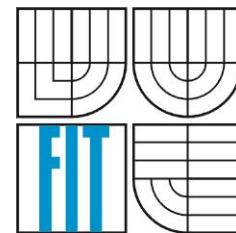


Obvodové realizace dělení: Shrnutí

- Sekvenční děličky
 - viz předchozí slidy
- Kombinační dělička
 - založena na úplné odčítáčce
 - obvodová struktura je podobná kombinační násobičce
- Iterační dělička
 - viz další přednáška

Operace v FP a iterační algoritmy

INP 2015
FIT VUT v Brně



Př. Latence instrukcí (mikroarchitektura Intel Haswell, 2013)

Instrukce	Latence
MOV r,r	1
MOV m,r	3
ADD r,r	1
MUL r32	4
DIV r32	22 – 29
ROR, ROL	1
FADD	3
FMUL	5
FDIV	10 – 24
FSQRT	10 – 23
FSIN	47 – 106
FPTAN	130

Jsou uvedeny minimální hodnoty latence jako počet hod. taktů jádra.
www.agner.org/optimize/instruction_tables.pdf

Operace FP

- Číslo X s pohyblivou řádovou čárkou $X = M_X \cdot B^{Ex}$ zapíšeme jako dvojici (M_X, E_X) , kde **mantisa** M_X je ve dvojkovém doplňkovém kódu, nebo v přímém kódu se znaménkem na n_M bitech, **exponent** E_X je v kódu s posunutím na n_E bitech. Třetím definičním údajem je hodnota **základu B**.
- Základní aritmetické operace pro dvojici čísel X, Y s plovoucí čárkou jsou:

$$X + Y = (M_X \cdot 2^{Ex - Ey} + M_Y) \cdot 2^{Ey}, \text{ kde } E_X \leq E_Y$$

$$X - Y = (M_X \cdot 2^{Ex - Ey} - M_Y) \cdot 2^{Ey}, \text{ kde } E_X \leq E_Y$$

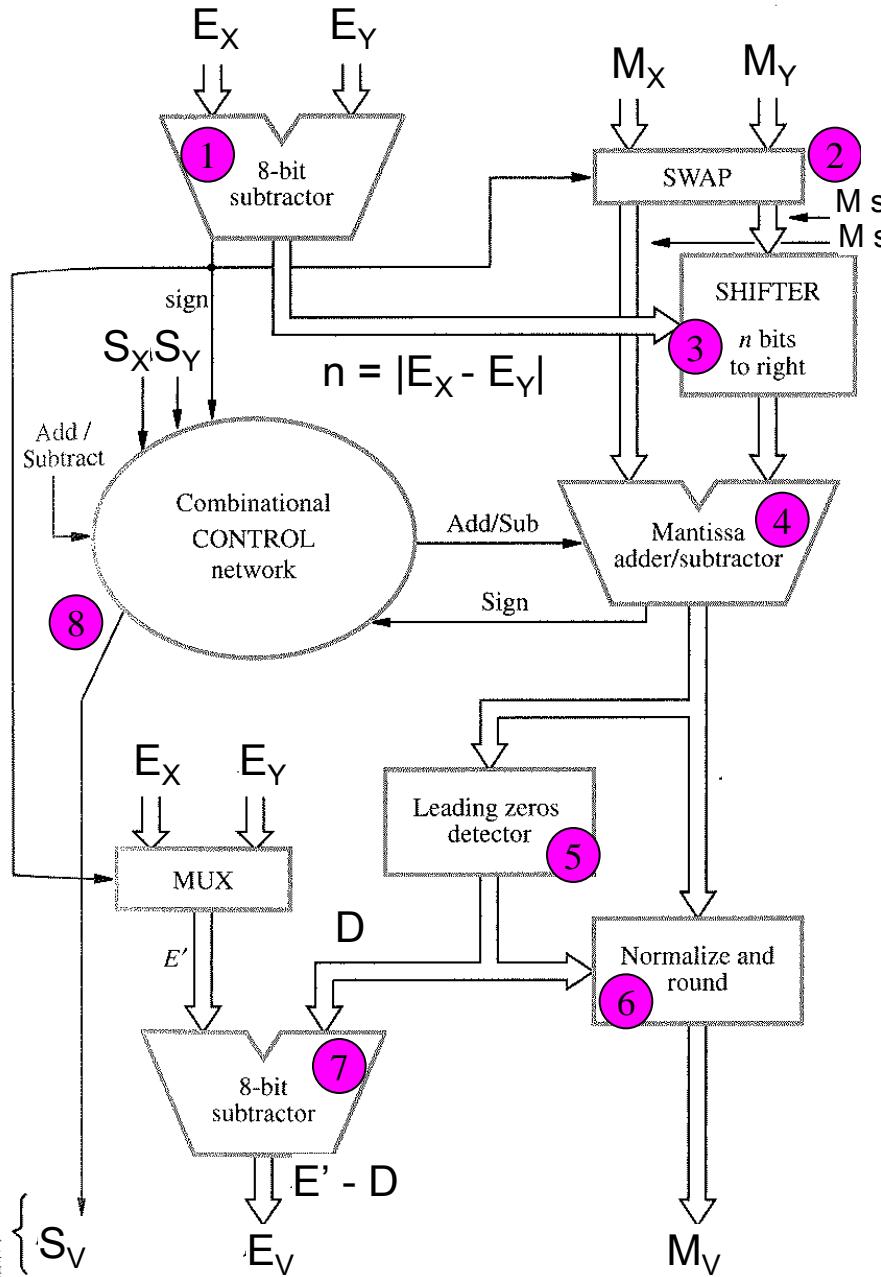
$$X * Y = (M_X \cdot M_Y) \cdot 2^{Ex + Ey}$$

$$X : Y = (M_X : M_Y) \cdot 2^{Ex - Ey}$$

Požadované operace

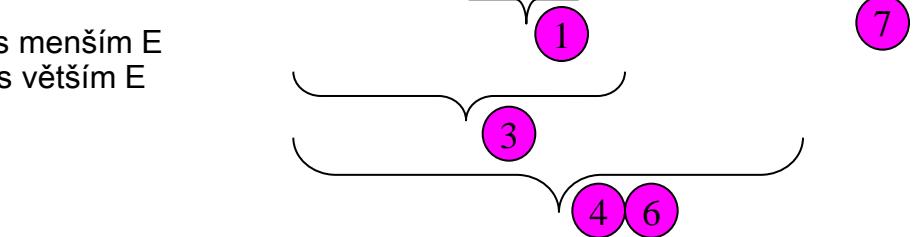
- Pro sčítání a odčítání:
 - 1. Vypočte se v pevné čárce rozdíl $E_X - E_Y$
 - 2. Posune se M_X o $E_X - E_Y$ bitů (tj. doprava, pokud je $E_X \leq E_Y$)
 - 3. Vypočte se v pevné čárce $M_X \cdot 2^{E_X - E_Y} +/- M_Y$
- Dále je zapotřebí provést **normalizaci a zaokrouhlení výsledku**. Nechť $V = (M_V, E_V)$ označuje výsledek. Normalizovat výsledek znamená posouvat mantisu vlevo (vpravo) a podle toho zmenšovat (zvětšovat) exponent E_V tak dlouho, až se do sledovaného bitu (v_0 nebo v_1) dostane 1.
- **Sčítání exponentů** se provádí v binární sčítáčce s korekcí, nebo ve speciální sčítáčce pro posunutý kód.
- **Sčítání mantis** se provádí v binární sčítáčce se šírkou $n_M + 2$ bitů s doplněným záhytným klopným obvodem S .
- V případě násobení (dělení) v FP se využije pro násobení (dělení) mantis dedikovaná násobička (dělička) mantis pracující v FX.

Princip obvodové realizace sčítacíky/odčítacíky FP



Pokud $E_X \leq E_Y$, potom

$$X +/ - Y = (M_X \cdot 2^{E_X - E_Y} +/ - M_Y \cdot 2^{E_Y})$$



Obecný postup (nevíme, zda $E_X \leq E_Y$):

- 1 Rozdíl exponentů, nastavení sign a n
- 2 Prohození mantis podle sign
- 3 Posuv mantisy o n bitů vpravo
- 4 Sečtení/odečtení mantis podle Add/Sub
- 5 Zjištění počtu nul D v horních bitech mantisy
- 6 Normalizace mantisy, detekce spec. případů
- 7 Úprava exponentu podle D a sign
- 8 Logika pro výpočet znaménka

Obvodová realizace operací v FP

- Pro operace s pohyblivou čárkou prováděné v rámci uvedených algoritmů by bylo možno teoreticky použít sčítáčku a násobičku ALU s pevnou čárkou. V praxi se to však takto neděje. Obvody aritmetiky s pohyblivou čárkou jsou konstruovány jako zcela **nezávislá** jednotka s vlastním řadičem.
- Dělení čísel s pohyblivou čárkou a celou řadu dalších operací (sinus, kosinus atd.) je možné v HW provádět **iteračními algoritmy**.

Newtonův iterační algoritmus

Na základě odhadu x_i hledáme přesnější odhad x_{i+1} v bodě průsečíku tečny funkce f s osou x . Rovnice přímky procházející bodem $f(x_i)$ je

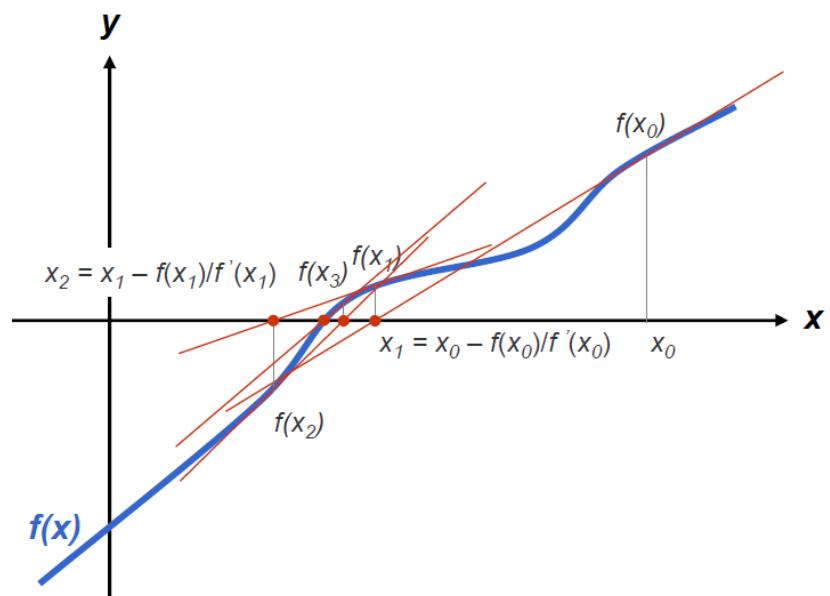
$$y - f(x_i) = f'(x_i)(x - x_i).$$

Pokládáme-li přímku za approximaci funkce $f(x)$, můžeme psát

$$f(x_{i+1}) - f(x_i) = f'(x_i)(x_{i+1} - x_i).$$

V průsečíku tečny s osou x je $f(x_{i+1}) = 0$, takže odtud dostáváme iterační vzorec

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$



Newtonův algoritmus - dělení

Iterační vzorec

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

použijeme pro případ dělení. Máme-li dělit číslem b , zvolíme $f(x) = 1/x - b$, a hledáme bod průchodu této funkce osou x , tedy bod x , kde $f(x) = 0$, takže pak platí $1/x = b$, resp. $x = 1/b$. Operaci dělení číslem b nahradíme násobením číslem $1/b$.

Derivace

$$f'(x) = -1/x^2, \text{ odtud}$$

$$\begin{aligned} x_{i+1} &= x_i - (1/x_i - b)/(-1/x_i^2) = \\ &= x_i + x_i - bx_i^2 = x_i(2 - bx_i) \end{aligned}$$

Rychlosť konvergencie závisí na volbě x_0 a je typicky kvadratická.

Newtonův algoritmus dělení - prakticky

Postup výpočtu a / b je následující:

1. Posune se b tak, aby padlo do intervalu $<1, 2)$ a pomocí tabulky odhadů zvolíme první odhad x_0 .
2. Provedeme krok iteračního výpočtu $x_{i+1} = x_i (2 - bx_i)$. Krok 2 opakujeme tak dlouho, až se dosáhne požadované přesnosti na p bitů, kdy je relativní chyba $(x_i - 1/b)/(1/b) = 2^{-p}$. V následujícím kroku $i+1$ je relativní chyba

$$(x_{i+1} - 1/b)/(1/b) = 2^{-2p}$$

3. Výsledek n -té iterace x_n vynásobíme číslem a , výsledek $x_n \cdot a$ posuneme o odpovídající počet bitů podle kroku 1.

Newtonův algoritmus dělení - příklad

Spočtěte binárně $1/b$ pro $b = 20$. $(20)_{10} = (10100)_2$ $x_{i+1} = x_i(2 - bx_i)$

- řádovou čárku posuneme tak, aby $b \in (1, 2)$, tedy:

$$10100 \rightarrow 1,0100 \text{ (o 4 bity doleva)}$$

- zvolíme např. $x_0 = 1$, potom:

$$x_1 = x_0(2 - bx_0) = 1.(10 - 1,01) = 1.0,11 = 0,11$$

$$x_2 = 0,11(10 - 1,01.0,11) = 0,11(10 - 0,1111) = 0,11.1,0001 = 0,110011$$

$$x_3 = 0,110011(10 - 1,01.0,110011) = 0,110011(10 - 0,11111111) =$$

$$= 0,110011.1,00000001 = 0,11001100110011$$

atd.

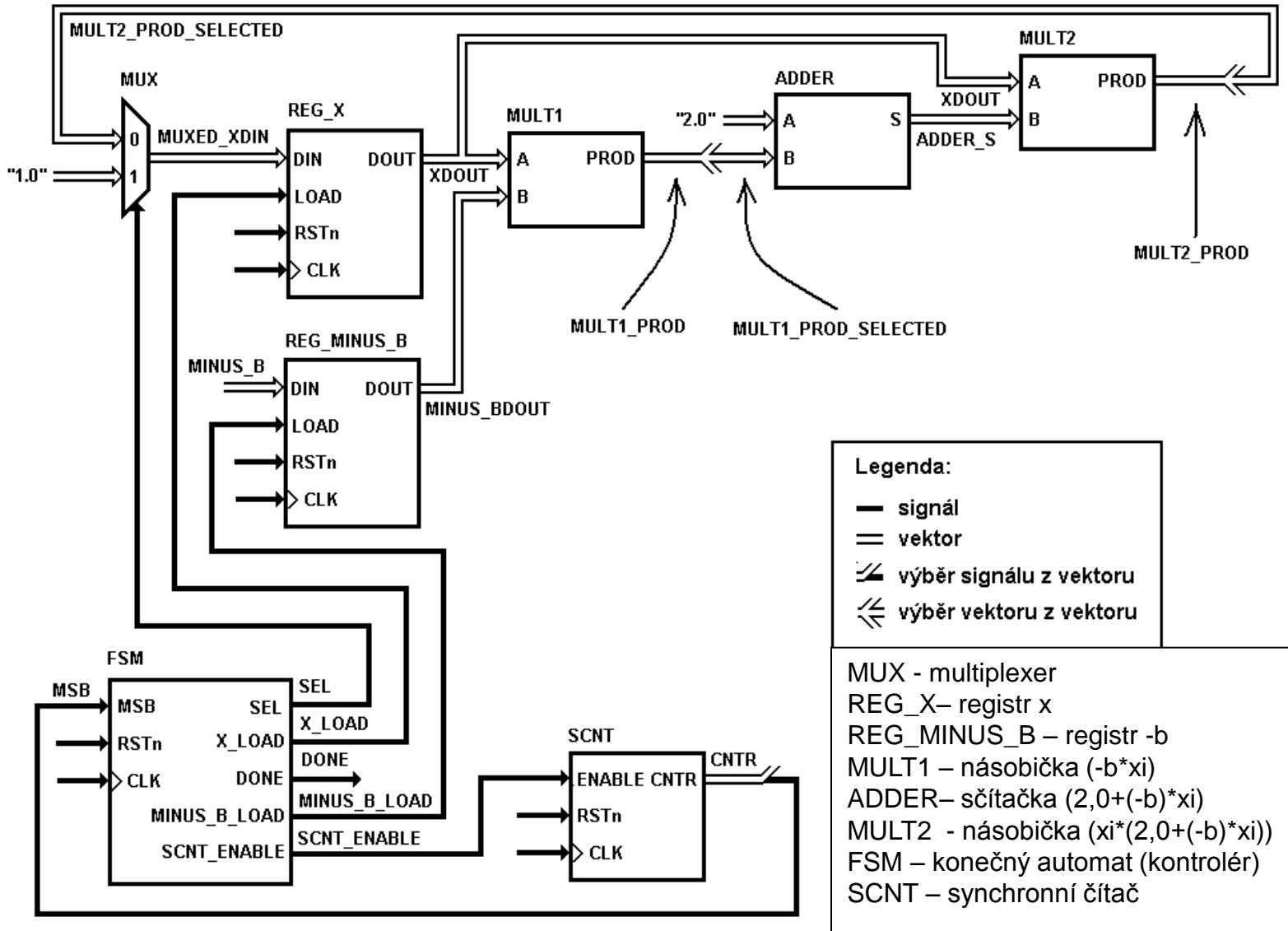
- řádovou čárku posuneme o 4 bity doleva: 0,000011001100110011

Ověření správnosti: $1/20 = 0,05$ a

$$(0,000011001100110011)_2 = (0,051952362)_{10}$$

Newtonův algoritmus dělení v HW

$$x_{i+1} = x_i (2 - bx_i)$$



Další funkce pomocí iteračních algoritmů

Zde platí, že postupy vhodné pro programovou implementaci, jako MacLaurinův rozvoj nebo Čebyševovy polynomy atd., nemusí být pro obvodovou realizaci iteračních výpočtů výhodné, a proto byla odvozena řada modifikovaných nebo nových algoritmů, např. Goldschmidtův algoritmus apod.

Všeobecná zásada je najít co nejrychlejší algoritmy, založené pouze na operacích sčítání (odčítání), posuvů a případně i násobení.

CORDIC

Algoritmus CORDIC (Coordinate Rotational Digital Computer) publikoval J. E. Volder v r. 1959. Následně byl zobecněn pro další typy výpočtů.

Myšlenka: Využitím jednoho algoritmu můžeme počítat řadu matematických funkcí pouhým vyčíslováním funkce ve tvaru

$$a \pm b \cdot 2^{-i}$$

tedy pomocí součtů, rozdílů a bitových posunů (tzn. rychlá a levná HW implementace).

Použití: CORDIC je používán typicky ve vestavěných zařízeních s jednoduchým procesorem (kapesní kalkulačky, jednočipové mikrokontrolery, apod.), čímž umožňuje efektivně počítat mnoho funkcí. Používá se také např. v koprocessorech Intel počínaje I 8087, pro číslicovou Fourierovu transformaci, číslicovou filtraci signálů apod.

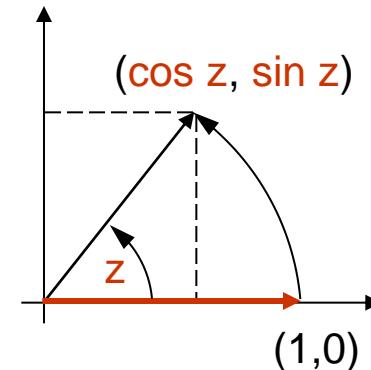
CORDIC – rotační režim

Algoritmus lze provozovat ve dvou režimech – rotačním a vektorovém. Každý režim umožňuje vypočítat jiné funkce.

Rotační režim

Princip: rotací vektoru $(1,0)$ o úhel z získáváme hodnotu $\cos z$ a $\sin z$.

Rotaci provádíme postupným přičítáním nebo odčítáním vhodně zvolených úhlů (postupně se zmenšující hodnoty) tak, aby získaná hodnota konvergovala k požadované hodnotě z .



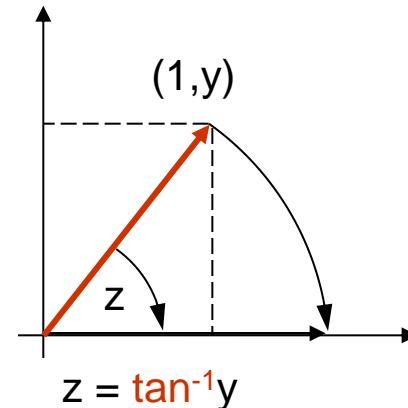
1. počátek v $(1,0)$,
2. rotace tak, aby úhel byl z ,
3. $x = \cos z$ a $y = \sin z$

CORDIC – vektorový režim

Algoritmus lze provozovat ve dvou režimech – rotačním a vektorovém. Každý režim umožňuje vypočítat jiné funkce.

Vektorový režim

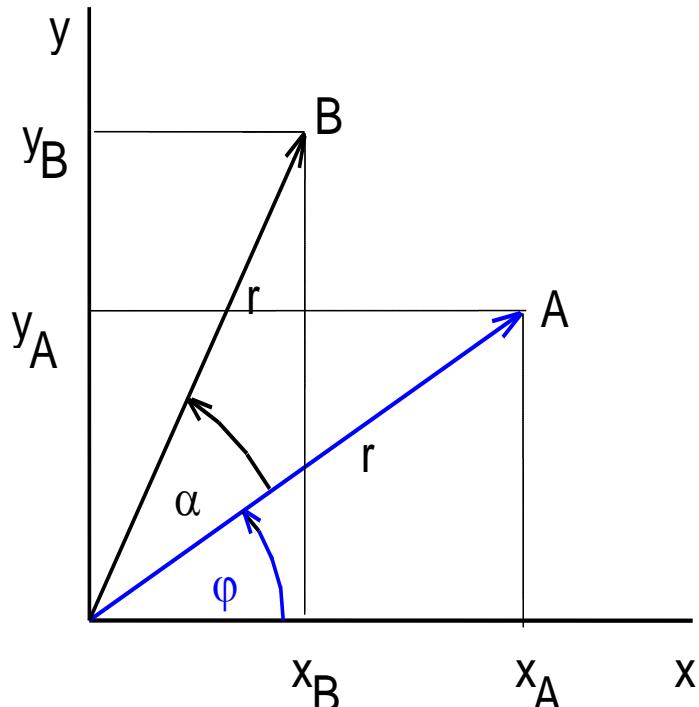
Princip: rotací vektoru $(1,y)$ o úhel z tak, aby výsledný vektor byl rovnoběžný s osou X, získáváme hodnotu $\arctan y$.



Nulování hodnoty y provádíme postupným přičítáním nebo odčítáním vhodně zvolených hodnot tak, aby získaná hodnota postupně konvergovala k 0.

1. počátek v $(1,y)$,
2. rotace o z tak, aby y bylo 0,
3. z odpovídá $\tan^{-1} y$

Rotace vektoru



Přechod od bodu A k bodu B lze obecně vyjádřit jako:

$$x_B = r \cdot \cos(\varphi + \alpha) = \\ r \cdot \cos \varphi \cdot \cos \alpha - r \cdot \sin \varphi \cdot \sin \alpha \quad (1)$$

$$y_B = r \cdot \sin(\varphi + \alpha) = \\ r \cdot \sin \varphi \cdot \cos \alpha + r \cdot \cos \varphi \cdot \sin \alpha \quad (2)$$

$$x_A = r \cdot \cos \varphi$$

$$y_A = r \cdot \sin \varphi$$

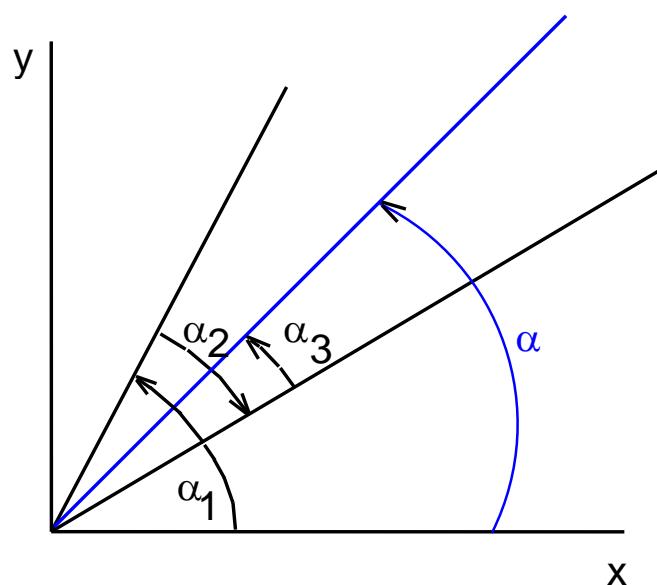
Dosadíme-li za r do (1) $r = x_A / \cos \varphi$, $r = y_A / \sin \varphi$ pořadě, a do (2) v opačném pořadí, dostaneme výrazy

$$x_B = \cos \alpha (x_A - y_A \cdot \tan \alpha)$$

$$y_B = \cos \alpha (y_A + x_A \cdot \tan \alpha)$$

Rotace vektoru

Požadovaný úhel natočení α můžeme složit z n úhlů s kladným i záporným znaménkem (viz obrázek), tedy z orientovaných úhlů α'_i .



Příklad:
rotace vektoru orientovaného do
osy x o úhel α dosažená pomocí
tří iterací s postupně se snižujícími
absolutními hodnotami úhlů α_i

CORDIC – rotace vektoru

Výsledný úhel natočení $\alpha = \alpha_0' + \alpha_1' + \dots + \alpha_{n-1}'$

Postup iteračního výpočtu je pak následující:

Položíme $x_0 = x_A$, $y_0 = y_A$ a určíme x_{i+1}, y_{i+1} postupně pro $i = 0, 1, \dots, n-1$.

Dostáváme pak iterační vzorce

$$x_{i+1} = \cos \alpha_i' \cdot (x_i - y_i \cdot \tan \alpha_i') \quad (3)$$

$$y_{i+1} = \cos \alpha_i' \cdot (y_i + x_i \cdot \tan \alpha_i') \quad (4)$$

Problém: uvedený vztah vyžaduje implementovat obecné násobičky:

- každý krok obsahuje násobení odlišným koeficientem $\tan \alpha_i'$
- každý krok obsahuje násobení odlišným koeficientem $\cos \alpha_i'$

Řešení:

- vhodná volba úhlů α_i' tak, aby hodnota $\tan \alpha_i'$ byla mocninou dvou
- zavedení pevného počtu iterací a odložení násobení koeficienty $\cos \alpha_i'$

CORDIC – rotace vektoru

Vhodná volba úhlů α'_i

Ve dvojkové soustavě volíme takové úhly α'_i , že platí $\operatorname{tg} \alpha'_i = \pm 2^{-i}$.

Tato volba nemá vliv na konvergenci, neboť je-li $\operatorname{tg} \alpha'_i$ klesající posloupnost, pak také α'_i je klesající posloupnost (viz tabulka dále).

Tím se v rovnicích (3), (4) nahradí násobení **posuvem o i bitů** (vpravo).

Odložení násobení

Další zjednodušení provedeme odložením násobení hodnotami $\cos \alpha'_i$ nakonec výpočtu, kdy výsledek vynásobíme hodnotou (agregační konstanta)

$$K_{n-1} = \cos \alpha_0 \cdot \cos \alpha_1 \dots \cos \alpha_{n-1} = 0,60725\dots$$

kde $\alpha_i = |\alpha'_i|$, protože platí $\cos \alpha'_i = \cos |\alpha'_i|$.

Hodnota aggregační konstanty závisí pouze na počtu kroků, který je fixní a je volen na základě požadované přesnosti.

Tabulka úhlů CORDIC

Pro $i = 0, 1, \dots$ sestavíme tabulku hodnot α_i , pro něž platí $\alpha_i = \arctg 2^{-i}$

i	$\alpha [^\circ]$	$\operatorname{tg} \alpha$	$\cos \alpha$	K
0	45,000	1	0,707107	0,707107
1	26,565	0,5	0,894427	0,632456
2	14,036	0,25	0,970143	0,613572
3	7,125	0,125	0,992278	0,608834
4	3,576	0,0625	0,998053	0,607648
5	1,790	0,03125	0,999512	0,607352
6	0,895	0,015625	0,999878	0,607278
7	0,448	0,0078125	0,999969	0,607259
...

Agregační konstanta K konverguje se zvětšujícím se počtem iterací k hodnotě 0,60725293.

CORDIC - finální podoba

Dosazením dostaneme konečný tvar iteračních vzorců

$$x_{i+1} = x_i - (+) y_i 2^{-i}$$

$$y_{i+1} = y_i + (-) x_i 2^{-i}$$

$$z_{i+1} = z_i + (-) \alpha_i$$

Rotační režim: vyjdeme-li z vektoru $x_0 = 1$, $y_0 = 0$, přičemž $z_0 = \alpha$, pak aplikací iteračních vzorců tak, aby $z_n \rightarrow 0$ (tzn. natočili jsme vektor o úhel α), dostáváme po n iteracích hodnoty

$$x_n = \cos \alpha / K_{n-1}, y_n = \sin \alpha / K_{n-1}.$$

Abychom získali požadované hodnoty kosinu a sinu, musíme obě složky násobit agregační konstantou K_{n-1} .

Optimalizace: Položíme-li $x_0 = K_{n-1}$, $y_0 = 0$, vyhneme se závěrečnému násobení a výsledek je přímo roven

$$x_n = \cos \alpha, y_n = \sin \alpha.$$

Poznámka: Pro úhly větší než 90° využíváme symetrie a opakování grafu goniometrických funkcí.

Výpočet sin a cos pomocí CORDIC v jazyce Python

```
def cordic(alpha, n=16): #úhel alpha, počet iterací

    alphatab = [math.atan(2**(-i)) for i in range(n)] Tabulka úhlů

    K = reduce(lambda x,y: x*y, [math.cos(a) for a in alphatab])
    x, y, z = K, 0.0, alpha Výpočet aggregační konstanty

    for i in range(n):
        if z < 0:
            xn = x + y * 2**(-i)
            yn = y - x * 2**(-i)
            z += alphatab[i]
        else:
            xn = x - y * 2**(-i)
            yn = y + x * 2**(-i)
            z -= alphatab[i]
        x, y = xn, yn

    return (x, y) # cos(alpha), sin(alpha)
```

Na základě hodnoty znaménka přičítáme nebo odčítáme úhel z tabulky úhlů.

Po n iteracích získáváme v x a y hodnoty $\cos(\alpha)$ a $\sin(\alpha)$

Př. S využitím sedmi iterací algoritmu CORDIC vypočtěte
 $\sin(28.027^\circ)$ a $\cos(28.027^\circ)$

Rotujeme vektor $(1,0)$ o úhel 28.027° , průběžný úhel označme z.

$$X_0 = 1 \quad Y_0 = 0 \quad z_0 = 28.027^\circ$$

Odečti úhel $\alpha_0 = 45^\circ \rightarrow z_1 = z_0 - \alpha_0 = 28.027 - 45 = -16.973^\circ$

Průběžný úhel je kladný, úhel α_0 odečítáme

$$X_1 = X_0 - Y_0 / 1 = 1 - 0 / 1 = 1$$

$$Y_1 = Y_0 + X_0 / 1 = 0 + 1 / 1 = 1$$

Přičti úhel $\alpha_1 = 26.565^\circ \rightarrow z_2 = z_1 + \alpha_1 = -16.973 + 26.565 = 9.592^\circ$

Průběžný úhel byl záporný, přičítáme a měníme znaménka

$$X_2 = X_1 + Y_1 / 2 = 1 + 1 / 2 = 1.5$$

$$Y_2 = Y_1 - X_1 / 2 = 1 - 1 / 2 = 0.5$$

Odečti úhel $\alpha_2 = 14.036^\circ \rightarrow z_3 = z_2 - \alpha_2 = 9.592 - 14.036 = -4.444^\circ$

$$X_3 = X_2 - Y_2 / 4 = 1.5 - 0.5 / 4 = 1.375$$

$$Y_3 = Y_2 + X_2 / 4 = 0.5 + 1.5 / 4 = 0.875$$

Přičti úhel $\alpha_3 = 7.125^\circ \rightarrow z_4 = z_3 + \alpha_3 = -4.444 + 7.125 = 2.680^\circ$

$$X_4 = X_3 + Y_3 / 8 = 1.375 + 0.875 / 8 = 1.484375$$

$$Y_4 = Y_3 - X_3 / 8 = 0.875 - 1.375 / 8 = 0.703125$$

Odečti úhel $\alpha_4 = 3.576^\circ \rightarrow z_5 = z_4 - \alpha_4 = 2.680 - 3.576 = -0.895^\circ$

$$\begin{aligned} X_5 &= X_4 - Y_4 / 16 &= 1.484375 - 0.703125 / 16 &= 1.440429 \\ Y_5 &= Y_4 + X_4 / 16 &= 0.703125 + 1.484375 / 16 &= 0.795898 \end{aligned}$$

Přičti úhel $\alpha_5 = 1.790^\circ \rightarrow z_6 = z_5 + \alpha_5 = -0.895 + 1.790 = 0.894^\circ$

$$\begin{aligned} X_6 &= X_5 + Y_5 / 32 &= 1.440429 + 0.795898 / 32 &= 1.465300 \\ Y_6 &= Y_5 - X_5 / 32 &= 0.795898 - 1.440429 / 32 &= 0.750884 \end{aligned}$$

Odečti úhel z $\alpha_6 = 0.895^\circ \rightarrow z_7 = z_6 - \alpha_6 = 0.894 - 0.895 = -0.0007^\circ$

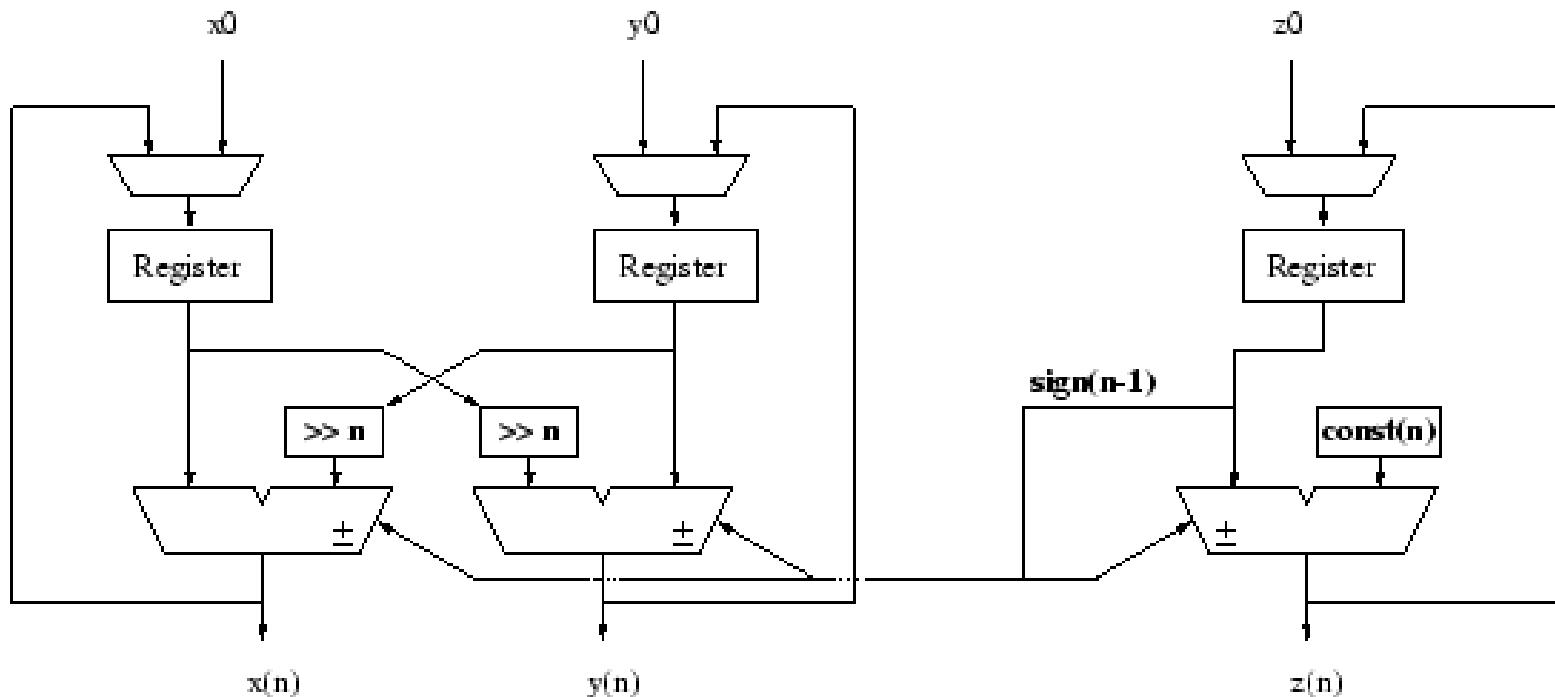
$$\begin{aligned} X_7 &= X_6 - Y_6 / 64 &= 1.465300 - 0.750884 / 64 &= 1.453567 \\ Y_7 &= Y_6 + X_6 / 64 &= 0.750884 + 1.465300 / 64 &= 0.773779 \end{aligned}$$

Následuje násobení agregační konstantou odpovídající použitému počtu iterací, tzn. $K_6 = 0.607278$ a získání požadovaných hodnot:

$$\sin(28.027^\circ) = 0.607278 * Y_7 = 0.46990$$

$$\cos(28.027^\circ) = 0.607278 * X_7 = 0.88272$$

CORDIC – v HW (základní implementace)



úhel v kroku i

d_i je znaménko úhlu

$$x_{i+1} = x_i + y_i \cdot d_i \cdot 2^{-i}$$

$$y_{i+1} = y_i - x_i \cdot d_i \cdot 2^{-i}$$

$$z_{i+1} = z_i - d_i \cdot \text{arctanTab}[i]$$

$d_i = -1$ pokud $z_i < 0$, jinak $+1$

Zobecněný CORDIC

$$x_{i+1} = x_i - \mu d_i y_i 2^{-i}$$

$$y_{i+1} = y_i + d_i x_i 2^{-i}$$

$$z_{i+1} = z_i + d_i \alpha_i$$

- $\mu = 1$ kruhové rotace (\sin, \cos) $\alpha_i = \tan^{-1} 2^{-i}$
- $\mu = 0$ lineární rotace $\alpha_i = 2^{-i}$
- $\mu = -1$ hyperbolické rotace $\alpha_i = \tanh^{-1} 2^{-i}$
- Přímý výpočet funkcí:
 $\sin, \cos, \tan^{-1}, \sinh, \cosh, \tanh^{-1}, \tan^{-1}(y/x), y + xz, \sqrt{x^2 + y^2}, e^z,$
násobení, dělení
- Nepřímý výpočet funkcí:
 $\tan, \tanh, \ln, \log, a^b, \cos^{-1}, \sin^{-1}, \cosh^{-1}, \sinh^{-1}, \sqrt{a}$

Násobení a dělení pomocí algoritmu CORDIC (lineární rotace)

```
def cordicdiv (a,b, n=16):
    x, y, z = b, a, 0.0

    for i in range(n):
        if y < 0:
            y += x * 2**-i
            z -= 2**-i
        else:
            y -= x * 2**-i
            z += 2**-i

    return z # a/b
```

```
def cordicmul (a,b, n=16):
    x, y, z = a, 0.0, b

    for i in range(n):
        if z >= 0:
            y += x * 2**-i
            z -= 2**-i
        else:
            y -= x * 2**-i
            z += 2**-i

    return y # a*b
```

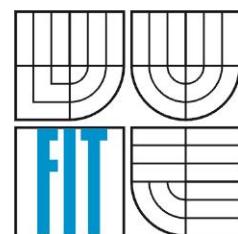
Využíváme vektorový režim

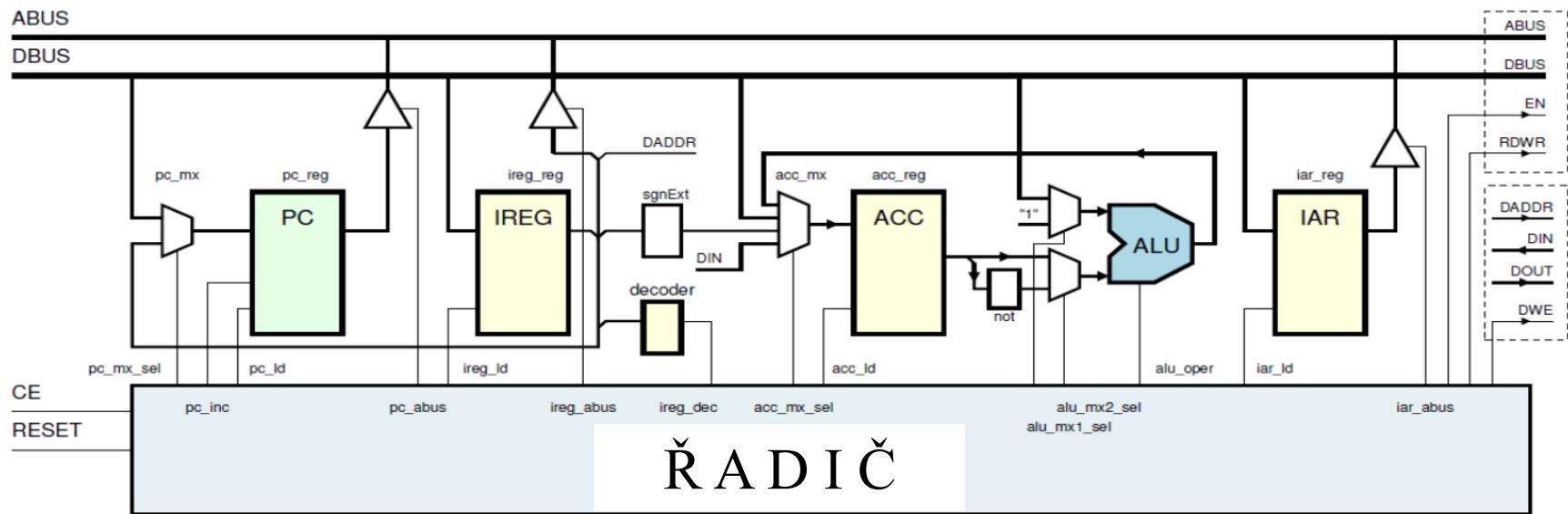
Využíváme rotační režim

Konec kapitoly o implementaci
aritmetických operací v HW

Řadiče

INP 2015
FIT VUT v Brně



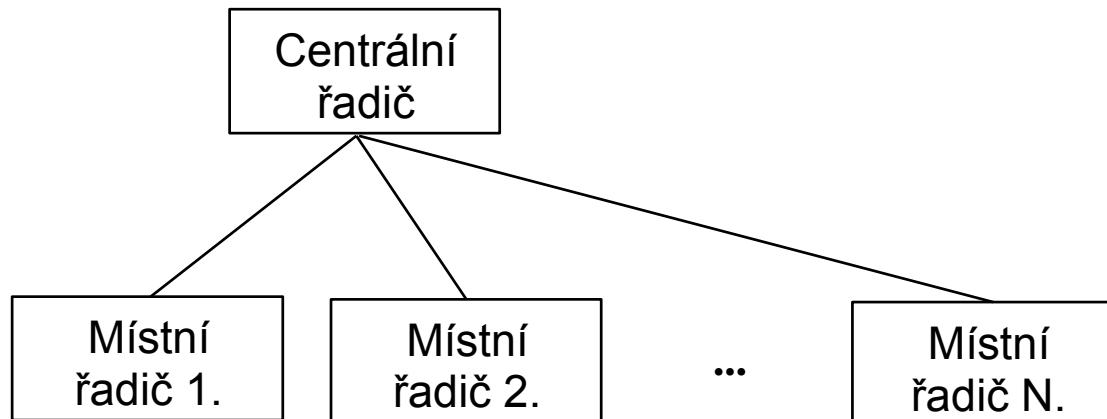


Co dělají a jak jsou realizovány řadiče „reálných“ procesorů?

Hlavní funkce řadiče

- interpretace instrukcí – dekódování a provedení
- krokování instrukcí – vytváření toku instrukcí
- řízení systémových procesů – přerušení, obsluha RVP (cache), řízení segmentace a stránkování, atd.

Řadič se obvykle realizuje jako **centrálně řízená soustava místních řadičů**, které mají svoje funkce rozdělené a specializované.



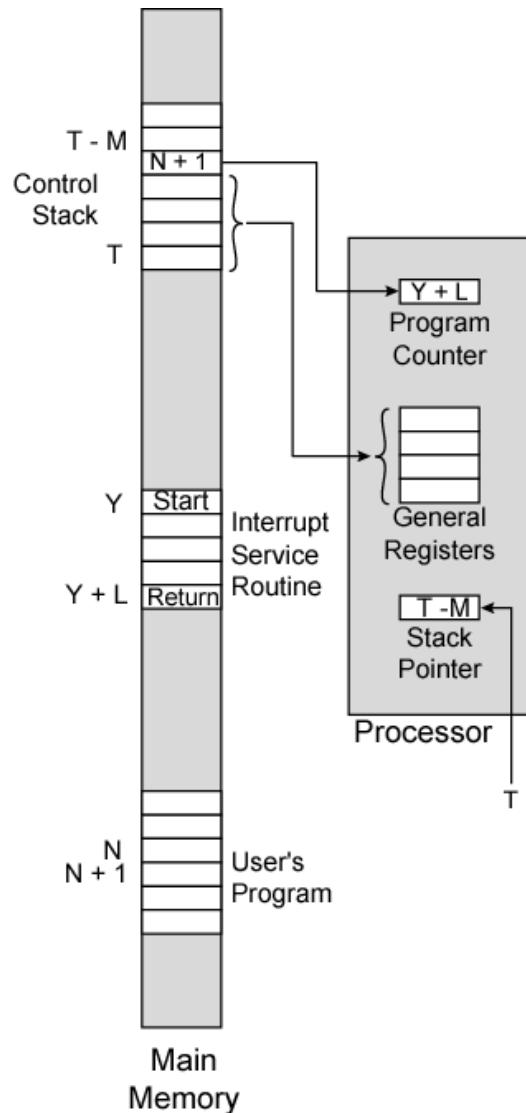
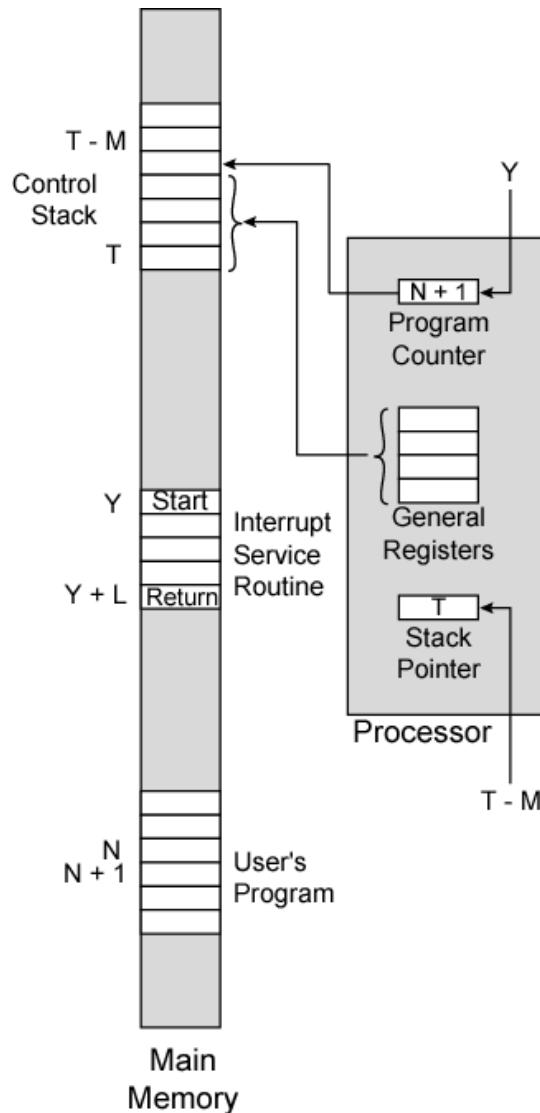
Soustava místních řadičů “reálného” počítače

- řadič provedení instrukce ALU s pevnou řádovou čárkou včetně výpočtu efektivní adresy operandu, řízení algoritmu operace, vyhodnocení podmínky skoku apod.
- řadič provedení instrukce FPU s pohyblivou řádovou čárkou
- řadič přerušení
- řadič vstup-výstupních operací, který má u počítače s nezávislými I/O kanály podobu řadiče kanálu
- řadič paměti
- řadič kanálu přímého přístupu k paměti (DMA - Direct Memory Access)
- řadiče jednotlivých periferních zařízení
- řadič sběrnice
- atd.

Přerušení

- Přerušení bylo u počítačů zavedeno pro ošetření nestandardních situací, jako
 - dělení nulou
 - chyba v datech přečtených z paměti
 - výpadek síťového napájení
 - připravenost diskové jednotky číst nebo zapsat blok dat
 - přeplnění disku síťového serveru
 - výpadek bloku dat RVP (cache miss)
 - atd.
- Má-li se přerušení správně obsloužit, musí se v první řadě správně identifikovat **zdroj** a **typ přerušení**, a pak spustit příslušná obslužná rutina.
- Klasifikace přerušení podle typu není zcela ustálená, a proto u různých typů procesorů můžeme nacházet terminologické i principiální rozdíly.

Princip přerušení



Př. Pokud byla stisknuta klávesa, procesor přeruší uživatelský program, uschová obsahy registrů a návratovou adresu na zásobník a obslouží klávesnici (rutina na adrese Y). Potom obnoví obsahy registrů a vrátí se do uživatelského programu.

Klasifikace přerušení

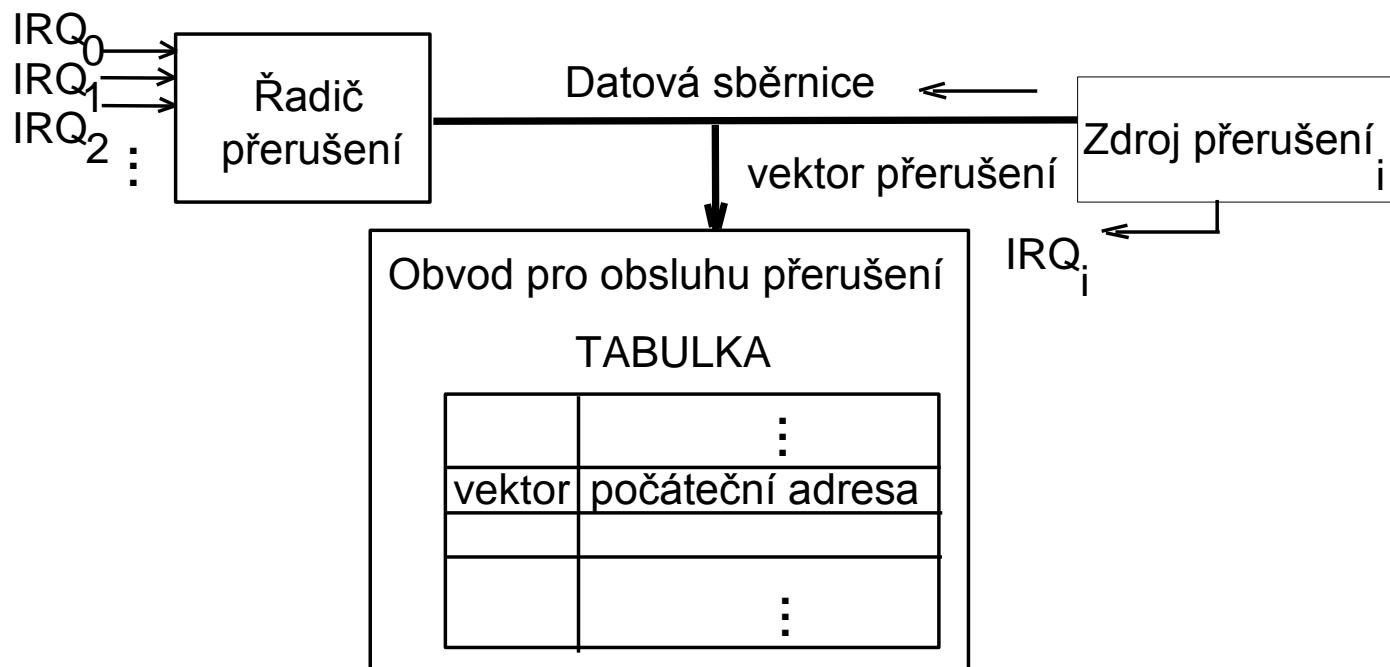
- Podle typu zdroje rozeznáváme přerušení
 - **programové**, vyvolané prováděním instrukcí, např. instrukcí INTn, přeplněním při sčítání, dělením nulou, neexistující adresou paměti apod.
 - **technické**, vyvolané poruchou některé jednotky, výpadkem napájení, překročením časového limitu (watchdog) atd.
 - **vstup-výstupní**, jako přerušení od časovače, připravenost V/V operace, zpráva z počítačové sítě aj.
- Přerušení můžeme rozdělit na
 - vnitřní a
 - vnější,
- nebo z hlediska jejich synchronismu s hodinovým taktem procesoru na
 - synchronní (trap), např. při krokování instrukcí
 - asynchronní (interrupt)
- nebo z hlediska naléhavosti rozlišují některé procesory přerušení
 - maskovatelná
 - nemaskovatelná.
- Zdrojem přerušení je přiřazena priorita – dle důležitosti.

Mechanismy přerušení

- Přerušení vyvolá zařízení žadající o obsluhu aktivací signálu **IRQ** (Interrupt Request). Žádosti vyhodnocuje řadič přerušení.
- Vlastní mechanismus přerušení může být založen na
 - vnučení adresy
 - vnučení instrukce
 - vnučení vektoru – nejobvyklejší způsob

Vnucení přerušovacího vektoru

- Každý zdroj přerušení generuje přerušovací číslo, zvané vektor přerušení (např. 8 bitů), které je ukazatelem do tabulky počátečních adres přerušovacích rutin.
- Obsahem tabulky mohou být např. 4 byty čísla segmentu a adresy uvnitř segmentu, nebo 8 bytů, které se vloží do tabulky přerušovacího deskriptoru (I 80 286 a další) a definují tak začátek obslužné rutiny.



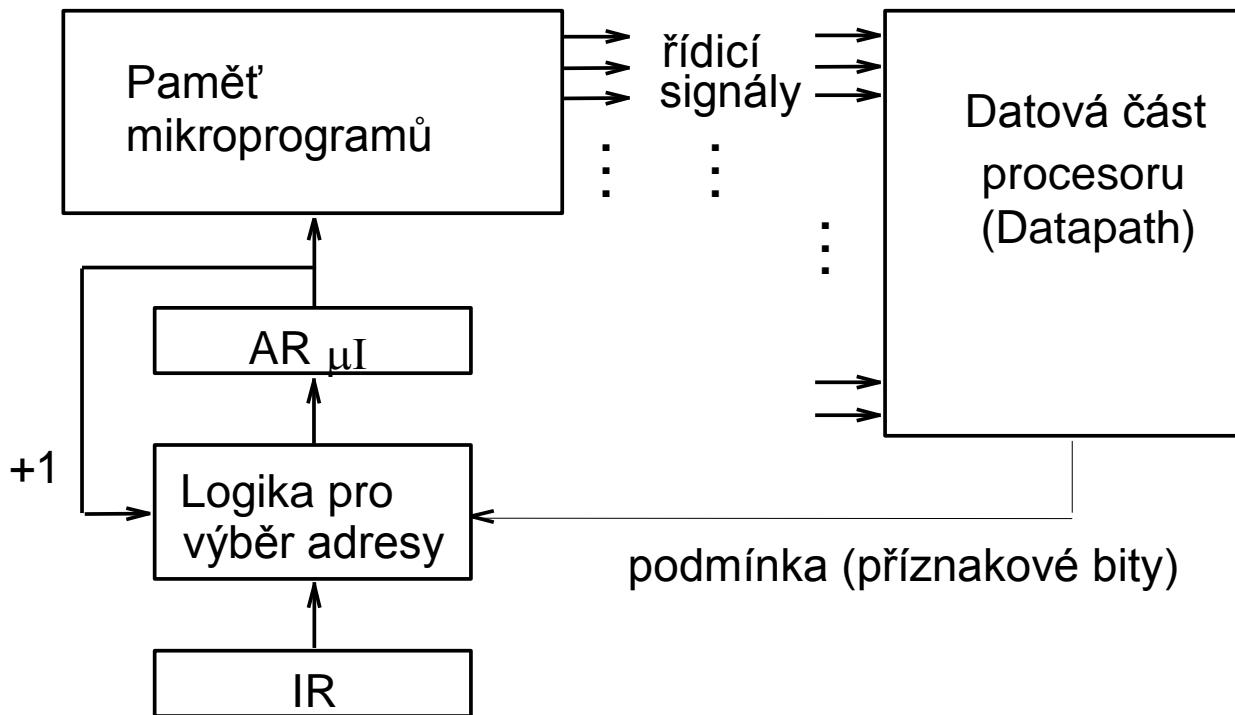
Implementace řadiče

- **Obvodový řadič**
 - Řadič je definován pomocí automatu a tento automat je implementován běžným způsobem.
 - viz první přednáška
- **Mikroprogramový řadič**
 - Namísto řadiče je vytvořen „malý procesor“, který vykonává mikroprogramy. Vykonáním jednoho mikroprogramu je provedena jedna instrukce procesoru.

Mikroprogramový řadič

- Konstrukce mikroprogramového řadiče a podpůrná disciplina mikroprogramování jsou velmi rozšířené postupy, které se po mnoho let staly převažující metodikou návrhu řadičů.
- Mikroprogramování je systematický postup, který za pomoci překladače mikroprogramů napsaných v mikroassembleru vytvoří binární obsah permanentní řídící paměti, která je základem mikroprogramového řadiče.
- Převažující část problému syntézy řadiče operací procesoru je tak přesunuta na programovou úroveň, která byla v historii vybavena komfortními podpůrnými prostředky dříve než disciplina obvodového návrhu.
- Mikroprogram slouží jako interpret instrukce v reálném čase.

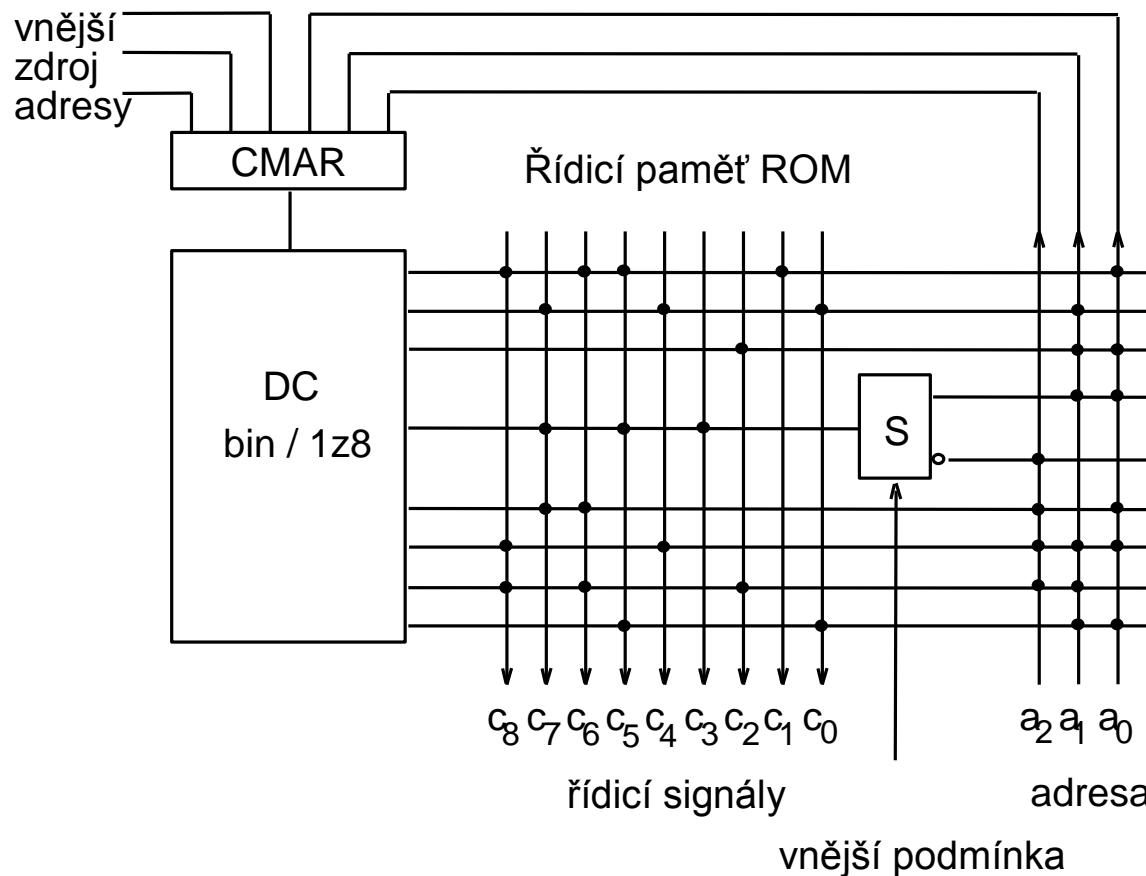
Základní schéma mikroprogramového řadiče



- Začátek mikroprogramu zajišťujícího provedení určité instrukce se určí logikou pro výběr adresy podle kódu operace v IR.
- Jednotlivé mikroinstrukce se čtou z paměti mikroprogramů podle obsahu adresového registru mikroinstrukce AR μ I v nejjednodušším případě sekvenčně, tedy adresa příští mikroinstrukce se určí ze současné adresy inkrementací.
- Jednotlivé bity mikroinstrukce jsou řídicí signály pro datovou část procesoru, tj. ALU, a dále M a I/O.

Základní implementace mikrořadiče

(Wilkesovo schéma, 1951)



Registr CMAR je adresový registr řídicí paměti (Control Memory Address Register), přepínač S (demultiplexor) řízený vnější podmínkou určuje adresu příští mikroinstrukce; buď je to 011, nebo 100.

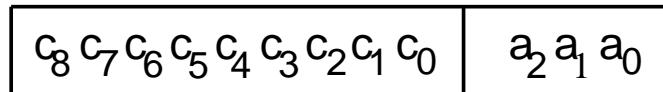
Paměť mikroprogramu a firmware

- Paměť mikroprogramů díky své pružnosti umožňuje poměrně rychlé a snadné modifikace, opravy a rozšiřování instrukčního souboru.
 - Dříve se jednalo o ROM, v poslední době s používají Flash paměti.
- **Firmware** = obsah této paměti (je to firemní záležitost).
- Další využití je pro diagnostické účely
 - Po startu systému se do paměti mikroprogramu uloží diagnostický mikroprogram, který velmi podrobně a rychle provedou diagnózu systému. Teprve po úspěšném testu se uloží do řídicí paměti mikroprogramy pro standardní instrukční soubor.

Nevýhody mikroprogramování

- Mikroprogramový řadič je **pomalejší**.
 - O 10% u HP 2116 (obvodový řadič) vs HP 2100S (mikroprogramový řadič).
 - Tato ztráta se však dalším technologickým vývojem brzy nahradila a výhody mikroprogramového řadiče na dlouhou dobu jednoznačně převázily.
- Zásah do mikroprogramů vyžaduje vysokou kvalifikaci a **není** to v žádném případě běžná **uživatelská záležitost**.
 - Změna mikroprogramů oproti firemní verzi se uplatňovala pouze ve výjimečných situacích, jako při řešení závažného selhání systému vlivem poruchy a neměla trvalý charakter.
 - U současných procesorů je paměť mikroprogramů pro uživatele **nepřístupná**.
- Mikroprogramy **nejsou přemístitelné** a optimalizace mikroprogramů z hlediska činnosti celého systému je náročná a dlouhodobá záležitost.
 - Může totiž snadno dojít k potlačení či přibrzdění některého podsystému, což se může projevit poklesem výkonnosti I/O podsystému, paměti, přerušovacího systému atd.

Formát Wilkesovy mikroinstrukce



funkce: řízení operací krokování sledu mikroinstrukcí

Základní dvě funkce mikroinstrukčního řadiče jsou *řízení operací*, resp. mikrooperací a *krokování sledu mikroinstrukcí*.

Kromě těchto funkcí musí mikroprogramový řadič obecně provádět tyto činnosti: *přiřadit každé instrukci mikroprogram a provádět nepodmíněné i podmíněné skoky*. K těmto funkcím se postupně přidaly další.

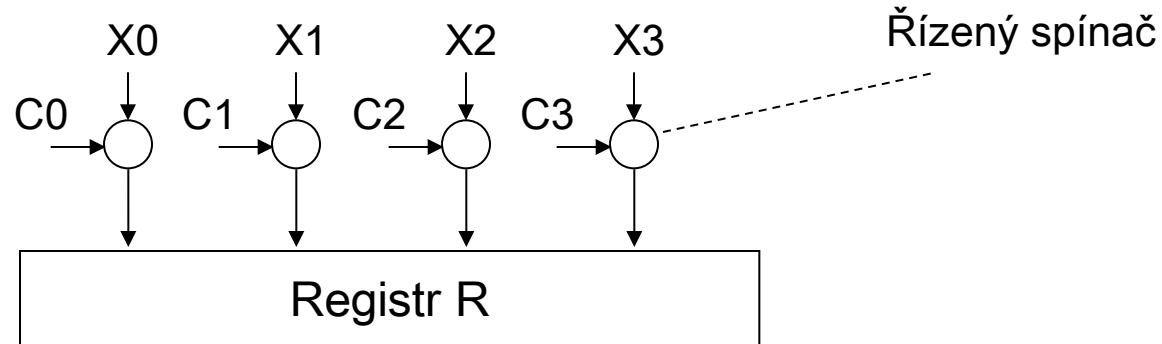
Obecný formát mikroinstrukce

zdroj 1	zdroj 2	ALU	cíl	M R/W	konst	různé	CC	adresa	modadr
---------	---------	-----	-----	-------	-------	-------	----	--------	--------

Mikroinstrukce je **tříadresová**. Pole *zdroj 1* a *zdroj 2* určují vstupní operandy, pole *ALU* určuje operaci, *cíl* je adresa pro výsledek operace, polem *M R/W* se může spustit operace paměti M, *konst* je pole pro výběr konstant z permanentní paměti, zvláštní situace se mohou ošetřit v poli *různé*, **adresa** je pole udávající adresu příští **mikroinstrukce**, která se může dále modifikovat polem *modadr*. Tento formát je s jistými modifikacemi použit ve všech mikroprogramových řadičích.

Kódování povelů

Př. Zápis do registru ze 4 nezávislých zdrojů



a) Nekódovaný formát (horizontální)

...	C0	C1	C2	C3	...
-----	----	----	----	----	-----

1	0	0	0	R ← X0
0	1	0	0	R ← X1
0	0	1	0	R ← X2
0	0	0	1	R ← X3
0	0	0	0	nop

b) Kódovaný formát (vertikální)

...	K0	K1	K2	...
-----	----	----	----	-----

0	0	1		R ← X0
0	1	0		R ← X1
0	1	1		R ← X2
1	0	0		R ← X3
0	0	0		nop

$\lceil \log_2(n+1) \rceil$ bitů

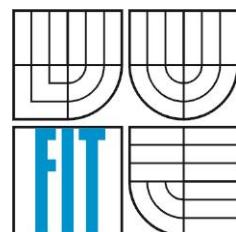
Nevýhoda: nutnost dekódovat, tj. zpoždění

Poznámky

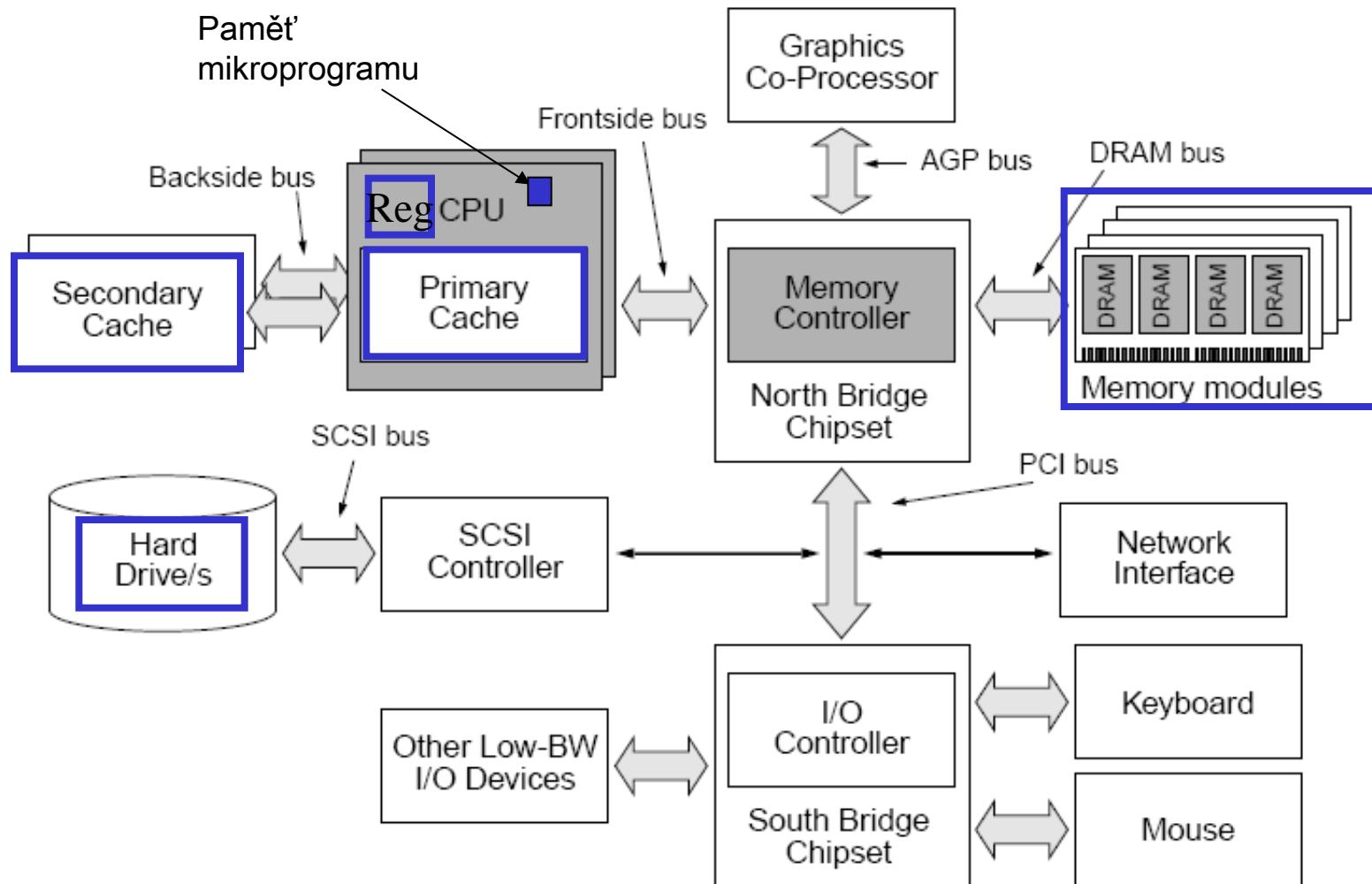
- Mikroprogramový řadič je často implementován jako **řetězená jednotka**
 - Musí se řešit stejné problémy jako u řetězeného zpracování na úrovni instrukcí.
- Mikroprogramová realizace instrukcí je typická pro procesory se složitým instrukčním souborem (**CISC**).
 - Pružnost této koncepce se přímo nabízí pro doplňování dalších složitějších funkcí. Typickou aplikací je realizace mikrodiagnostického systému, který se spustí po startu počítače a provede základní testy paměti M, procesoru, periferií apod.
- Často doplňovanými funkcemi jsou mikroprogramové realizace typických instrukčních posloupností generovaných překladači vyšších jazyků.
 - Tyto prostředky přinášely zrychlení provádění programů řádově v jednotkách až desítkách procent.

Paměti

Návrh počítačových systémů
INP 2015

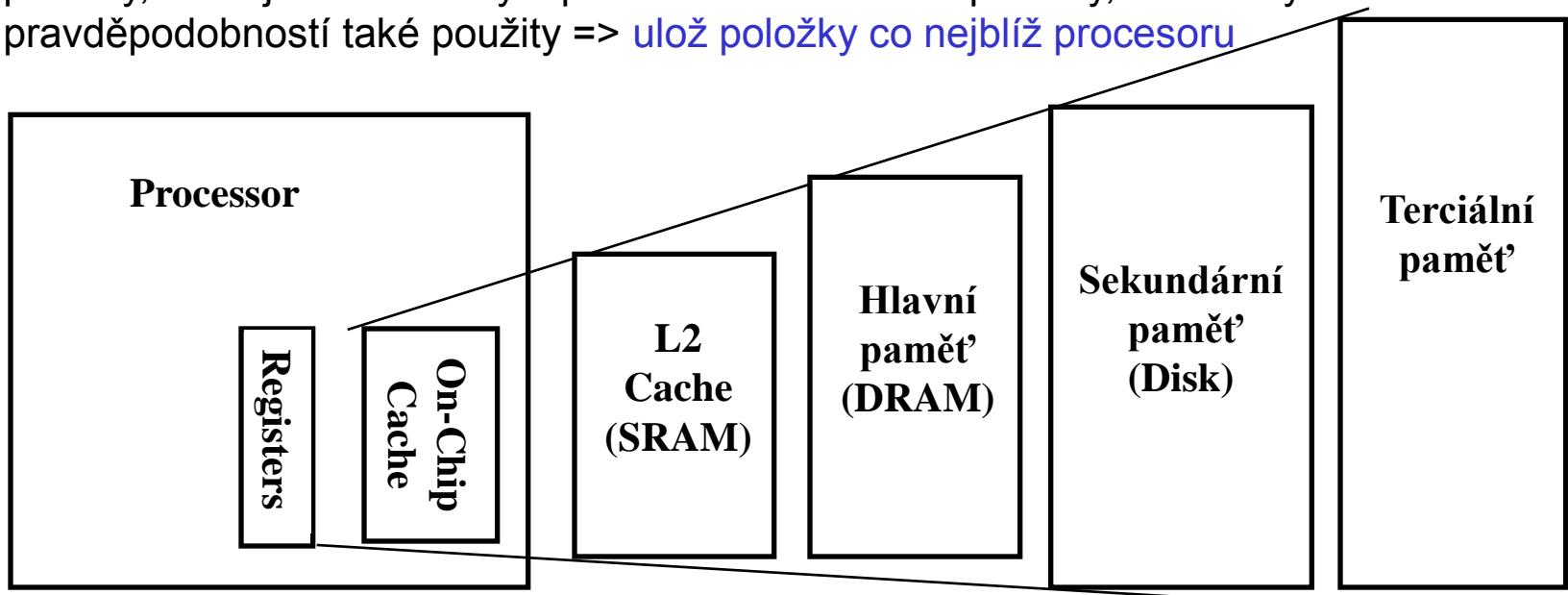


Paměťové prvky v reálném počítači



Proč paměťová hierarchie?

- chceme maximalizovat výkonnost počítače (tj. poměr výkon/cena)
- potřeba **nevolutilní** paměti, která nepotřebuje napájecí napětí
- **časová lokalita** - pokud procesor používá nějakou položku v paměti, je vysoká pravděpodobnost, že ji bude používat znovu => **ulož položku co nejblíž procesoru**
- **prostorová lokalita** – pokud procesor pracuje s nějakou položkou v paměti, potom položky, které jsou umístěny v paměti v blízkosti této položky, budou s vysokou pravděpodobností také použity => **ulož položky co nejblíž procesoru**



Přístupová doba: ns <10ns desítky ns jednotky ms sekundy

Kapacita: stovky B kB jednotky MB jednotky GB stovky GB desítky TB

Cena/MB: nejvyšší střední nejnižší

Paměti - typy a parametry

Paměťové prvky se používají v počítači na všech hierarchických úrovních v těchto funkcích:

- **vnitřní paměť procesoru** - registry, registrové sady, zásobníky, fronty, tabulky pro různé účely a paměť mikroprogramů v řadiči procesoru,
- **hlavní paměť** včetně rychlých vyrovnávacích pamětí,
- **vnější paměti.**

Ve všech těchto hlavních skupinách se používá mnoho různých typů pamětí, které se zásadně liší svými **parametry**, **fyzikálními principy**, **způsobem výběru dat** a dalšími vlastnostmi.

Parametry pamětí:

- **kapacita** se udává ve tvaru respektujícím organizaci paměti, tedy jako součin *počtu paměťových míst* a *délky paměťového místa*, tj. $N \times n$ bitů, jako např. $16K \times 1$ bit, $64K \times 1$ Byte, $4M \times 4$ Bytes atd.
- **přístupová doba** t_a je doba od zahájení čtení (tj. udáním adresy paměťového místa a povelu R) po získání obsahu paměťového místa.
- **doba cyklu** je doba od zahájení čtení nebo zápisu (Č/Z) až po skončení této operace, kdy je možno spustit další operaci C/Z.

Další parametry pamětí

Přenosová rychlosť je parametr udávající počet datových jednotek (bitů, bytů atd.) přenášených do nebo z paměti za sekundu, např. 1Gb/s u disku.

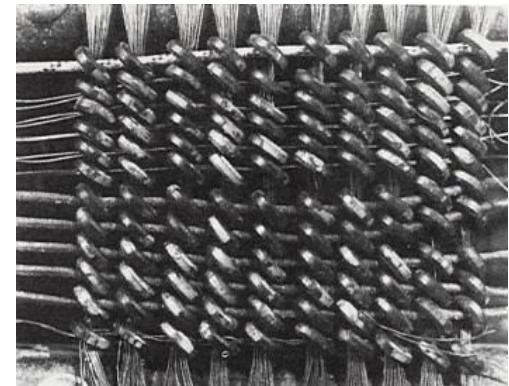
Pro cenové rozvahy je důležitým parametrem *cena/bit*.

K dalším významným parametrům patří *chybovost* paměti udávaná např. v počtu chyb na 1000 hodin a *poruchovost*, nejčastěji popisovaná parametrem *střední doba mezi poruchami*.

Výkonnost je u pamětí udána parametry: kapacita, přístupová doba a přenosová rychlosť.

Klasifikace pamětí podle fyzikálního principu

- paměti polovodičové - bipolární a
 - unipolární
- magnetické - disketové,
 - diskové,
 - páskové
 - atd.
- optické - CD
 - DVD
- magnetooptické
- nekonvenční, např. molekulární



feritová pamět z IBM 405

Mnoho typů pamětí zmizelo a další se objevují.

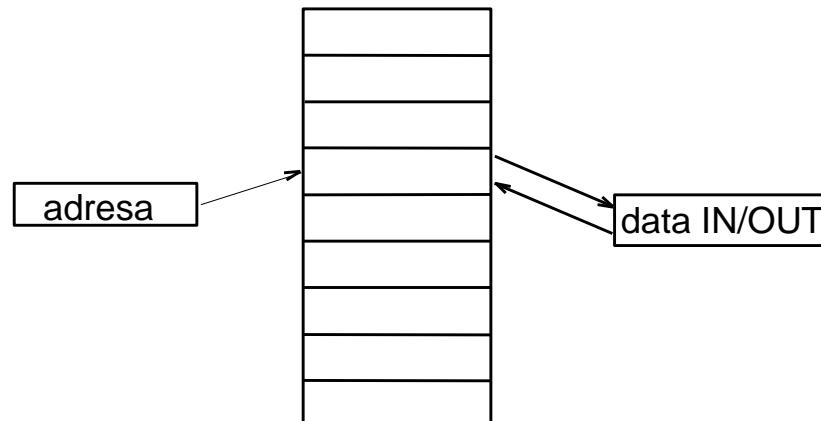
Přístup k datům – libovolný a sériový

Pokud přístupová doba nezávisí na umístění požadované položky, jde o paměť s *libovolným přístupem RAM* (Random Access Memory).

Paměť se *sériovým přístupem SAM* (Serial Access Memory) vybavuje položky s různou dobou přístupu podle toho, jak dlouho to trvá, než se paměťové médium přisune k čtecí hlavě.

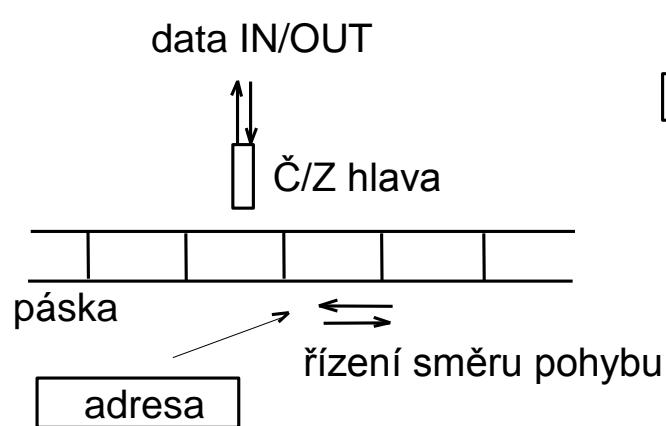
Disk s několika záznamovými povrchy je reprezentantem *smíšeného přístupu*; výběr záznamového povrchu je libovolný přístup, vystavování hlavy na požadovanou stopu a otáčení disku při čekání na požadovaný záznamový sektor je sériový přístup. Čistě sériový přístup má magnetická páska.

Libovolný přístup RAM

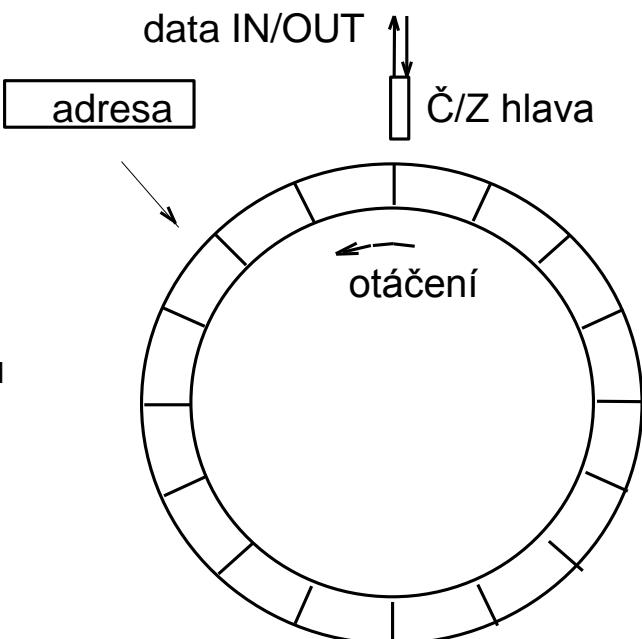


a) RAM

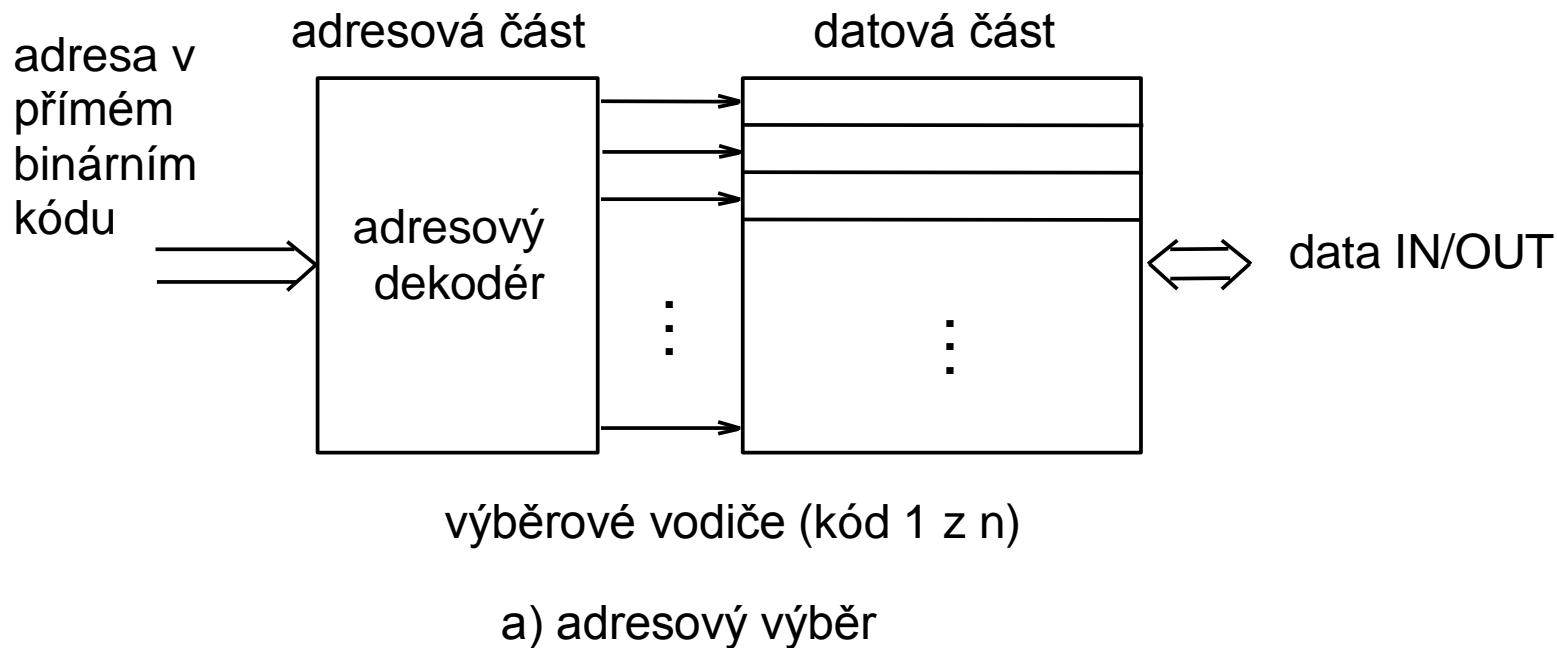
Sériový přístup SAM



b) SAM, resp. smíšený přístup

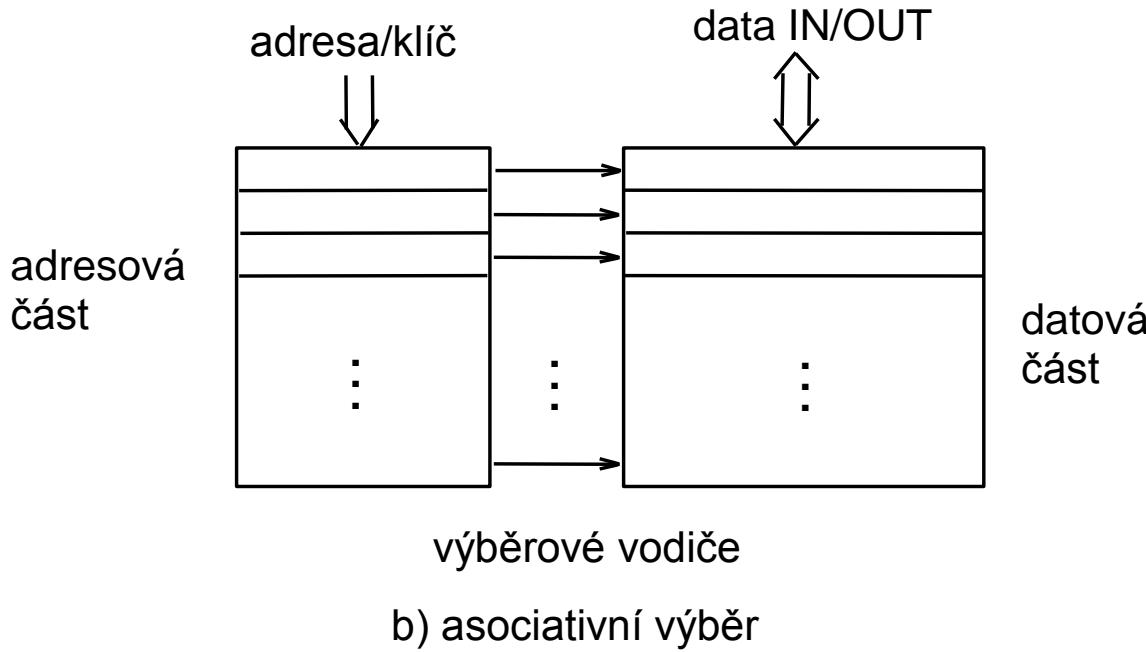


Výběr z paměti – adresový



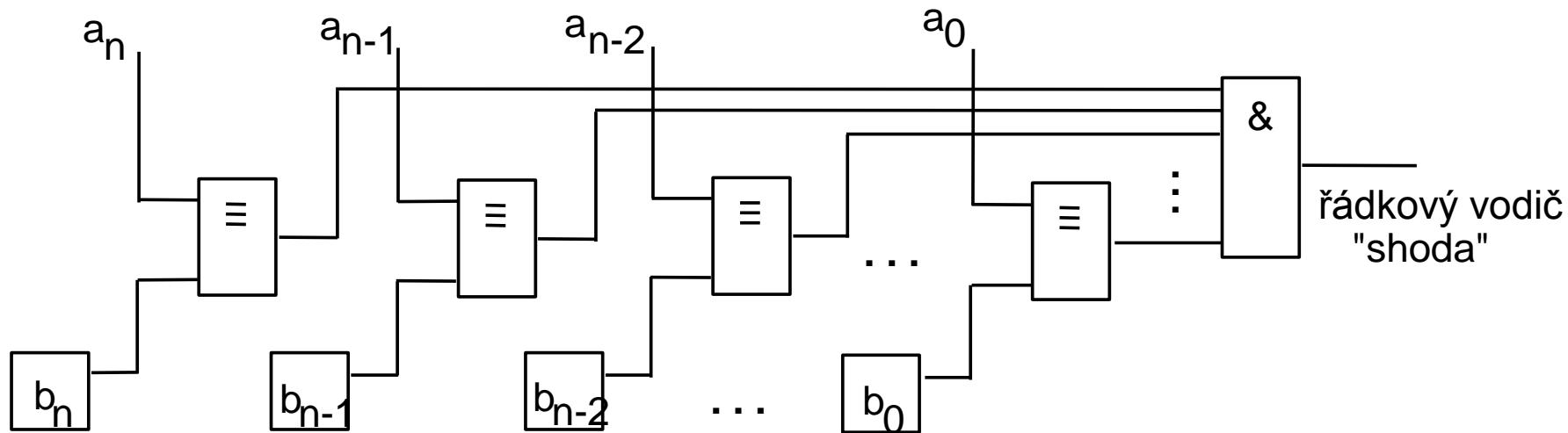
Paměťová místa jsou uspořádána podle adres vzestupně, *adresový prostor je uspořádaný a souvislý*.

Výběr z paměti - asociativní



- U *asociativního výběru* jsou v adresové části paměti poznamenány adresy paměťových míst datové části.
- Paměťová místa mohou být vzhledem k adresám uspořádána libovolným způsobem, některým adresám nemusí odpovídat žádné paměťové místo.
- Adresový prostor je *neuspořádaný a nesouvislý*. Výběrový vodič se aktivizuje na základě shody hledané adresy (klíče) s adresou daného řádku. Navíc se může pomocí *masky adresy* určit, které bity adresy se mají při porovnávání použít, a které ne. Proto je vhodné říkat adrese **klíč**.
- Princip asociativního výběru vyžaduje, aby ve všech řádcích adresové části paměti byl *komparátor adres - klíčů*.

Komparátor adres - klíčů asociativní paměti



a – klíč na vstupu paměti dodaný uživatelem

b – jedna konkrétní hodnota klíče v paměti

Uvedená obvodová struktura musí být zopakována k -krát,
kde k je počet položek uložitelných v paměti.

Měnitelnost obsahu paměti – R/W

- **RWM** - Read/Write Memory - paměť umožňující čtení i zápis
- **ROM** - Read Only Memory - paměť umožňující pouze čtení, zapisovat nelze.

Varianty:

- **PROM** - programovatelná ROM; „čistá“ nenaprogramovaná paměť umožňuje jedno naprogramování, další změna již není možná.
- **EPROM** - vymazatelná PROM; naprogramovaná paměť se dá vymazat a znova naprogramovat. Paměti s tímto označením se mažou ultrafialovým zářením.
- **EEPROM** - elektricky vymazatelná PROM. Zde je řada modifikací podle toho, zda je možno mazat vybraný řádek, nebo pouze celou paměť, jak rychle proces mazání probíhá atd. (např. Flash EEPROM).

Stálost obsahu paměti

Podle energetické závislosti dělíme paměti na *nevolutilní* a *volatilní*. Je-li záznam stálý i po vypnutí napájecího napětí, jde o nevolutilní paměť.

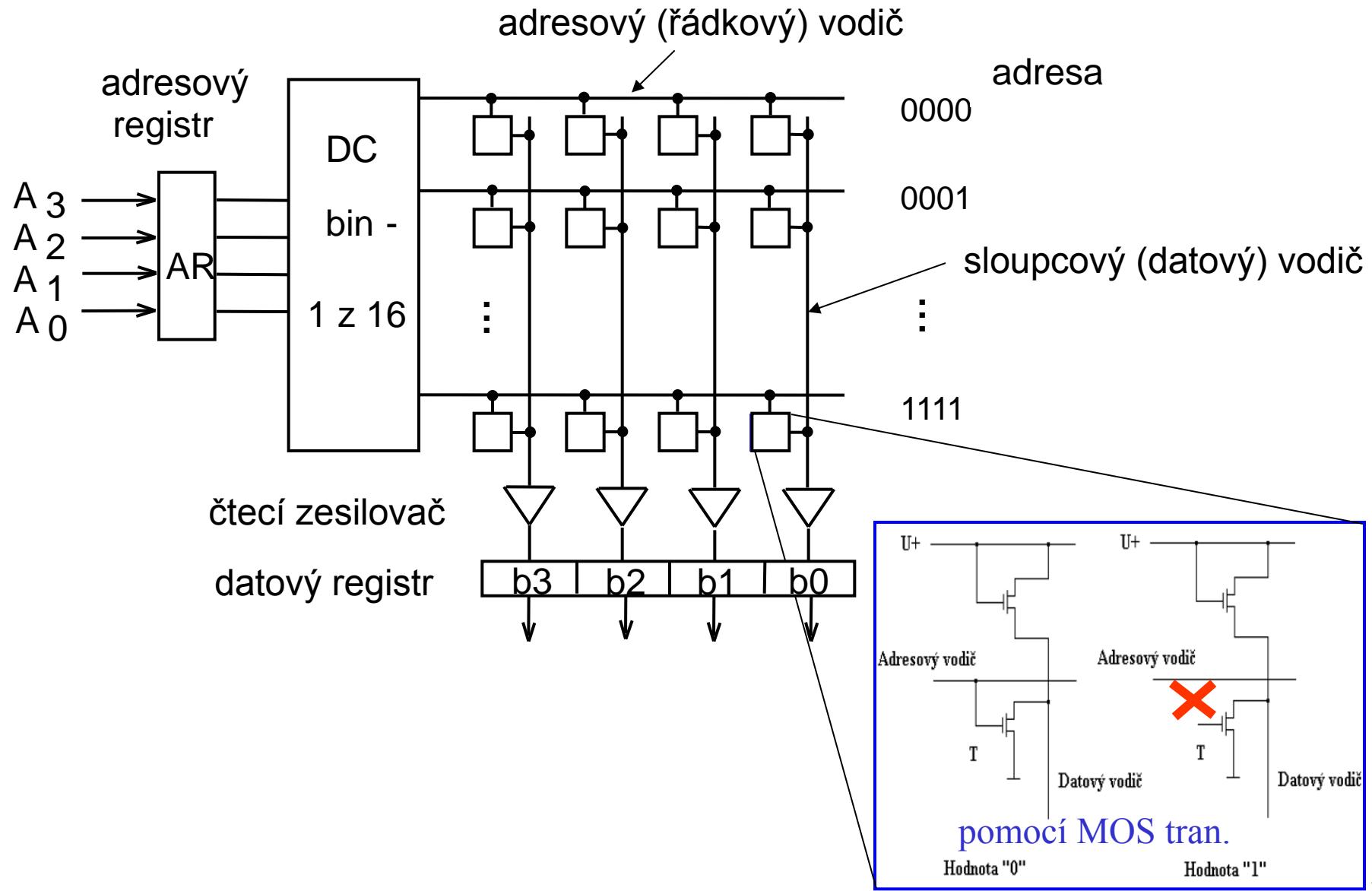
Některé fyzikální principy vedou na paměť, u které se čtením záznam vymaže. Znamená to, že se po cyklu čtení musí zařadit vždy cyklus zpětného zápisu. Takové paměti se označují jako *destruktivní*.

Podle *doby uchování informace* dělíme paměti na statické (**SRAM**), které při dodržení jistých provozních parametrů drží informaci libovolně dlouho, a dynamické (**DRAM**), které "zapomínají", a to docela rychle. U tohoto typu pamětí se proto musí zavčas informace obnovit (*refresh*).

Polovodičové paměti

- ROM
 - PROM, EPROM
- RWM
 - RAM
 - SSRAM, SRAM
 - DRAM
 - FPM DRAM, EDO DRAM, BEDO DRAM
 - SDRAM
 - DDR SDRAM, DDR2 SDRAM, DDR3 SDRAM
- EEPROM a FLASH

Struktura paměti ROM 16 x 4b

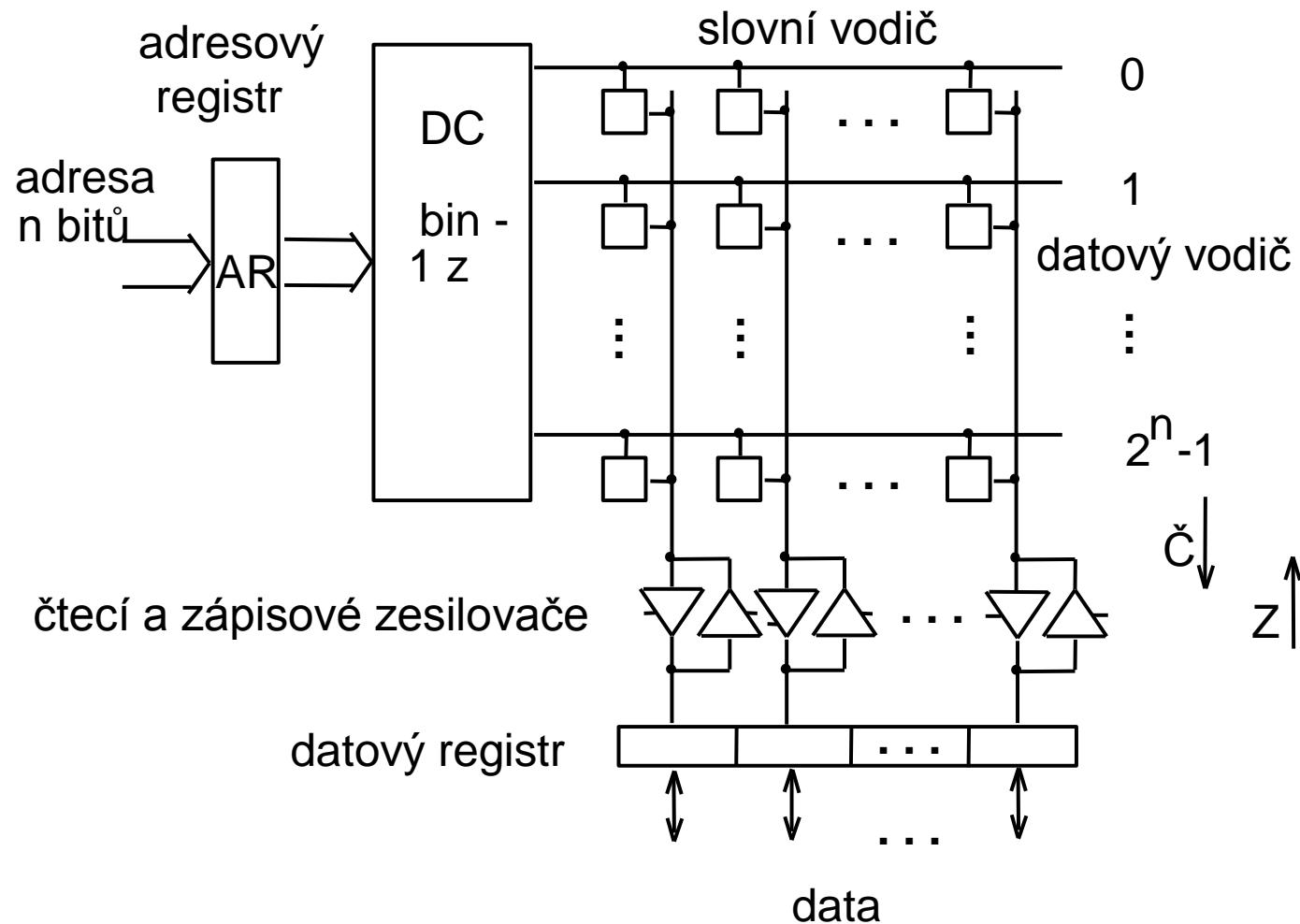


Komentář k příkladu paměti ROM

Čtyřbitová binární adresa se zachycuje do adresového registru AR a dále se dekóduje do kódu 1 z 16, což znamená, že pro jistou adresu se aktivizuje příslušný adresový vodič. Obsah vybraného čtyřbitového adresového místa se objeví na sloupcových datových vodičích a po zesílení pomocí čtecích zesilovačů ČZ se zapíše do výstupního datového registru DR.

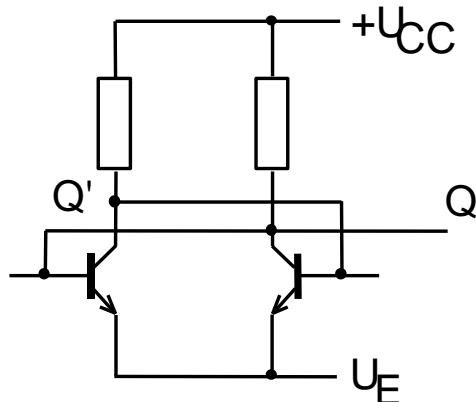
Ve funkci paměťových prvků pamětí ROM se v historii počítačů vystřídaly všechny základní pasivní i aktivní elektrické prvky. Byly tak použity rezistory, indukčnosti, transformátory, feritová jádra, kapacitory, diody, tranzistory bipolární i unipolární. Nejrozšířenější typ EPROM využívá kapacity izolovaného hradla tranzistoru MOS. U pamětí PROM se programuje přepalováním chromniklových nebo křemíkových propojek.

Struktura paměti RWM

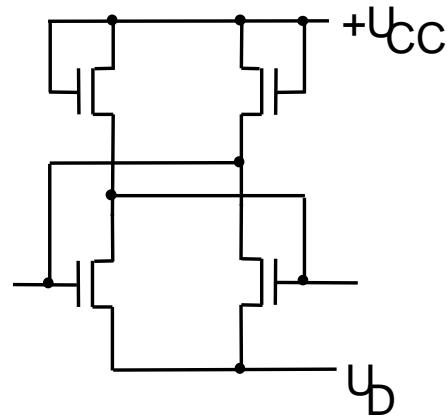


Komentář k RWM a typy paměťových členů

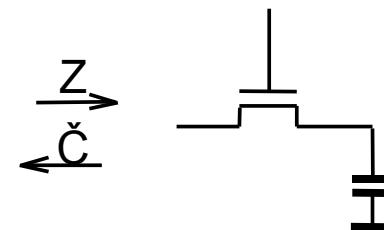
Struktura paměti RWM (nepřesně označovaná jako RAM) je principiálně velmi podobná struktuře paměti ROM. Rozdíl je v tom, že sloupcové datové vodiče nyní přenášejí data **obousměrně**, a jsou doplněny zápisové zesilovače ZZ. Rovněž vnější datová sběrnice je obousměrná.



a) Bipolární statický člen



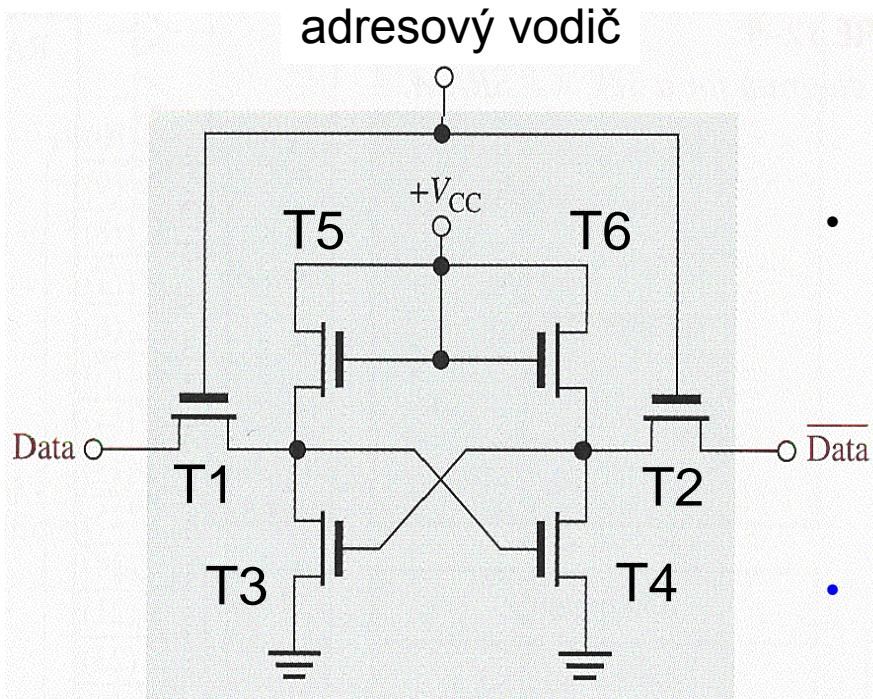
b) Unipolární statický člen



c) Unipolární dynamický paměťový člen DRAM

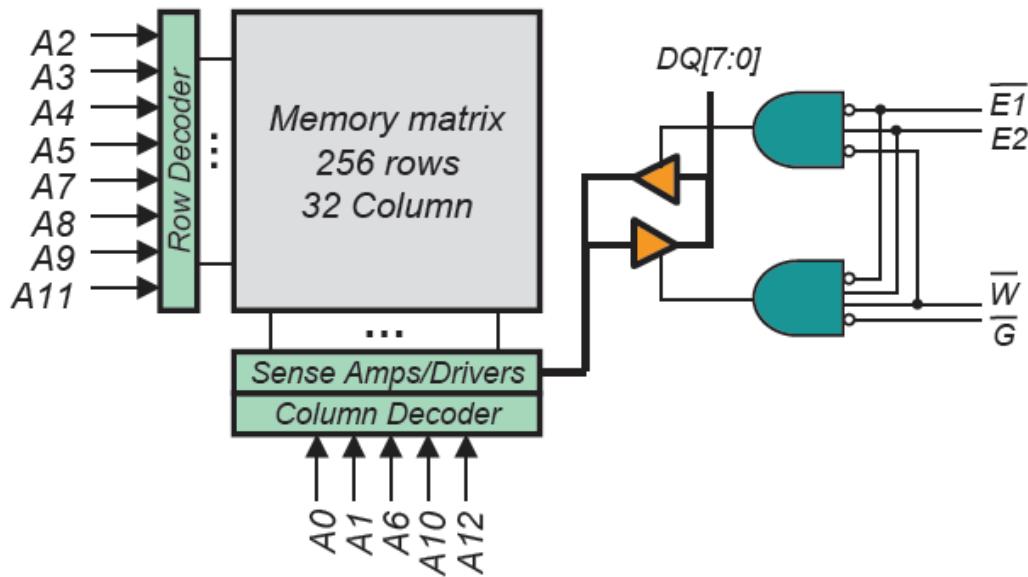
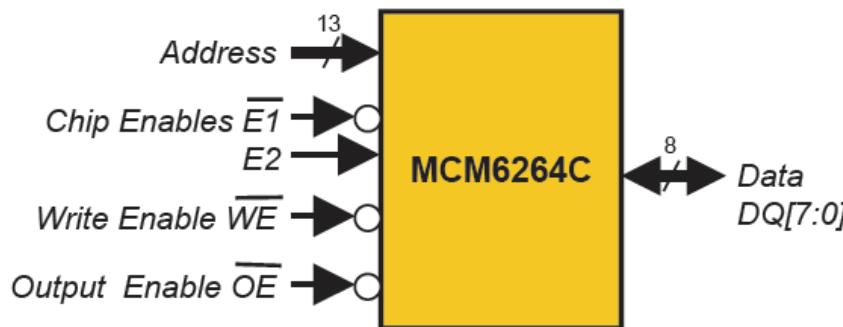
Každý paměťový člen má jiný způsob čtení a zápisu informace. Podle typu použitého paměťového členu se proto modifikuje vnitřní zapojení paměťové matice, čtecích a zápisových zesilovačů, případně se modifikuje celková struktura paměti. U statických paměťových členů se zápis a čtení provádí pomocí dvojic sloupcových datových vodičů, které nesou komplementární proudové nebo napěťové impulsy.

Paměťová buňka SRAM (Static RAM)



- Vodič **Data** je určený k zápisu dat do paměti. Vodič označený jako **-Data** se používá ke čtení. Hodnota na tomto vodiči je vždy opačná než hodnota uložená v paměti => na konci procesu čtení je nutno ji ještě negovat.
- Při **zápisu** se na adresový vodič umístí hodnota logická 1, na vodič Data se přivede zapisovaná hodnota (např. 1). Tranzistor T1 se otevře => jednička na vodiči Data otevře tranzistor T4 => uzavře se tranzistor T3. Tento stav obvodu představuje uložení hodnoty 0 do paměti.
- **Čtení**: na adresový vodič je přivedena hodnota logická 1 => otevřou se tranzistory T1 a T2. Jestliže byla v paměti zapsána hodnota 1, je tranzistor T4 otevřen (tj. na jeho výstupu je hodnota 0), čtenou hodnotu obdržíme na vodiči **-DATA**. V případě uložené hodnoty 0 - tranzistor T4 je uzavřen (tj. na jeho výstupu je hodnota 1).

Př. SRAM (8k x 8b)



Same (bidirectional) data bus used for reading and writing

Chip Enables ($\overline{E1}$ and $E2$)

$E1$ must be low and $E2$ must be high to enable the chip

Write Enable (WE)

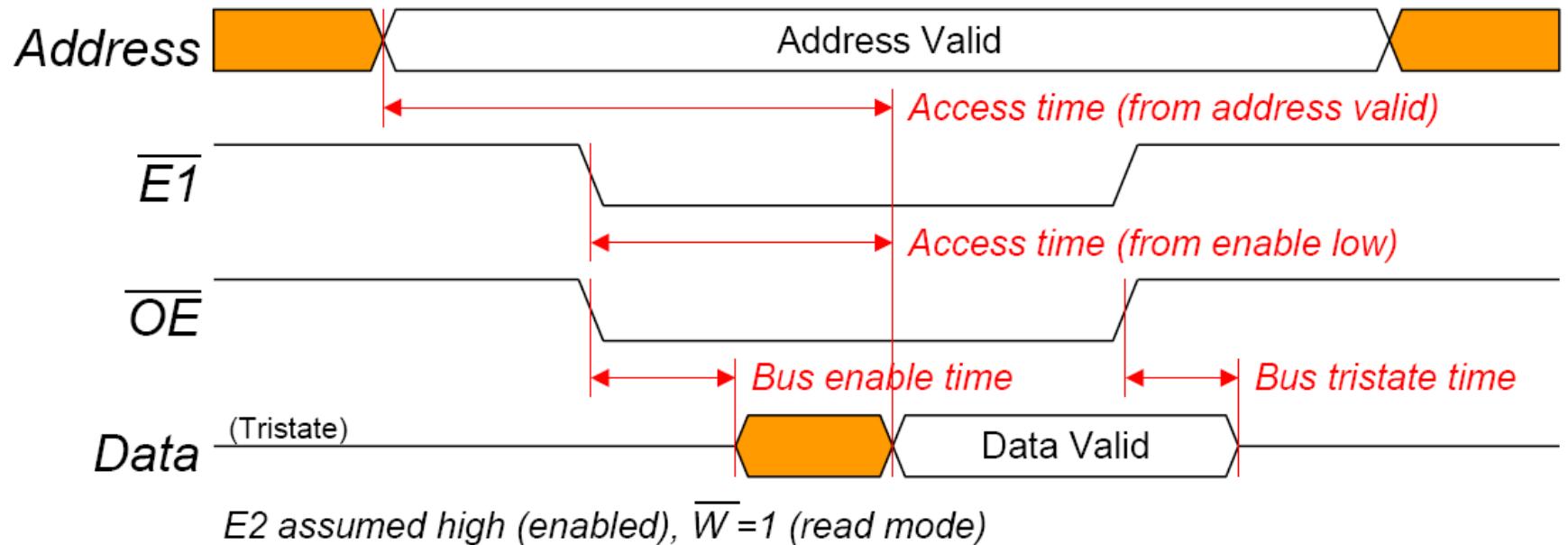
When low (and chip enabled), values on data bus are written to location selected by address bus

Output Enable (OE or G)

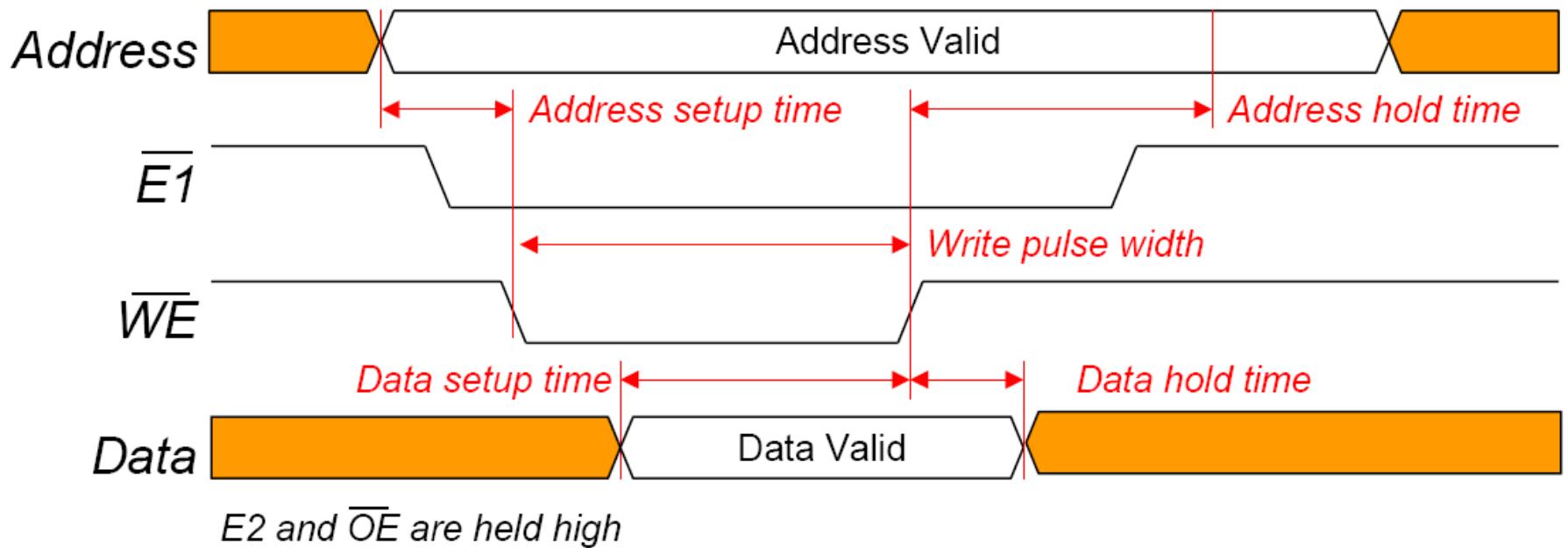
When low (and chip is enabled), data bus is driven with value of selected memory location

NC	1	28	V _{CC}
A ₁₂	2	27	\overline{W}
A ₇	3	26	$E2$
A ₆	4	25	A ₈
A ₅	5	24	A ₉
A ₄	6	23	A ₁₁
A ₃	7	22	\overline{G}
A ₂	8	21	A ₁₀
A ₁	9	20	\overline{ET}
A ₀	10	19	DQ ₇
DQ ₀	11	18	DQ ₆
DQ ₁	12	17	DQ ₅
DQ ₂	13	16	DQ ₄
V _{SS}	14	15	DQ ₃

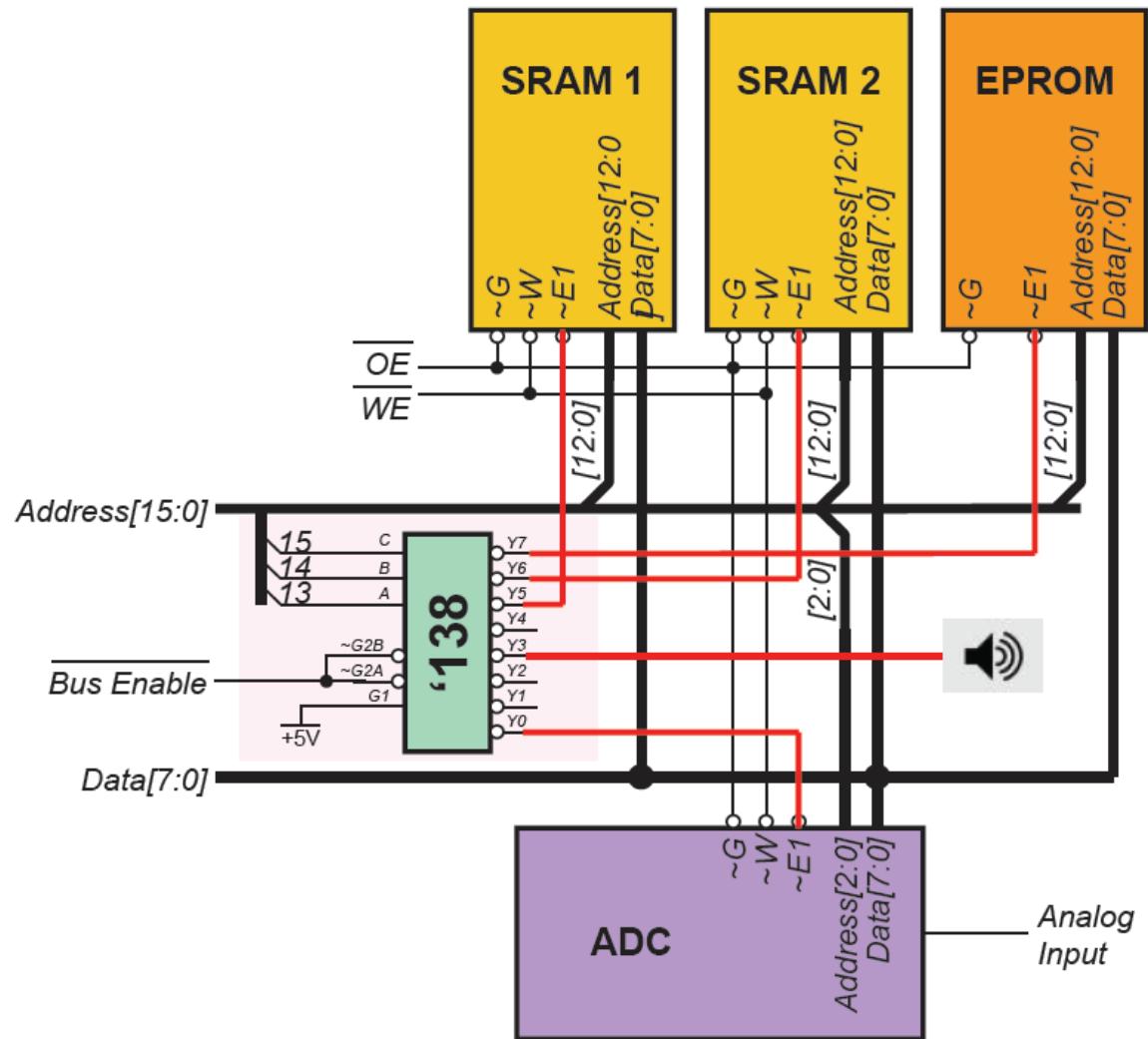
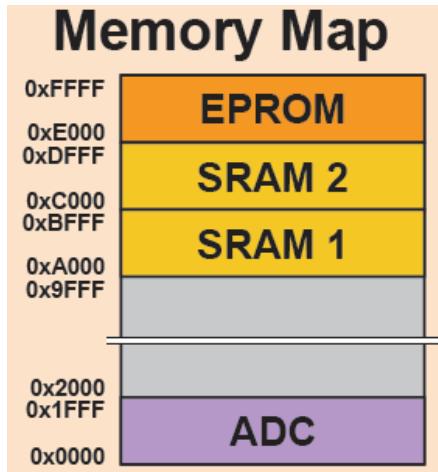
Čtení asynchronní SRAM



Zápis do SRAM

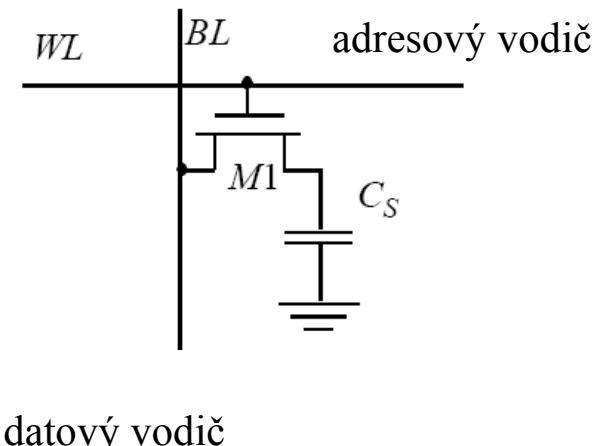


Adresování dvou a více pamětí



Dynamická paměť (DRAM) – od 70. let

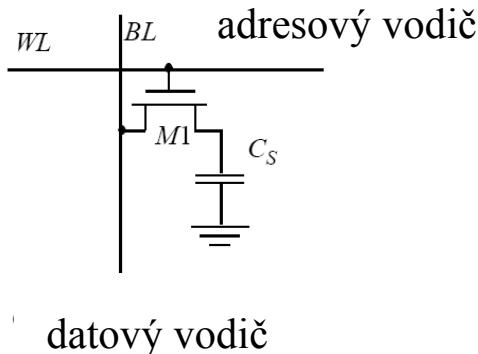
Informace je uložena pomocí elektrického náboje na kondenzátoru.



Při **zápisu** se na adresový vodič (WL – word line) přivede hodnota logická 1 => tranzistor se otevře. Na datovém vodiči (BL – bit line) je umístěna zapisovaná hodnota (např. 1), tato hodnota projde přes otevřený tranzistor a nabije kondenzátor. V případě zápisu nuly dojde pouze k případnému vybití kondenzátoru (pokud byla dříve v paměti uložena hodnota 1).

Při **čtení** se využívá obvodu nazývaného diferenciální zesilovač, který umí porovnat napětí „přečtené“ z kondenzátoru s rozhodovací úrovní (typicky $U_{cc}/2$), výsledek interpretovat jako log. 0 či log. 1 a dále tuto hodnotu uložit zpět (tzv. refresh). K přečtení informace z dané buňky je přivedena log. 1 na adresový vodič, čímž je sepnut tranzistor, napětí na kondenzátoru se objeví na datovém vodiči a je zpracováno diferenciálním zesilovačem. Kondenzátor je tímto vybit a přečtenou hodnotu je tedy nutno znovu uložit (čtení je destruktivní operace).

DRAM – obnova informace

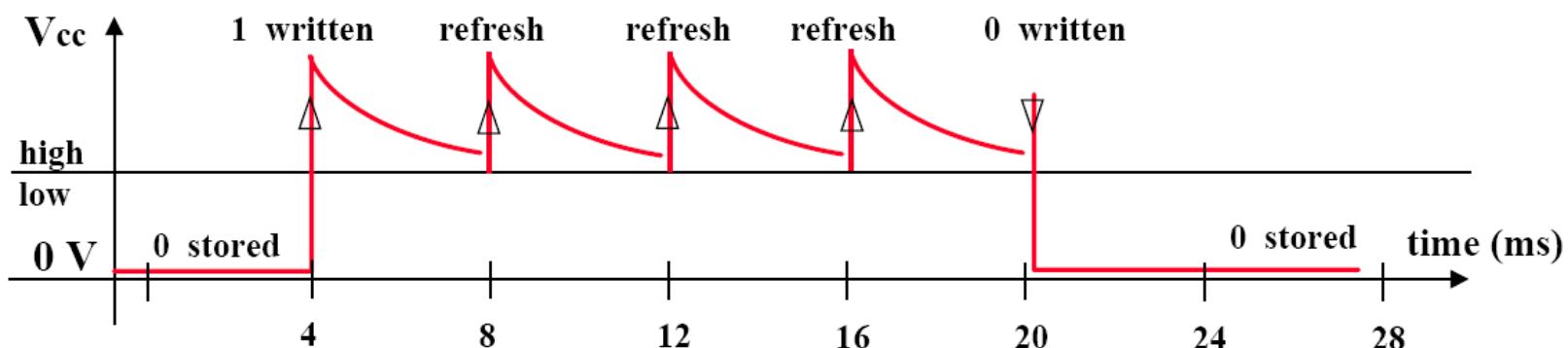


Náboj na kondenzátoru se **vybíjí** i v době, kdy je paměť připojena ke zdroji elektrického napájení => je nutné periodicky provádět tzv. **refresh**, tj. obnovení paměťové buňky. Tuto funkci plní některý z obvodů čipové sady.

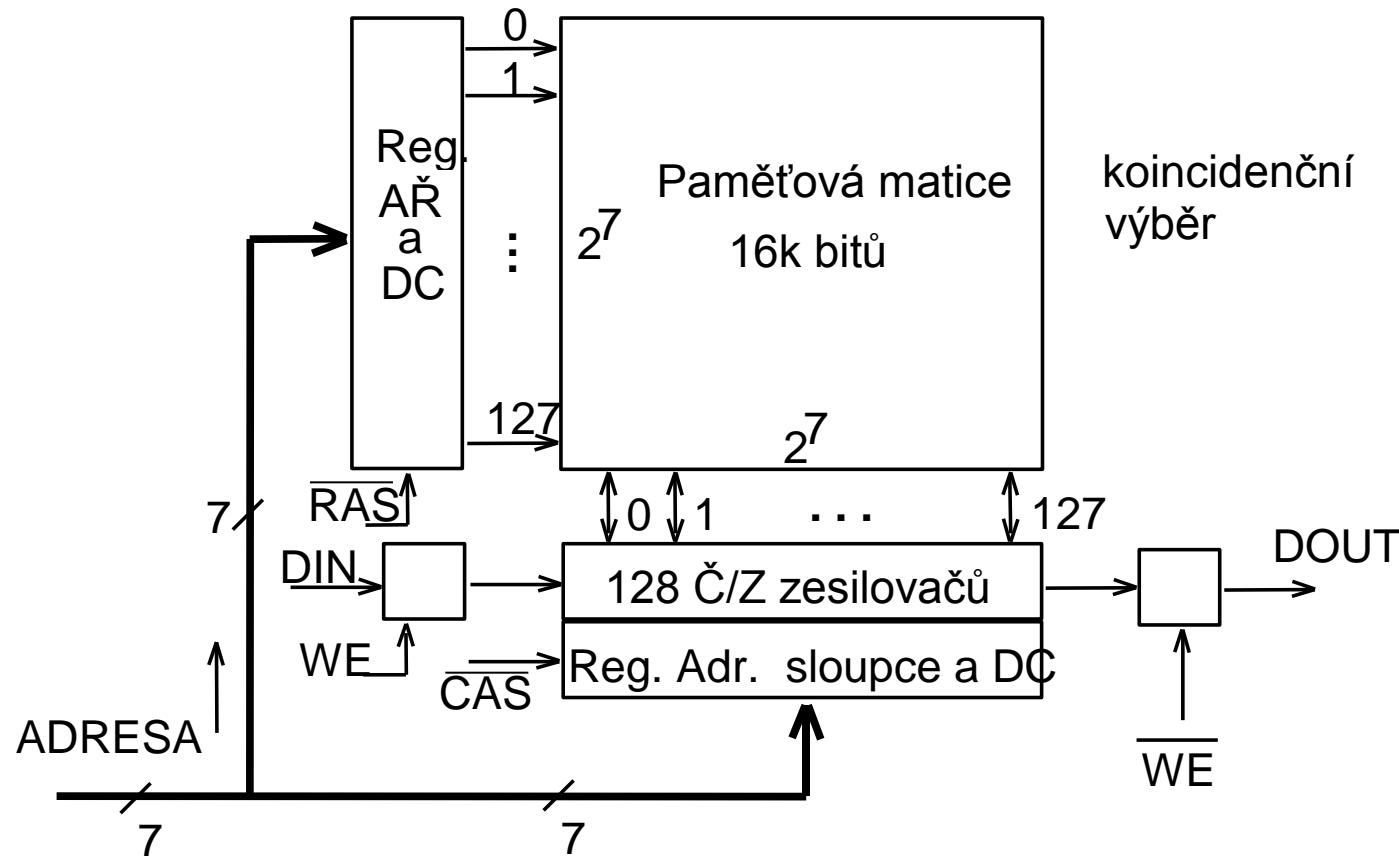
datový vodič

Jednou za dobu t_r (t_r je např. 4 ms) dojde k obnově informace v paměti. Zdegenerovaný obsah řádku je načten do registru a vzápětí je obsah registru (již s upravenými log. úrovněmi) zapsán zpět do buněk paměti.

Obrázek ukazuje průběh napětí na kondenzátoru po zápisu log.1.

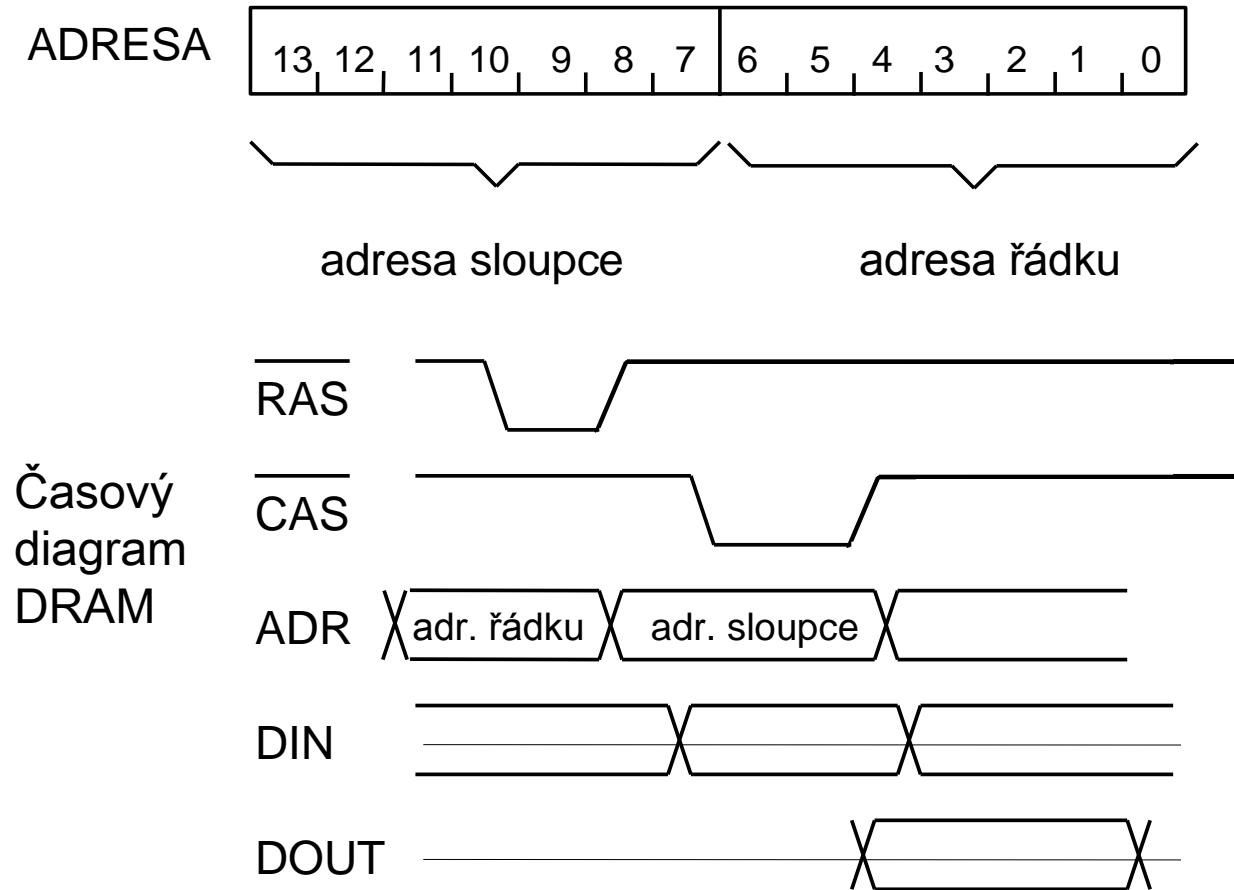


Struktura DRAM 16K x 1bit



Předávání adresy DRAM

Pro minimalizaci počtu vývodů pouzder DRAM se ustálilo předávání adresy *multiplexním způsobem* nadvakrát:



Komentář k adresování DRAM

Nejdříve se předává **adresa řádku AŘ**; zápis této části adresy do registru AŘ z adresové sběrnice ADR je řízen signálem **RAS** - *Row Address Select*.

Poté se na adresové sběrnici ADR objeví **adresa sloupce AS** a ta se zapíše do registru AS signálem **CAS** - *Column Address Select*. Jde-li o zápis, musí být na datovém vstupu DIN ve stanoveném intervalu platný datový bit, který se zapíše do adresovaného místa. Jde-li o čtení, pak za dobu danou katalogovými hodnotami se na datovém výstupu DOUT objeví přečtený bit. Funkce čtení/zápis se řídí signálem WE (často se používá označení R/-W).

Adresový výběr provádějí dva adresové dekodéry s výstupem v kódu 1 z 128, a *koincidence* (tedy logický součin) *aktivovaného řádkového a sloupcového vodiče* určí adresované paměťové místo (1 z 16 384). *Koincidenčním výběrem* se výběr paměťového místa z paměťové matice výrazně zjednoduší.

Řízení obnovy dat

Mechanismus obnovy musí zajistit, aby se před uplynutím zaručené doby uchování informace adresovala všechna paměťová místa. Základem *řadiče obnovy* je čítač adresy řádku, který inkrementuje po 1 a po naplnění čítá znovu od nuly. Pro pouzdro DRAM 16K x 1 je čítač obnovy sedmibitový.

Obnova dat se zajišťuje většinou jako vnější obnova, speciální paměti mají vnitřní mechanismus obnovy, tedy obvody pro řízení obnovy mají přímo na čipu. Vnější obnova je uspořádána jako

- rozložená
- dávková
- transparentní
- během normální činnosti

Obnova informace - komentář

U **rozložené obnovy** se spouštějí pravidelně cykly obnovy tak, aby se v daném intervalu adresovaly všechny paměťové buňky.

U **dávkové obnovy** následují všechny obnovovací cykly těsně za sebou.

U **transparentní obnovy** se využívá volných intervalů v činnosti paměti, takže obnova pak nezdržuje činnost procesoru. Tento způsob je však možno použít jen někdy.

Obnovy **normální činností** se využívá tam, kde je zaručeno adresování všech paměťových míst normální činností, např. u VIDEO paměti, ze které se cyklicky čtou data pro zobrazení na monitoru.

Časové diagramy rozložené a dávkové obnovy jsou na obr. a), b). Cykly běžné činnosti paměti jsou označeny N, obnovovací cykly jsou označeny symbolem R.

a) rozložená

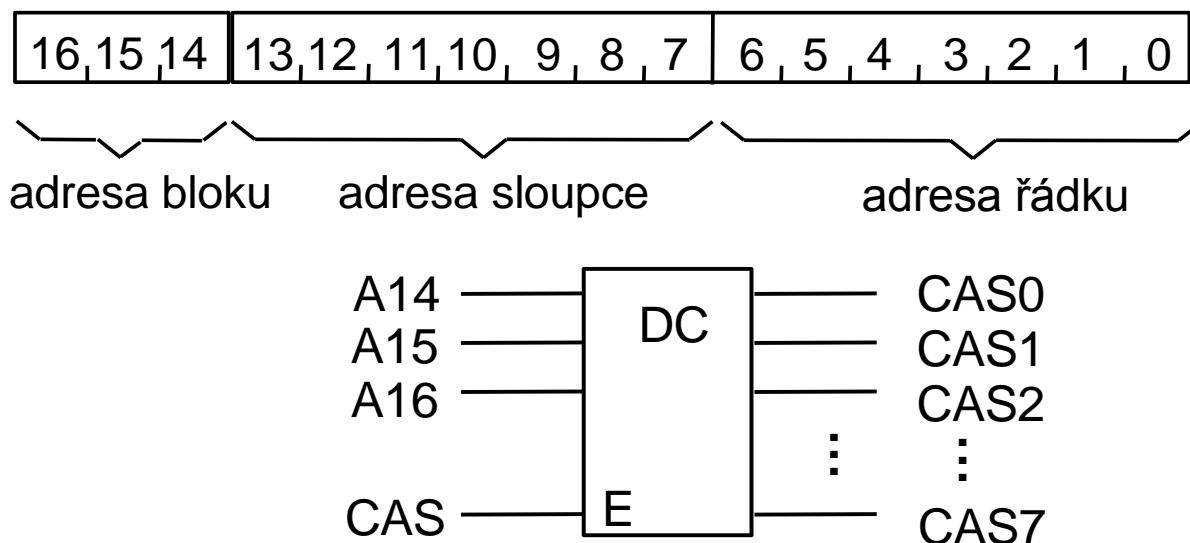


b) dávková

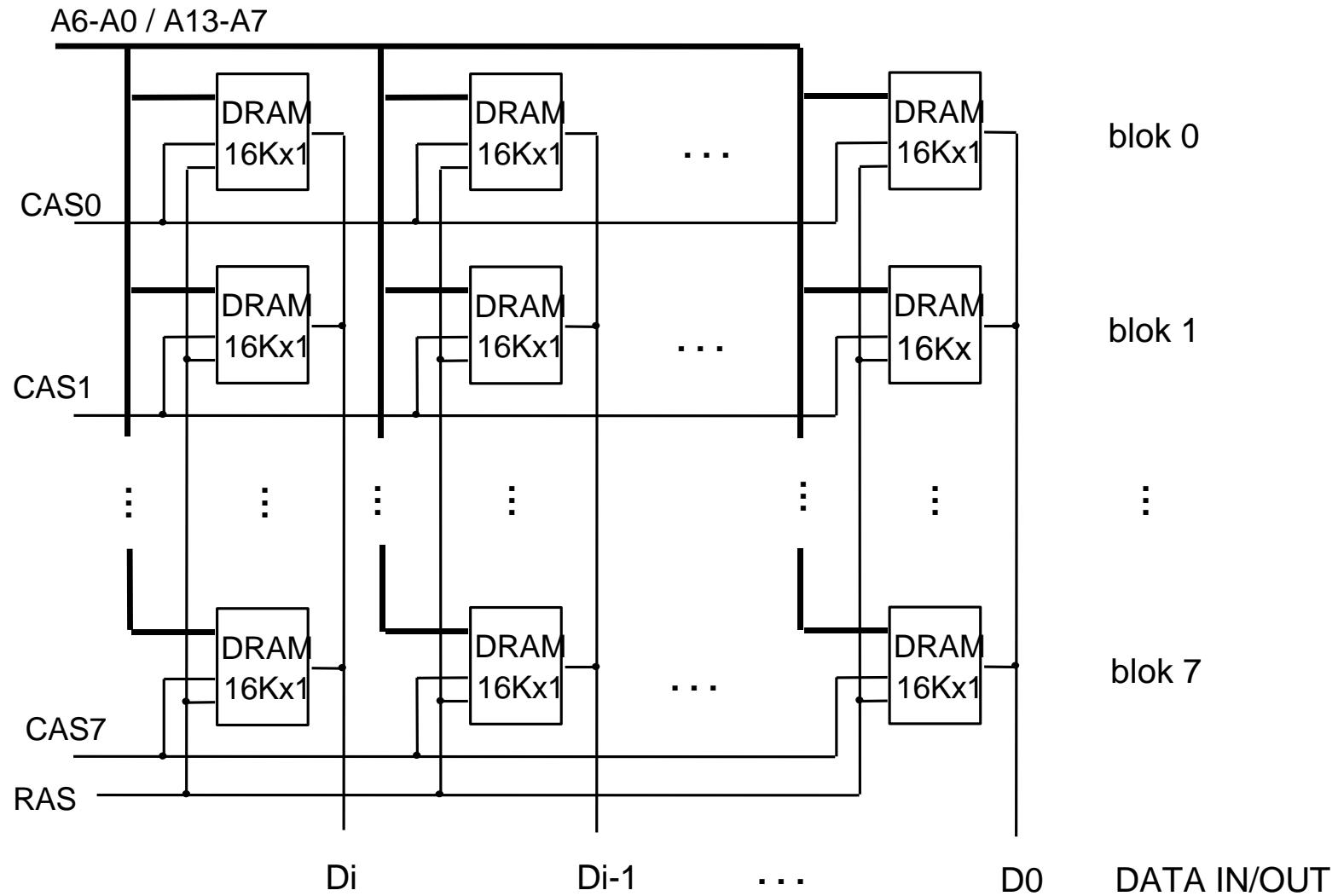


Bloková struktura DRAM (1)

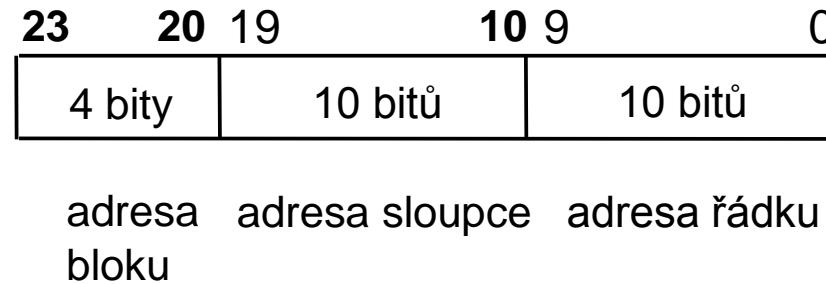
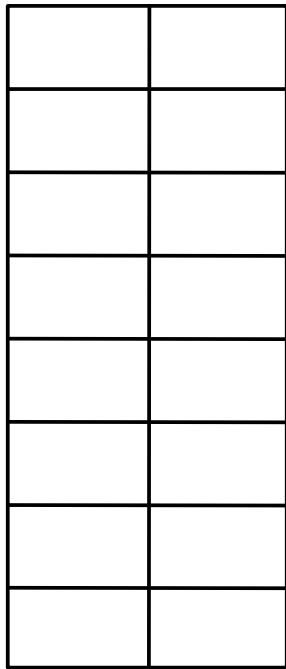
Zvětšení kapacity paměti (adresového prostoru) se provádí uspořádáním paměti do **bloků**. Na obrázku je naznačena *rozšířená adresa* a *dekodér adresy bloku*. Zde je použito jedno z více možných uspořádání. Protože u DRAM nejsou k dispozici výběrové signály *Chip Select*, jsou pomocí dekodéru vytvářeny v kódu 1 z n *selektivní signály CAS_i*. Signál RAS je rozveden do všech bloků. Výsledná bloková struktura paměti je na dalším obrázku. Zvětšování *šířky datového slova* je omezeno pouze výkonem výstupů zdrojů signálů CAS_i a RAS.



Bloková struktura DRAM (2)



Př. Topologie čipu DRAM 16 M x 1 bit

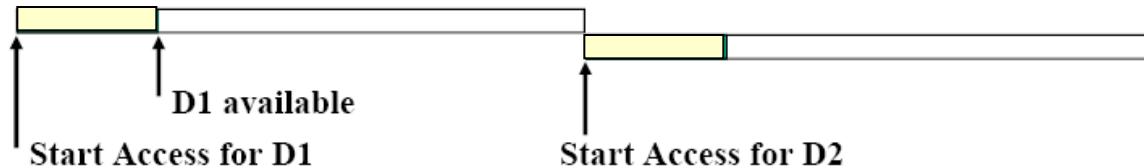


Blokový režim paměti:

Jeden blok má kapacitu 1 Mbit, třetí rozměr adresy má 4 bity. **Přepnutí bloku** změnou nejvyšších čtyř bitů (bity 20 až 23) beze změny adresy řádku a sloupce proběhne rychleji, než výběr podle nové adresy. Toho se využívá u většiny pamětí zavedením tzv. *blokového* nebo též *stránkového režimu*. Po běžné sekvenci nastavení adresy pomocí signálů RAS a CAS se impulsním průběhem na CAS adresují postupně další bloky (hodnota RAS zůstává nezměněna).

Princip prokládání paměťových operací

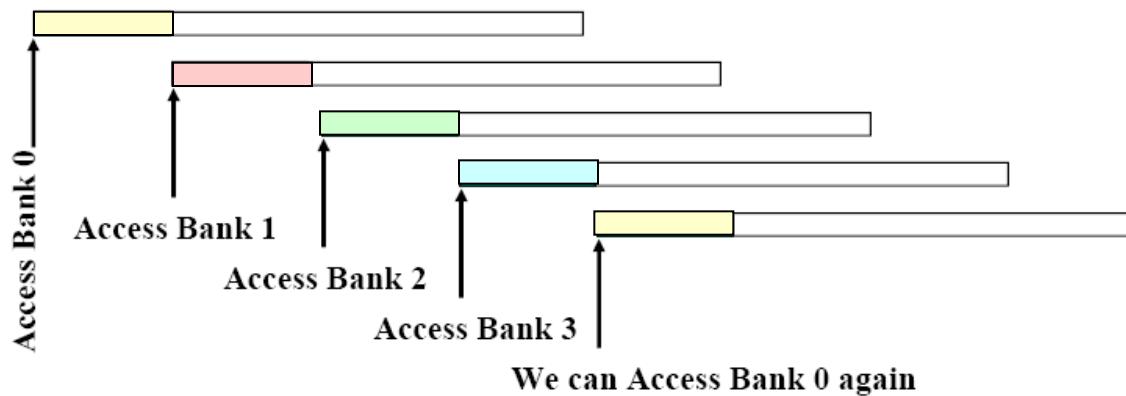
Bez prokládání



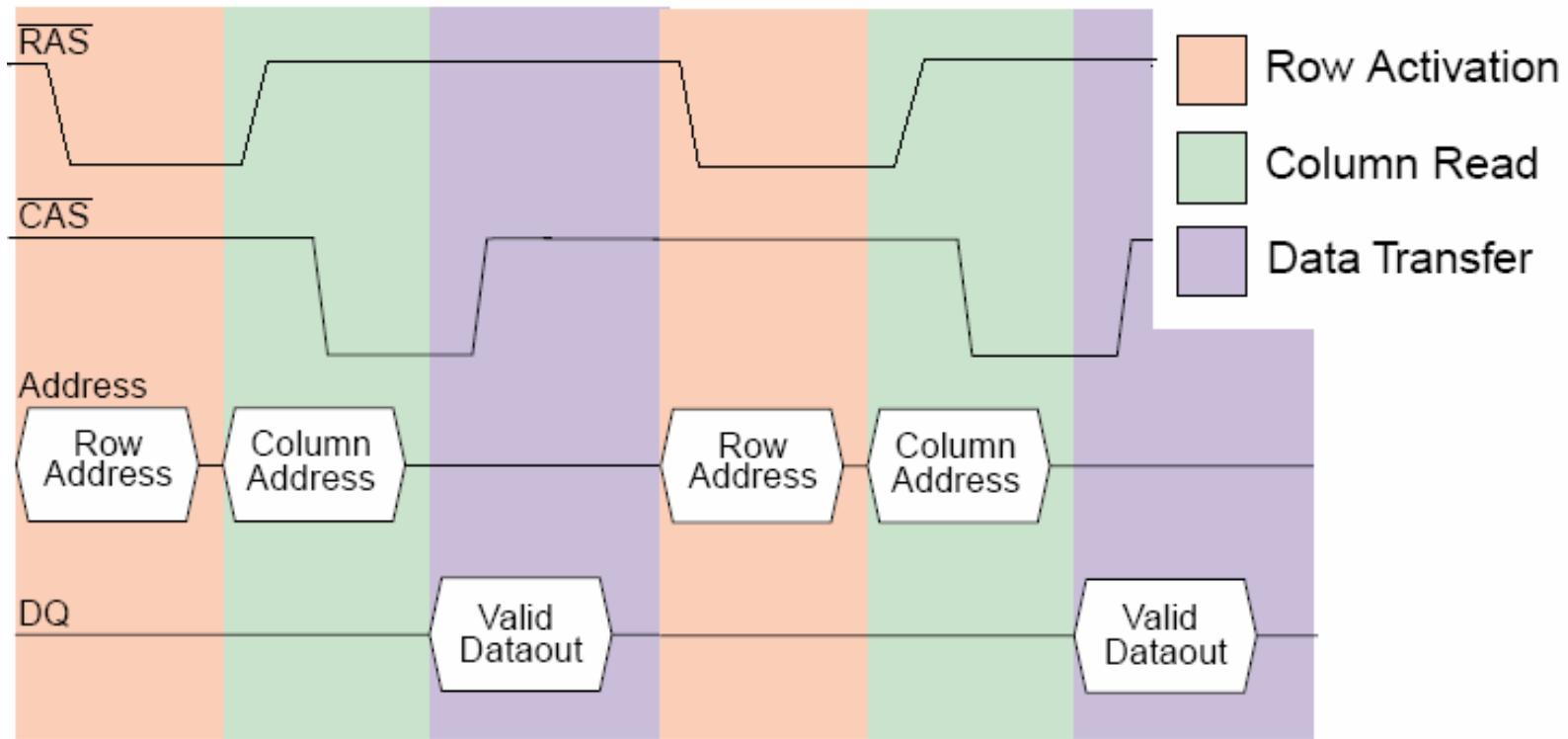
Bank Bez prokládání S prokládáním

Bank	Bez prokládání	S prokládáním
0	1	1
1	2	5
2	3	9
3	4	13
4	5	2
5	6	6
6	7	10
7	8	14
8	9	3
9	10	7
10	11	11
11	12	15
12	13	4
13	14	8
14	15	12
15	16	16

Čtyřcestné prokládání – urychlení 4x



Časování základní varianty DRAM



Neustálé střídání RAS' a CAS' vede k tomu, že jen v určitých okamžicích (fialová barva na obrázku) dochází k přenosu dat do procesoru. Jak zvýšit propustnost? Změnit časování paměti!

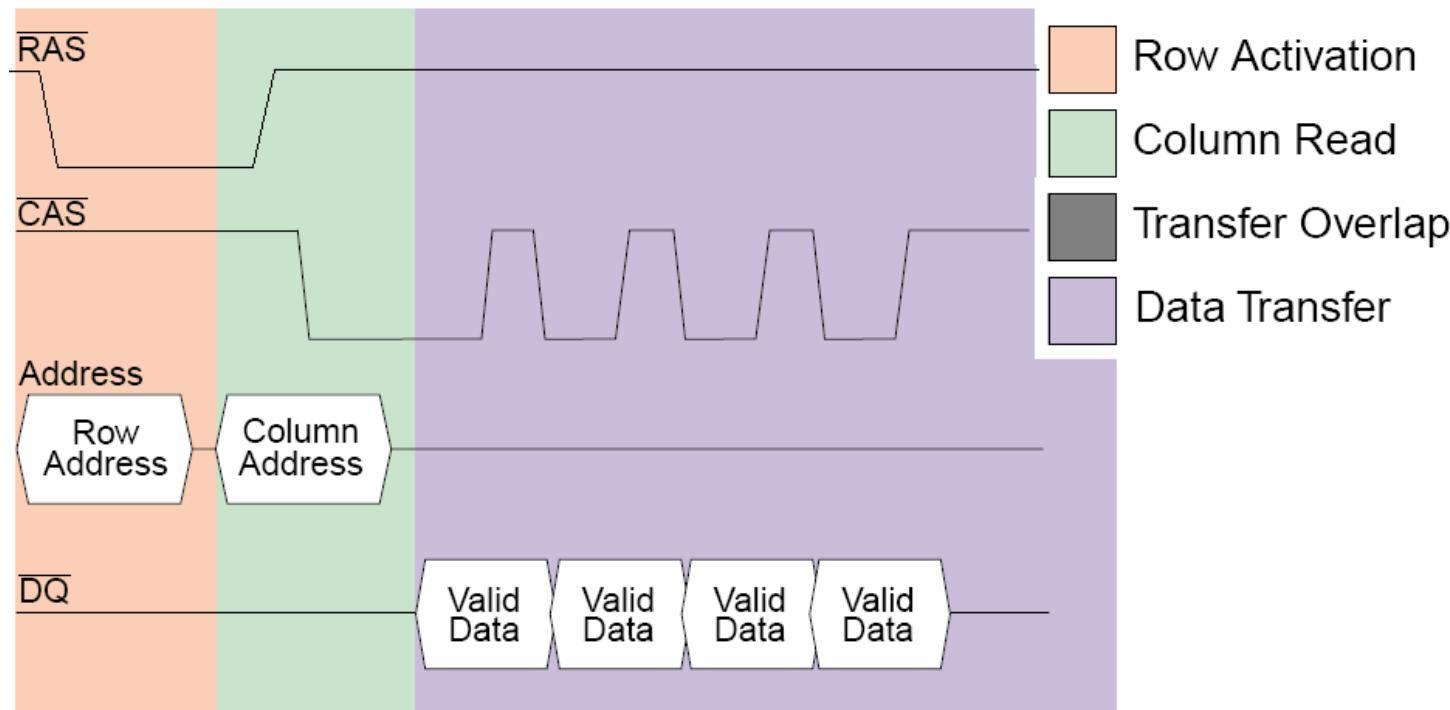
Vývoj pamětí DRAM

(princip uložení dat zůstává, paměti se liší v zásadě pouze frekvencí a časováním)

- Fast page mode DRAM (**FPM DRAM**)
 - umožňuje realizovat čtení tak, že je nastaven ROW na určitou hodnotu a přičítá se hodnota COL
- Extended data out DRAM (**EDO DRAM**)
 - přidán latch - prodlužuje se doba, po niž jsou přečtená data k dispozici na datové sběrnici => řadič paměti má více času na to, aby předal data přes sběrnici do procesoru.
- Burst Enhanced Data-Out DRAM (**BEDO DRAM**)
 - interní čítač adres urychlí operace s pamětí (4 adresy v dávce)
- Synchronous DRAM (**SDRAM**)
 - jiný princip oproti předchozím, synchronizace paměti, ovládání příkazy
- RamBus™ DRAM (**RDRAM**)
 - speciální typ paměti – seriové propojení, sběrnice pouze 16 bitů
- Double data-rate synchronous DRAM (**DDR SDRAM**)
 - přenosové děje se ovíjejí od nástupné i sestupné hrany synchronizačních pulsů – je tak možné dvakrát zrychlit přenos
- DDR2 SDRAM, DDR3, DDR4

BEDO – Burst Enhanced Data-Out DRAM

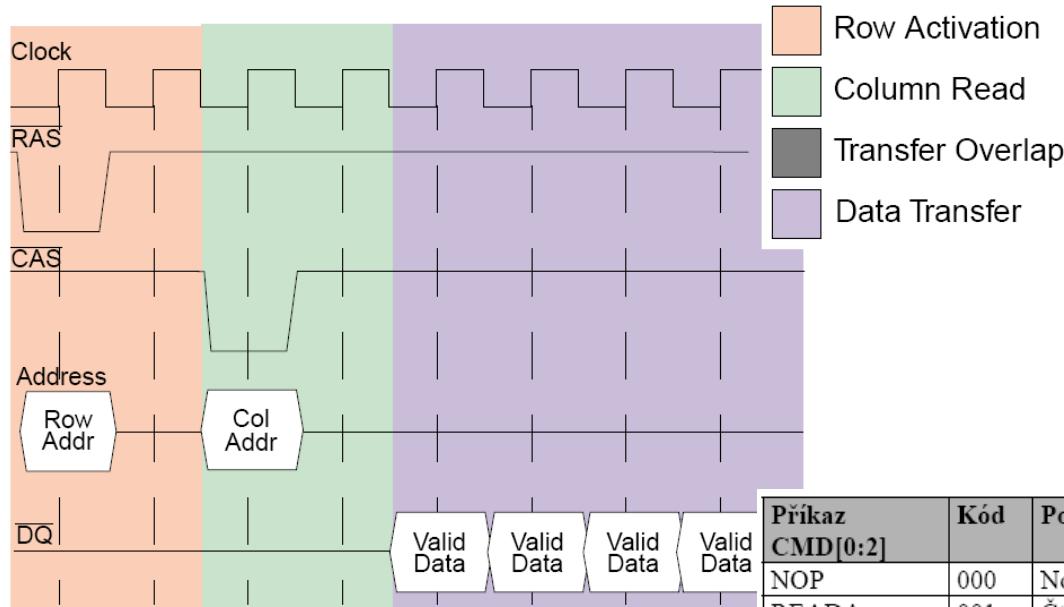
Paměti typu BEDO obsahují **interní čítač adres**, takže se do adresového registru zavádí pouze adresa první, zbývající čtyři se odvodí v čítači postupnou inkrementací => redukce objemu komunikace mezi řadičem paměti a pamětí (**burst = dávkové zpracování**). BEDO urychluje o 30% oproti EDO.



Zrychlení získáno za cenu, že data nejsou čtena z libovolných adres.

Synchronní DRAM - SDRAM

(1993, 66 – 133 MHz, až 512MB, 3.3V)



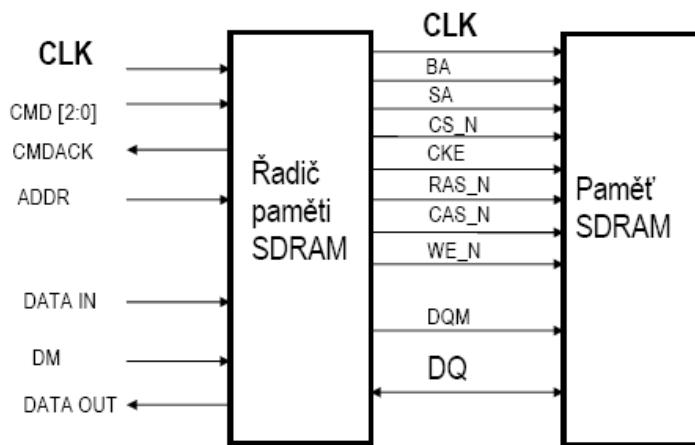
Synchronní DRAM – operace jsou synchronizovány s ext. sběrnicí
Dávkový (burst) režim, podobně jako BEDO.

Paměť SDRAM je řízena kombinací signálů RAS, CAS a WE, které představují kód příkazu. Každou nástupnou hranou signálu CLK jsou tyto signály dekódovány a SDRAM pak provede požadovanou funkci.

Příkazy přijímané řadičem paměti a jejich kódy

Příkaz CMD[0:2]	Kód	Popis činnosti
NOP	000	No operation.
READA	001	Čti SDRAM + auto precharge
WRITEA	010	Piš do SDRAM + auto precharge
REFRESH	011	refresher
PRECHARGE	100	Uzavří všechny banky
LOAD MODE	101	Zapiš do registru mode
LOAD REG1	110	Zapiš do konfiguračního registru R1
LOAD REG2	111	Zapiš do konfiguračního registru R2

Příkazy generované řadičem pro SDRAM



Příkaz	Zkratka	RASN	CASN	WEN
No operation	NOP	H	H	H
Activate	ACT	L	H	H
Read	RD	H	L	H
Write	WR	H	L	L
Burst terminate	BT	H	H	L
Precharge	PCH	L	H	L
Autorefresh	ARF	L	L	H
Load mode register	LMR	L	L	L

Burst režim (dávkový režim)

- Umožňuje čtení nebo zápis **bloku dat** o předem známé délce.
- V bloku jsou obsažena data z „nějak sousedních“ adres (např. řádek 100, sloupce 1, 2, 3, 4), ne z libovolných adres.
- Díky synchronnímu provozu stačí pouze inicializovat přenos (nastavit adresu řádku i adresu sloupce) a poté je v každém taktu provedeno čtení nebo zápis jedné hodnoty.
- Prvotní zpoždění způsobené výběrem řádku představuje tzv. **latenci paměti**.
- Dávkový přenos představuje hlavní přínos synchronních DRAM oproti asynchronním DRAM.
- V dávkovém režimu paměť pracuje na své maximální možné rychlosti. Je žádoucí používat přenosy bloků dat a nikoli čtení či zápis na náhodné adresy.

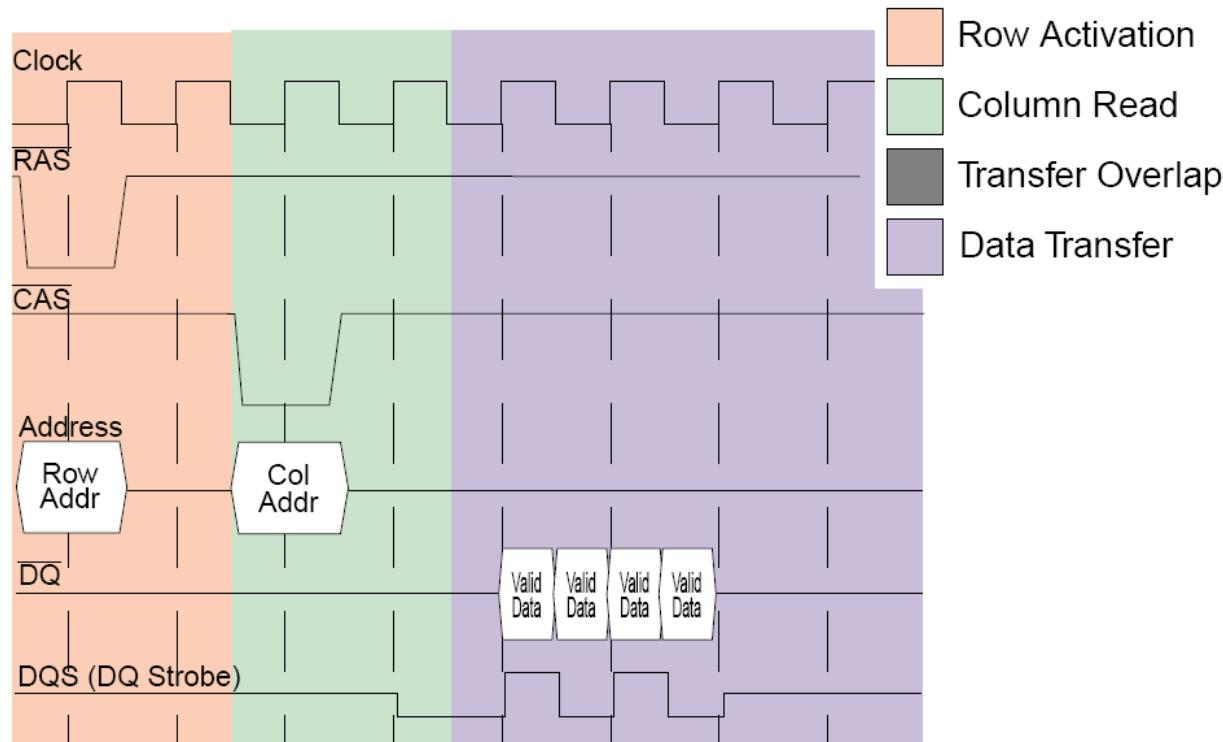
DDR SDRAM – Double Data Rate

(r. 2000, 200 – 400 MHz, 2,5V, 64MB - 2GB)

DDR - data jsou přenášena na **náběžné i sestupné hraně** hodinového pulsu (propustnost: 1,6 - 3,2 GB/s).

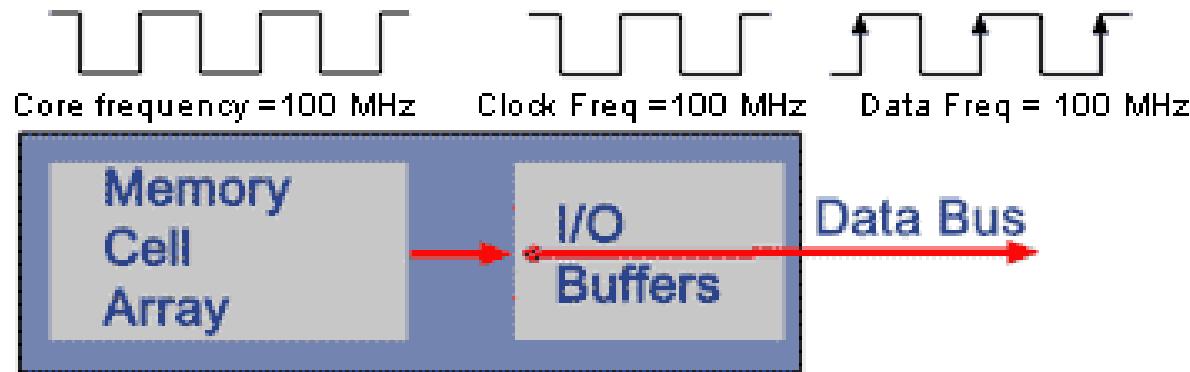
Startovací frekvence jsou od 266 MHz, které se označují standardem DDR266.

Maximální rychlosti technologie DDR se pohybují okolo 600 MHz efektivně, ale obvykle je pro jejich dosažení nutné značně vysoké napájecí napětí (i přes 3 V).

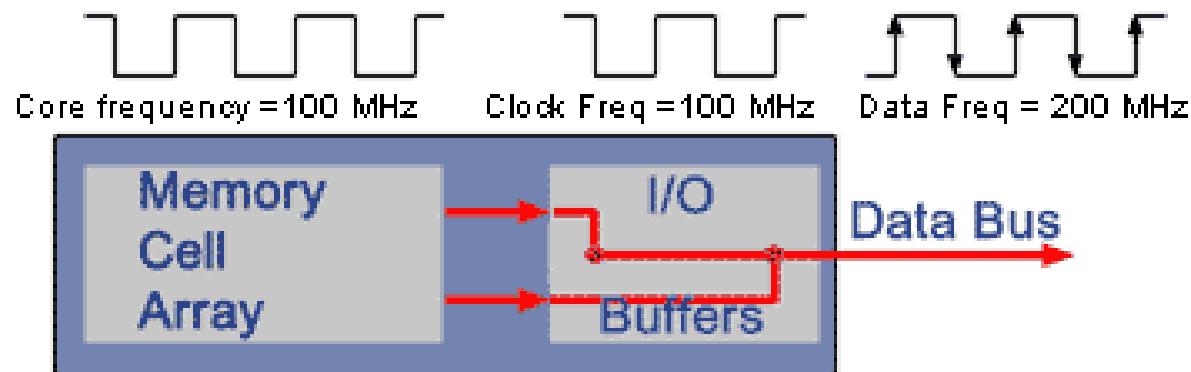


SDRAM vs DDR

SDRAM



DDR I



DDR2 a DDR3

DDR2-400 a DDR2-800

pracovní frekvence: 400 – 800 MHz, propustnost: 3,2 – 6,4 GB/s, 1,8 V,
128 MB až 8 GB

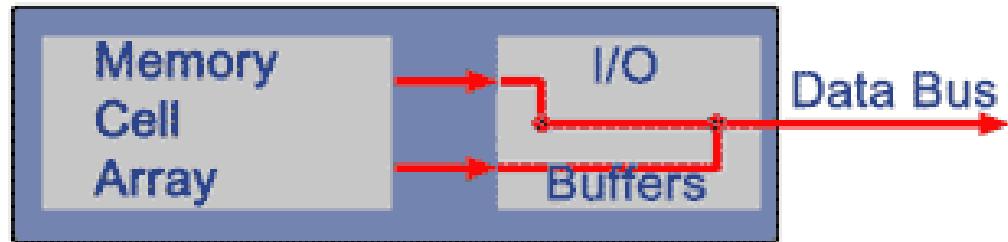
DDR3-800 a DDR3-1600

pracovní frekvence 800 – 1600 MHz, propustnost 6,4 GB/s – 12,8 GB/s,
napájecí napětí 1,5 V
kapacita: 8 GB (Infineon)

DDR vs DDR2

DDR I

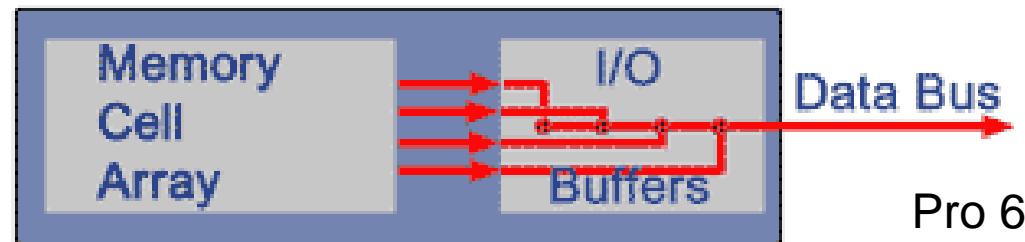
Core frequency = 100 MHz Clock Freq = 100 MHz Data Freq = 200 MHz



prefetch 2

DDR II

Core frequency = 100 MHz Clock Freq = 200 MHz Data Freq = 400 MHz



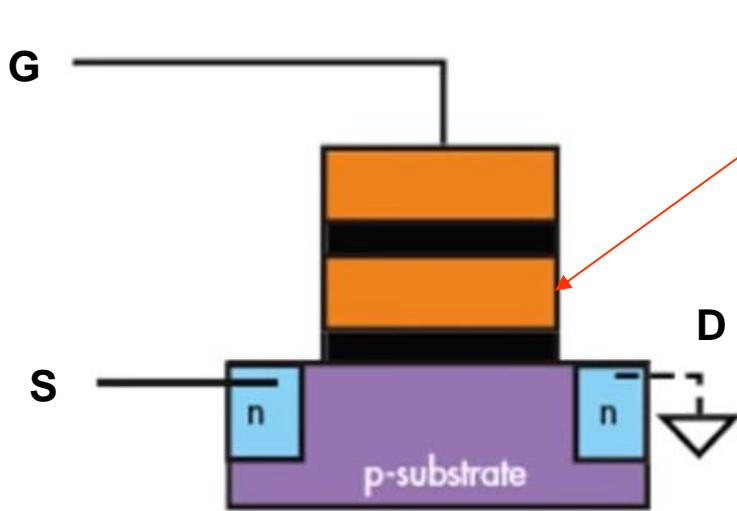
prefetch 4

Pro 64 b. sběrnici je max. propustnost:
100 MHz x 2 x 2 x 64/8 = 3,2 GB/s

Paměti Flash (princip objeven 1980)

- Nevolatilní semipermanentní paměť (bez napájení uchová data řádově roky)
- Paměťové buňky jsou tvořeny tranzistory s plovoucím hradlem (1 tranzistor na buňku) – levné, vysoká kapacita ~ stovky GB
 - Plovoucí hradlo uchovává informaci v podobě elektrického náboje
- Jedna buňka může uchovávat informaci jednoho bitu (Single Level Cell – SLC), případně i více bitů (Multi-Level Cell – MLC)
- Dva základní typy: NOR Flash (Intel, 1988), NAND Flash (Toshiba, 1989)
 - Liší se hlavně organizací paměťových buněk, způsobem adresování, řízení a rozhraním
- Před zápisem (programováním) je nutno provést mazání (paměti či bloku)

Tranzistor s plovoucím hradlem - princip



Informace je uchována v podobě přítomnosti či nepřítomnosti záporného náboje na plovoucím hradle. Princip činnosti flash je tedy založen na schopnostech řízené manipulace s tímto nábojem.

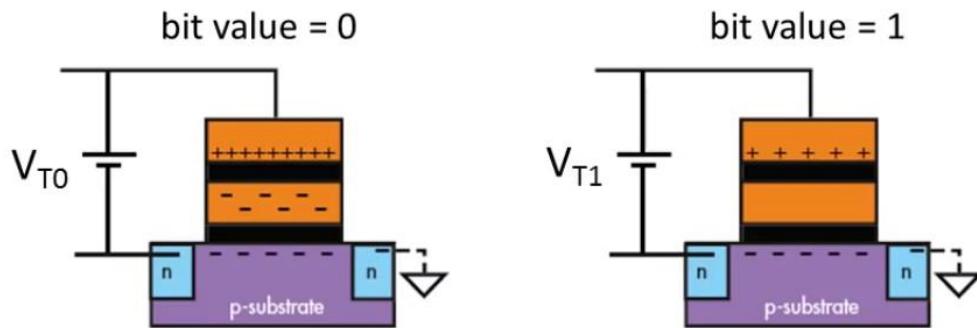
FGFET = Floating Gate FET

Předpokládejme, že FG bez náboje reprezentuje log. 1 („prázdná“ buňka – viz dálší slide).

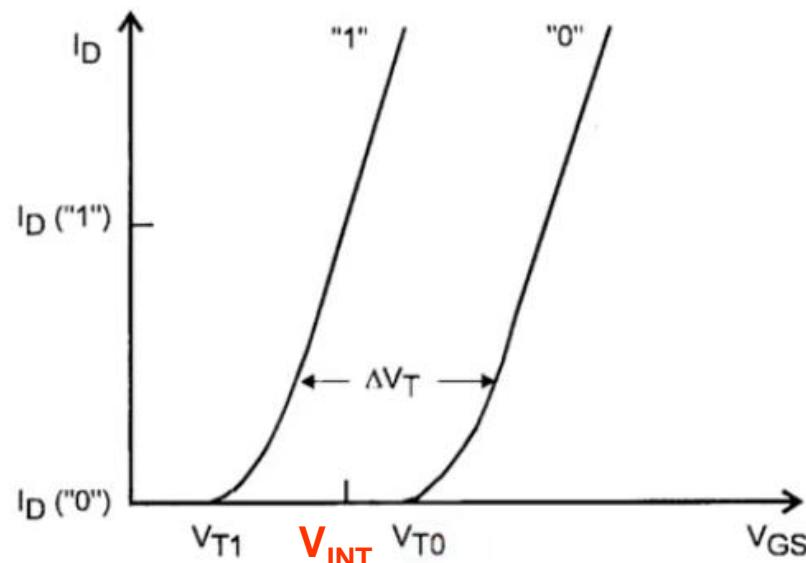
Sepnutím tranzistoru ($+V_{GS}$ dostatečné velikosti) jím začne procházet proud a je-li i V_{SD} dostatečně vysoké, některé elektrony začnou prostupovat izolační vrstvou do FG (hot electron inject.). Tranzistor je tak „naprogramován“ - do buňky je „zapsána“ log. 0.

„Vymazání“ buňky (nastavení do log. 1) se napětím opačné polarity $-V_{GS}$ (quantum tunneling).

Interpretace informace na FGFET



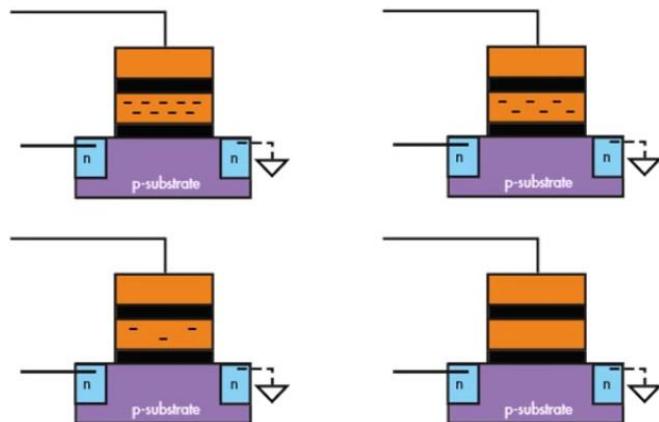
Přítomnost či nepřítomnost náboje na plovoucím hradle ovlivňuje prahové napětí V_T potřebné k otevření tranzistoru.



Položíme-li $V_{GS}=V_{INT}$, můžeme na základě proudu protékajícího tranzistorem (otevřený/uzavřený tranzistor) rozhodnout o log. hodnotě uloženého bitu (1 nebo 0) – operace **čtení**.

Single Level Cell (SLC):
Tranzistor bez náboje na plovoucím hradle si „pamatuje“ hodnotu log. 1 (ekvivalent „vymazané“ paměti), záporný náboj je interpretován jako log. 0.

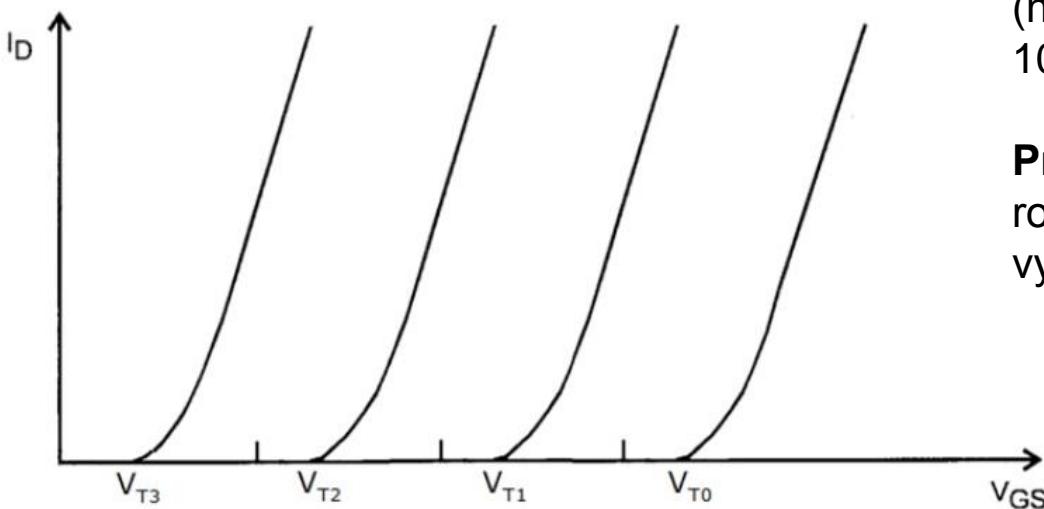
Koncept Multi-Level Cell



Dříve uvedený princip SLC lze zobecnit, když budeme rozlišovat více hodnot náboje na plovoucím hradle a podle toho i více hodnot proudu protékajícího tranzistorem – koncept **Multi-Level Cell (MLC)**.

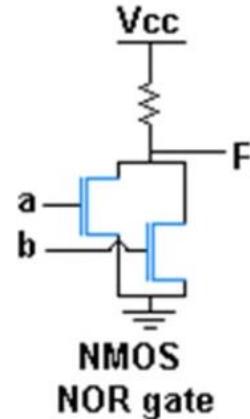
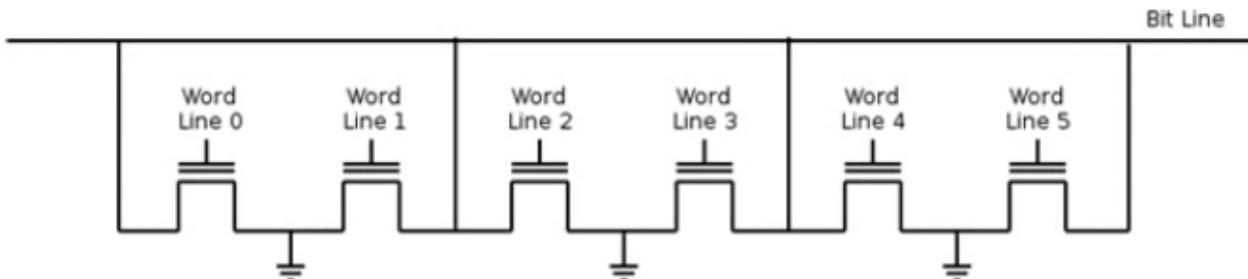
Tím je pak umožněno uložit v paměti flash více bitů na FGFET (např. 2 bity – kombinace 00, 01, 10, 11).

Problém: je třeba precizně rozlišovat více hodnot proudu, vyšší náchylnost k chybám.

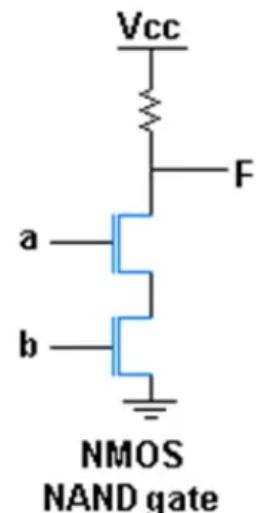
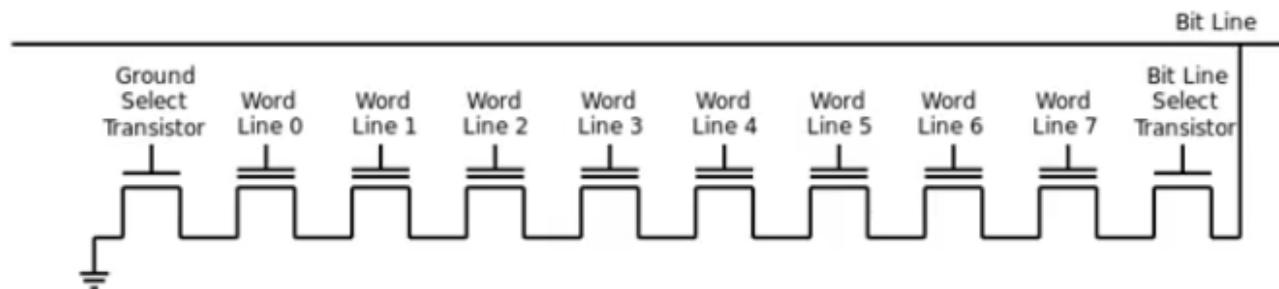


Typy Flash (podobnost s hradly NOR, NAND)

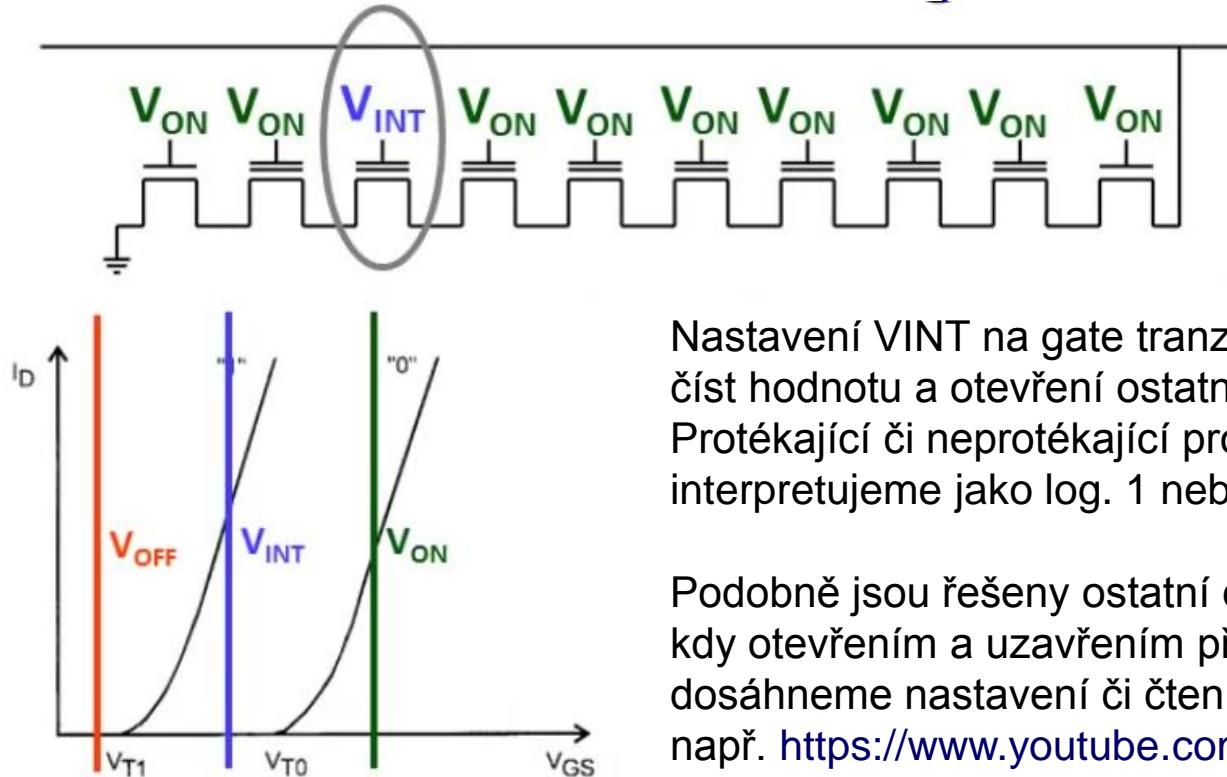
- **NOR Flash** – čtení/zápis po jednotlivých buňkách (k dispozici kompletní adresové a datové rozhraní), mazání jako celek či po blocích, pomalé. Náročné na plochu => nízká kapacita (řádově MB). Použití: paměti pro firmware, uložení programu v mikrokontrolérech, OS v mobilech apod.



- **NAND Flash** – manipulace po stránkách (~ 1000 bitů), komplexní řadič. Vysoká hustota a kapacita (řádově GB), rychlé operace čtení/zápis. Nižší spolehlivost, nutnost zavedení redundantních buněk a opravných kódů. Použití jako datová média – USB flash, SSD disky pro NB apod.



Příklad čtení z buňky NAND Flash



Nastavení V_{INT} na gate tranzistoru, z něhož chceme číst hodnotu a otevření ostatních tranzistorů v buňce. Protékající či neprotékající proud tranzistory interpretujeme jako log. 1 nebo 0.

Podobně jsou řešeny ostatní operace (též v NOR flash), kdy otevřením a uzavřením příslušných tranzistorů dosáhneme nastavení či čtení určité hodnoty (více viz např. <https://www.youtube.com/watch?v=s7JLXs5es7I>)

Srovnání NOR a NAND flash z pohledu základních parametrů

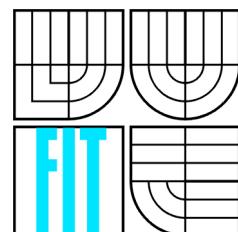
	NOR Flash (typ.)	NAND Flash (typ.)
Random Read Access Time	80 ns / 16 bit word	15 us / 528 byte page
Sector Read Speed	13.2 MB/s	12.7 MB/s
Program Speed	1 word / 10 us = 0.2 MB/s	528 bytes / 250 us = 2.1 MB/s
Erase Speed	64 kB / 0.8 sec = 0.08 MB/s	16 kB / 3 ms = 5.3 MB/s

Příště: Rychlá vyrovnávací paměť cache,
paměťová hierarchie

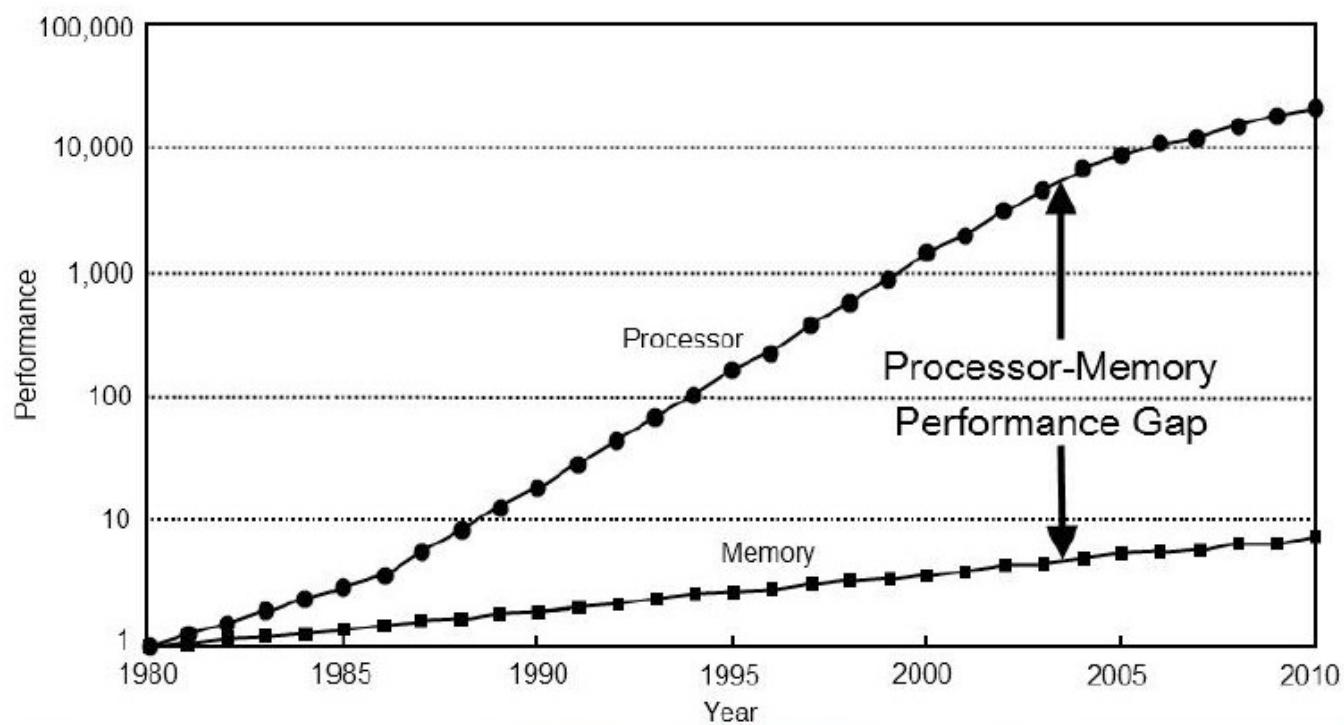
Paměťová hierarchie

INP 2015

FIT VUT v Brně



Výkonová mezera mezi CPU a pamětí



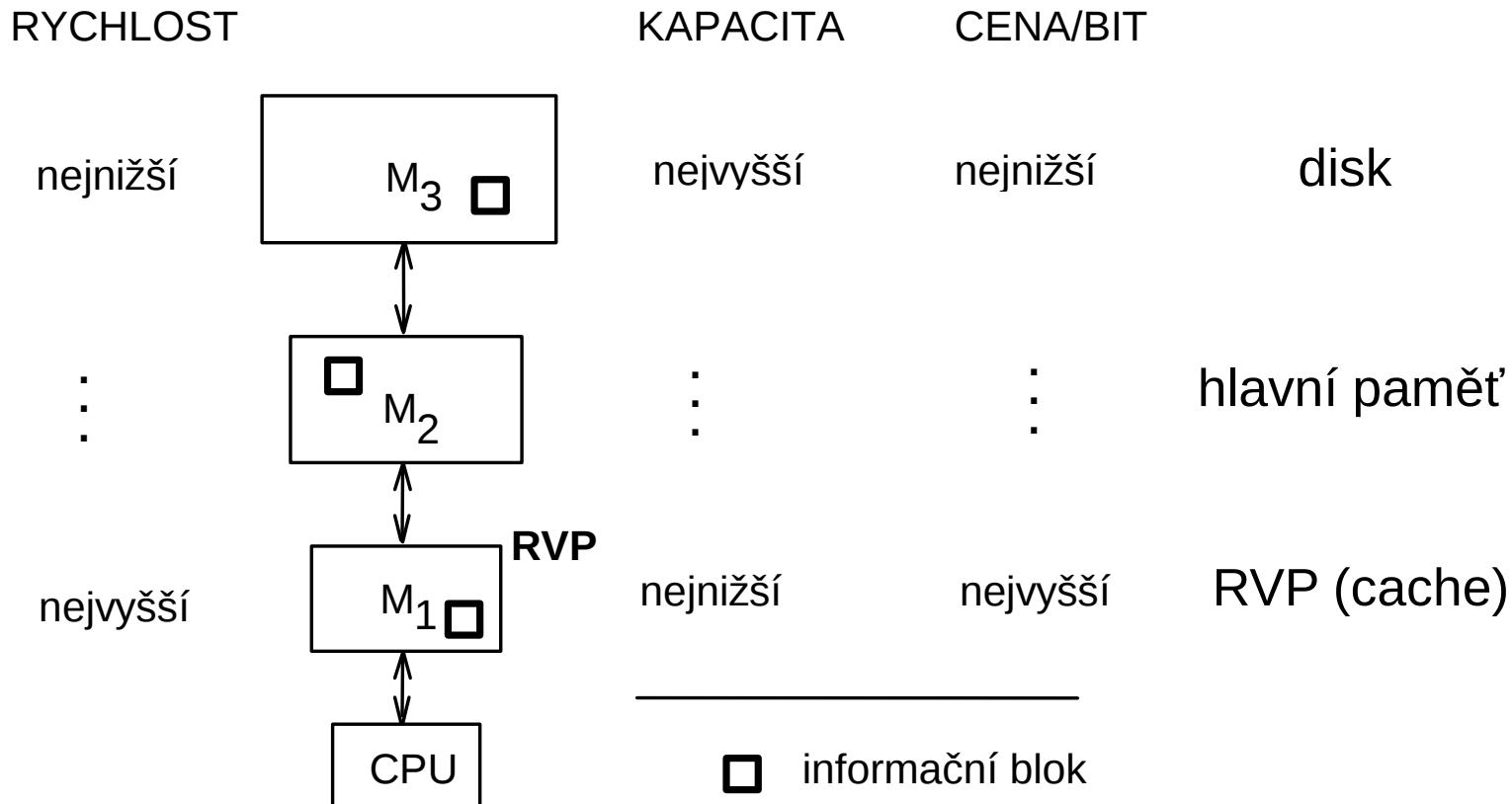
3

Slide 17

Computer architecture: a quantitative approach
By John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau

Kvůli výkonové mezeře není užitečné, aby procesor přímo pracoval s hlavní pamětí, která je dnes levná, ale pomalá. Mezi CPU a hlavní paměť je proto umístěna jedna nebo několik rychlých (ale malokapacitních) vyrovnávacích pamětí (RVP), angl. cache. Vznikne hierarchický paměťový systém. Předpokladem jeho funkčnosti je existence časové a prostorové lokality odkazů k paměťovým místům („80 % času stráví program pouze ve 20 % kódu“).

Základní vlastnosti hierarchického paměťového systému



Jistý datový nebo instrukční blok (blok I/D) pak může v systému existovat až ve třech kopíích.
(na obrázku předpokládáme existenci jedné rychlé vyrovnávací paměti).
V moderních počítačích existuje více úrovní RVP: L1, L2 i L3.

Typické parametry

Procesor/ Parametr	80286 (1982)	Core i7-4770K (2013)	Paměť	Latence
Jádra (vlákna)	1 (1)	4 (8)	CPU Registr	<= 1
CPU	8 MHz	3,5 GHz	L1 Cache	4
Hlavní paměť, frekvence	8 MHz	1,6 GHz	L2 Cache	11
Hlavní paměť, kapacita	128 KB	8 GB	L3 Cache	40
RVP	Není	L1: 4 x 64 KB L2: 4 x 256 KB L3: 8 MB	RAM	~200

RVP

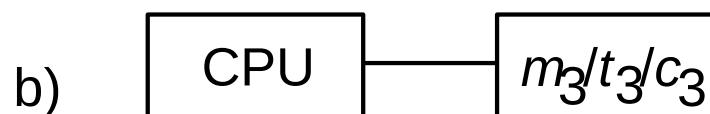
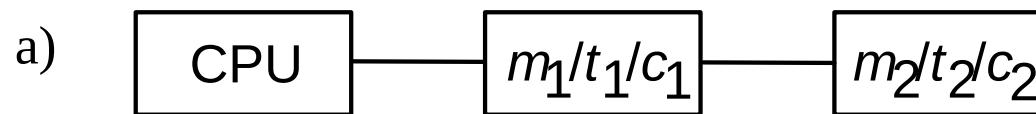
- RVP je rozdělena do **bloků** o konstantní velikosti (ideálně dle velikosti dat dodaných z hlavní paměti při blokovém přenosu).
 - Př. Vyrovnávací paměť o kapacitě 512 kB a velikosti bloku 32 bytů obsahuje celkem 16384 bloků.
- Hlavní paměť (např. DDR) je rozdělena na bloky o stejné velikosti. Těchto bloků je však mnohem více, než bloků ve vyrovnávací paměti => tudíž ne všechny bloky operační paměti mohou být v RVP.
- Musí existovat mechanismy umožňující potřebné bloky do RVP nahrávat a nepotřebné bloky odstraňovat.

Účinnost RVP

- Základní údaj o účinnosti RVP je **pravděpodobnost nalezení bloku** (p_{hit}),
 - resp. **pravděpodobnost neúspěchu** (miss rate), neboli **pravděpodobnost výpadku bloku** ($1 - p_{hit}$).
 - Tyto parametry mohou být definovány zvlášť pro čtení a zápis, pro data i instrukce (data hit/miss rate, instruction hit/miss rate, atd.)
- Doba potřebná k nalezení bloku je **přístupová doba** RVP (ale jen v případě, kdy blok v RVP je).
- V případě neúspěchu (blok v RVP není) se přičítá **ztrátová doba** (miss penalty), což je doba potřebná na přesunutí bloku z hlavní paměti.
 - Je daná dobou potřebnou k uvolnění místa v RVP, přístupovou dobou k prvnímu slovu požadovaného bloku ve vzdálenější paměti plus doba přenosu celého bloku.
- Cílem je navrhnut organizaci a správu RVP tak, aby hodnota hit rate byla co nejvyšší (pozor, vždy závisí i na datech/programech)!!!
 - v praxi $p_{hit} \sim 95\text{-}99\%$

Analýza hierarchické paměti z hlediska ceny a výkonu

- Máme dvě konfigurace počítače podle obrázku s parametry podle tabulky, kde m_i je kapacita paměti, t_i je doba přístupu a c_i je cena v centech/Kbit.



i	m_i [B]	t_i [ns]	c_i [centů/Kbit]
1	5K	100	35
2	2M	500	14
3	2M	300	20

Výpočet

Máme vypočítat průměrnou cenu na KB v obou konfiguracích a máme určit, kdy bude výhodnější varianta a).

Střední ceny paměti v obou konfiguracích jsou ($1B = 9b$, 100 centů = 1 \$)

$$C_a = 9(m_1c_1 + m_2c_2)/100(m_1 + m_2) \text{ \$/KB}$$

$$C_b = 9c_3/100 \text{ \$/KB}$$

V konfiguraci a) máme pravděpodobnost úspěchu při čtení z M_1 danou p_h . Zjednodušeně můžeme napsat výraz pro střední dobu přístupu

$$t_a = t_1p_h + t_2(1 - p_h)$$

Má-li být konfigurace a) výhodnější než b) z hlediska výkonu, musí platit

$$t_a \leq t_3, \text{ tedy}$$

$$t_3 \geq t_1p_h + t_2 - t_2p_h$$

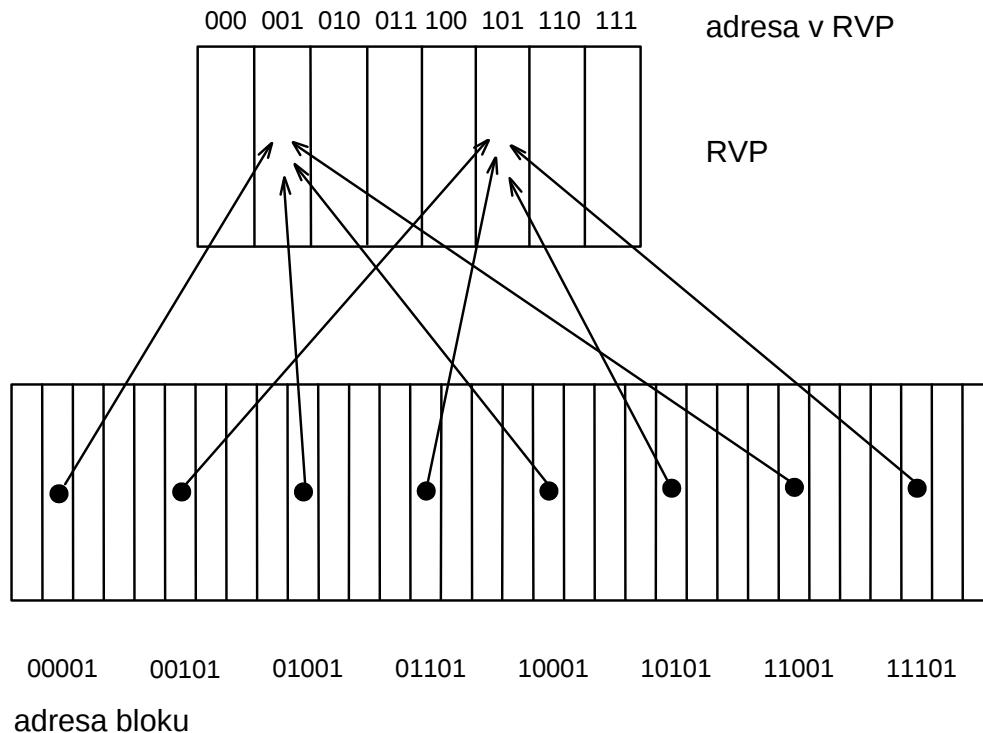
Odtud dostáváme podmínu pro pravděpodobnost úspěchu

$$p_h \geq (t_3 - t_2)/(t_1 - t_2)$$

Po dosazení hodnot podle tabulky dostáváme

$$p_h \geq 0,5, \text{ což je velmi mírný požadavek.}$$

RVP s přímým mapováním



Př. RVP má 8 blokových rámců s adresami 000 až 111, adresa bloku je pětibitová. Adresa polohy bloku v RVP se určí podle nejnižších tří bitů.

$$\text{adresa v RVP} = \text{adresa bloku} \bmod \text{počet bloků v RVP}$$

Výhody: jednoduchý koncept

Nevýhody: dva bloky, které mají stejnou adresu v RVP, nemohou být současně v RVP

Činnost RVP s přímým mapováním

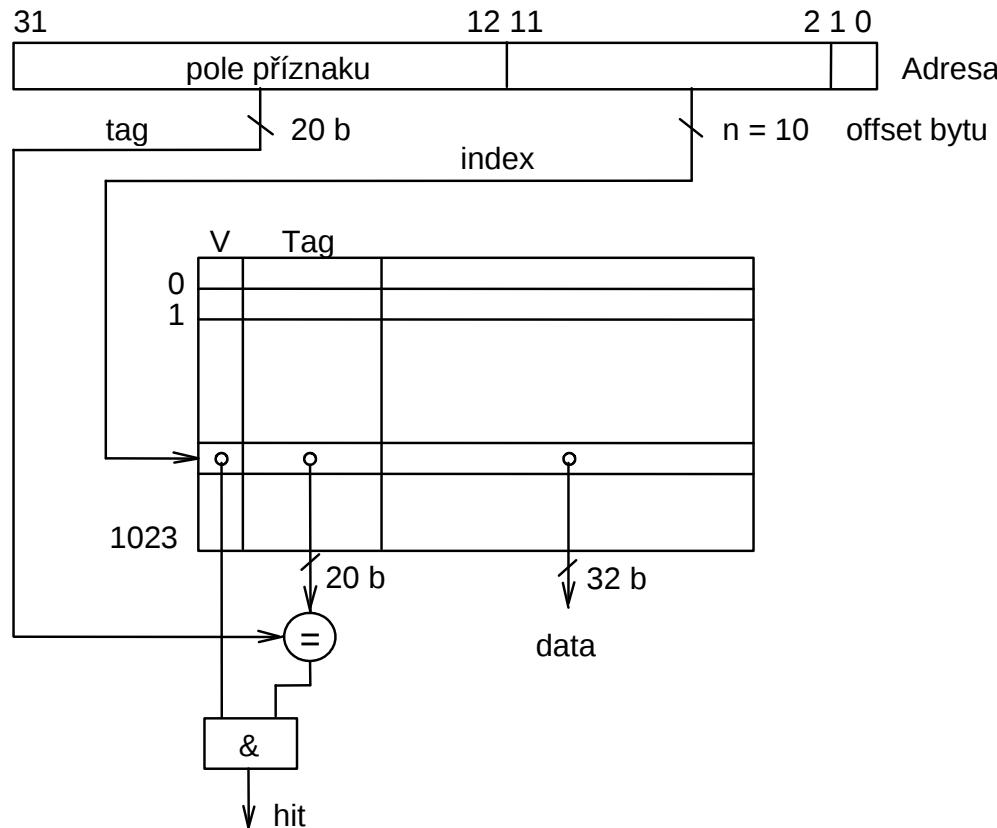
- U RVP se dále musí poskytnout informace, zda je požadovaný blok přítomen, příp. zda je informace v bloku platná. K tomu slouží **adresový příznak** (tag), což jsou zbývající horní bity adresy, a **příznak platnosti** (valid bit - V).
- Činnost RVP je pro zadanou posloupnost adres bloků ilustrována v tabulce. Předpokládáme, že je RVP na počátku činnosti prázdná.

Adresa	Úspěch/neúspěch	Přidělený blok RVP	Tag	Validity
10110	Miss	110	10	0→1
11010	Miss	010	11	0→1
10110	Hit	110	10	1
11010	Hit	010	11	1
10000	Miss	000	10	0→1
00100	Miss	100	00	0→1
10000	Hit	000	10	1
10010	miss	010	11→10	1

čas



RVP s přímým mapováním s 32-bitovou adresou



- Levá část RVP je adresová, pravá je datová.
- Celkový počet bloků o velikostí jednoho slova je 2^{30} .
- V RVP je umístěno 2^{10} bloků.
- Dolní odhad pravděpodobnosti úspěchu je $p_{hit} = 2^{10}/2^{30} = 2^{-20}$
- Díky lokalitě odkazů se v praxi dosahuje hodnot p_{hit} 0,9 až 0,98.

Proč je důležité programovat s ohledem na RVP?

Př.

Datový typ int je na 4B.

Blok dat RVP má velikost 4 x 4B

Mějme 2D pole **int a [4][5];**

Úloha: Sečtěte hodnoty všech položek pole **a**.

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

RVP:

a[0][0], a[0][1], a[0][2], a[0][3]
a[0][4], a[1][0], a[1][1], a[1][2]
a[1][3], a[1][4], a[2][0], a[2][1]
a[2][2], a[2][3], a[2][4], a[3][0]
atd.

Hlavní paměť

adr: data

100: a[0][0]
104: a[0][1]
108: a[0][2]
10c: a[0][3]
110: a[0][4]
114: a[1][0]
118: a[1][1]
11c: a[1][2]
120: a[1][3]
124: a[1][4]
128: a[2][0]
12c: a[2][1]
130: a[2][2]
134: a[2][3]
138: a[2][4]
13c: a[3][0]
140: a[3][1]
144: a[3][2]
148: a[3][3]
14c: a[3][4]
150: ...

Miss rate = 25%

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

RVP:

a[0][0], a[0][1], a[0][2], a[0][3]
a[1][0], a[1][1], a[1][2], a[1][3]
a[2][0], a[2][1], a[2][2], a[2][3]
a[3][0], a[3][1], a[3][2], a[3][3]
atd.

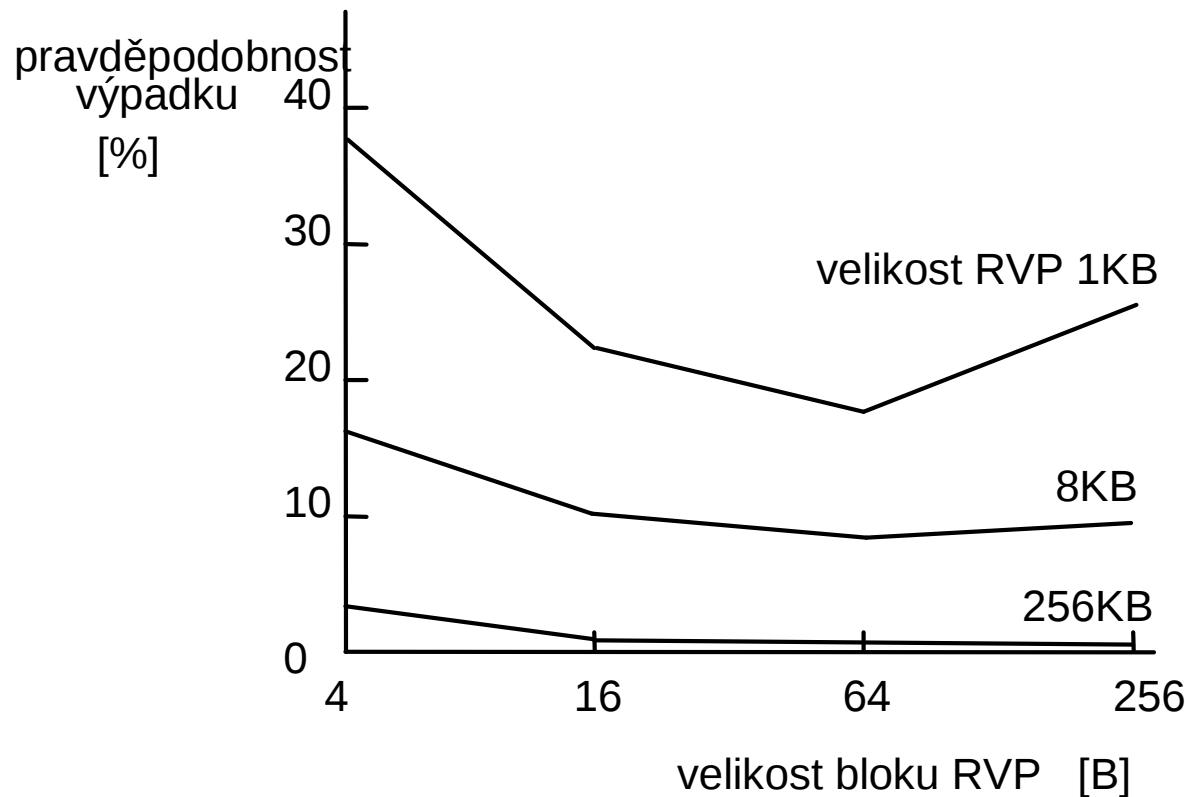
Miss rate = 100%

Koherence dat

- Změní-li se data v bloku zápisem, ztratí bloky na vzdálenějších úrovních platnost a nesmí se již použít. Vzniká tak **datová nekonzistence**, neboli **nekoherence**.
- Pro udržování koherence dat ve všech kopiích bloků je možno použít tyto strategie:
 - **přímý zápis** (write-through, průpis)
 - Při přímém zápisu do RVP se zapisuje okamžitě i do bloku v M. Má výhodu ve snadné realizaci a rovněž ztráty při R-neúspěchu jsou nízké, protože nový blok se přisune bez dalšího zdržování. Je-li však větší rozdíl mezi rychlostí primární a sekundární (vzdálené) paměti, pak přímý zápis častými opravnými zápisy příliš zdržuje a je tedy nevýhodný.
 - **zápis s mezipamětí** (write buffer)
 - Umožňuje odložit opravné zápisy až do okamžiku uvolnění přístupu k vzdálenější paměti, takže nedochází ke zdržování procesoru.
 - **zpětný zápis vždy**
 - Při zpětném zápisu se opraví celý blok v sekundární paměti až při jeho odsouvání. Strategie zpětného zápisu vždy je nepraktická, protože blok se zapisuje do sekundární paměti i v případě, že k žádné změně nedošlo.
 - **zpětný zápis podle příznaku změny** (write-back, copy-back, store on flag)
 - Prakticky použitelný je proto zpětný zápis podle příznaku změny (**dirty bit**). To je výhodné, protože střední počet zápisů do sekundární paměti je nižší než u přímého zápisu. Navíc blokový přesun lze zrychlit zvětšením šířky sběrnice mezi vyrovnávací a hlavní pamětí.

Typická pravděpodobnost výpadku vzhledem k velikosti bloku

Růst velikosti bloku má příznivý vliv jen do určité míry. Je-li velikost bloku vzhledem k velikosti RVP příliš značná, je v RVP málo bloků a příliš často se musí vyměňovat, protože požadovaná data nejsou často k dispozici.



Jak umístit do RVP více položek se stejnou adresou bloku?

Problém vzájemného vytlačování položek se stejným ukazatelem se řeší zvýšením **stupně asociativity**.

U dvoucestné paměti mohou být v paměti uloženy současně dvě položky se stejným ukazatelem.

Organizace b) však přináší oproti uspořádání a) jen malé zlepšení, protože kapacita paměti se nezměnila a do jednoho řádku se mapuje dvojnásobné množství adres.

Stupeň asociativity lze dále zvyšovat, až dospějeme k **plně asociativní paměti**, kdy již je příznak celá adresa, která může být umístěna v kterékoli pozici (v praxi nepoužitelné, příliš drahé).

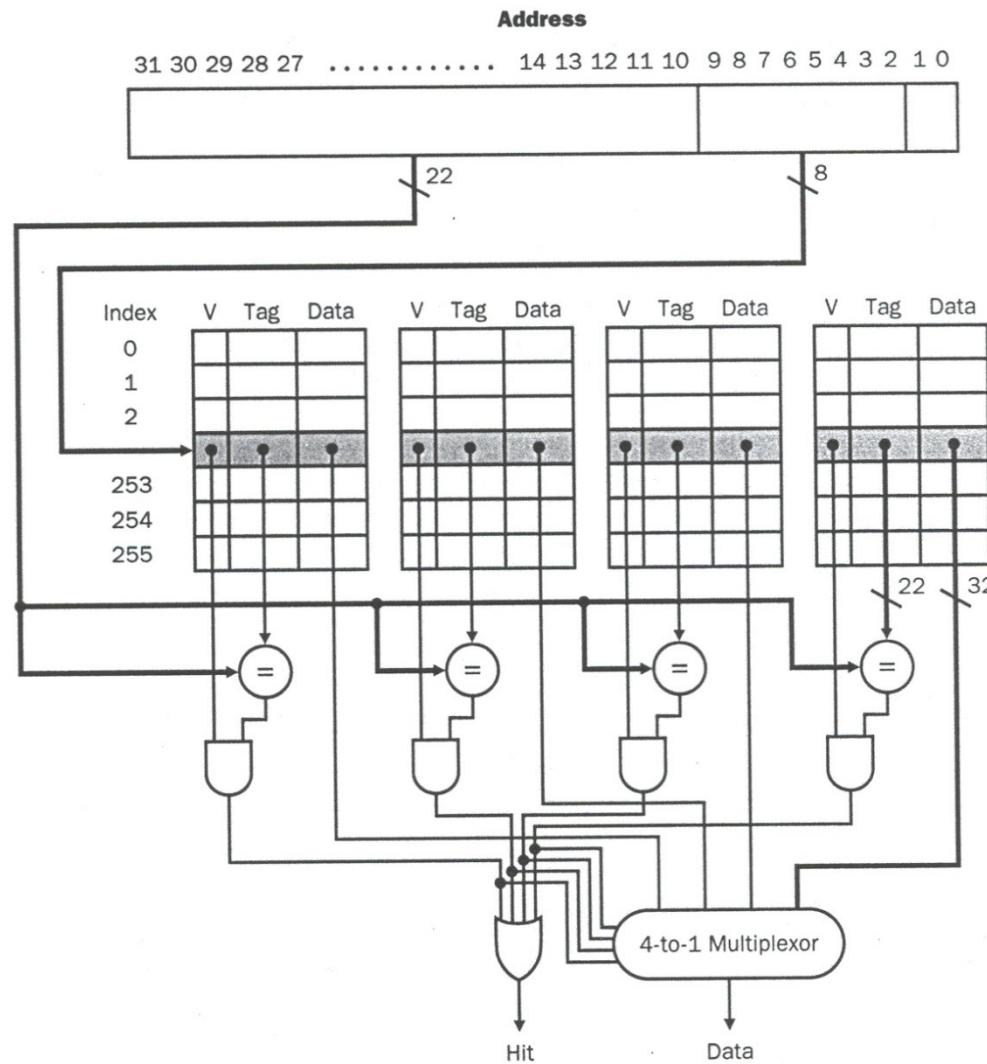
číslo bloku	příznak (tag)	data	stupeň asociativity
0			
1			
2			
3			
4			
5			
6			
7			

a) 1-cestná (s přímým mapováním)

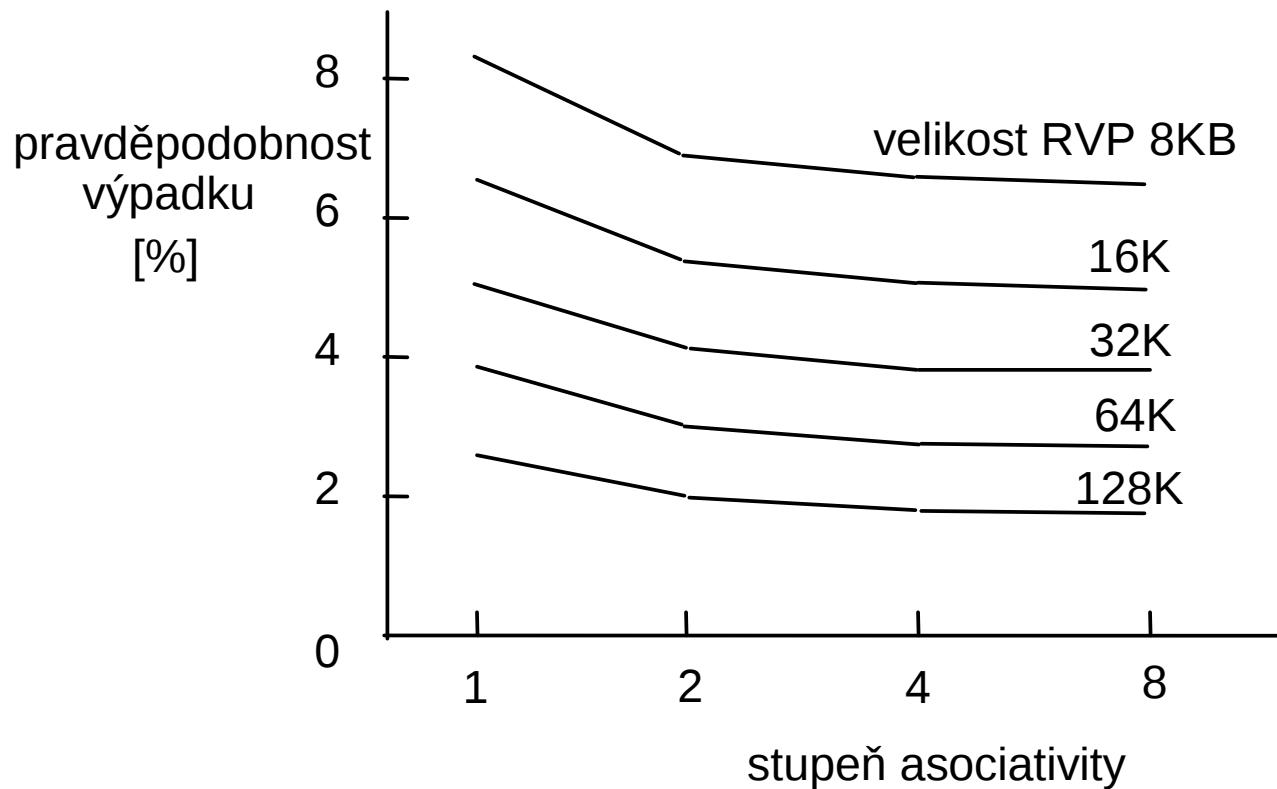
množina (sada)	tag	data	tag	data	stupeň asociativity
0					
1					
2					
3					

b) 2-cestná

Implementace 4-cestné RVP



Typická závislost pravděpodobnosti výpadku bloku na stupni asociativity



Výběr oběti

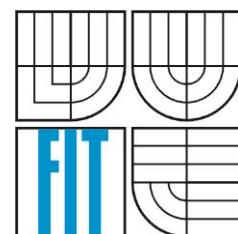
- Jsou-li všechny položky pro daný ukazatel obsazeny, je třeba rozhodnout, kterou položku zrušíme, a uvolníme tak místo pro novou položku. U dvoucestné RVP a obecně pro stupeň asociativity >1 tak vzniká problém **výběru oběti**.
- Tento problém řeší některá ze strategií náhrady:
 - **Least Recently Used** (LRU) - ponechávají se položky používané v poslední době a ruší se nejdéle nepoužitá položka
 - **Most Frequently Used** (MFU) - ruší nejčastěji použitou položku
 - **RAND** - náhodný výběr oběti
 - **FIFO** - oběť je položka, která je v RVP nejdéle
- Strategie LRU, MFU i FIFO vyžadují další obvodové doplňky, jako registry pro udržování času použití, resp. jejich úsporné modifikace, nebo čítače četnosti použití. Zde se musí řešit např. otázka přeplnění čítačů, a jsou navrženy algoritmy, které zavčas zahájí dekrementaci vybraných čítačů.

Související problematika – viz IOS

- virtuální paměť'
- překlad adresy
- fyzický adresový prostor
- logický adresový prostor
- stránka
- segment
- rámec
- Translation Look-Aside Buffer (TLB)

Sběrnice a periferní zařízení

INP 2015
FIT VUT v Brně



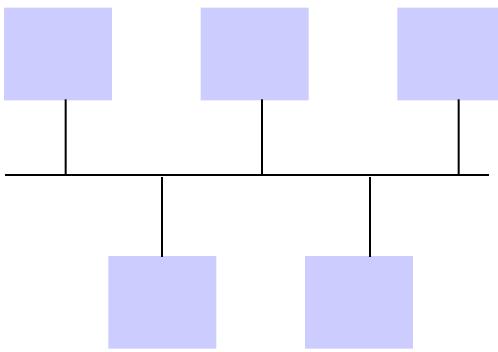
Přehled přednášky

- Sběrnice
 - Princip
 - Varianty sběrnic
 - Arbitrace na sběrnici
- Periferní zařízení
 - Připojování
 - Obsluha

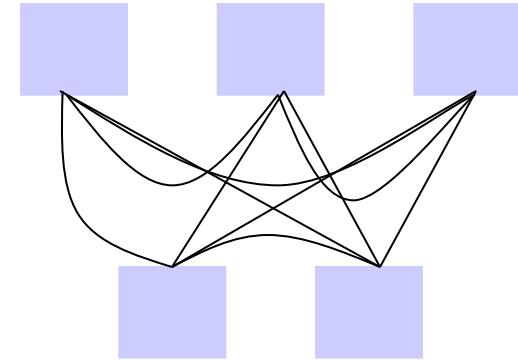
Zkratky:

- I/O = V/V – vstup/výstup
- PZ – periferní zařízení
- RVP – rychlá vyrovnávací paměť
- M – hlavní paměť
- IOP – I/O procesor

Proč sběrnice? Rozhoduje poměr výkon/cena!



Varianty propojení



Sběrnice:

- všechna zařízení sdílí relativně nízký počet vodičů
- dlouhé vodiče => nižší kmitočty
- způsob komunikace: jedna komponenta vysílá, všechny mají možnost přijímat
- relativně nízká propustnost pokud je připojen vyšší počet zařízení

Point-to-point linky:

- samostatné vodiče pro každé zařízení (vyšší cena)
- relativně vysoký počet vodičů
- krátké vodiče => vyšší kmitočty
- komunikace: point to point
- vysoká propustnost (souběžná komunikace mezi několika páry jednotek je možná)

Pojmy

Sběrnice: propojovací soustava umožňující komunikaci mezi více než jedním **zdrojem dat a přijímači dat.**

Rozhraní: definuje logický význam a elektrické vlastnosti vodičů určených pro komunikaci dané komponenty

Komunikační (sběrnicový) protokol: určuje pořadí aktivace signálů rozhraní tak, aby komponenta mohla komunikovat s okolím

Sběrnicová transakce (cyklus sběrnice) – operace zahrnující vydání adresy a zaslání (nebo přečtení) dat

Základní typy sběrnic:

- adresová vs. datová vs. řídicí (*signály RD, WR atd.*)
- interní vs. externí
- sériové vs. paralelní
- synchronní vs. asynchronní
- multiplexovaná vs. dedikovaná
- atd.

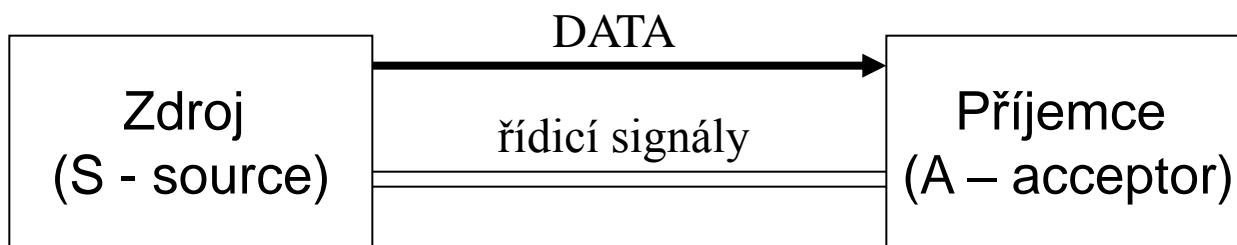
Parametry sběrnice podstatným způsobem ovlivňují výkonnost systému!

Šířka sběrnice:

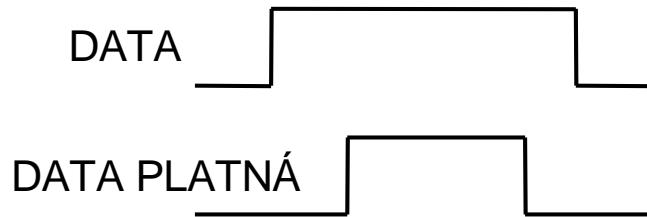
- datová sběrnice – počet bitů významně ovlivňuje výkonnost
- adresová sběrnice – počet bitů určuje max. paměťovou kapacitu systému

Asynchronní sběrnice

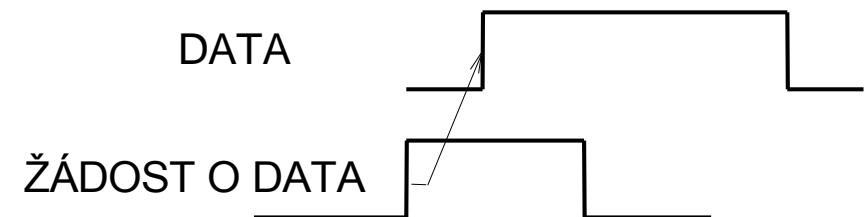
- Součástí dějů na sběrnici **nejsou** synchronizační signály, tj. generování nějakého signálu je vázán na výskyt události předcházející.
- Řízení asynchronní sběrnice
 - Jednostranně – řízení zajišťuje buď zdroj nebo příjemce
 - Oboustranně – na řízení se podílí zdroj i příjemce



Asynchronní přenos řízený jednostranně



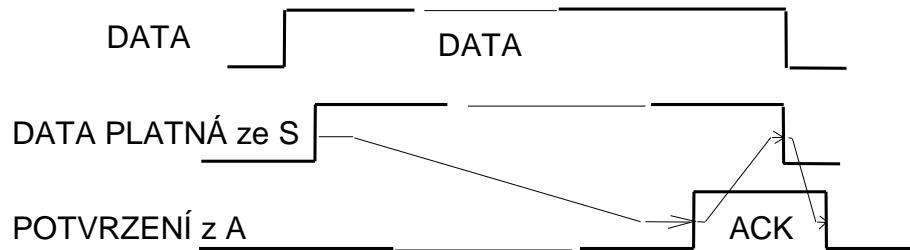
Řízení zdrojem S



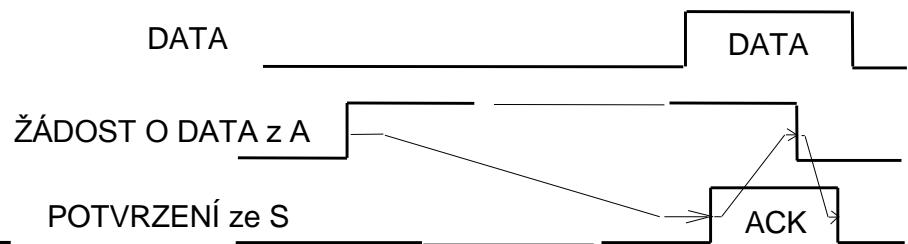
Řízení příjemcem A

- K vzorkování (převzetí) dat dochází hranou, nebo hladinou signálu DATA PLATNÁ, nebo ŽÁDOST O DATA.
- Nevýhodou jednostranného řízení je, že není potvrzováno správné převzetí dat, což může nastat, opozdí-li se příjemce, má-li poruchu, či jiné problémy vyžadující servis či opravu.
- Proto se doplňuje **potvrzovací signál ACK (acknowledge)**, kterým se mění jednostranné řízení na **dvoustranné** a dostáváme tak tzv. **korespondenční protokol** (handshake).

Asynchronní protokol řízený oboustranně (handshake)



Přenos vyvolaný zdrojem S



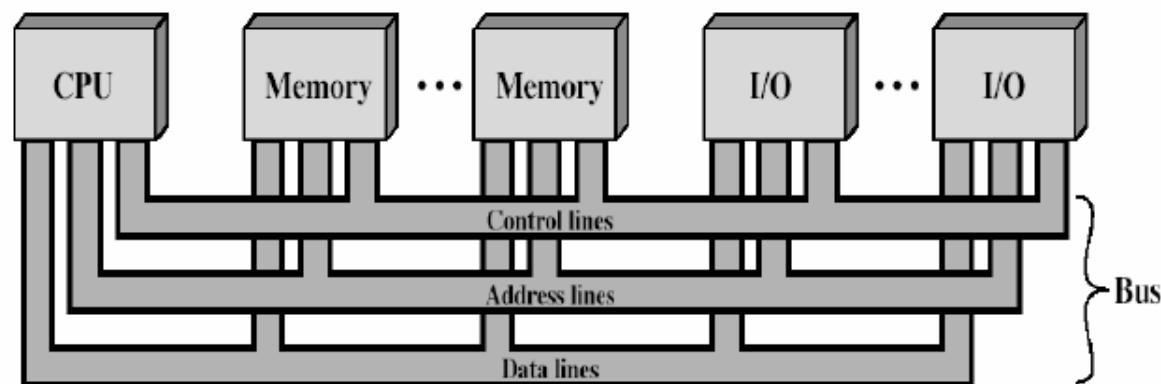
Přenos vyvolaný příjemcem A

- Máme zde naznačen úplný (uzavřený) korespondenční cyklus, který se vyznačuje pevným sledem hran, vázaných šipkami vzájemné závislosti (full interlock).

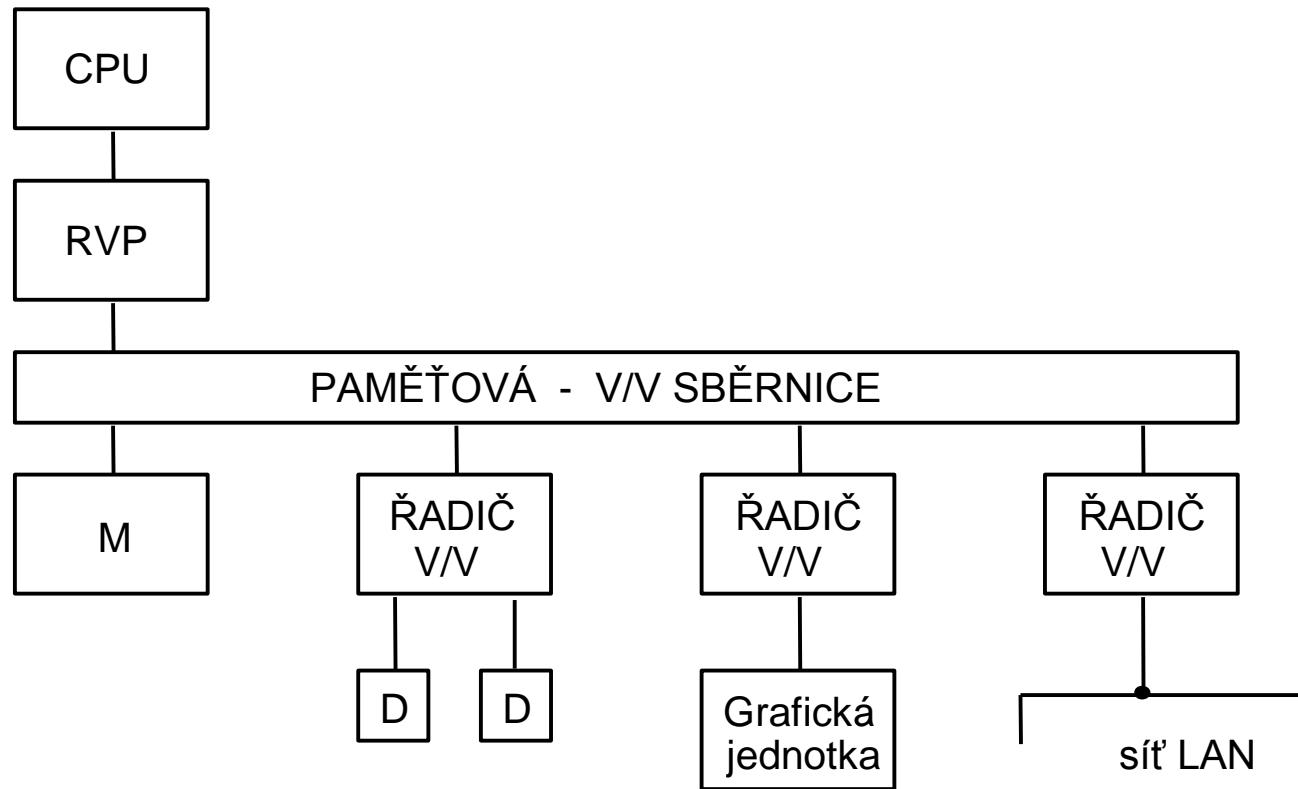
Typické signály řídicích sběrnic

- Memory Write (zápis do paměti)
- Memory Read (čtení obsahu paměti)
- I/O Write (zápis do registru PZ)
- I/O Read (čtení obsahu registru PZ)
- Transfer ACK (potvrzení příjmu)
- Bus Request (žádost o přidělení sběrnice)
- Bus Grant (přidělení sběrnice)
- Interrupt Request (žádost o přerušení)
- Interrupt ACK (potvrzení žádosti o přerušení)
- Clock (systémová synchronizace)
- Reset (systémové nulování)

Základní
počítačová
architektura
(jednosběrnicová,
bez RVP).

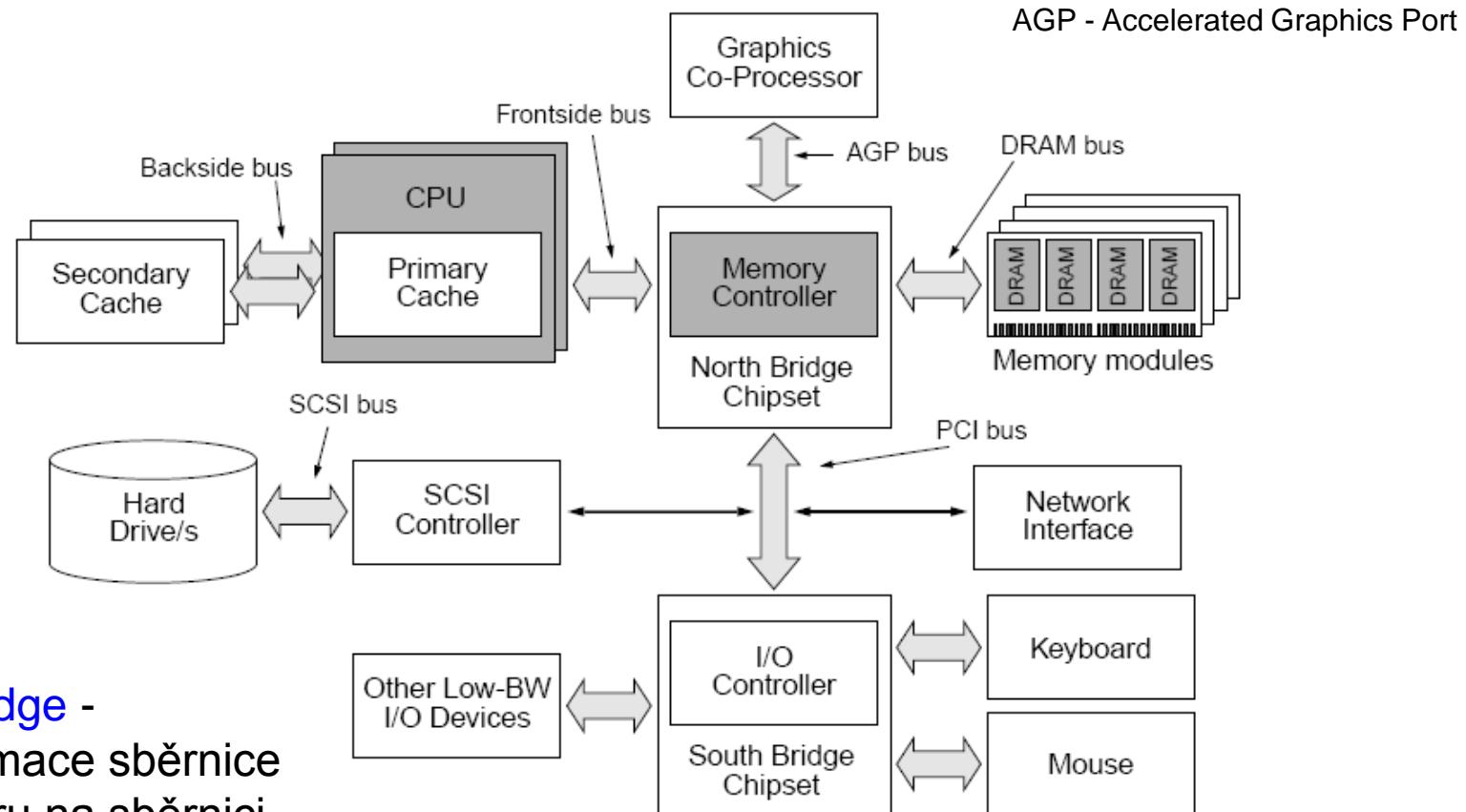


Základní uspořádání počítače – jediná sběrnice



- Veškerá komunikace probíhá přes jedinou sběrnici, která je sdílena všemi jednotkami.

Sběrnicová hierarchie v PC s PCI sběrnicí



Northbridge -
transformace sběrnice
procesoru na sběrniči
PCI

Southbridge -
transformace sběrnice
PCI na rozhraní PZ

- Moderní počítače používají **hierarchii sběrnic**. „Pomalá“ zařízení jsou připojena na jinou sběrniči než „rychlá“ zařízení.

Příklady nejčastěji vyskytujících se standardů sběrnic

<http://cs.wikipedia.org/wiki/Sběrnice>

- **ISA** - starší typ pasivní sběrnice, šířka 8 nebo 16 bitů, přenosová rychlosť < 8 MB/s
- **PCI** - novější typ „inteligentní“ sběrnice, šířka 32 nebo 64 bitů, burst režim, přenosová rychlosť < 130 MB/s (260 MB/s)
- **AGP** - jednoúčelová sběrnice určená pro připojení grafického rozhraní (karty) k systému, přenosová rychlosť 260 MB/s - 2 GB/s
- **PCI-X** - zpětně kompatibilní rozšíření sběrnice PCI
- **PCI-Express (PCIe)** - nová sériová implementace sběrnice PCI
- **USB** - sériová polyfunkční sběrnice, 2 diferenciální datové vodiče + 2 napájecí vodiče 5 V/500 mA, široké použití, verze 1.1 přenosová rychlosť 12 Mb/s (~1,43 MB/s), 2.0 přenosová rychlosť 480 Mb/s (~57 MB/s), 3.0 přenosová rychlosť 4800 Mb/s (~572 MB/s)
- **FireWire** - sériová polyfunkční sběrnice, široké použití, 50 MB/s
- **RS-485** - sériová průmyslová sběrnice (někdy jako proudová smyčka), do prostoru s vysokým elektromagnetickým rušením
- **I²C** - sériová sběrnice, < 100 kb/s, adresace 32 zařízení, komunikace a řízení v elektronických zařízeních

Způsoby rozhodování o přidělení sběrnice při současné žádosti několika jednotek

Rozhodování:

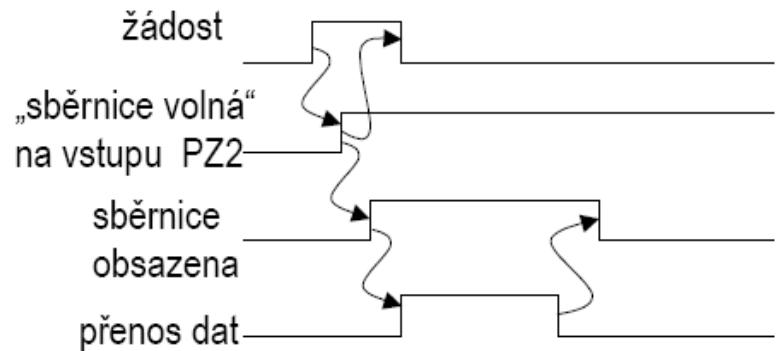
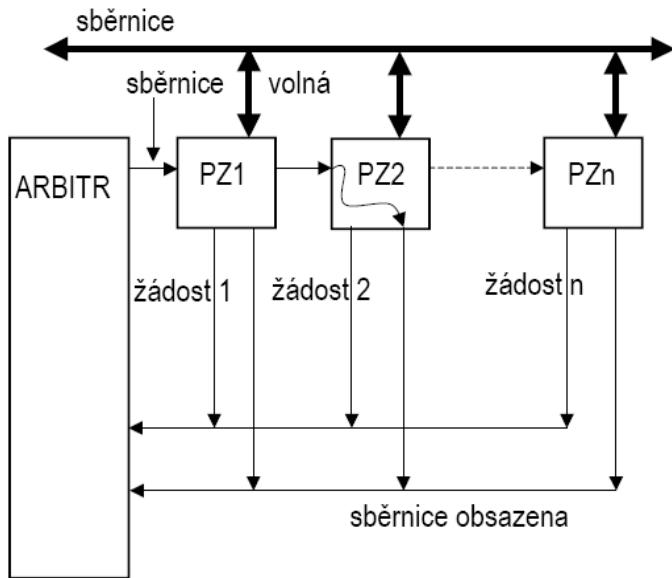
Centralizované vs. decentralizované

Prioritní vs. spravedlivé

U **centrálního řízení** je použita centrální **rozhodovací jednotka (arbitr)**, která rozhoduje buď *podle priority*, nebo *spravedlivě*, např. podle pořadí vzniku žádostí (udržuje se fronta požadavků), cyklicky (pomocí centrálního čítače), nebo náhodně.

Prioritní dekodér je základem centrálního rozhodovacího členu, který po vyhodnocení priority žádostí oznámí jednotce s nejvyšší prioritou, že získala sběrnici. Vlastní mechanismus oznámení může využívat vyzývání (polling), kdy se na adresovou, datovou, nebo rozhodovací (vyzývací) sběrnici vyšle číslo vítězné jednotky. Jednotka, která zjistí svoje číslo, může zahájit přenos po sběrnici.

Příklad: Prioritní linka



- Jednotlivá zařízení vysílají signály „žádost x“ do společného vodiče.
- Arbitr odpoví vysláním signálu „sběrnice volná“, který je jednotlivými zařízeními postupně přijímán a vyhodnocen.
- Zařízení, které vyslalo „žádost“, zablokuje odeslání signálu „sběrnice volná“ do následujícího PZ a vyšle signál „sběrnice obsazena“.
- Jakmile je ukončen přenos dat, signál „sběrnice obsazena“ je shoven.
- Prioritní systém je uplatněn pořadím zařízení na kabelu „sběrnice volná“ – zařízení, která jsou blíže arbitra, mají vyšší prioritu.

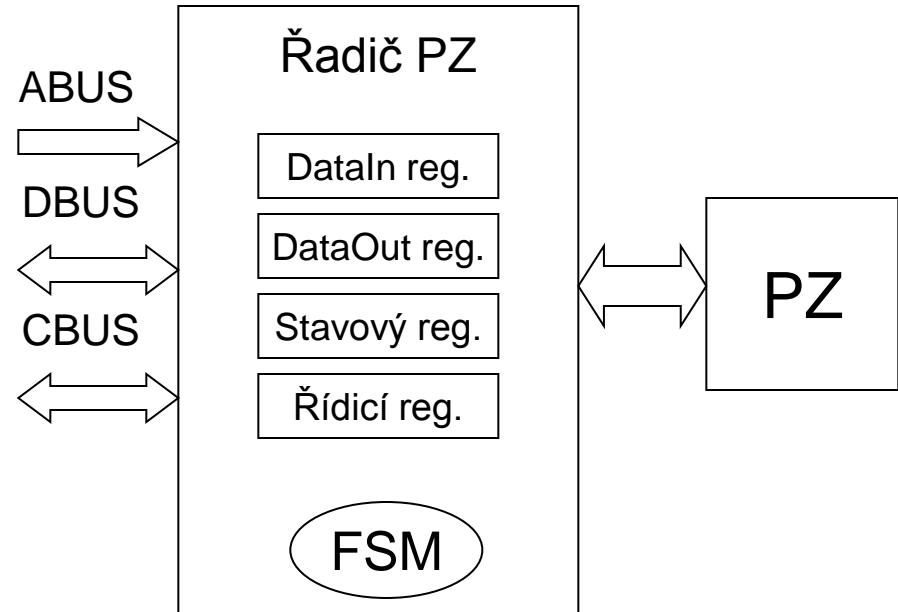
Způsoby připojování a řízení periferních zařízení

- Procesor komunikuje s registry řadiče periferního zařízení.
- Používá k tomu buď instrukce pro práci s pamětí nebo speciální instrukce pro ovládání V/V.
- Poznámky:
 - Např. není možné přímo realizovat přenos mezi registrem řadiče a pamětí – adresy obou by musely být ve stejném okamžiku vystaveny na sběrnici.
 - Přímá komunikace mezi periferií a pamětí (bez účasti procesoru) je možná zavedením dalších obvodů – např. DMA.

Řadič periferního zařízení

Funkce řadiče:

komunikace s CPU,
vyvolání přerušení,
řízení a časování operací PZ,
realizuje vyrovnávací paměť,
detekuje a reportuje poruchu



Registry jsou programovatelné ze strany procesoru.

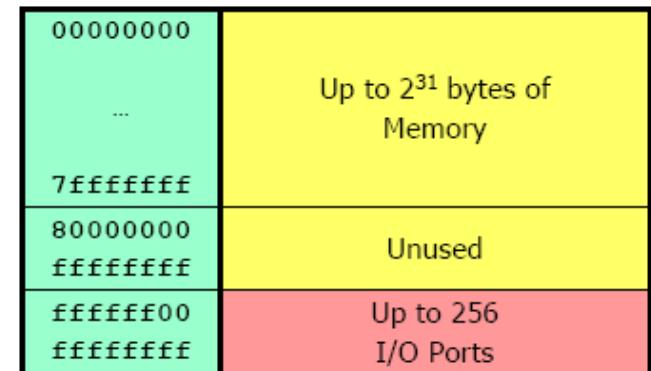
Datový registr - přes něj se přenášejí data.

Řídicí registr - určuje způsob provedení operace PZ.

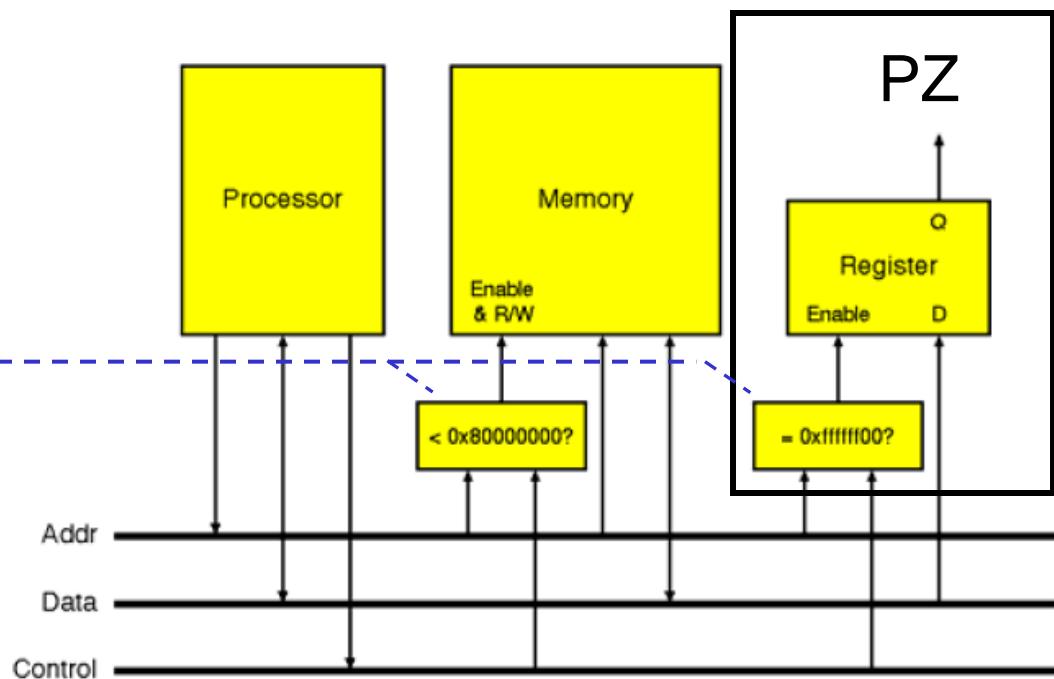
Stavový registr - stav řadiče a PZ.

Možnosti připojení PZ: 1) mapovaný V/V

- Registry PZ jsou namapovány do na určité adresy (hlavní) paměti.
- PZ a paměť sdílí stejný adresový prostor.
- Operace s PZ se provádí stejně jako operace s pamětí (instrukcemi pro čtení a zápis).

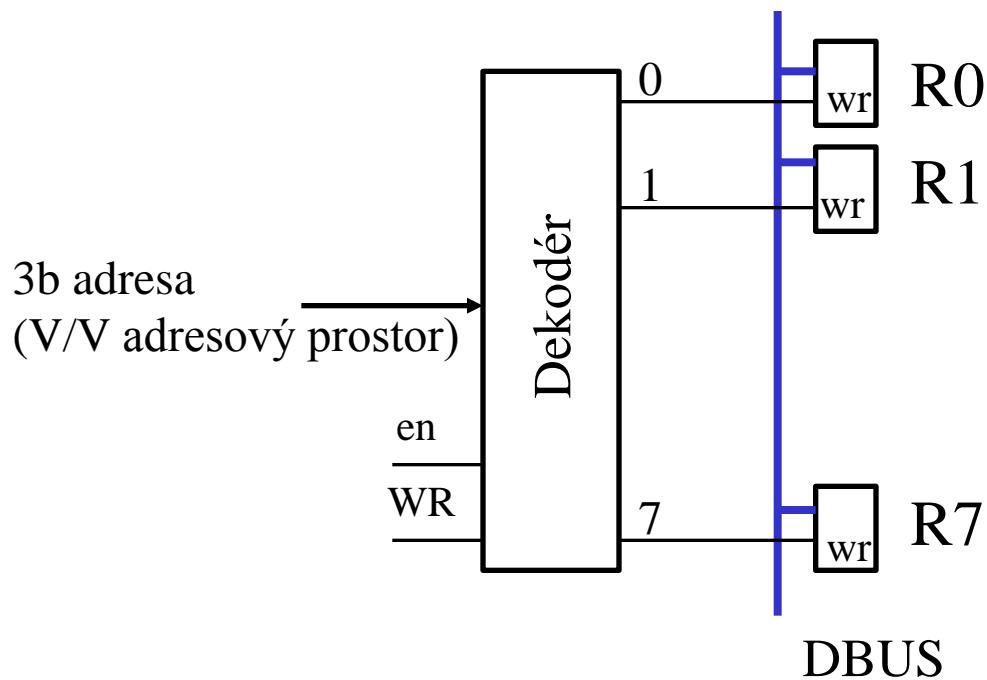


Řešení je založeno
na použití
adresového
dekodéru.



Možnosti připojení PZ: 2) izolovaný V/V

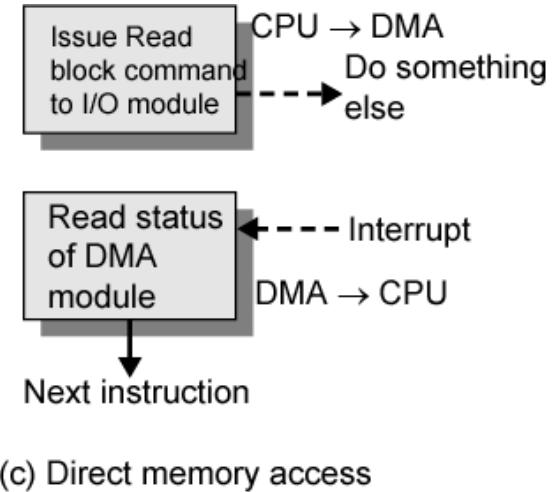
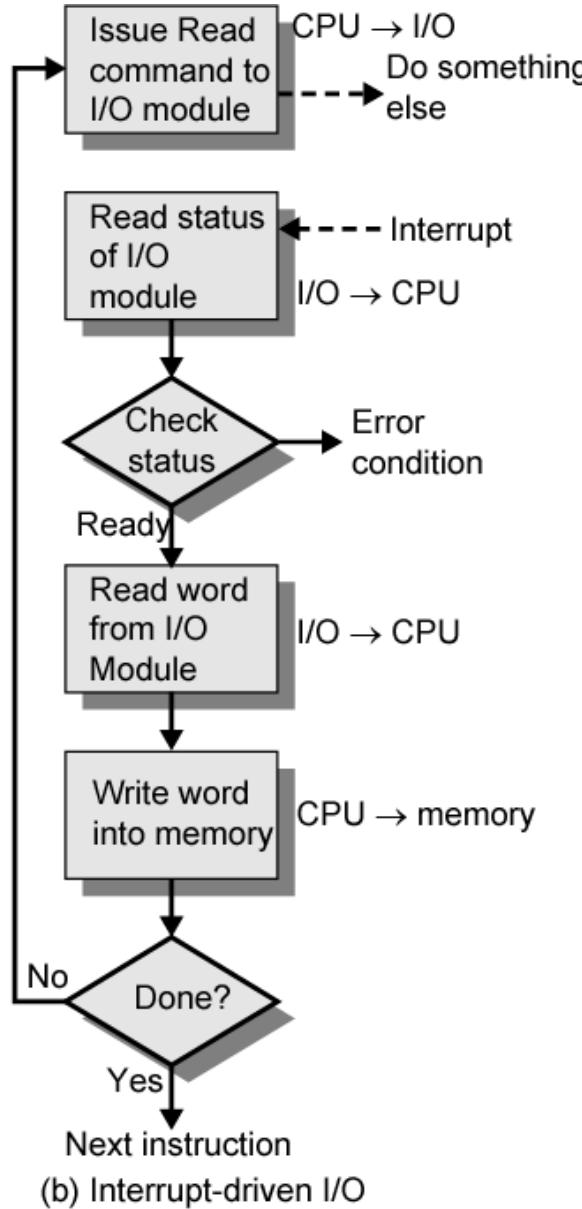
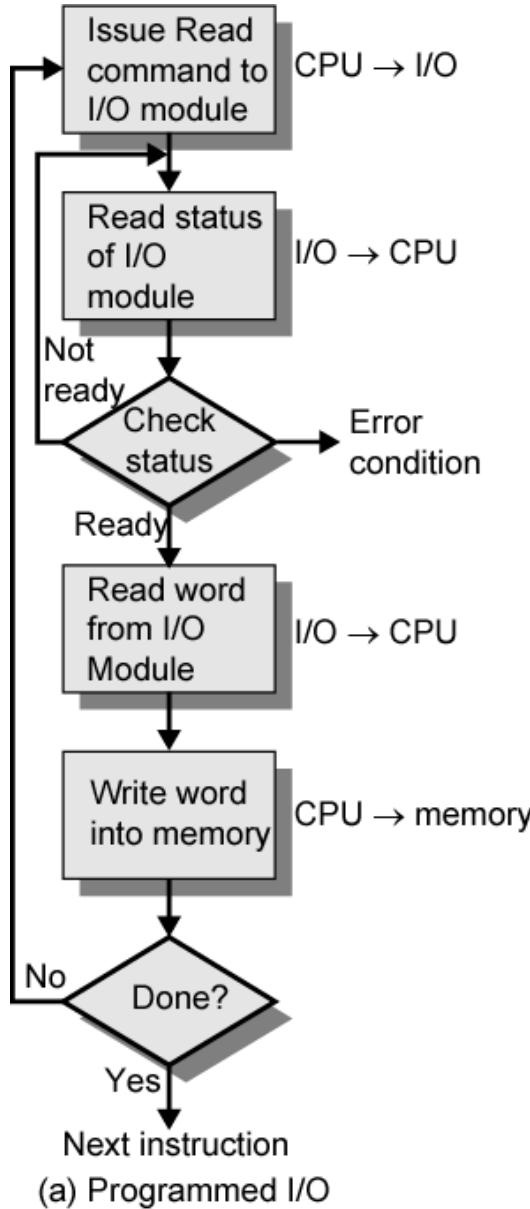
- Oddělení adresového prostoru paměti a periferií
 - Registry periferních zařízení mají svůj **vlastní adresový prostor** (uveden příklad, kdy je možné adresovat až 8 registrů v periferních zařízeních)
 - Operace s PZ se provede pomocí speciální instrukce **IN** a **OUT** (např. signál „zápis do registru“ (viz wr na obrázku) je odvozen od instrukce OUT).



Způsoby obsluhy periférií

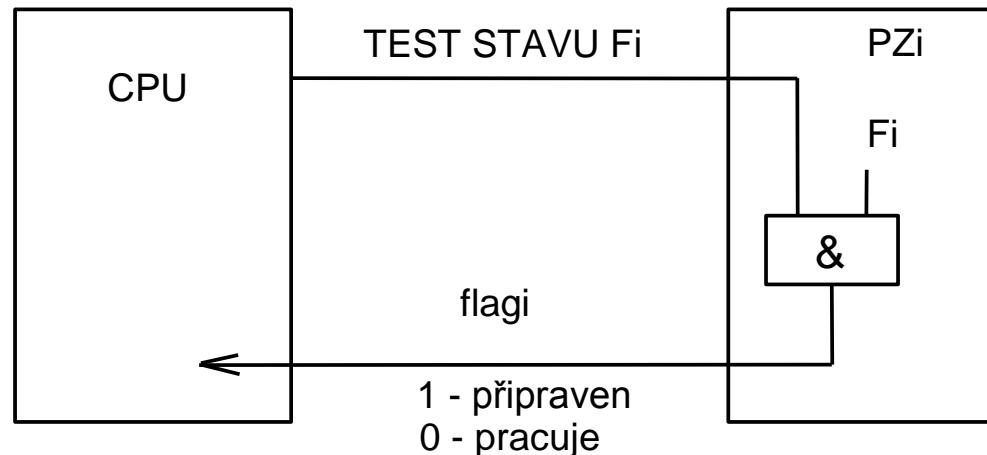
- Programový V/V
 - Procesor testuje ve smyčce stav všech PZ (stisknuta klávesa, přišla data ze sítě?), což je neefektivní.
- Využití přerušení
 - Pokud PZ potřebuje obsluhu vyvolá přerušení a procesor jej obslouží.
- Využití obvodů pro řízení blokových přenosů DMA (direct memory access)
 - Přenos větších objemů dat mezi PZ a pamětí zajistí DMA bez použití procesoru
- Využití IO procesorů
 - Zobecnění konceptu DMA

Porovnání přístupů (př: čtení dat z PZ do paměti)

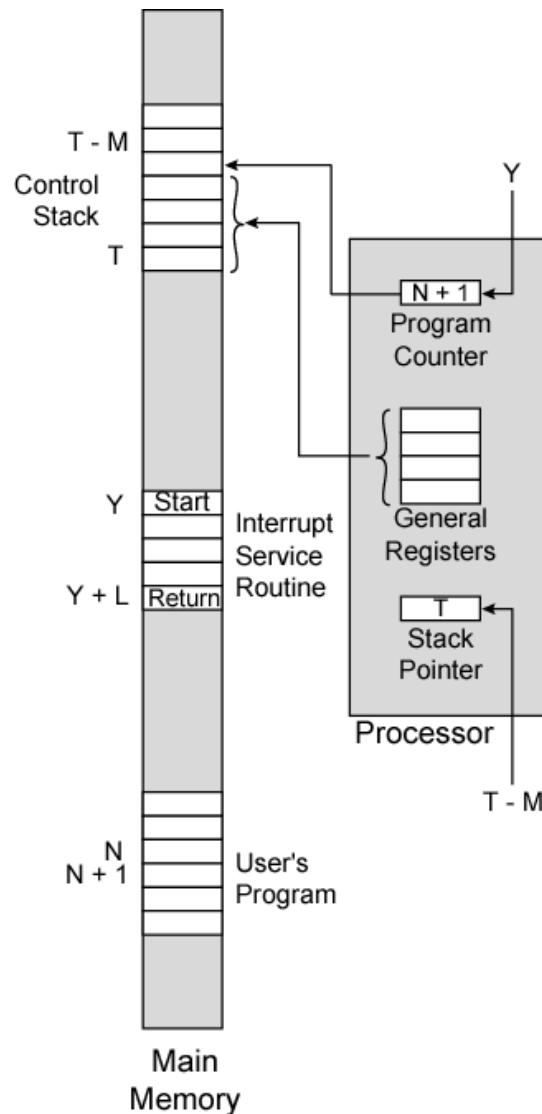


A. Programovaný V/V (polling)

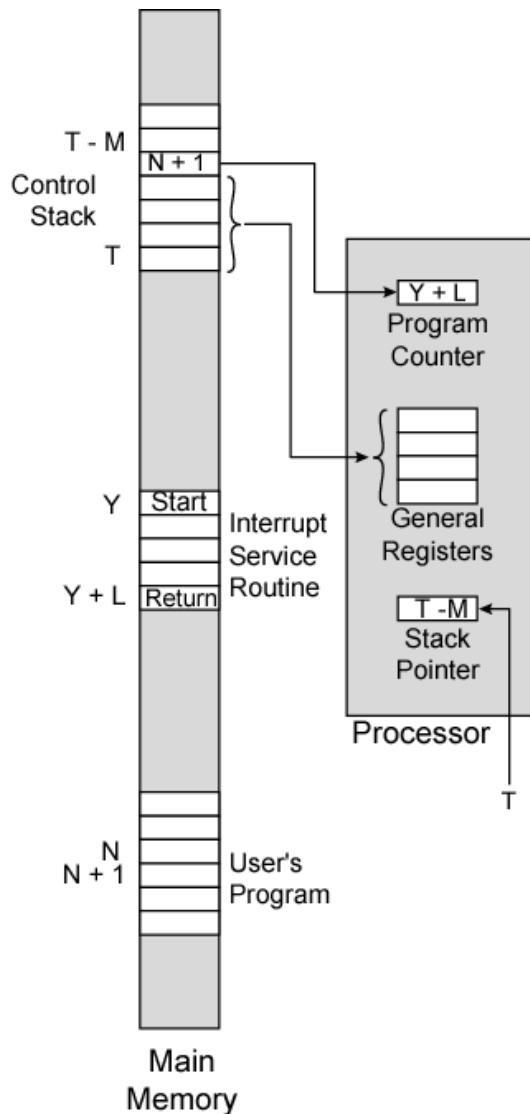
Procesor testuje ve smyčce stav PZ (byla stisknuta klávesa?). Pokud je možné provést obsluhu PZ (klávesa stisknuta), provede obsluhu (přečte kód znaku a zpracuje ho), jinak opět testuje ve smyčce stav PZ (byla stisknuta klávesa?). Je zřejmé, že čekací smyčka zatěžuje procesor, který pak nemůže provádět užitečnější práci. Rovněž sběrnice je zbytečně zatěžována.



B. Obsluha využívající přerušení: O obsluhu žádá PZ!!!



Přerušení programu po vykonání instrukce na adrese N.



Návrat z obsluhy přerušení

Př. Pokud byla stisknuta klávesa, procesor přeruší uživatelský program, uschová obsahy registrů a návratovou adresu na zásobník a obslouží klávesnici (rutina na adrese Y). Potom obnoví obsahy registrů a vrátí se do uživatelského programu.

Jak se řeší priorita, když dvě a více PZ žádají o obsluhu?

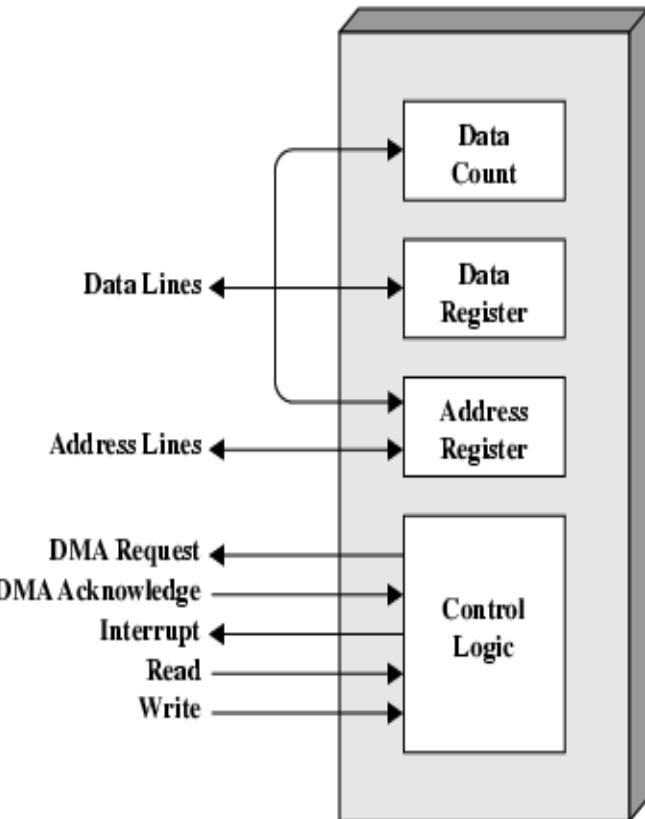
- Viz předchozí přednášky

C. Přenos DMA (Direct Memory Access)

Koncept přerušení je nevýhodný, pokud je třeba přenášet větší objemy dat. Při obsluze přerušení se procesor podílí na datových přenosech, což ho zatěžuje => zaveden DMA.

Řadič DMA je speciální obvod zajišťující blokové přenosy mezi PZ a pamětí (popř. mezi pamětí a pamětí).

- CPU zadá požadavek obvodu DMA, tj. **adresu PZ, adresu v paměti** (ta se uloží do Address Register v DMA) a **počet slov** (ten se uloží do Data Count v DMA).
- DMA obvod provede přesun dat, slovo za slovem, bez zásahu CPU.
 - data bud' prochází přes řadič DMA
 - nebo data neprochází přes řadič DMA
- CPU mezitím může vykonat další kód (je však omezen ve využívání sběrnice).
- Po ukončení přenosu (popř. při vzniku chyby) obvod DMA vyvolá přerušení.



Obvod řadiče DMA

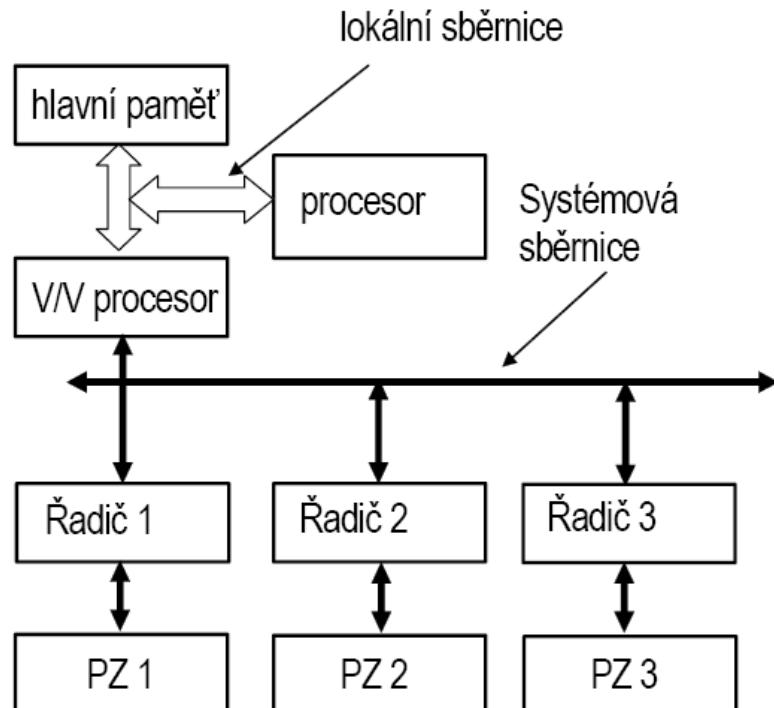
- typické komponenty a rozhraní
- je připojen ke sběrnici

D. IO procesory

Řadič DMA je v podstatě jednoduchý FSM s několika registry.

Rozšířením principu DMA se dospělo ke koncepci **IO procesoru**.

IO procesor (V/V procesor) je další procesor počítače, který má ale speciální instrukční soubor (zejména IO instrukce).



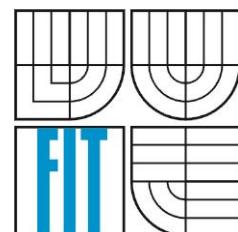
- Procesor nemá přístup k registrům řadičů PZ.
- IO procesor komunikuje s řadiči jednotným způsobem.
- Procesor zadá IO procesoru program, který má vykonat. Program je uložen v hlavní paměti.

Literatura

- Drábek, V.: Výstavba počítačů. Skriptum VUT Brno, 1995
- Kotásek, Z.: Periferní zařízení. Texty k přednáškám. FIT VUT Brno, 2006
- Stallings, W.: Computer Organization and Architecture, 7th ed., Pearson Prentice Hall, 2006
- <http://cs.wikipedia.org/wiki/Sběrnice>

Měření výkonnosti počítačů

INP 2015
FIT VUT v Brně



Amdahlův zákon o urychlení výpočtu

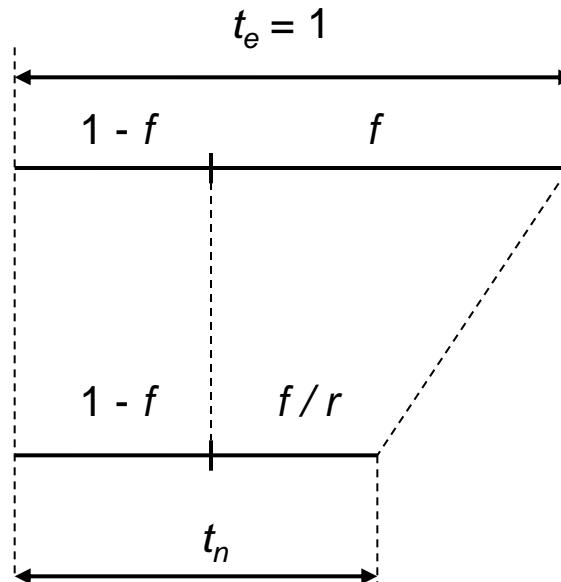
- Amdahlův zákon udává urychlení výpočtu zavedením rychlejšího zpracování jisté části úlohy. Předpokládáme, že zbylou část úlohy nelze urychlit.
- Část úlohy, kterou lze zrychlit r -krát, označíme f , $f < 1$. Zbylou část označíme $1-f$.
- Doby provedení výpočtu před a po zavedení urychlení si označíme jako t_s a t_n .

$$t_n = t_s \left((1-f) + \frac{f}{r} \right)$$

Celkové urychlení v_r je rovno

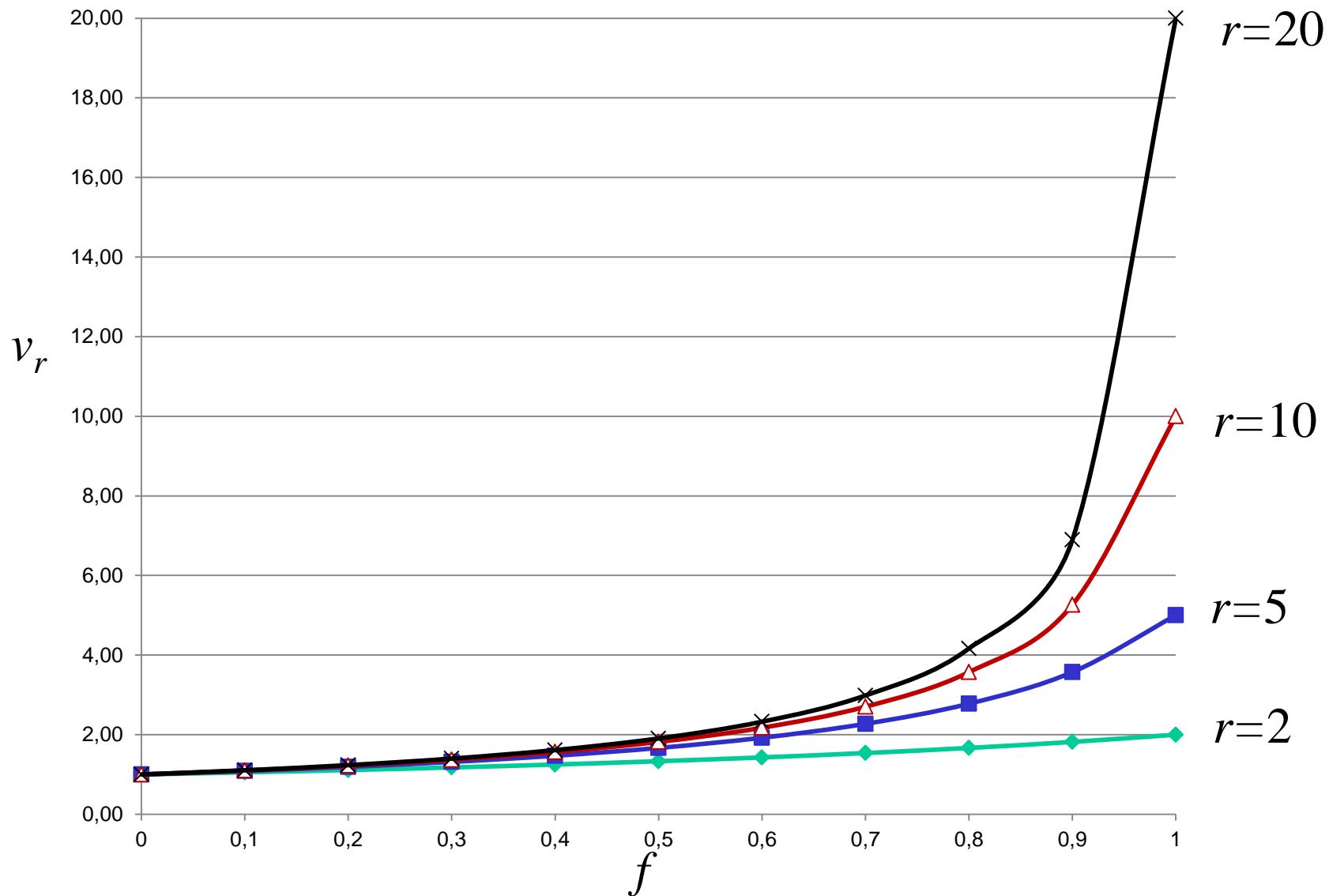
$$v_r = \frac{t_s}{t_n}$$

$$v_r = \frac{1}{(1-f) + \frac{f}{r}}$$



Zrychlení jako funkce f , $r = \text{konst.}$

Př. Jakého urychlení dosáhneme, pokud vložíme do počítače RVP, která je r-krát rychlejší než hlavní paměť a je využita v x% případů?



Definice výkonnosti počítače

- Výkonnost počítače se posuzuje několika způsoby, nejčastěji jako:
 - doba (provedení) výpočtu t_e (execution time).
 - propustnost systému P_e - počet transakcí, které je systém schopen uskutečnit za časovou jednotku.
- Absolutní měření výkonnosti je velmi problematické.
- Schůdnější je hodnotit výkonnost **relativně**, tj. vzhledem k referenčnímu počítači, jako např. PC XT 4,7 MHz, nebo VAX-11/780.
- Je-li výkonnější počítač označen X a méně výkonný Y, lze pro poměr jejich výkonností (propustností) P_e a dob výpočtu t_e , které byly získány pro nějaký program H, psát

$$P_{eX}/P_{eY} = t_{eY}/t_{eX} = n$$

a říkáme "počítač X je **n krát** výkonnější než počítač Y".

Počítač X je **o k% výkonnější** než Y, pokud platí:

$$(t_{eY}/t_{eX} - 1) * 100 = k$$

Měření výkonnosti

- Převážná část počítačů je zkonstruovaná s využitím konstantního hodinového kmitočtu f_C (doba cyklu $T_C = 1/f_C$).
- Doba výpočtu:
 $t_e = \text{Počet hodinových cyklů během programu} \times T_C$ (1)
- Můžeme spočítat počet provedených instrukcí programu Počet I a určit důležitý parametr procesoru - **počet cyklů na provedení jedné instrukce CPI - Cycles Per Instruction.**
- $CPI = \text{Počet hodinových cyklů během programu} / \text{Počet I}$ (2)
- $\text{Počet hod. cyklů během programu} = CPI \times \text{Počet I}$ (3)
- Po dosazení (3) do rovnice (1) dostaneme
 $t_e = \text{Počet I} \times CPI \times T_C$ (4)
- Počet instrukcí v programu je dán řešenou úlohou a je ovlivněn architekturou instrukčního souboru (Instruction Set Architecture - ISA) a technologií komplikátoru. Parametr CPI je dán architekturou ISA, a T_C je dána použitou obvodovou technologií a organizací obvodů počítače.

Měření výkonnosti v jednotkách MIPS

- Alternativou k době provedení je počet milionů instrukcí provedených za sekundu, udávaný v jednotkách MIPS.

$$P_{\text{MIPS}} = \text{Počet I} / (t_e \times 10^6) = f_C / (\text{CPI} \times 10^6) \quad (5)$$

- Pozn.: Poslední výraz byl získán pomocí vztahu

$$t_e = (\text{Počet I} \times \text{CPI}) / f_C \quad (6)$$

- Poznámky:

- Výhodou této definice je její jednoduchost, ale její použití pro srovnávání výkonnosti počítačů nese některé problémy.
- Např. P_{MIPS} je závislý na instrukčním souboru, takže je problematické srovnávat počítače s různými instrukčními soubory.
 - Př. Na počítači RISC i CISC trvá výpočet stejného problému 1 ms. CISC potřebuje 20 instrukcí, RISC potřebuje 60 instrukcí. Je opravdu RISC 3krát výkonnější??? Ne!
- MIPS se hodí např. pro srovnání různých generací procesorů jednoho výrobce.

Měření výkonnosti v jednotkách MIPS

- Řešením by mohlo být definovat výkonnost relativně vzhledem k referenčnímu počítači, a pracovat s **relativním P_{MIPS}**.
- Relativní P_{MIPS} = (t_{e ref. počítače} × P_{MIPS ref. počítače}) / t_{e měř. počítače}
- P_{MIPS ref. počítače} je dohodnuté číslo. Např. počítače VAX - 11/780 bývá označen za "1 MIPS" stroj. Výkonnost v MIPS dalších procesorů je např. zde: http://en.wikipedia.org/wiki/Instructions_per_second

Měření v jednotkách MFLOPS

- Pro počítače s jednotkou pro práci s čísly v pohyblivé řádové čárce (FP) se udává výkonnost v počtu operací s pohyblivou řádovou čárkou (**FLOPS**).
$$P_{MFLOPS} = \text{Počet FP operací} / (t_e \times 10^6)$$
- Problém je, že instrukční soubory FP nejsou u všech počítačů stejné.
- Složitost FP operací je různá. Operace sčítání je značně rychlejší než operace dělení. Pro program se 100% operací sčítání pak bude zjištěna vyšší výkonnost než pro program se 100% operací dělení.
- Aby parametr P_{MFLOPS} tyto rozdíly správně zachytil, byly definovány kanonické počty FP operací ve zkušebním programu. Každá operace FP se pak ohodnotí vahou, která reprezentuje její složitost, a dostaneme parametr normalizovaný P_{MFLOPS} .
- Váhy FP operací podle Livermore Loops
 - ADD, SUB, CMP, MPY 1
 - DIV, SQRT 4
 - EXP, SIN 8

Programy pro hodnocení výkonnosti

- Pro měření se používají
 - reálné programy,
 - jádra (kernels) - nejvýznamnější části skutečných programů,
 - demonstrační zkušební úlohy (např. Quicksort), a
 - syntetické zkušební úlohy - mají obdobnou filosofii jako jádra, snaží se vystihnout průměrné frekvence operací a dat na rozsáhlých množinách programů.
- V minulosti bylo nejrozšířenější použití benchmarků, jako Whetstone benchmark s převahou operací v pohyblivé řádové čárce a Dhrystone benchmark s převahou operací v pevné řádové čárce.
- Dnes se nejvíce používají
 - SPEC – System Performance Evaluation Cooperative
 - TPC – Transaction Processing Performance Council

SPEC (od 1988)

- Nejdokonalejší jsou série reálných benchmarků SPEC (System Performance Evaluation Cooperative): SPEC89, SPEC92, SPEC95, SPEC2000
- Skupiny měřicích programů se neustále mění tak, aby se objektivizovalo měření a pokrývaly se moderní aplikace
 - SPECint (CINT2006) pro operace s pevnou řádovou čárkou (12 programů).
 - SPECfp (CFP2006) pro operace s pohyblivou řádovou čárkou.
- Pro každou úlohu se zjistí relativní doba výpočtu (vzhledem k referenčnímu počítači). Ve skupinách int a fp se pak odděleně vypočte tzv. geometrický průměr, což je n -tá odmocnina ze součinu n relativních dob provedení. Výsledkem jsou dva výkonové parametry SPECint a SPECfp.

SPECint 2006

Benchmark	Language	Category
400.perlbench	C	Programming Language
401.bzip2	C	Compression
403.gcc	C	C Compiler
429.mcf	C	Combinatorial Optimization
445.gobmk	C	Artificial Intelligence
456.hmmmer	C	Search Gene Sequence
458.sjeng	C	Artificial Intelligence
462.libquantum	C	Physics / Quantum Computing
464.h264ref	C	Video Compression
471.omnetpp	C++	Discrete Event Simulation
473.astar	C++	Path-finding Algorithms
483.xalancbmk	C++	XML Processing

$$SPEC \text{ int } 2006 = \sqrt[12]{\prod_{i=1}^{12} tr_i} \quad tr = \frac{t_{ref}}{t_{hod}}$$

t_{ref} – doba výpočtu referenčního počítače
 t_{hod} – doba výpočtu hodnoceného počítače

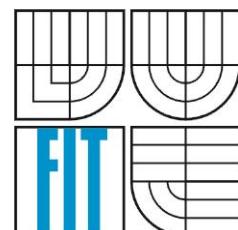
Měření výkonnosti se provádí odděleně pro programy s operacemi FX a FP.

Shrnutí

- Výkonnost závisí na optimalizaci CPU (program, kompilátor, ISA a HW), paměťového subsystému a I/O subsystému.
- Principy zrychlování popisuje Amdahlův zákon.
- Neexistuje jediná univerzální metrika pro hodnocení výkonnosti.
- Nejobjektivnější je použít sadu benchmarků
 - Problém aritmetický vs. geometrický průměr při summarizaci výsledků
 - aritmetický průměr – výsledek závisí na volbě referenčního počítače; zkresluje, pokud jsou naměřená data velmi variabilní.
 - geometrický průměr - není lineární (např. 2x vyšší geometrický průměr neznamená 2x vyšší kvalitu).

Spolehlivost

INP 2015
FIT VUT v Brně



Obsah

- Definice spolehlivosti
- Ukazatele spolehlivosti
- Modelování spolehlivosti
 - Kombinatorické modely
 - Markovské modely
- Systémy odolné proti poruchám

Pozn.: Bude prezentován jen velmi základní přehled problematiky, podrobněji v (magisterském) kurzu Systémy odolné proti poruchám.

Spolehlivost

- Spolehlivost **byla** definována jako obecná vlastnost *objektu* spočívající ve schopnosti plnit požadované funkce při zachování hodnot stanovených *provozních ukazatelů* v daných mezích a v čase podle stanovených *technických podmínek* (ČSN 01 0102 „Názvosloví spolehlivosti v technice“).
 - *Objekt* – součástka, obvod, funkční jednotka, nebo systém
 - *Provozní ukazatele* – produktivita, rychlosť, výkonnost, spotřeba energie, ...
 - *Technické podmínky* – souhrn specifikací technických vlastností, předepsaných pro požadovanou funkci objektu, způsob jeho provozu, skladování, přepravy, údržby a oprav
- Spolehlivost **je** souhrnný termín používaný pro popis pohotovosti a činitelů, které ji ovlivňují: bezporuchovost, udržovatelnost a zajištěnost údržby (platná terminologická norma ČSN IEC 50 (191))
- Spolehlivost je komplexní vlastnost objektu, která je číselně nekvantifikovatelná – v angličtině **dependability**.

Porucha, chyba, selhání

- **Porucha** – např. poškození obvodu v důsledku přehřátí, což může a nemusí způsobit chybu.
- **Chyba** – odchýlení od korektního stavu systému – např. dojde v důsledku poruchy k chybnému sečtení, což může a nemusí způsobit selhání.
- **Selhání** – systém nepracuje podle specifikace.

Fault (porucha) → **Error (chyba)** → **Failure (selhání)**

Příčiny selhání

- chyba software
- návrhová chyba hardware
- náhodná chyba (v datech)
- vnější rušení – např. chyba v datech, uložených v paměti, např. částicí alfa
- porucha hardware
 - destrukcí – průraz statickou elektřinou, aj.
 - korozí
 - mechanickým poškozením
 - atd.

Klasifikace poruch a chyb

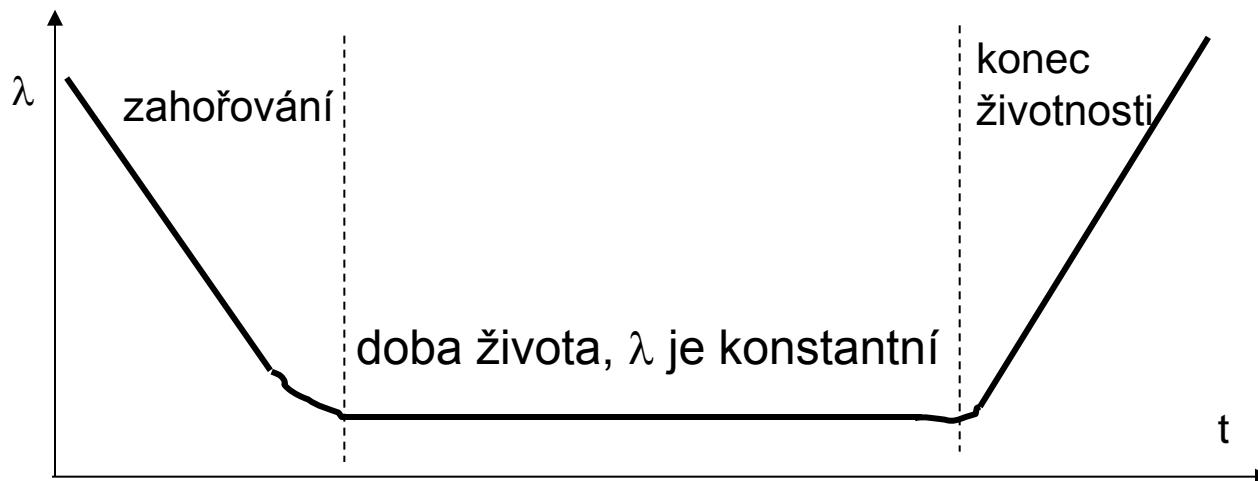
- časově stálé – permanentní, trvalý defekt (hard defect)
- přechodné – objeví se chyba v datech a zase zmizí (soft error, transient error)
- občasné – opakovaně se občas objeví a zase zmizí - intermittent error, např. zlomený drát, nebo vadný mechanický kontakt („vakl“ kontakt)

Klasifikace objektů (systémů)

- neopravované – neopravují se, protože to není ekonomické (např. žárovka)
- opravované – obsahují mechanismy umožňující opravit určité poruchy

Intenzita poruch

- Intenzita poruch λ – četnost, s jakou se systém porouchá.
Vyjadřuje se jako počet poruch za jednotku času.
- Časový průběh funkce $\lambda(t)$ je popsán vanovou křivkou.



- V praxi je často intenzita poruch zjednodušeně modelována konstantou, např. $\lambda = 0,001$ porucha/hod.

Ukazatele spolehlivosti

$R(t)$ – pravděpodobnost bezporuchové činnosti v intervalu $<0, t>$, v angl. reliability. Je to podmíněná pravděpodobnost, a to tím, že v čase 0 je objekt bez poruchy.

$R(t)$ se chová podle exponenciálního zákona

$$R(t) = e^{-\lambda t}$$

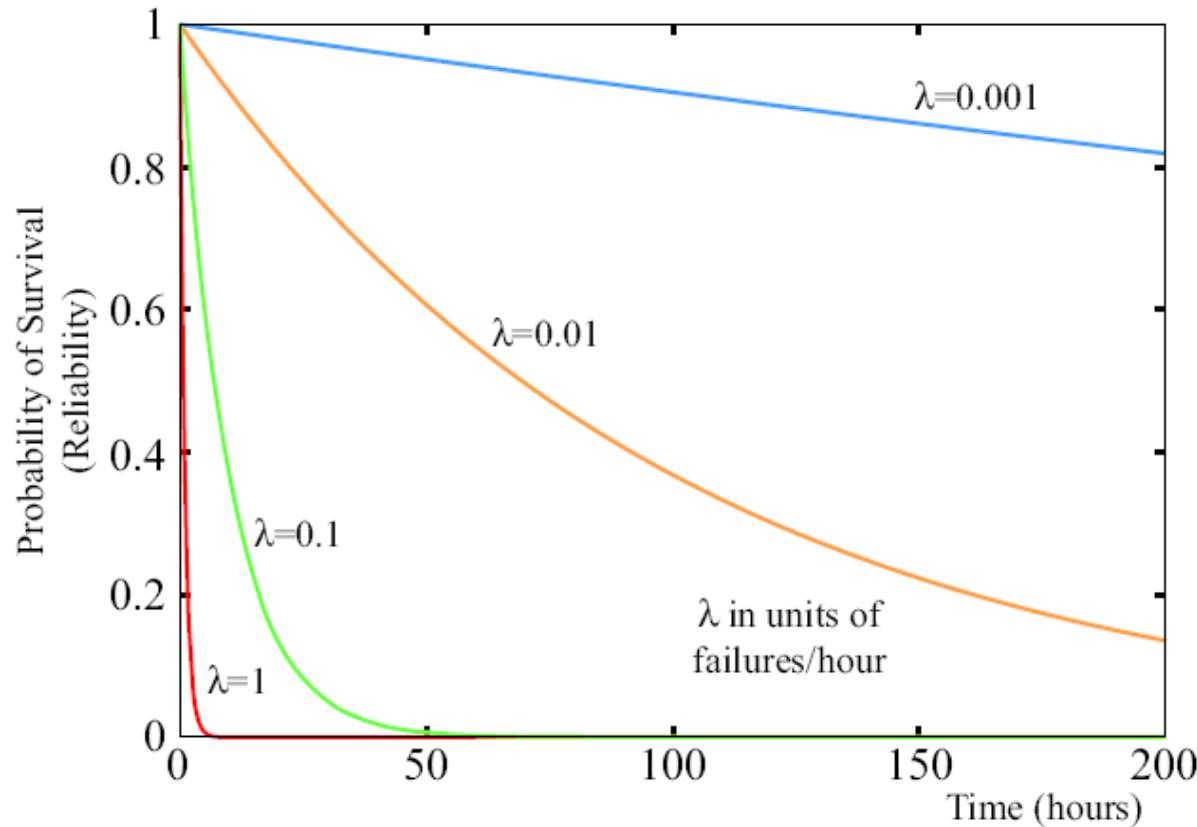
Komplementární veličina je pravděpodobnost výskytu poruchy

$$Q(t) = 1 - R(t).$$

Sledovaný časový interval činnosti (doba mise – mission time) je zásadní:

- u počítače v kosmických aplikacích je např. 10 let (let kosmické sondy na hranici sluneční soustavy)
- u počítače pro letadla je např. 15 hod., tj. asi max. doba letu

Funkce $R(t)$ při různých hodnotách λ



Odolnost proti poruchám – schopnost systému pracovat bezchybně i za přítomnosti poruch. Je to něco jiného, než pravděpodobnost bezporuchové činnosti!

Jak? – díky použití opravných kódu, a maskováním chyb/poruch systémem TMR – viz dále.

Ani velmi spolehlivý systém, postavený z velmi spolehlivých součástek, nemusí být odolný proti poruchám.

Pohotovost (availability) je pravděpodobnost, že v okamžiku t bude systém funkční.

Koeficient pohotovosti $K_p(t)$, v angl. $a(t)$.

Př. Počítač v bance může mít občas poruchu, ale musí se rychle opravit tak, aby to klienti nepoznali. Takže nemusí být odolný proti poruchám.

Střední doba bezporuchové činnosti T_S

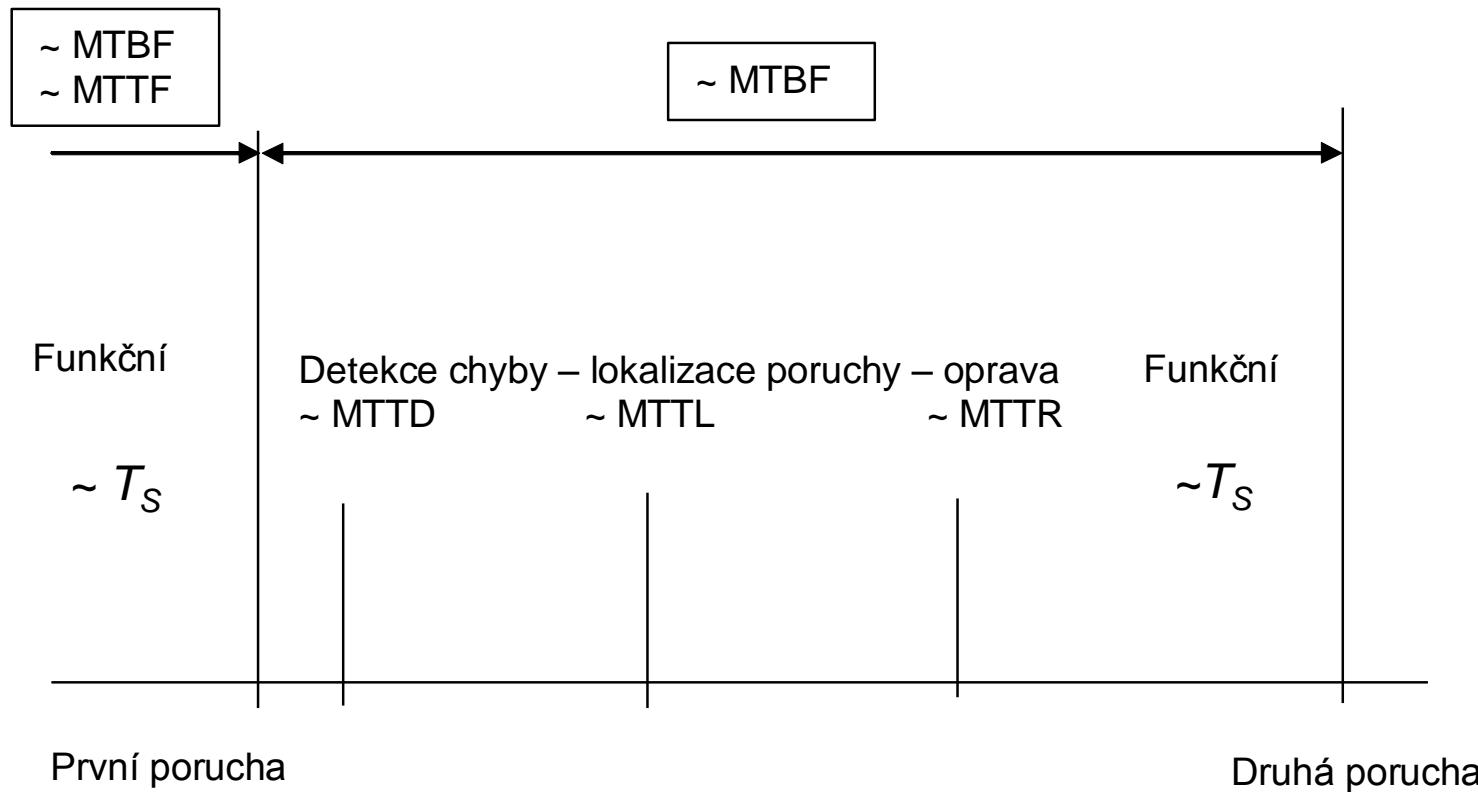
$$T_S = \int_0^{\infty} R(t) d(t)$$

- Pro exponenciální zákon platí $T_S = 1/\lambda$.

$$T_S = \int_0^{\infty} e^{-\lambda t} dt = \frac{1}{\lambda} \quad \text{pokud } R(t) = e^{-\lambda t}$$

- T_S nemá stejný význam jako další veličina, používaná pro opravované systémy, tzv. střední doba mezi poruchami – Mean Time Between Failures (MTBF)
- Pro neopravované systémy lze T_S lze ztotožnit se s střední dobou do (první) poruchy Mean Time To a Failure (MTTF)

Vztahy mezi středními ukazateli spolehlivosti



MTTD – Mean Time To Detect

MTTL – Mean Time To Locate

MTTR – Mean Time To Repair

Pro opravované systémy je analogicky zavedena intenzita oprav μ jako převrácená hodnota střední doby opravy $T_O = 1/\mu$

$$\text{Pohotovost } a = T_S / (T_S + T_O) = \mu / (\mu + \lambda)$$

Bezpečnost (safety) $S(t)$ je pravděpodobnost, že systém buďto pracuje správně, nebo hlásí poruchu, případně chybu v datech. Hodnota $S(t)$ je tedy větší než $R(t)$.

Příklady:

- Systém pro řízení dopravní křižovatky nebo železničního přejezdu se konstruuje jako bezpečný. Když selže, nesmí nikdy nastavit v obou směrech zelenou, resp. nesmí blikat bílé světlo. Bezpečný stav je: nesvítí nic, všude je červená, nebo oranžová, atd.
- Když se porouchá hydraulika letadla, lze je řídit ručně.
- Když se porouchá ventil, tak nejde otevřít.

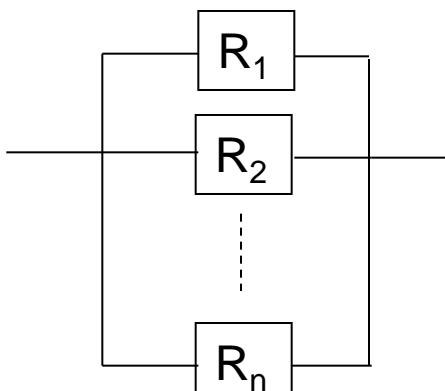
Modelování spolehlivosti: Kombinatorický výpočet $R(t)$

Sériové spolehlivostní zapojení



$$R(t) = \prod_{i=1}^n R_i(t) \quad \text{za dobu } t$$

Paralelní spolehlivostní zapojení



$$Q(t) = \prod_{i=1}^n Q_i(t) \Rightarrow R(t) = 1 - \prod_{i=1}^n (1 - R_i(t))$$

Výčet provozuschopných stavů:

111 ... 1

Výčet provozuschopných stavů:

111 ... 1

011 ... 1

101 ... 1

...

000 ... 1

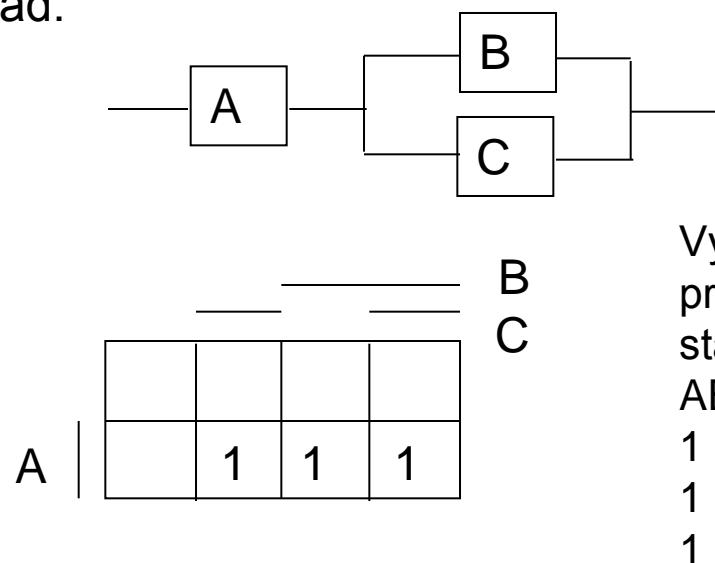
000 ... 0 Ne!

Sériově-paralelní spolehlivostní zapojení se řeší modifikovanou Karnaughovou mapou.

Postup výpočtu:

1. Vytvořit spolehlivostní model analyzovaného systému.
2. Nakreslit spolehlivostní Karnaughovu mapu – 1 odpovídá bezporuchovému modulu, 0 modulu s poruchou. Počet proměnných je roven počtu modulů v systému. U složitých systémů sdružujeme moduly do modulů vyšší úrovně, a ty pak řešíme postupně.
3. Provozuschopné stavy systému vyznačíme v mapě jedničkami.
4. Najdeme disjunktní pokrytí mapy, tj. každá jednička je pokryta pouze jedenkrát. Formálně zaměníme logické operace AND, OR, NOT za součin, součet a jedničkový komplement.

Příklad:



A, B, C jsou pravděpodobnosti bezporuchové činnosti

Výčet provozuschopných stavů:

ABC

1 1 1

1 1 0

1 0 1

Disjunktní pokrytí a úprava:

$$R = A \cdot B + A \cdot (1-B) \cdot C$$

Další metody kombinatorického modelování

Zálohování „m z n“

$$R(t)_{m \text{ z } n} = \sum_{i=0}^{n-m} \binom{n}{i} R^{n-i}(t) [1 - R(t)]^i$$

kde n je počet všech modulů

m je počet požadovaných fungujících modulů

i je počet přijatelných poruch

Př. viz dále - TMR

Další metody kombinatorického modelování

Binomický zákon – vyjadřuje pravděpodobnost P , že nastane r nezávislých událostí na n místech.
Předpokládá, že pravděpodobnost výskytu události p (vadný výrobek, průraz izolace) je stejná.

$$P = \binom{n}{r} \cdot p^r \cdot (1-p)^{n-r} = \frac{n! p^r (1-p)^{n-r}}{r!(n-r)!}$$

Markovské spolehlivostní modely

- Kombinatorické modelování často selhává, protože
 - blokové modely se obtížně sestavují
 - spolehlivostní modely jsou složité
 - proces oprav se popisuje obtížně
- Proto se nejčastěji používají **Markovské spolehlivostní modely**
 - Spolehlivostní model je vytvářen pomocí teorie stochastických procesů s diskrétními stavami a spojitým časem.
 - Markovský proces – náhodná posloupnost hodnot, kde k-tá hodnota závisí pouze na hodnotě k-1.
 - Spolehlivostní model lze vyjádřit grafem, maticí nebo soustavou rovnic. Model popisuje pravděpodobnosti přechodů mezi důležitými stavami systému. Z modelu lze vypočítat klíčové spolehlivostní ukazatele (příklady uvidíme při popisu TMR, NMR atd.).
 - mimo možnosti INP

Zvyšování spolehlivosti systému

Základní princip zvyšování spolehlivosti je **zálohování** součástek, funkčních jednotek, nebo celých systémů (redundance).

Typy záloh:

- **technické vybavení** (zálohy zdvojení, ztrojení, ...)
- **programové vybavení** (alternativní programy, testovací a diagnostické programy)
- **informační** (detekční a opravné kódy – např. parita, Hammingův kód)
- **časové** (opakování operace)

Typy substitučních záloh:

- **zatížená** – intenzita poruch je stejná jako u funkční jednotky – stejný pracovní režim
- **odlehčená** – intenzita poruch je snížená, např. snížením napájecího napětí
- **nezatížená** – intenzita poruch je (teoreticky) nulová

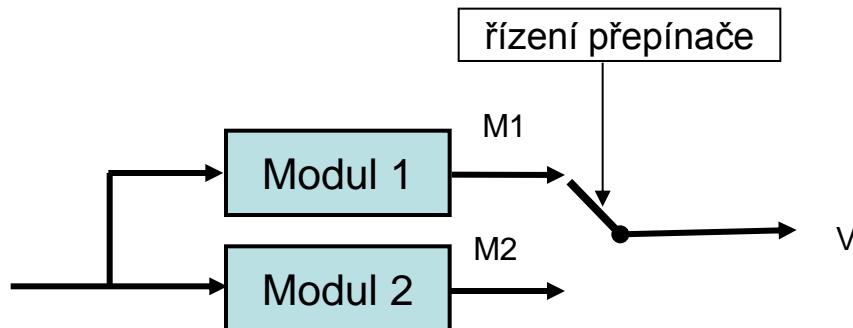
Typy záloh podle využití v čase:

- **statická** – pracuje nepřetržitě po celou dobu funkce systému, je trvale připojená, nebo jako záloha bez přepínání
- **dynamická** – neboli s přepínáním podle potřeby

100% spolehlivost nelze u reálných systémů zajistit nikdy!!!

Techniky zajištění odolnosti proti poruchám: Duplexní systém

- Jednoduchá metoda, která využívá dvě stejně pracující (nicméně ne nutně stejně implementované) verze modulu M.
- Pokud jsou výstupy M1 a M2 různé, není zřejmé na základě jejich pozorování, který výsledek je správný.
- Nelze dosáhnout **maskování chyby** a řízená soustava je ohrožena chybou nebo výpadkem řídicího signálu během přepínání na záložní prvek.
- Cena navíc oproti nezabezpečené verzi: 1 x modul, přepínač, řídicí logika



Techniky zajištění odolnosti proti poruchám

- Statická redundance
 - třímodulová redundance (TMR)
 - NMR – zobecnění TMR
 - N – celkový počet modulů
 - n – počet modulů, jejichž porucha je tolerována

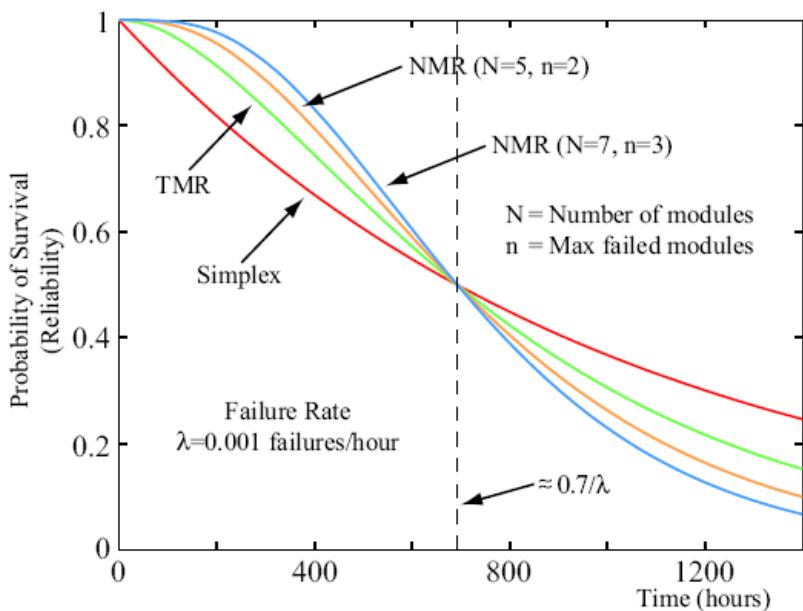
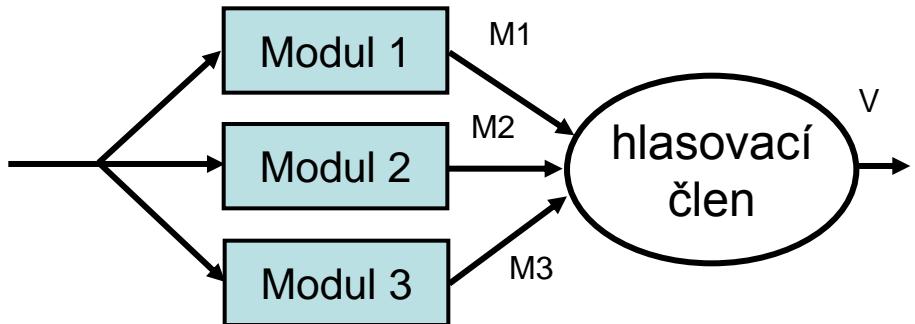


Figure 2.15: The TMR architecture only provides improved dependability over its simplex alternative up to a specific point in the systems lifetime. This point is determined by the failure rate of the simplex system.

Pozn: Simplex = 1 modul



Příklad realizace hlasovacího člena pro 1b výstup
(majoritní funkce)

M1	M2	M3	V
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$R(t) = 3e^{-2\lambda t} - 2e^{-3\lambda t}$$

Techniky zajištění odolnosti proti poruchám

- **Dynamická redundance**

- Při poruše aktivního modulu se systém přepne na záložní modul. Musí existovat detektor poruchy.

- **Hybridní redundance**

- TMR + náhrada poškozených modulů záložními moduly

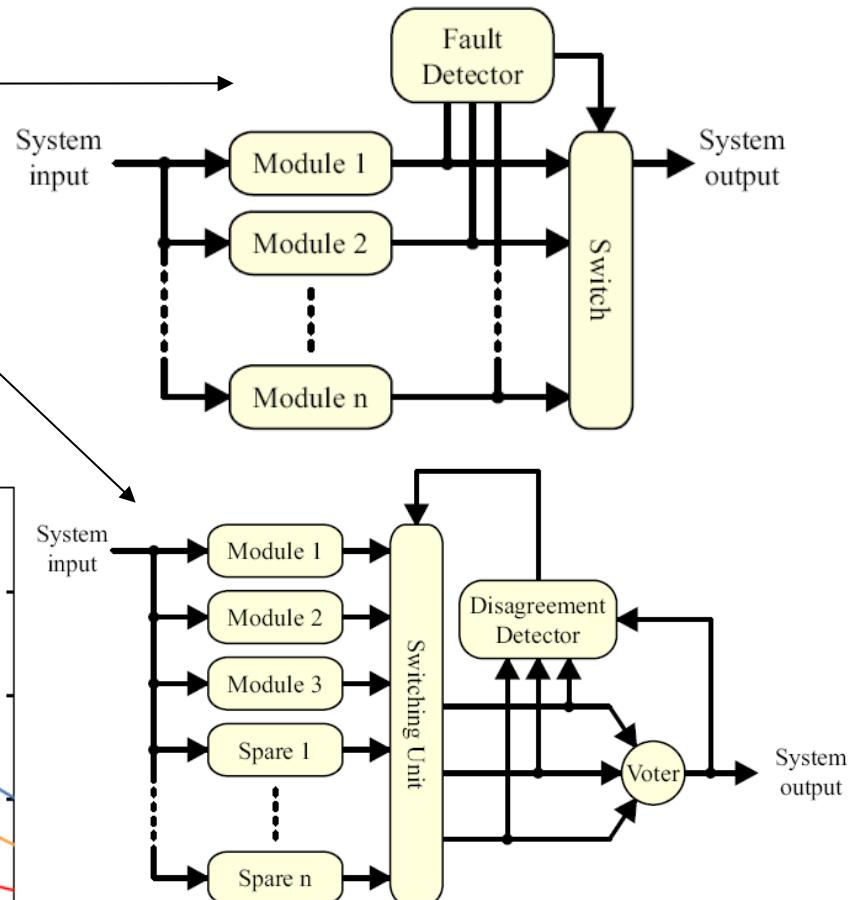
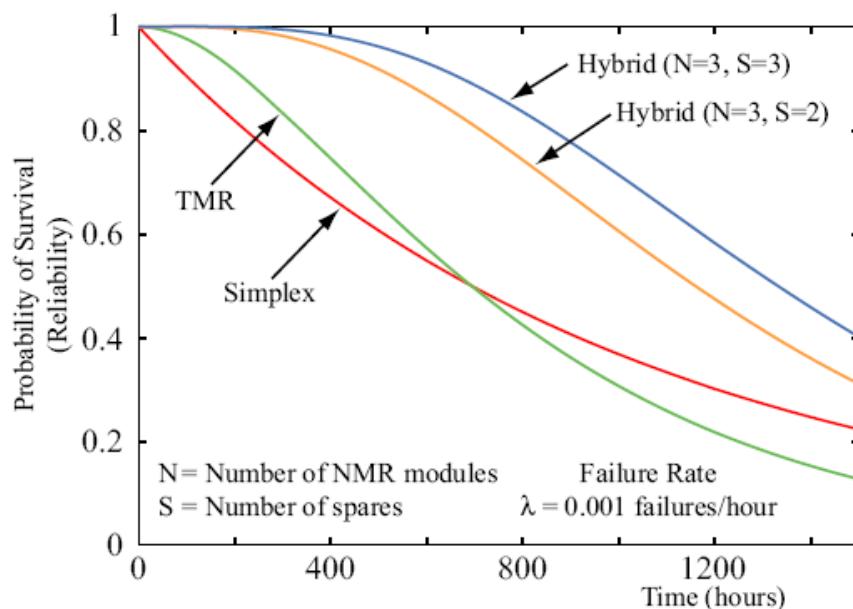


Figure 2.18: This graph shows the change in reliability with time of two hybrid redundancy systems compared to that of simplex and TMR systems. One hybrid system has two spares, $S = 2$, the other three, $S = 3$, both use TMR for masking. The voter, switch and disagreement detector are assumed to never fail.

Spolehlivost z pohledu návrhu

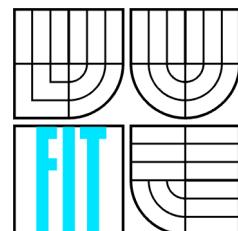
- **Pasivní přístup:** volba spolehlivějších součástek, řízení kvality, stínění, chlazení, bezpečná návrhová pravidla, verifikace návrhu, důsledná dokumentace, testování a diagnostika, nepoužívá se redundance – jde o předcházení poruchám (**fault avoidance**)
- **Aktivní přístup:** použití hw i sw (i časové) redundance, návrh bezpečných a odolných obvodů, detekce chyb v datech, oprava chyb v datech pomocí opravných kódů, hlavně jde o tolerování (maskování) poruch, izolaci poruch a rekonfiguraci s cílem vyřadit vadné součásti (**fault tolerance**).
- Praxe: kombinace pasivního i aktivního přístupu

Literatura

- Drábek V.: Systémy odolné proti poruchám. Přednášky FIT VUT 2009
- Hlavička J. et al.: Číslicové systémy odolné proti poruchám. Vydavatelství ČVUT Praha 1992
- Polsterová H.: Spolehlivost v elektrotechnice, skriptum FEKT VUT v Brně, 2003
- Greensted A.: A reliability Engineered Multicellular Architecture Inspired by Endocrinology: The BioNode System. PhD Thesis, The University o York, 2004, 184 s.

Úvod do paralelních systémů

INP 2015



Obsah

- Paralelní přístup z pohledu praxe
- Moderní procesory – přehled vývoje architektur
 - Skalární
 - Superskalární
 - Vícevláknové
 - Vektorové
- Technologické důvody pro zavedení paralelismu na vyšších úrovních
 - Proč nestačí jedno jádro?
 - Vícejádrové procesory
- Distribuované systémy
 - Základní koncepty architektury
 - Propojovací sítě a jejich topologie
 - Vliv architektury na výkonnost paralelního systému

Podrobně je problematika paralelních systémů probírána v magisterských kurzech:

- Architektura procesorů
- Paralelní a distribuované algoritmy
- Architektura a programování paralelních systémů

Proč paralelní přístup?

- Paralelní přístup je nutný pro řešení **složitých úloh**, které vyžadují vysoký výpočetní výkon.
 - paralelní počítání (na vhodně propojených počítačích běží paralelní algoritmus)
- Stávající **technologická omezení** nutí pro zajištění co nejvyšší výkonnosti používat paralelní přístup na všech úrovních (instrukcí, dat, vláken atd.)
 - Jednoprocесоровé systémy potřebují k navýšení výkonnosti neúměrně vysoký příkon.
 - Vícejádrové procesory poskytují pro daný povolený příkon vyšší výkonnost.

Klasifikace procesorů

- Architekturu procesoru charakterizují parametry:
 - m - počet instrukcí, které se v jednom okamžiku **vydávají** ke zpracování
 - r - počet současně prováděných (rozpracovaných) instrukcí
- **Subskalární procesory** (von neumannovské) $r = 1$, $m = 1$
 - doba provádění programu je součtem dob trvání jednotlivých instrukcí.
 - nová instrukce může být vydána až po dokončení předchozí.
- **Skalární procesory** $r > 1$, $m = 1$
 - využívají **řetězené zpracování instrukcí**, kdy je v každém taktu vydávána maximálně jedna instrukce, ale současně jich může být rozpracováno několik.
- **Superskalární (vícecestné) procesory** $r > 1$, $m > 1$
 - vydávají k zpracování více než jednu instrukci v jednom taktu
 - obvykle kombinováno s řetězeným zpracováním

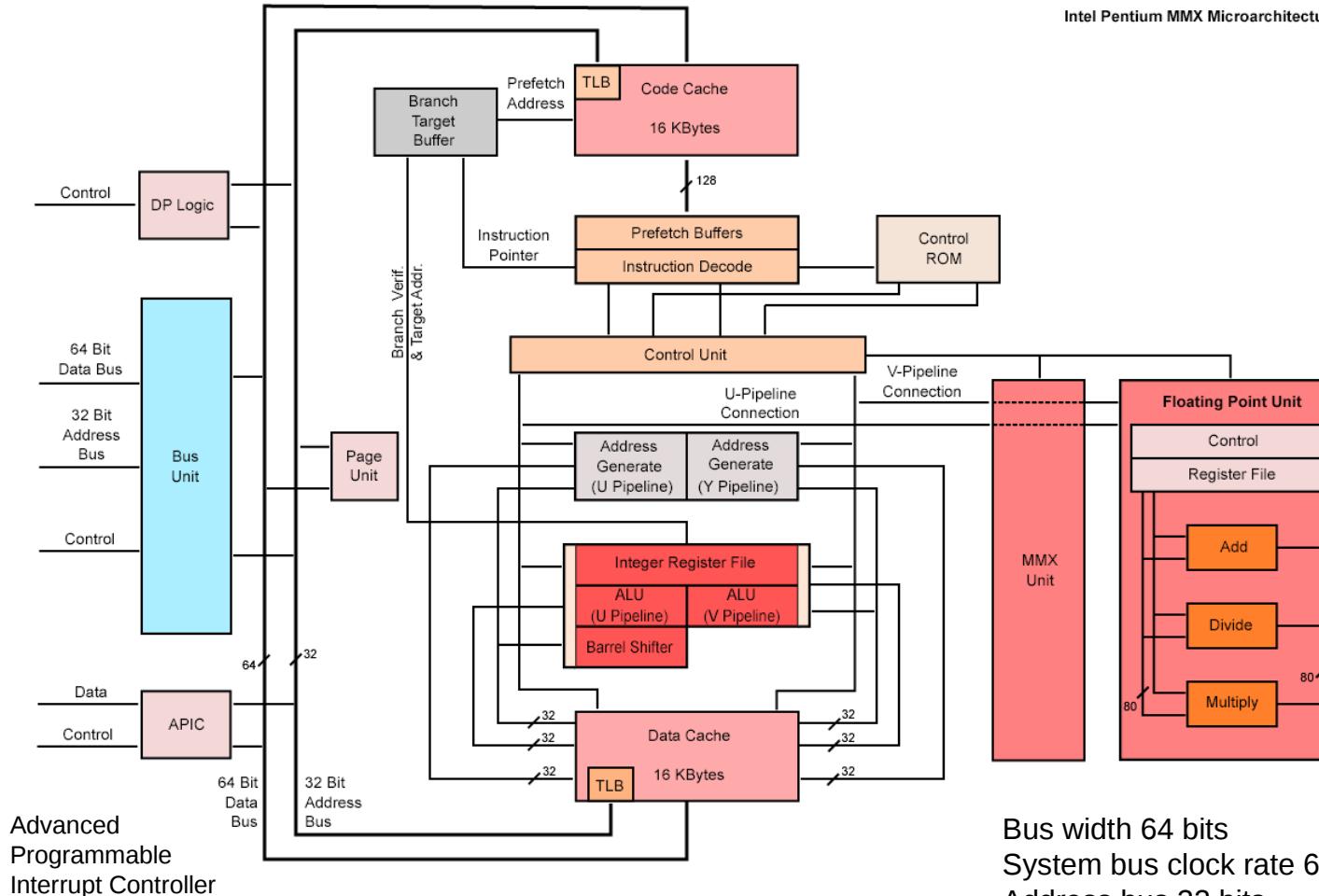
Techniky paralelismu

- Funkční paralelismus
 - na úrovni instrukcí (**ILP** – Instruction level Parallelism)
 - Pipelining, vícecestné zpracování
 - na úrovni vláken (**TLP** – Thread Level Parallelism)
 - **Vícevláknové procesory**
 - na úrovni procesorů (jader)
 - **Vícejádrové procesory**
- Datový paralelismus
 - provádění stejných operací nad různými daty → **vektorové zpracování** (**SIMD** – Single Instruction, Multiple Data)
 - V současných procesorech implementován jako specializované sady instrukcí: **SSE**, **AVX**, dříve např. **MMX** (Intel), **3DNow** (AMD) apod.

Superskalární procesory

- m-cestný (řetězený) procesor
 - Dovoluje vydávat v jednom taktu až m instrukcí.
 - Podporuje řetězené zpracování instrukcí, kdy se využívá několik paralelně pracujících linek (paralelní načítání, paralelní dekódování instrukcí atd.)
 - Mezi stupni řetězené linky jsou vícemístné **buffery** namísto registrů.
 - **Data-flow** zpracování - rozpracované instrukce, které čekají na operandy, je možné odložit do bufferu a probudit je, až budou operandy připraveny.
- Provádění instrukcí má tři části
 - Načtení/dekódování/rozesílání (pořadí instrukcí dodrženo)
 - Provedení instrukcí v paralelně pracujících linkách
 - Dokončení vykonávání (podle pořadí instrukcí)
- Charakteristika
 - Dynamické plánování instrukcí
 - Provádění instrukcí mimo pořadí
 - Spekulativní provádění skoků (Branch Target Buffer)
 - Pokročilé řešení konfliktů ILP – přejmenování registrů (HW podpora), přeskládání instrukcí, pokročilý přístup do cache atd.

Př. Intel Pentium (1993)



Advanced
Programmable
Interrupt Controller

Bus width 64 bits
System bus clock rate 60 or 66 MHz
Address bus 32 bits
Addressable Memory 4 GB
Virtual Memory 64 TB
Superscalar architecture
Runs on 5 volts
16 KB of L1 cache
U and V pipeline (m=2)

Superskalární procesory

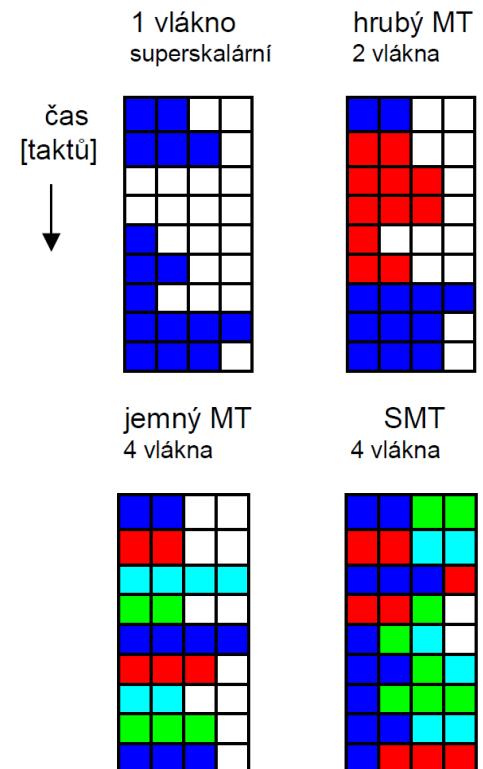
- Superskalární přístup dosáhl svých mezí
 - Výkonnost = IPC x f
 - IPC (Instructions Per Clock) a f [MHz] jdou proti sobě!
 - Vysoký IPC (až 6) => složitý HW => nízká f
 - Vysoká f => jednoduší HW => nízké IPC
 - Pro $m > 4-8$ již většinou nemá smysl (zrychlení je <3x)
 - Zvyšování f je nevýhodné z hlediska příkonu - plánované projekty superskalárních procesorů s vysokými kmitočty (9,2 GHz a 10,2 GHz firmy Intel) byly opuštěny pro vysoký příkon
- Pro srovnání: výkonnost skalární koncepce
 - Výkonnost = f / CPI
 - CPI – Clocks Per Instruction
- Další vývoj: vícevláknové a vícejádrové procesory

Multivláknové procesory

- **Vlákno** (thread) je posloupnost instrukcí vyžadující určitý adresový prostor a čas CPU. Vlákna jsou (oproti procesům) lehká, přepnutí kontextu je rychlejší, kopíruje se méně dat.
- Vlákna tvořená v rámci procesu sdílí jeho adresový prostor a další prostředky – kód, hromadu aj. K vláknu patří jen ukazatele IP a SP (instruction pointer, stack pointer), PSW (program status word) včetně myid a priority, sada registrů a zásobník.
- Nejčastěji je to programátor, kdo explicitně vlákna vytvoří při paralelizaci programu.
- **Multivláknový provoz** (MT - MultiThreading)
 - **Časový** – vlákna se střídají na jednom CPU (TMT)
 - **Prostorový** – vlákna běží paralelně v multiprocesorovém systému se sdílenou pamětí (P procesorů / P vláken)
 - **Časoprostorový** (na P procesorech běží R vláken a R>P)

Multivláknové procesory

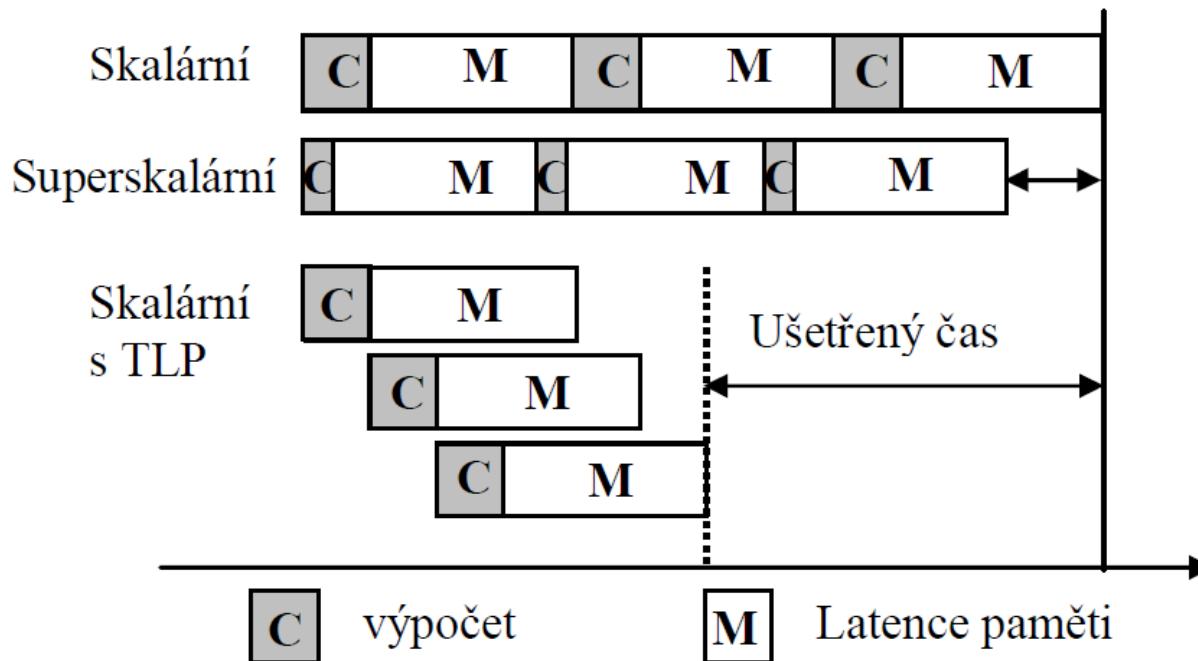
- Z pohledu HW
 - Prostředky sdílené vlákny:
 - Procesor, cache, prediktory skoků
 - Prostředky replikované pro každé vlákno
 - Sada registrů, PC, SP, PSW, řadič přerušení
 - Je třeba dodat nový HW – např. pro výběr vlákna
- Techniky TMT
 - **Hrubý MT**
 - Jedno vlákno běží řadu taktů, k přepnutí kontextu dochází pouze při výskytu události s dlouhou latencí, která by vedla k zastavení linky.
 - **Jemný MT**
 - každém taktu se přepíná na jiné vlákno (prokládání vláken).
 - **SMT, souběžný (současný) MT** (Simultaneous Multithreading).
 - V každém taktu přepíná kontext několika vláken současně, protože v jednom taktu se zpracovávají instrukce z několika vláken.



(4-cestný superskalární CPU, TNT)

Multivláknové procesory

- Podpora více vláken v HW je asi nejzajímavější technika pro překlenutí vysoké latence přístupu do paměti.
 - Musí však existovat dostatek vláken.
 - Doba přepnutí vlákna (přepnutí kontextu) musí být krátká.
 - Na obrázku je porovnání typického výpočtu pro různé architektury:



Př. Počet vláken pro TMT

- Jestliže chceme tolerovat latenci přístupů do paměti (např. $L = 70$ taktů) multivláknovými CPU, kolik vláken (N) bude třeba pro získání maximální účinnosti, když jedno vlákno běží průměrně $R=10$ taktů a přepnutí kontextu trvá $S = 3$ takty?

$$(N - 1)R + NS \geq L$$

$$(N - 1)10 + 3N \geq 70$$

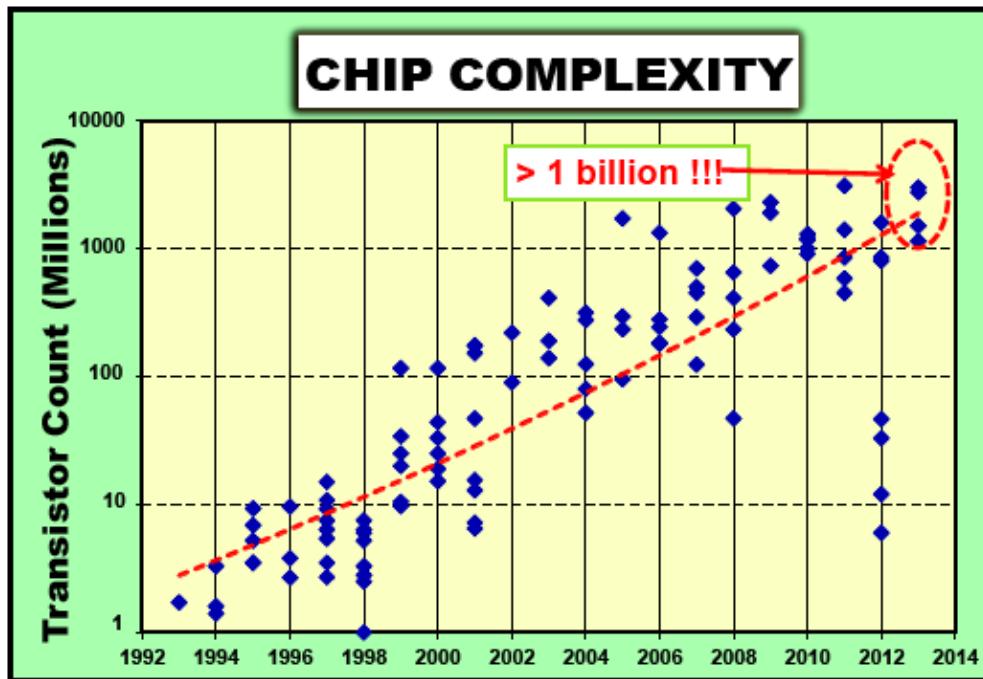
$$13N \geq 80$$

$$N \geq 6,15$$

$$N \geq 7$$

Moorův zákon a výkonnost procesorů

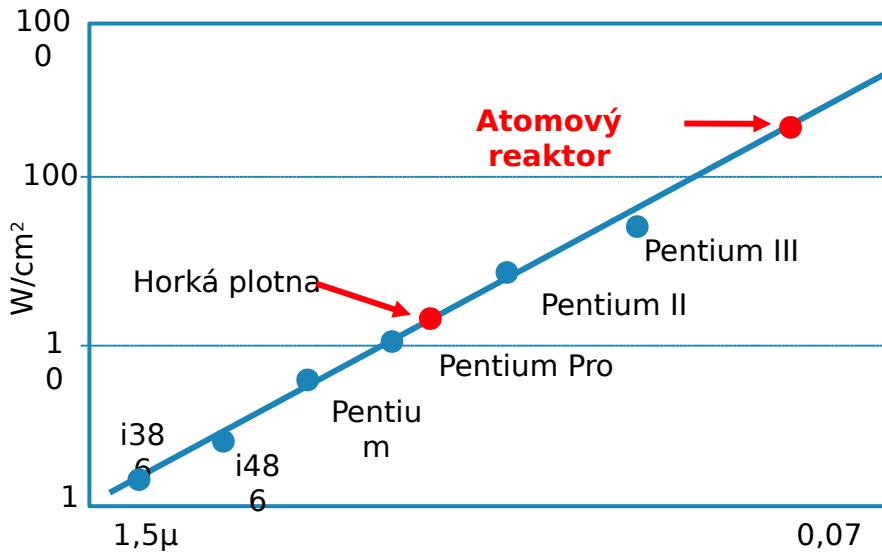
- Dříve (~ do 2005): výkonnost procesorů rostla zejména díky možnosti zvyšování pracovní frekvence a miniaturizaci tranzistorů
 - Možnost implementovat složitější, sofistikovanější HW
 - Rostoucí složitost a frekvence však vyžaduje vyšší příkon
- Současnost: roste složitost, ale příkon (frekvenci) už příliš zvyšovat nelze (CMOS)
 - Vyšší výkonnost lze dosáhnout **zvýšením stupně paralelismu**



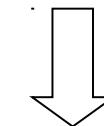
Zdroj: International Solid-State Circuits Conference, 2013

Technologické limity

- Frekvenci f nelze zvyšovat nad jistou mez => problém s odvedením tepla
- Jaká je rozumná mez pro příkon? Dnešní procesory mají cca $100 \text{ W}\cdot\text{cm}^{-2}$
- **Při snížení Vdd je nutné snížit frekvenci** (CMOS) – min. Vdd je cca 0,5V
- Dynamická kapacita C – přibližně odpovídá počtu tranzistorů a jejich aktivitě



$$P = C \cdot V_{dd}^2 \cdot f$$



$$P \approx V_{dd}^3$$

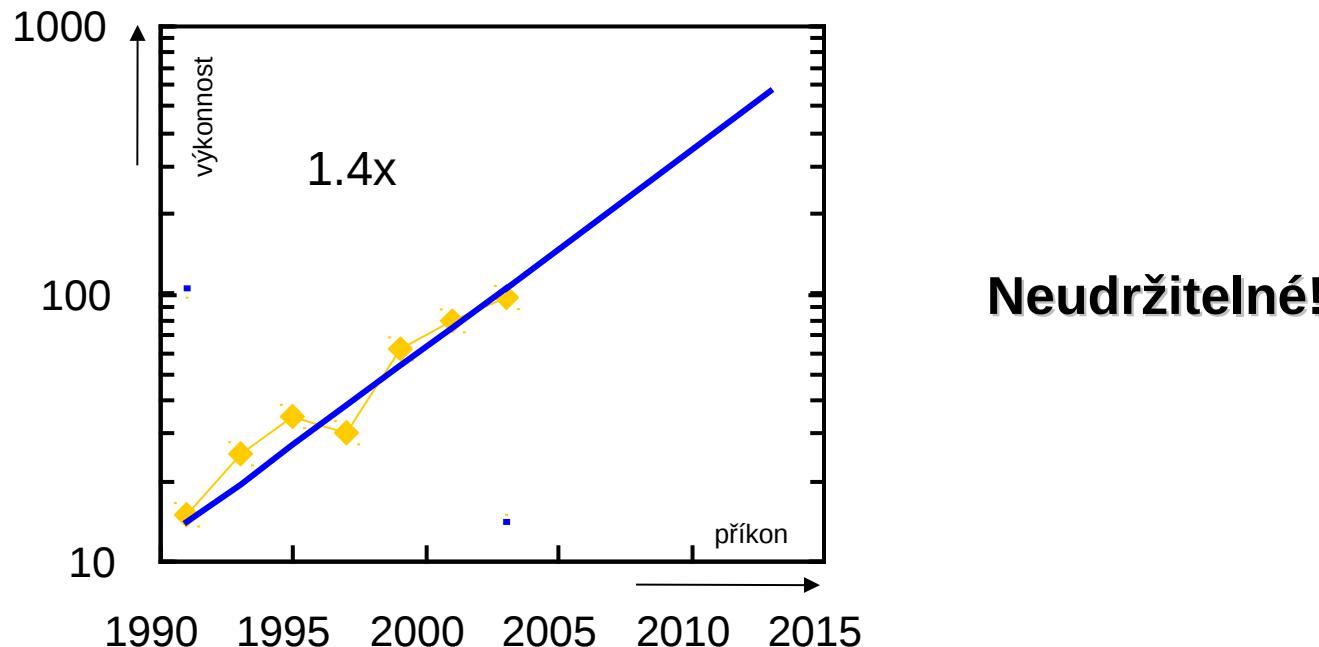
Na příkon má největší vliv Vdd.

- **Požadujeme** co nejvyšší výkonnost za rozumnou cenu (počet tranzistorů) a s rozumným příkonem.

[Pollack, F. J. : "New microarchitecture challenges in the coming generations of CMOS process technologies", <http://research.ac.upc.edu/HPCseminar/SEM9900/Pollack1.pdf>]

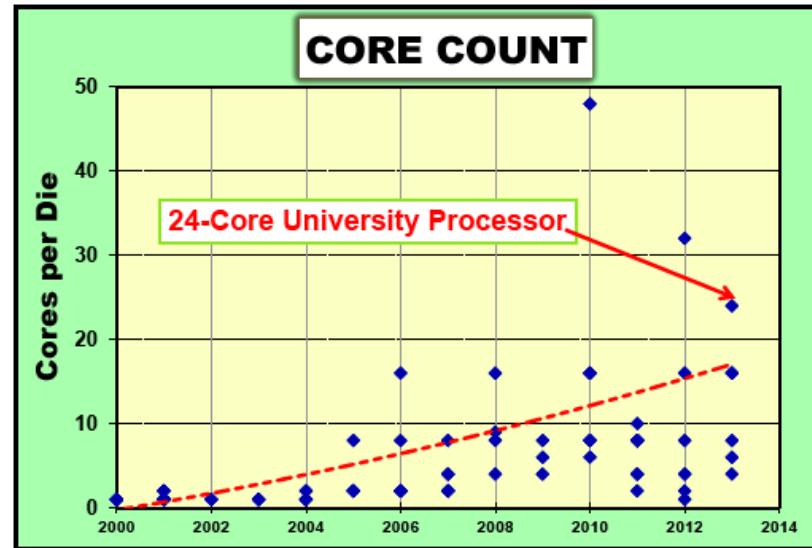
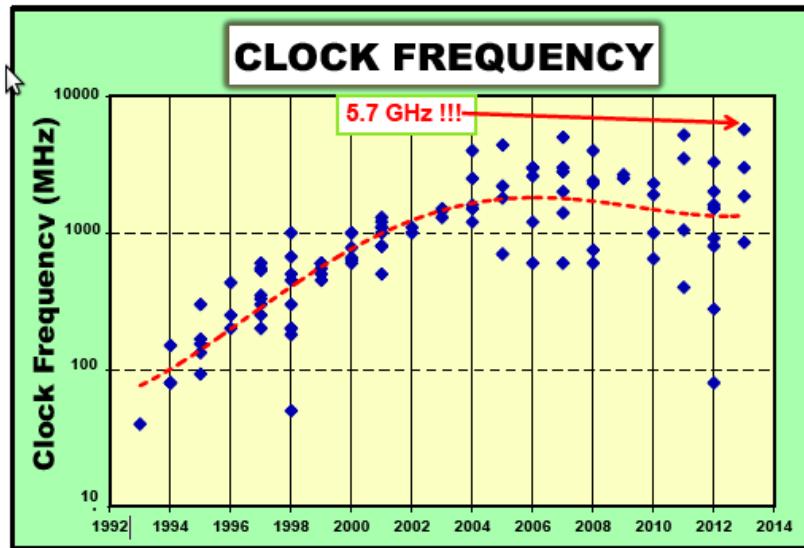
Příkon vs. výkonnost

- Příkon roste úměrně složitosti
- **Pollackovo pravidlo** (empirické)
 - Efektivní (užitečná) výkonnost procesorů je přibližně úměrná druhé odmocnině jejich složitosti
 - 2x více tranzistorů => jen 1,4x vyšší výkonnost



Trendy v oblasti vývoje výpočetních systémů

Zdroj: International Solid-State Circuits Conference, 2013



- Redukce příkonu (snižování Vdd a frekvence)
=> snížení výkonosti
- Kompenzace v podobě zvýšení stupně paralelismu
- CLOCK: IBM System Z: 5,7 GHz, 2,75 mld. tranzistorů
- CORE: 24-jádrový procesor, Fudan University, Shanghai

Trendy v oblasti vývoje výpočetních systémů

International Technology Roadmap for Semiconductors, www.itrs.net

- Předpovědi vývoje různých parametrů v oblasti polovodičů a jejich aplikací
- Zde ukázka předpovědi pracovní frekvence procesorů

	<i>Year of Production</i>	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018
Předpověď 2008	Chip Frequency (MHz) On-chip local clock -- WAS	5.454	5.875	6.329	6.817	7.344	7.911	8.522	9.180	9.889	10.652
Předpověď 2011	Chip Frequency (MHz) On-chip local clock – 2011 IS*	3.462	3.600	3.744	3.894	4.050	4.211	4.380	4.555	4.737	4.927

	<i>Year of Production</i>	2019	2020	2021	2022	2023	2024	2025	2026
Předpověď 2008	Chip Frequency (MHz) On-chip local clock -- WAS	11.475	12.361	13.315	14.343	15.451	16.640	-	-
Předpověď 2011	Chip Frequency (MHz) On-chip local clock – 2011 IS*	5.124	5.329	5.542	5.764	5.994	6.234	6.483	6.743

Zdroj: ITRS 2011 Executive Summary, www.itrs.net

A New Era...

THE OLD

Performance
Equals Frequency

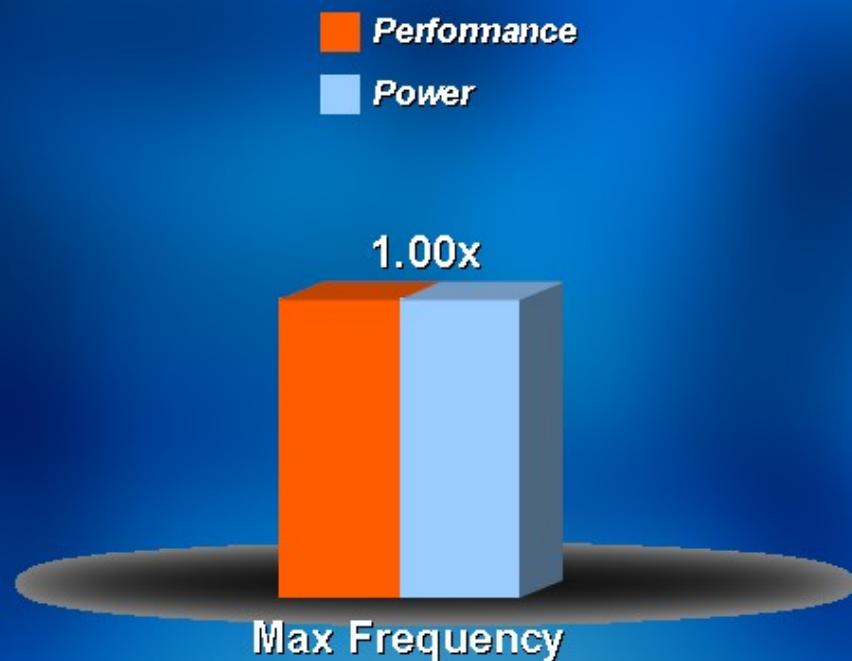
Unconstrained Power
Voltage Scaling

THE NEW

Performance Equals IPC
Multi-Core Power Efficiency
Microarchitecture Advancements



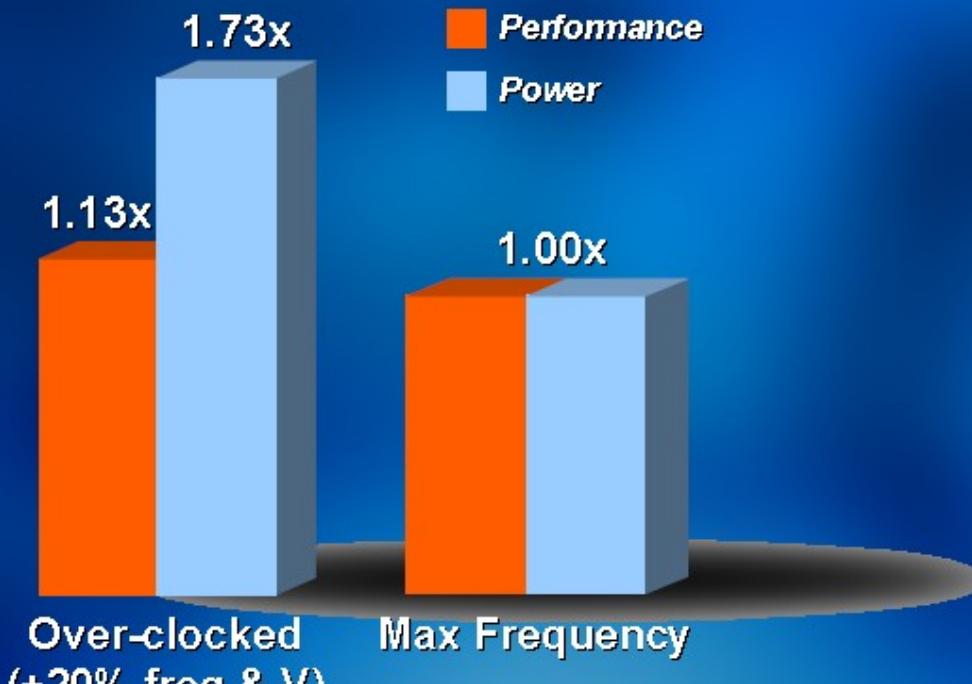
Why Multi-Core?



Relative single-core frequency and Vcc

Over-clocking

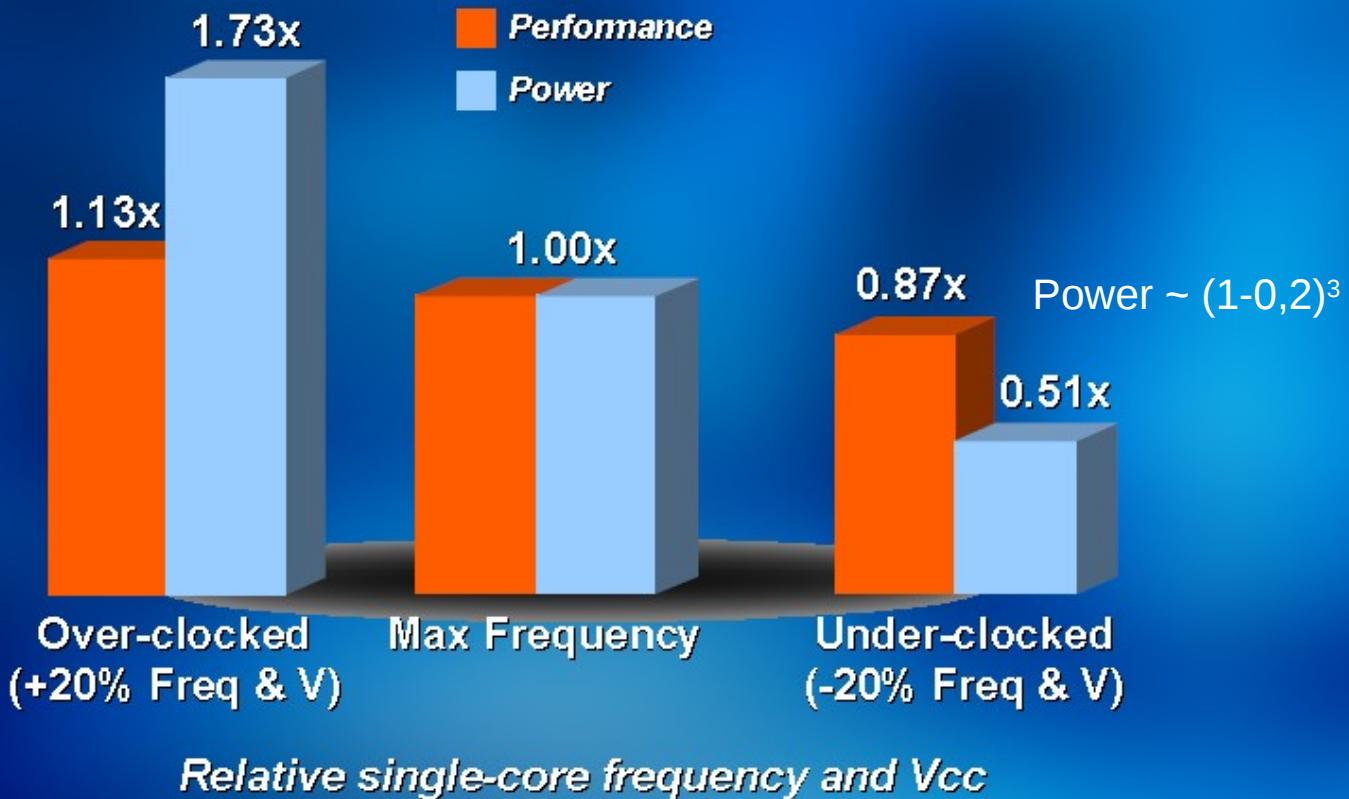
Power $\sim (1+0,2)^3$



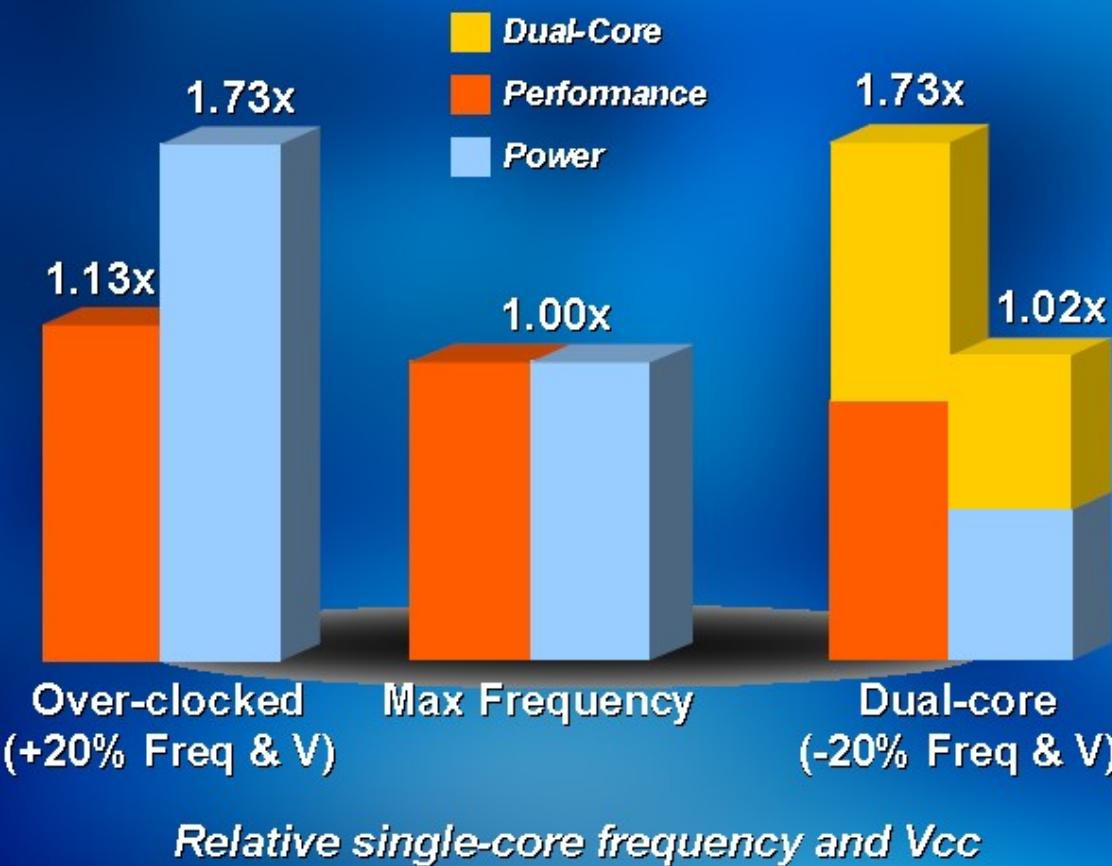
Relative single-core frequency and Vcc

Under-clocking

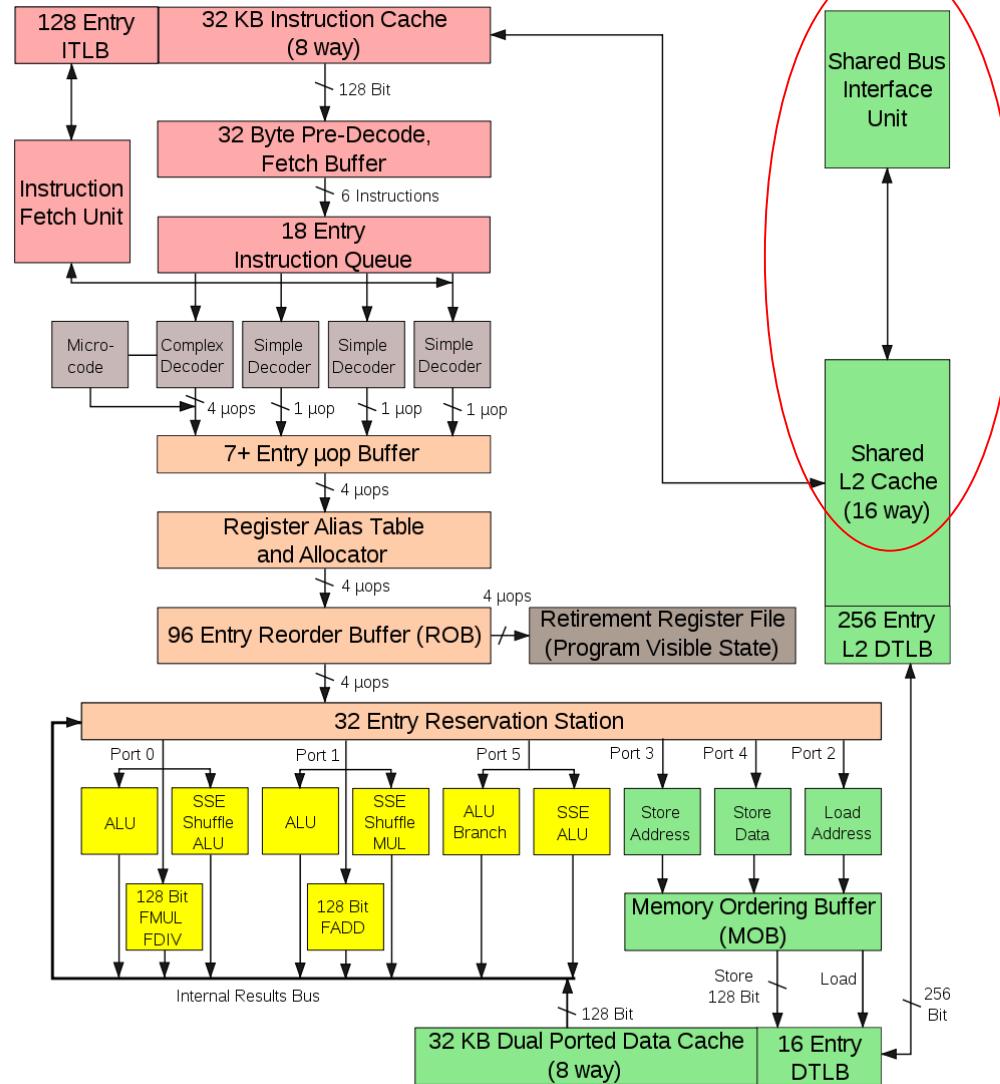
Power $\sim (1+0,2)^3$



Multi-Core Energy-Efficient Performance



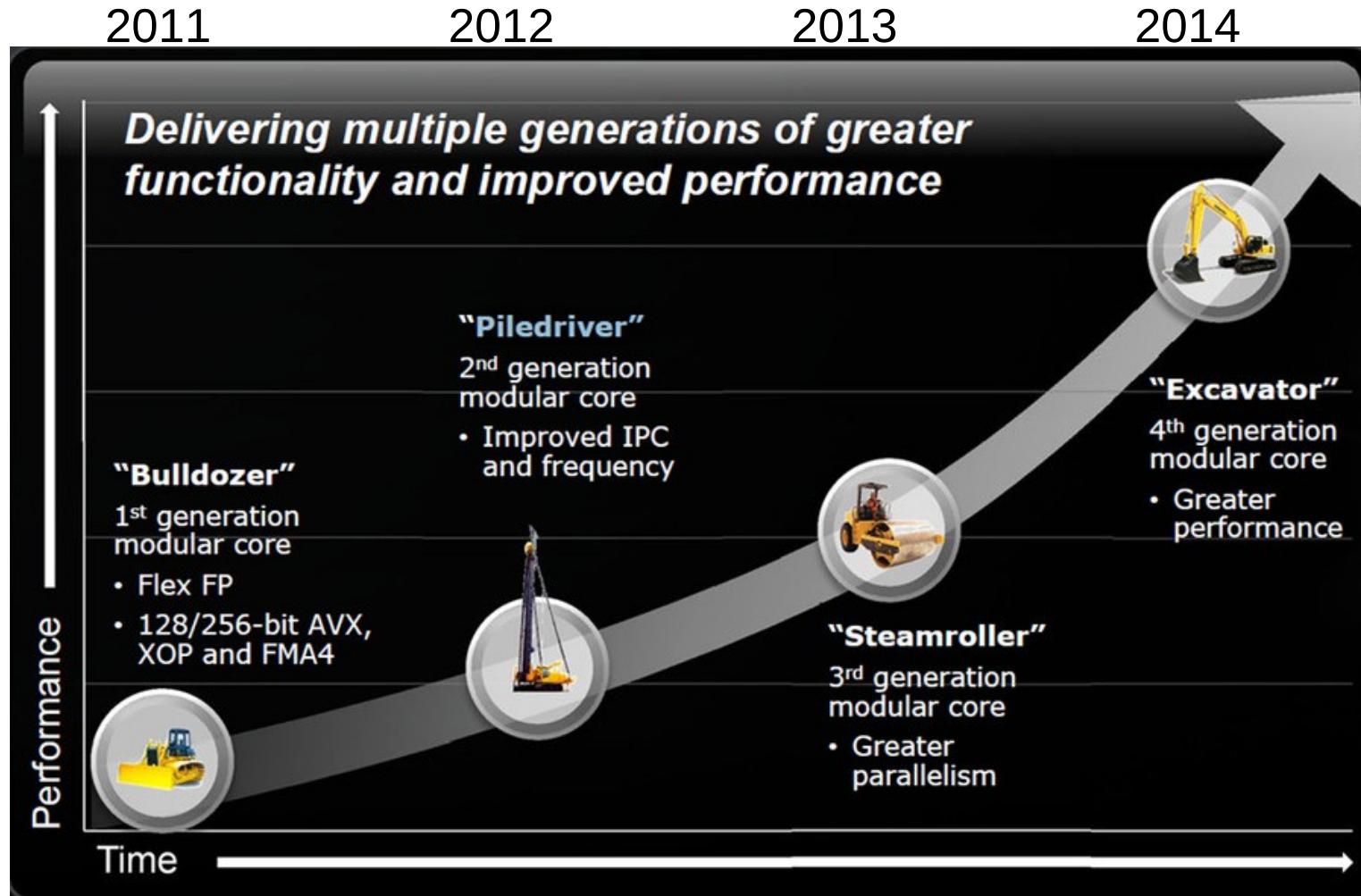
Intel Core 2



Sdíleno!

Intel Core 2 Architecture

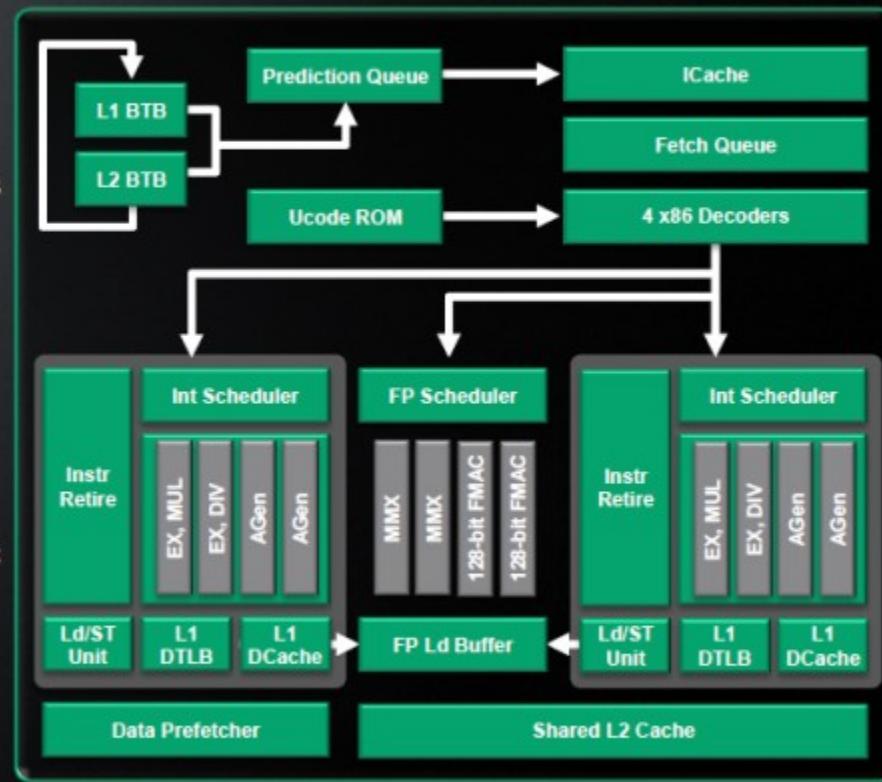
Vývoj moderních architektur AMD



Př.: AMD Piledriver (2012)

32nm "PILEDRIVER" COMPUTE MODULE x86 CORE REDESIGN, SECOND GENERATION BULLDOZER

- Shared Fetcher/prediction pipeline - 64KB I-Cache
- Shared 4-way x86 decoder
- Shared Floating Point Unit - dual 128-bit FMA pipes
- Shared 16-way 2MB L2; 1024-entry TLB cache
- Dedicated integer cores
 - Register renaming based on physical register file
 - Unified scheduler per core
 - Way-predicted 16KB L1 D-cache
 - Out-of-order Load-Store Unit
- ISA additions: AVX, AVX1.1, FMA3, AES and F16C
- Lightweight profiling support in HW
- "Piledriver" performance increase over "Stars"
 - 14% improvement for desktop5
 - 25% improvement for notebook2
 - AMD Turbo Core 3.0



Multithread, multicore - shrnutí

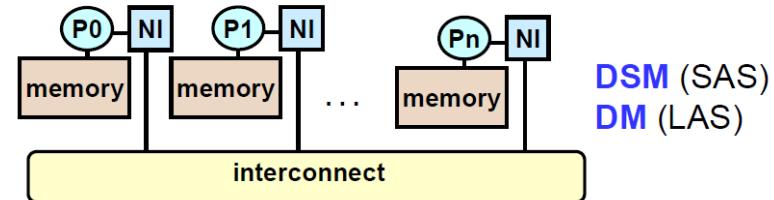
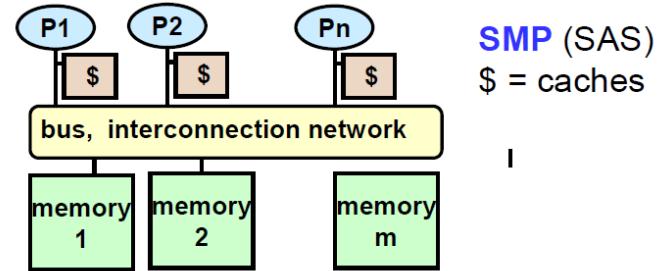
- Podpora vícevláknového zpracování (multitasking, multithreading) může mít různou podobu
 - Přepínání úloh pouze z **úrovně OS** – dnes se již samostatně téměř nepoužívá
 - Přepínání s **podporou HW CPU** – nejedná se o paralelní zpracování, úlohy běží v časovém multiplexu (hrubý/jemný MT)
 - **Vícecestné procesory** – souběžný MT, výhradně s HW podporou CPU, paralelní vykonávání instrukcí různých úloh (pokud je to možné – sdílení určitých komponent CPU)
 - **Vícejádrové procesory** – každé jádro může autonomně spouštět různé úlohy (vlákna nebo i procesy).
Běžně se používá v kombinaci s dříve uvedenými přístupy.
- Komunikace mezi vlákny se provádí přes sdílenou paměť.
- Programování nejčastěji pomocí OpenMP (C/C++)

Koncepty paralelního zpracování

- Procesory **nekomunikují** (n krát rychlejší než sekvenční počítač)
 - Mohu spustit současně více různých úloh, každou na jednom procesoru
 - Mohu spustit stejnou úlohu na n procesorech, ale s jinými parametry (např. program pro předpověď počasí s různým nastavením počátečních podmínek – zrychlení n -krát oproti sekvenčnímu počítači)
- procesory **komunikují** (spolupracují) za účelem vyřešení složité úlohy v rozumném čase, jedná se o **paralelní systém**
 - Návrh řešení určité úlohy na paralelním systému se nazývá **paralelní programování**
 - Procesory společně řeší jednu úlohu, musí být vhodně propojeny
 - Někdy je možné dosáhnout větší zrychlení než v předchozím případě!
 - Příklad: paralelní řazení Enumeration sort (pro k prvků)
 - Pokud je k dispozici k^2 procesorů, je časová složitost $O(\log k)$
 - Příklad: paralelní řazení Bucket sort (pro k prvků)
 - Pokud je k dispozici $\log(k)$ procesorů, je časová složitost $O(k)$

Základní varianty paralelních systémů

- Se sdílenou pamětí (shared memory, SM)
 - Obvykle řešeno jako tzv. UMA – uniform memory access, kdy latence přístupu do sdílené paměti je stejná u všech procesorů (SMP – symetrický multiprocesor).
 - Programování pomocí OpenMP
 - **Výhoda:** efektivní komunikace
 - **Nevýhoda:** max. pouze ~ 32 procesorů (v závislosti na použité propojovací síti)
- S distribuovanou pamětí (distributed memory, DM)
 - Každý procesor má svoji (privátní) paměť
 - Programování (komunikace) pomocí knihovny zasílání zpráv (MPI – Message Passing Interface), případně přes síťové sokety
 - **Výhoda:** škálovatelnost
 - **Nevýhoda:** vyšší komunikační režie
- Se distribuovanou sdílenou pamětí (distributed shared memory, DSM)
 - Obvykle řešeno jako tzv. CC-NUMA (cache coherent non-uniform memory access)



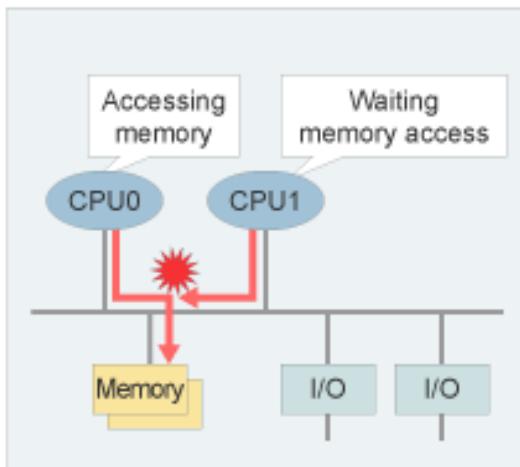
Propojení víceprocesorových systémů

- Komunikace mezi procesory a přístup ke sdíleným prostředkům jsou realizovány přes propojovací sítě

např. sběrnice,

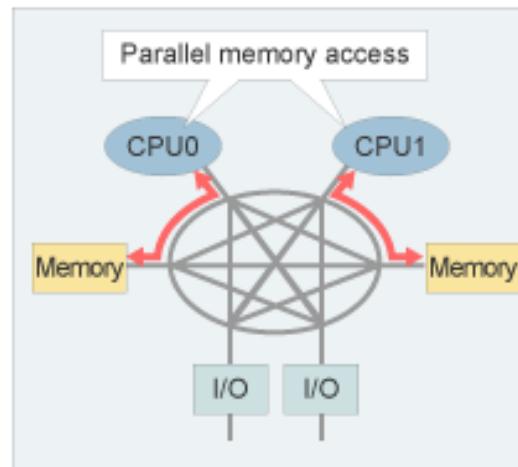
úplné propojení,

Xbar apod.



CPU1 can't access memory until
CPU0 completes memory access

Performance bottleneck

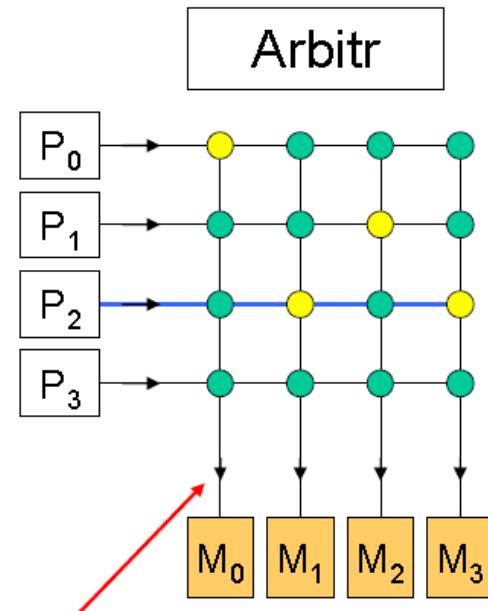


CPU0 and 1 access memory in parallel

(viz dále)

Křížový přepínač (X-bar)

- Propojení 1:1 mezi vstupy a výstupy.
- Je možné propojit 1 vstup k několika výstupům.
- Více vstupů k jednomu výstupu se připojit nesmí.
- Je nutné použít arbitra.
- Cena: p^2 přepínačů (řešení založeno na multiplexorech) – drahé řešení
- Použití: propojení procesorů s procesory nebo paměťovými moduly



Způsob propojení procesorů a typ linek má zásadní vliv na výkonnost!

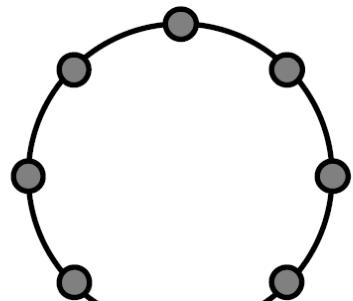
Zahltit sběrnici není obtížné

- Uvažme procesory (každý s I-cache a D-cache) propojené navzájem sběrnicí a připojené k paměti
 - Hit rate $h_i = 98\%$ v I-cache
 - Hit rate $h_d = 95\%$ v D-cache
 - Procesor má výkonnost 250 MIPS
 - Přístup do D-cache potřebuje 1/3 instrukcí (~ 75 MIPS)
- Počet výpadků v I-cache $2\% \times 250 \text{ MIPS} = 5 \text{ M výpadků /s}$
- Počet výpadků v D-cache $5\% \times 75 \text{ MIPS} = 3,75 \text{ M výpadků /s}$
- Celkem $8,75 \text{ M výpadků/s}$
- Při každém výpadku se po sběrnici přenáší blok o velikosti 16B.
- Potřebná šířka pásma je pro každý procesor $8,75 \text{ M výpadků/s} \times 16\text{B} = 140 \text{ MB/s}$
- Kolik procesorů zahltí sběrnici, která má propustnost 1GB/s?
- $N = 1000 / 140 \Rightarrow 7 \text{ procesorů}$
- Závěr: Větší počet procesorů nemá smysl propojovat sběrnicí!

Přímé propojovací sítě – příklady topologií



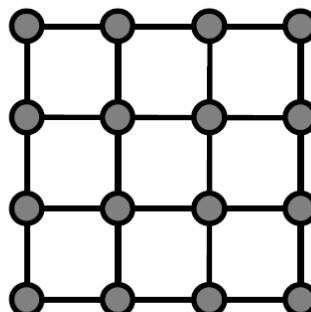
1D mřížka



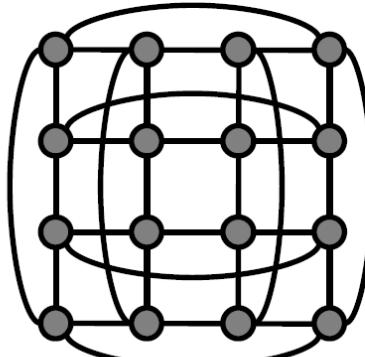
1D kruh



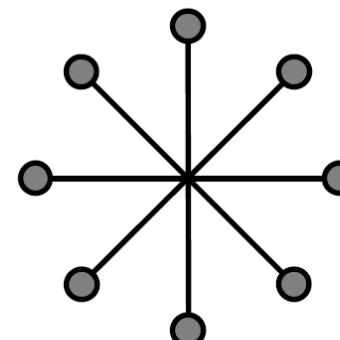
Sběrnice



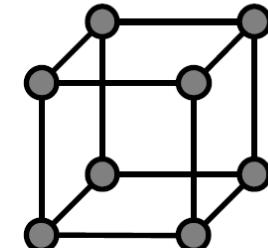
2D mřížka



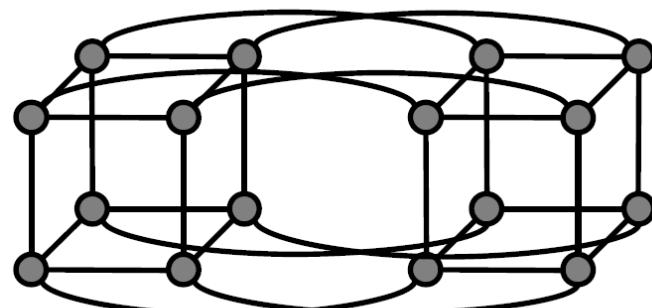
2D torus



Hvězda



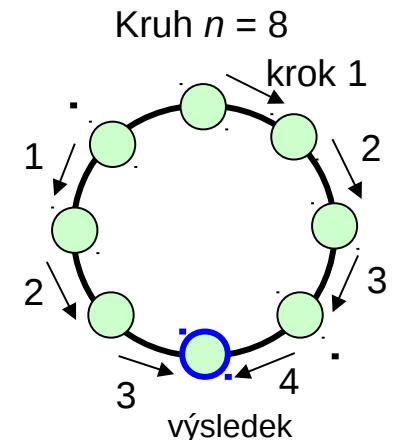
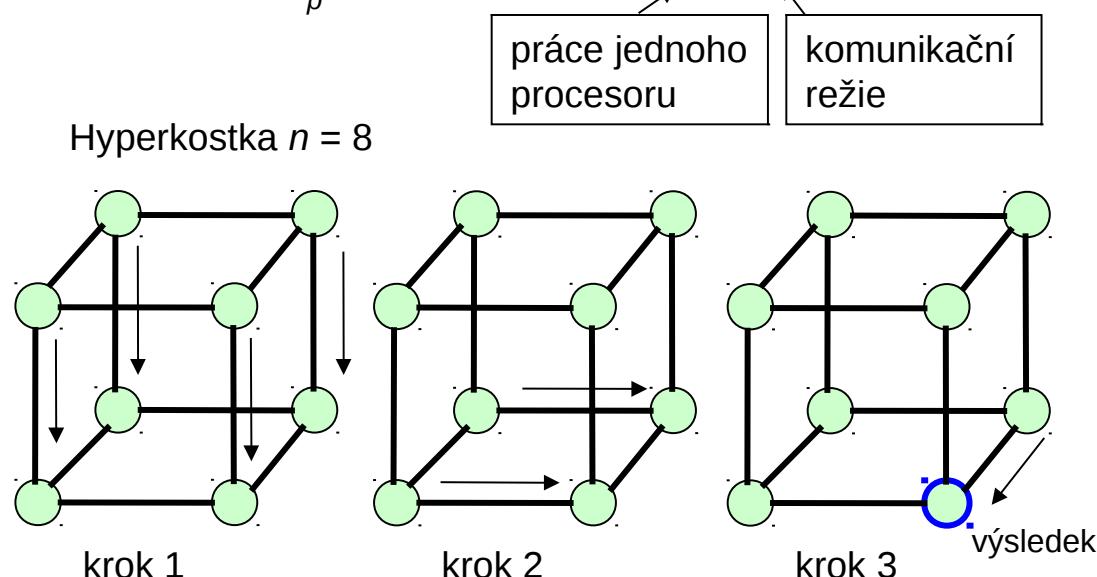
3D kostka



4D hyperkostka

Př. Sečtení k čísel na n procesorech

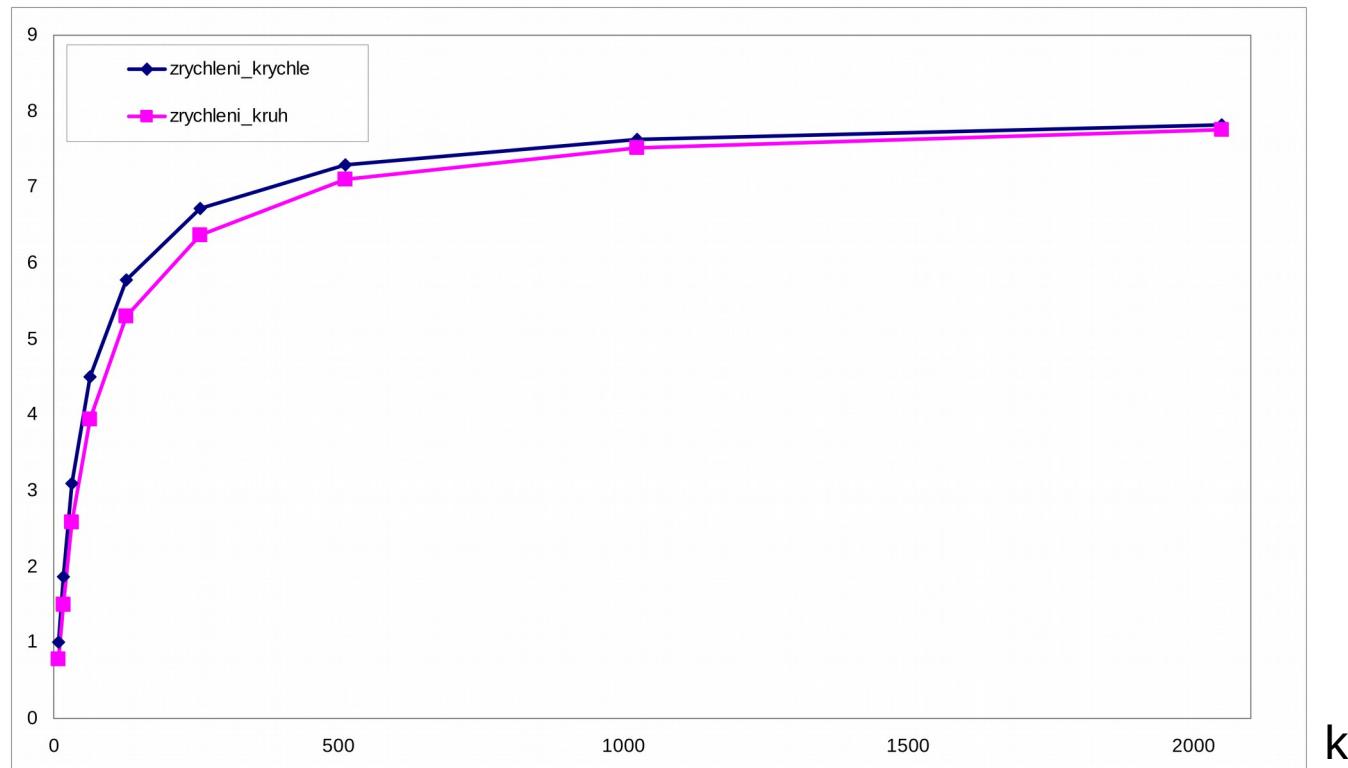
- Sekvenční řešení: $t_s = k - 1$ kroků
- Paralelní řešení (každý procesor má k/n čísel):
 - Hyperkostka: $t_p = k/n + 2 \log_2 n$ kroků
 - Kruh: $t_p = k/n + 2n/2 = k/n + n$ kroků



Pozn.: Operace "pošli mezivýsledek a sečti" je počítána jako 2 kroky.

Př. Sečtení k čísel na n procesorech

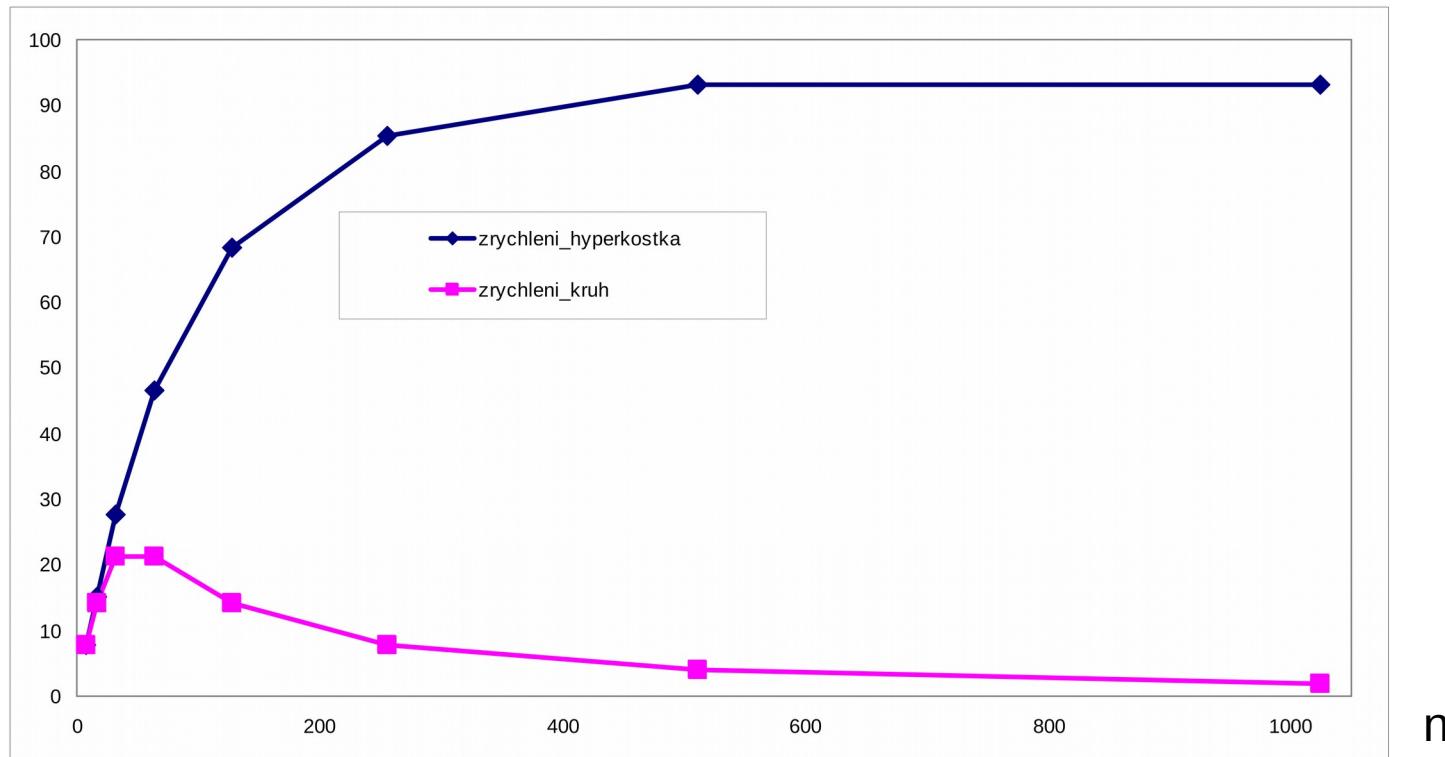
- zrychlení = t_s / t_p
8 procesorů



Je patrný velký vliv komunikační režie na zrychlení pro nízké hodnoty k.

Př. Sečtení k čísel na n procesorech

$k = 2048$



U kruhu je patrný velký vliv komunikační režie na zrychlení pro větší počet procesorů.

Vybrané paralelní výpočetní systémy v ČR

- Výpočetní cluster na FIT VUT v Brně (řízeno systémem [SGE](#))
 - Až 2400 procesů (dle stavu jednotlivých serverů/modulů)
 - V současnosti 102 uzelů, každý vybaven 2x CPU 4-16 jader, 4-256 GB RAM
- [Anselm](#), VŠB-TU Ostrava (spuštěn 2013), součást projektu IT4Innovations (VUT je účastníkem projektu)

Přes 200 uzelů, každý vybaven 2x CPU Intel Sandy Bridge 8-core, 96 GB RAM, některé navíc disponují GPU akcelerátorem Nvidia Tesla Kepler K20



Anselm ~ do 06/2015



Vybrané paralelní výpočetní systémy v ČR

Salomon VŠB-TU Ostrava ~ od 08/2015, 40. místo
v top500 v době spuštění, současně nejvýkonnější
clusterem s koprocesory **Intel Xeon Phi** v Evropě



Parametry:

- 2 000 TFLOPS teoretický výpočetní výkon
- 1 008 výpočetních uzelů s 24 192 výpočetními jádry Intel Xeon (Haswell EP) s celkem 129 TB paměti
- 432 akcelerovaných výpočetních uzelů, každý akcelerovaný dvěma Intel® Xeon Phi™, celkově 52 704 jader a 13,8 TB paměti
- Sdílená disková úložiště s kapacitou 500 TB HOME, 1 730 TB

Nejvýkonnější superpočítače (2015)

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
48	IT4Innovations National Supercomputing Center, VSB-Technical University of Ostrava Czech Republic	Salomon - SGI ICE X, Xeon E5-2680v3 12C 2.5GHz, Infiniband FDR, Intel Xeon Phi 7120P SGI	76,896	1,457.7	2,011.6	

Literatura

- Dvořák, V., Drábek, V.: Architektura procesorů. Studijní opora. FIT VUT v Brně 2006
- Dvořák, V.: Architektura a programování paralelních systémů. Skriptum VUT v Brně 2004
- Hanáček, P.: Paralelní a distribuované algoritmy. Přednášky FIT VUT v Brně, 2008
- Fučík O.: Hardware/Software codesign. Habilitační přednáška, FIT VUT v Brně 2009
- Bhandarkar D.: The Dawn of a New Era: Multi-Core Computing. Intel 2006
- Pospíchal P.: Akcelerace genetického algoritmu s využitím GPU. Diplomová práce FIT VUT v Brně, 2009
- Pozn. Většina obrázků převzata z internetu a uvedených publikací
- Podrobně je problematika paralelních systémů probírána v magisterských kurzích:
 - Architektura procesorů (ACH)
 - Paralelní a distribuované algoritmy (PRL)
 - Architektura a programování paralelních systémů (ARC)

Souhrn INP

- Procesor
 - Základní konstrukce von Neumannova procesoru, princip činnosti
 - Architektury: střadač, zásobník, registry, případně kombinace
 - Pokročilé koncepty: zřetězené zpracování, přehled moderních principů procesorů využívajících paralelní zpracování na vyšších úrovních (vícecestné, vícevláknové a vícejádrové procesory)

Souhrn INP

- Reprezentace dat
 - Základní principy kódování informace v počítači: celočíselné a floating-point reprezentace (norma IEEE 754), princip kódování instrukcí
 - Vybrané kódy zvláštního významu: odstranění redundance (Huffmanův kód), využití redundantních kódů pro zabezpečení, detekci a opravu chyb (Hammingův kód)

Souhrn INP

- HW realizace výpočetní jednotky (CPU ALU)
 - Základní aritmetické operace v celočíselné aritmetice (add, mult, div v různých variantách – zejména CLA add, Boothovo násobení a další optimalizace pomocí Wallaceova schématu, dělení se znaménkem - SRT)
 - Princip výpočtů ve floating point (IEEE 754, základní schéma pro výše uvedené aritmetické operace)
 - Algoritmy a princip HW realizace pokročilých funkcí (iterační algoritmy, CORDIC)

Souhrn INP

- Základní dovednosti konstrukce poč. HW
 - Transformace matematického vztahu či algoritmu na funkční schéma HW
 - Obecné urychlení výpočtů s využitím paralelního zpracování (opakování výpočetní jednotky, řetězení)
 - Adresace paměťových buněk (registrů): princip a použití adresového dekodéru 1 z N, pojem bázová adresa, připojení více modulů k procesoru a jejich jednoznačné adresování
 - + fundamentální techniky probírané v kurzu INC

Souhrn INP

- Základní komponenty PC mimo CPU
 - Paměti: typy, princip funkce a pužití, význam paměťové hierarchie, rychlá vyrovnávací paměť (cache); a nezapomenout, že i registry CPU představují paměť!
 - Řadiče: princip a základy implementace (obvodový řadič založený na FSM, mikroprogramový řadič, Wilkesovo schéma)
 - Sběrnice: základní přehled, principy funkce, řízení a přidělování sběrnice (handshake, prioritní schéma)
 - Periferní zařízení: základní uspořádání a řízení (polling, obsluha přerušení), přístup k registrům PZ (paměťově mapovaný V/V, izolovaný V/V), princip DMA přenosů

Souhrn INP

- Další související téma
 - Výkonnost (definice, měření, jednotky), Amdahlův zákon o urychlování výpočtu
 - Spolehlivost číslicových systémů (základní pojmy, ukazatele spolehlivosti, modelování spolehlinosti), zvyšování spolehlivosti systémů (zálohování, zajištění odolnosti proti poruchám)
 - Přehled technik paralelismu na úrovni procesorů: klasifikace, základní principy, vícevláknové zpracování, vícejádrové procesory (motivace k jejich zavedení), využití víceprocesorových (SMP) a distribuovaných paralelních systémů (vliv topologie propojovací sítě na výkonnost)

Připomenutí principů návrhu hardware pomocí VHDL, simulace a verifikace hardware

INP - cvičení 1

Zdeněk Vašíček, 2015
vasicek@fit.vutbr.cz

VHDL – základní pojmy

VHDL

- VHDL je jazyk umožňující popis hardware běžně používaný v elektrotechnickém průmyslu na celém světě
- Dovoluje popisovat číslicové, ale také analogové a smíšené systémy (viz rozšíření VHDL-AMS)
- VHDL je zkratkou VHSIC HDL (**V**ery **H**igh **S**peed **I**ntegrated **Circuit **H**ardware **D**escription **L**anguage)**
- VHDL prochází standardizací (poslední revize je z roku 2008)

VHDL a klasické programovací jazyky

- Odlišnosti od tradičních jazyků:
 - umožňuje modelovat souběžné děje
(systém popsán pomocí paralelně pracujících komponent).
 - specifické datové objekty a typy
(signály, sběrnice, vícehodnotová logika, fyzikální datové typy)
 - koncept času, schopnost modelovat elektrické vlastnosti
(zpoždění, parametry hradel – setup, hold apod.)
- VHDL je možné použít pro
 - simulaci (digital i mixed-signal, modely, validace, verifikace)
 - syntézu (popis hardware – FPGA i ASIC)
 - specifikaci (obecný prog. jazyk)

Zálužnosti VHDL

- Jazyk VHDL byl navržen s ohledem na efektivitu simulace chování elektronických systémů
 - nepochopení konstrukce může způsobit odlišné chování od simulace (např. sensitivity list)
- VHDL obsahuje konstrukce usnadňující tvorbu simulačních prostředí
 - ne každá konstrukce je syntetizovatelná (např. while)

Návrh hardware v praxi

Role VHDL

Zdrojový kód (VHDL, Verilog, ...)

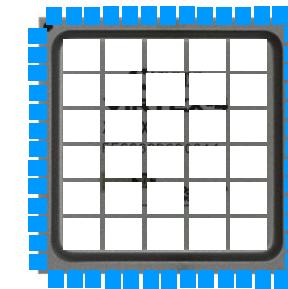
```
library IEEE;
use IEEE.std_logic_1164.all;

entity blk4 is
  port (
    i1: in STD_LOGIC;
    i2: in STD_LOGIC;
    i3: in STD_LOGIC;
    i4: in STD_LOGIC;
    o1: out STD_LOGIC
  );
end blk4;

architecture struc of blk4 is
begin
  o1 <= (i1 or i2) and (i3 and i4);
end struc;
```

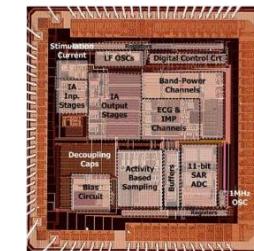
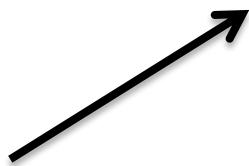
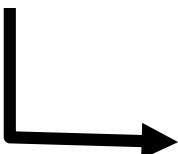
Virtex-7 (největší FPGA):
2.4 miliony buněk
53MB konfigurace

Spartan-3 (FITkit):
1728 buněk
54kB konfigurace



FPGA

syntéza,
place&route, map

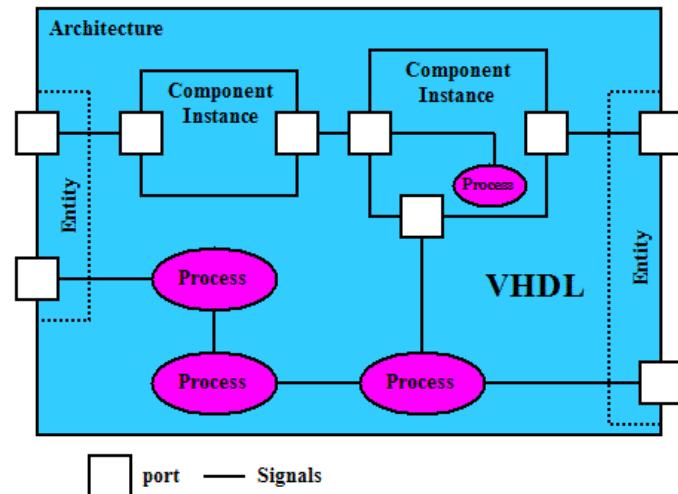


ASIC

Stručné připomenutí principů návrhu HW pomocí VHDL

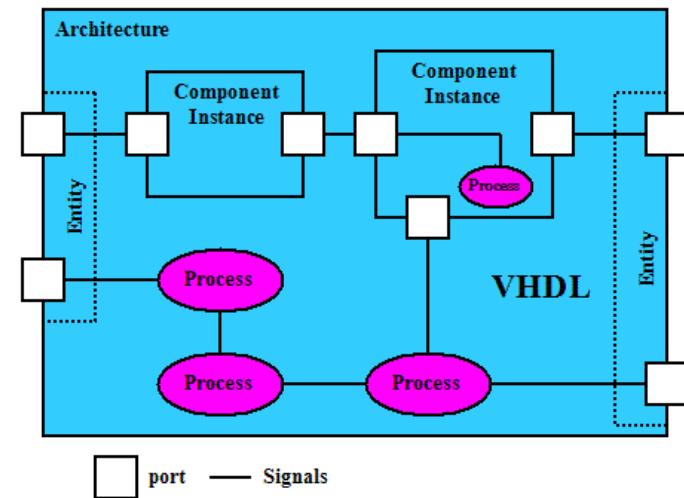
Princip modelování HW systémů pomocí VHDL

- Modelovaný systém je chápan jako množina vzájemně komunikujících komponent.
- Komunikace probíhá pomocí signálů (speciální k tomuto účelu vyhrazený datový objekt).
- Kompletní systém je typicky modelován jako hierarchie komponent (návrh shora dolů).



Způsob modelování komponent

- Ve VHDL je každá komponenta modelována dvojicí
 - entita (jak vypadá rozhraní)
 - architektura (jak se komponenta chová)
- Jednotlivé části popisu jsou oddělitelné
 - jedna entita může mít několik různých architektur (např. jednu pro simulaci, jinou pro syntézu)

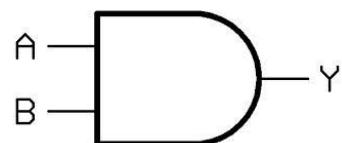


Entita

- Entita popisuje rozhraní mezi komponentou a jejím okolím
- Rozhraní komponenty definuje
 - **seznam signálů rozhraní** (volitelně)
 - **seznam generických parametrů** sloužících k předávání dalších parametrů do entity (volitelně)

```
library IEEE;
use IEEE.std_logic_1164.all;

-- entity
entity ANDGATE is
  port (
    A: in std_logic;
    B: in std_logic;
    Y: out std_logic
  );
end ANDGATE;
```



Signály entity

- Signály rozhraní mohou být v jednom ze čtyř režimů
 - **IN** (vstup, hodnota nemůže být uvnitř komponenty modifikována)
 - **OUT** (výstup, hodnota nemůže být uvnitř komponenty čtena)
 - **BUFFER** (výstup, hodnotu však lze uvnitř komponenty číst)
 - **INOUT** (vstup-výstup, obousměrný port, sběrnice)

Architektura

- Architektura popisuje chování a/nebo strukturu komponenty a je vždy svázána s entitou.
- Architektura obsahuje
 - Deklarační část učenou k deklaraci signálů, konstant, typů nebo funkcí použitých uvnitř architektury.
 - Sekci paralelních příkazů obsahující instance komponent a procesy propojené pomocí signálů. Nezáleží na pořadí příkazů!

```
library IEEE;
use IEEE.std_logic_1164.all;

-- entity
entity ANDGATE is
  port (
    A: in std_logic;
    B: in std_logic;
    Y: out std_logic
  );
end ANDGATE;
```

```
architecture RTL of ANDGATE is
  --deklarace
begin

  Y <= A and B;

end RTL;
```

Možnosti popisu chování

- V rámci sekce paralelních příkazů lze použít
 - data-flow popis
 - specifikace funkce pomocí logických výrazů
 - lze přepsat pomocí konstrukce process
 - strukturní popis
 - instancování komponent a jejich propojení pomocí signálů
 - behaviorální popis
 - popis algoritmů a funkčních bloků pomocí procesů komunikujících prostřednictvím signálů
 - ne vše co lze zapsat behaviorálně je syntetizovatelné!
- Jednotlivé přístupy lze libovolně kombinovat

Konstrukce process

Základní prvek behaviorálního popisu

- Konstrukce process vymezuje úsek kódu, jehož příkazy jsou vyhodnocovány v průběhu **simulace** sekvenčně.
Pořadí příkazů je podstatné!

```
[label:] process [(sensitivity list)]  
    deklarační část  
begin  
    část sekvenčních příkazů  
end process [label];
```

- Process obsahuje
 - **Sensitivity list** (nepovinný) – seznam spouštěčů procesu
 - **Deklarační část** (nepovinná) – je vyhrazena pro deklaraci proměnných, konstant nebo typů použitých uvnitř procesu.
 - **Část sekvenčních příkazů** – popisují chování komponenty nebo její části.

Záludnosti konstrukce process

Sensitivity list

- Sensitivity list
 - seznam signálů, jejichž změna jediné hodnoty má za následek provedení příkazů uvedených uvnitř procesu
 - informuje simulátor, kdy je nutné přepočítat signály, které jsou tímto procesem řízeny
 - slouží pouze k zefektivnění simulace, syntézní nástroje jej ignorují neboť provádějí analýzu kódu
- Časté chyby
 1. signál není omylem uveden v sensitivity listu
 2. sensitivity list obsahuje signály tak, aby simulace pracovala korektně
- Důsledek těchto chyb
 - v praxi se obvod chová jinak než v rámci simulace

Jednoduchá pravidla pro tvorbu sensitivity listu

1. Kombinační logika

- nemá hodinový signál
- výstup musí reagovat na změnu kteréhokoliv z použitých signálů
- sensitivity list obsahuje všechny signály na pravé straně přiřazení a všechny signály použité v podmínce či proceduře

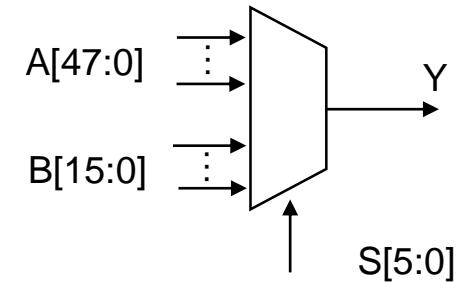
2. Sekvenční logika

- pracuje pouze na hraně hodinového signálu
- výstup se mění pouze je-li detekována hrana hodinového signálu (modeluje se podmínkou testující atribut ‘EVENT’)
- sensitivity list obsahuje zpravidla pouze hodinový signál CLK, případně **asynchronní** nulování RESET

Kombinační prvek pomocí konstrukce process

Multiplexor variantu 64-1

```
entity mux64 is
  port ( A: in std_logic_vector (47 downto 0);
         B: in std_logic_vector (15 downto 0);
         S: in std_logic_vector (5 downto 0);
         Y: out std_logic);
end mux64;
```



Multiplexor pomocí konstrukce for-loop

```
architecture beh of mux64 is begin
  process (A, B, S)
  begin
    for i in 0 to 63 loop
      if (i = conv_integer(S)) then
        if (i < 48) then
          Y <= A(i);
        else
          Y <= B(i-48);
        end if;
      end if;
    end loop;
  end process;
end architecture;
```

Multiplexor pomocí vektorové aritmetiky

```
architecture beh2 of mux64 is
begin
  process (A, B, S)
    variable muxin : std_logic_vector(63 downto 0);
  begin
    muxin := B & A;
    Y <= muxin(conv_integer(S));
  end process;
end architecture;
```

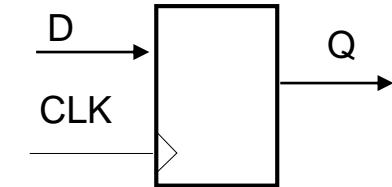
Zavedením proměnné odpadá nutnost používat přetypování, navíc $T'(B \& A)(...)$ není syntakticky validní

Sekvenční prvek pomocí konstrukce process

klopný obvod typu D

Klopný obvod typu D je synchronní obvod, který při každé vzestupné hodinové hraně kopíruje stav vstupu D na výstup Q

```
library IEEE;
use ieee.std_logic_1164.all;
entity reg is
  port (
    CLK : IN std_logic;
    D   : IN std_logic;
    Q   : OUT std_logic
  );
end reg;
```



```
architecture behavioral of reg is
begin
  reg: process (CLK)
  begin
    if CLK'event and (CLK = '1') then
      -- if rising_edge(CLK) then
        Q <= D;
      end if;
    end process;
  end architecture;
```

proces se aktivuje s každou změnou CLK, nikoliv změnou D

přiřazení je vykonáno pouze tehdy, došlo-li k události na CLK a CLK je v log. 1 (tzn. byla detekována vzestupná hrana)

Záludnosti konstrukce process

Příkaz přiřazení a sémantika procesu

- Co je nutné vědět o příkazech přiřazení
 - Přiřazení hodnoty signálu ($<=$) není vykonáno ihned, ale až po ukončení procesu (přirozené nebo voláním příkazu wait). Tato sémantika umožňuje modelovat paralelismus.
 - Přiřazení hodnoty proměnné ($:=$) je vykonáno ihned. Zavedeno proto, aby bylo možné zkracovat zápisu v testech a pravých stranách přiřazení (tzn. používat dočasné proměnné).
 - Ačkoliv jsou interpretovány příkazy sekvenčně, provádějí se jednotlivá přiřazení hodnot signálům paralelně.
 - Jeden konkrétní signál nelze ovládat z více procesů současně (nejedná-li se o sběrnici).
 - Proměnná je inicializována pouze při prvním vykonání process!

Process

proměnné versus signály

```
architecture VAR of EXAMPLE is
    signal CLK, RESULT: integer :=0;
begin

    process(CLK)
        variable a1: integer :=1;
        variable a2: integer :=2;
        variable a3: integer :=3;
    begin
        if CLK'event and CLK='1' then
            a1 := a2;
            a2 := a1 + a3;
            a3 := a2;
            RESULT <= a1 + a2 + a3;
        end if;
    end process;

end architecture;
```

RESULT: 12, 25, ...

```
architecture SIG of EXAMPLE is
    signal CLK, RESULT: integer := 0;
    signal a1: integer :=1;
    signal a2: integer :=2;
    signal a3: integer :=3;
begin

    process(CLK)
    begin
        if CLK'event and CLK='1' then
            a1 <= a2;
            a2 <= a1 + a3;
            a3 <= a2;
            RESULT  <= a1 + a2 + a3;
        end if;
    end process;

end architecture;
```

RESULT: 6, 8, ...

1. Proměnné jsou inicializovány pouze při elaboraci! (obdoba static v C)
2. Hodnota proměnné je přepsána bezprostředně po vykonání přiřazení

Process

proměnné versus signály

```
architecture VAR of EXAMPLE is
    signal CLK, RESULT: integer :=0;
begin

    process(CLK)
        variable a1: integer :=1;
        variable a2: integer :=2;
        variable a3: integer :=3;
    begin
        if CLK'event and CLK='1' then
            RESULT <= a1 + a2 + a3;
            a1 := a2;
            a2 := a1 + a3;
            a3 := a2;
        end if;
    end process;

end architecture;
```

RESULT: ?

```
architecture SIG of EXAMPLE is
    signal CLK, RESULT: integer := 0;
    signal a1: integer :=1;
    signal a2: integer :=2;
    signal a3: integer :=3;
begin

    process(CLK)
    begin
        if CLK'event and CLK='1' then
            RESULT  <= a1 + a2 + a3;
            a1 <= a2;
            a2 <= a1 + a3;
            a3 <= a2;
        end if;
    end process;

end architecture;
```

RESULT: ?

Process

proměnné versus signály

```
architecture VAR of EXAMPLE is
    signal CLK, RESULT: integer :=0;
begin

    process(CLK)
        variable a2: integer :=2;
        variable a3: integer :=3;
    begin
        if CLK'event and CLK='1' then
            RESULT <= 3*a2 + 2*a3;
            a2 := a2 + a3;
            a3 := a2;
        end if;
    end process;

end architecture;
```

RESULT: ?

```
architecture SIG of EXAMPLE is
    signal CLK, RESULT: integer := 0;
    signal a2: integer :=2;
    signal a3: integer :=3;
begin

    process(CLK)
    begin
        if CLK'event and CLK='1' then
            RESULT <= 3*a2 + 2*a3;
            a2 <= a2 + a3;
            a3 <= a2;
        end if;
    end process;

end architecure;
```

RESULT: ?

Process a sekvenční zpracování příkazů

Jakou bude mít hodnotu signál CNT a LED, pokud bude CNT "1110" a nastane hodinový signál.

```
process(CLK)
begin
  if (CLK'event) and (CLK='1') then

    if (cnt = "1110") then
      cnt <= "0000";
    else
      cnt <= cnt + 1;
    end if;

    led <= '0';
    if (cnt = "1110") then
      led <= '1';
    end if;

  end if;
end process;
```

Process a sekvenční zpracování příkazů

použití pravidla jeden process = jedna komponenta

Jakou bude mít hodnotu signál CNT a LED nyní?

```
process(CLK)
begin
  if (CLK'event) and (CLK='1') then
    if (cnt = "1110") then
      cnt <= "0000";
    else
      cnt <= cnt + 1;
    end if;
  end if;
end process;
```

```
process(CLK)
begin
  if (CLK'event) and (CLK='1') then
    led <= '0';
    if (cnt = "1110") then
      led <= '1';
    end if;
  end if;
end process;
```

Jak se vyhnout problémům

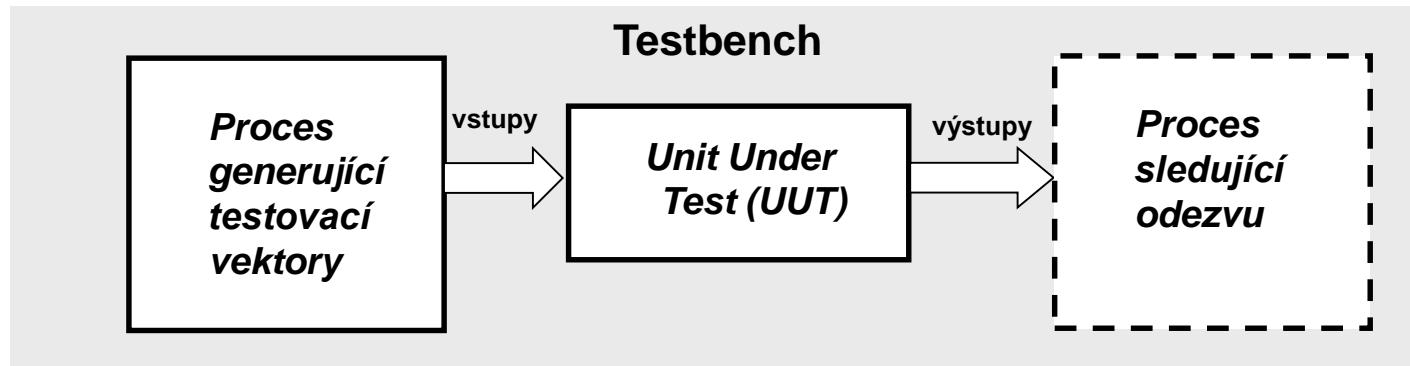
aneb pravidla pro popis HW pomocí VHDL

- Každý systém lze rozložit na kombinační a sekvenční část, tvorba HW tak spočívá ve vhodném spojování základních stavebních bloků (tzn. **syntetizovatelných šablon**)
 - V Xilinx ISE viz menu *Edit* položka *Language Templates* a dále **VHDL > Synthesis Constructs > Coding Examples**
 - Přehled popisu základních komponent lze nalézt zde:
http://merlin.fit.vutbr.cz/FITkit/docs/navody/synth_templates.html
<https://www.fit.vutbr.cz/study/courses/IVH/private/>
- Návrh je dobré začít **blokovým schematem**, následný přepis do VHDL je rutinní záležitost
- V rámci **jednoho procesu** je vhodné popisovat pouze **jedinou komponentu**, vyhneme se problémům s chybami v konstrukcích.
- **Nepoužívat proměnnou k uchování stavu** mezi hodinovými takty.

Verifikace a simulace pomocí nástroje Xilinx ISE Simulator

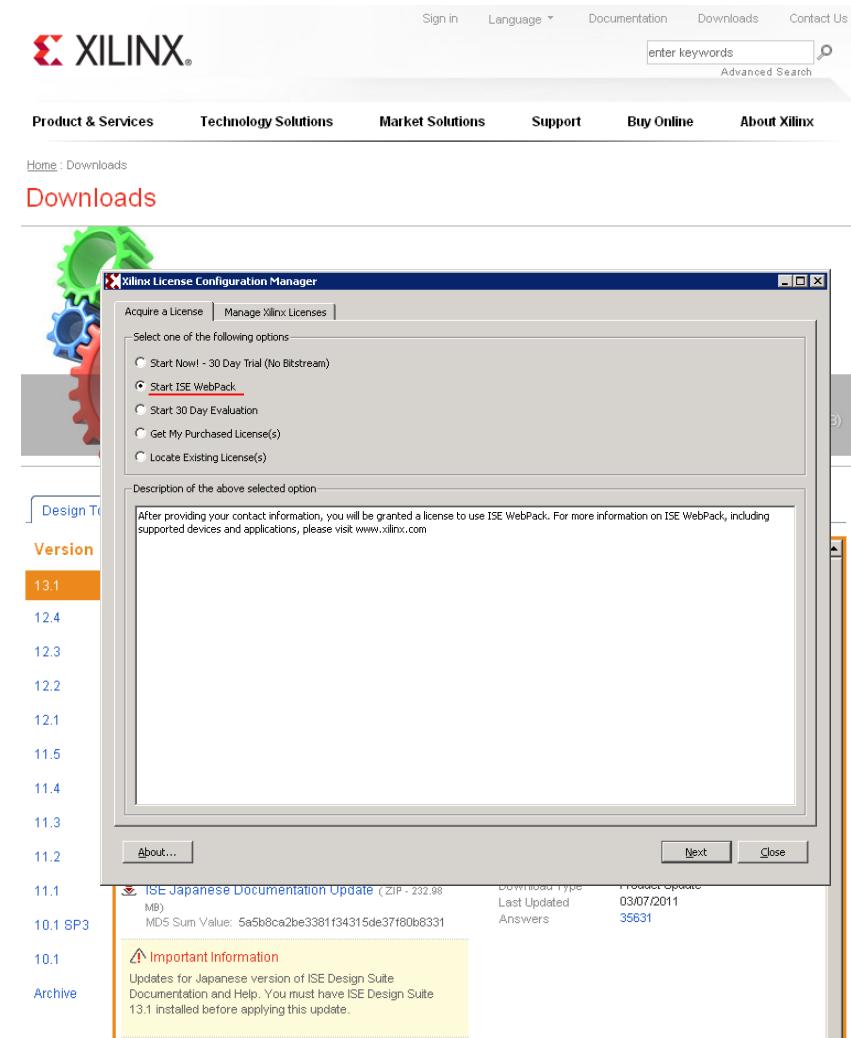
Ověření funkčnosti - testbench

- Testbench je obvod, který aplikuje testovací vektory na vstupy testovaného obvodu a verifikuje získanou odezvu (není nezbytně nutné).
- Výsledkem může být časový diagram (waveform) nebo textový/binární soubor.
- Ve VHDL lze napsat jednoduchý i komplexní testbech (snadná přenositelnost).



Nástroje pro simulaci VHDL

- Pro simulaci lze použít
 - ModelSIM
 - (vyžaduje licenci)
 - Xilinx ISE Simulator (ISIM)
(součást volně dostupného balíku Xilinx ISE Webpack)
- SW je k dispozici ve formě image pro VirtualBox dostupného na stránkách FITkitu
(<https://merlin.fit.vutbr.cz/FITkit/private/web/download.php>)



<http://www.xilinx.com/support/download/index.htm>

Xilinx ISE Simulator

Součásti simulátoru Xilinx ISE Simulator (ISIM)

- *vhcomp.exe*, *vlogcomp.exe* – parser VHDL, Verilog
- *fuse.exe* – HDL elaborace, linker
přeloží a slinkuje dohromady jednotlivé jednotky VHDL
výstupem je simulátor ve spustitelné podobě (např. *testbench.exe*)
- *testbench.exe* – spustitelný simulátor událostně řízené simulace
- *Isimgui.exe* – uživatelské rozhraní ISIM interagující se simulátorem

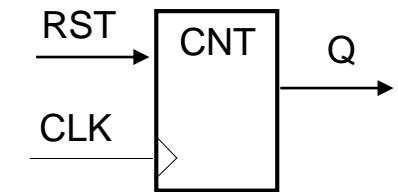
Práce se simulátorem:

- Celým procesem simulace lze projít z GUI (Xilinx ISE) i příkazové řádky
- Simulaci lze automatizovat pomocí TCL skriptů

Příklad: 8-bitový binární čítač (cntr.vhd)

Synchronní čítač - obvod, který při každé vzestupné (nebo sestupné) hodinové hraně inkrementuje hodnotu na výstupu Q

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity cntr is
    port (
        CLK : IN std_logic;
        RST : IN std_logic;
        Q   : BUFFER std_logic_vector(7 downto 0)
    );
end cntr;
```



```
architecture behavioral of cntr is
begin
    citac: process (RST, CLK)
    begin
        if RST='1' then
            Q <= (others => '0');
        elsif CLK'event and (CLK = '1') then
            -- elsif (CLK = '1') then -- chybny zapis
            Q <= Q + 1;
        end if;
    end process;
end architecture;
```

Příklad: 8-bitový binární čítač testbench pro čítač (tb.vhd)

```
library ieee; use ieee.std_logic_1164.all;
entity citac_tb is
end citac_tb;

architecture behavior of citac_tb is
    --vstupy
    signal clk : std_logic := '0';
    signal rst : std_logic := '1';
    --vystupy
    signal q : std_logic_vector(7 downto 0);
begin
    -- instance uut (unit under test)
    uut: entity work.cntr port map (
        CLK => clk,RST => rst, Q => q
    );
    clk <= not clk after 5 ns;

    process
    begin
        wait for 100 ns; wait until clk='0';
        rst <= '0';
        wait until clk='1'; wait for 1 ns;
        assert q = "00000001" report "Chyba 1";
        wait until clk='1'; wait for 1 ns;
        assert q = "00000010" report "Chyba 2";
        ...
        wait;
    end process;
end;
```

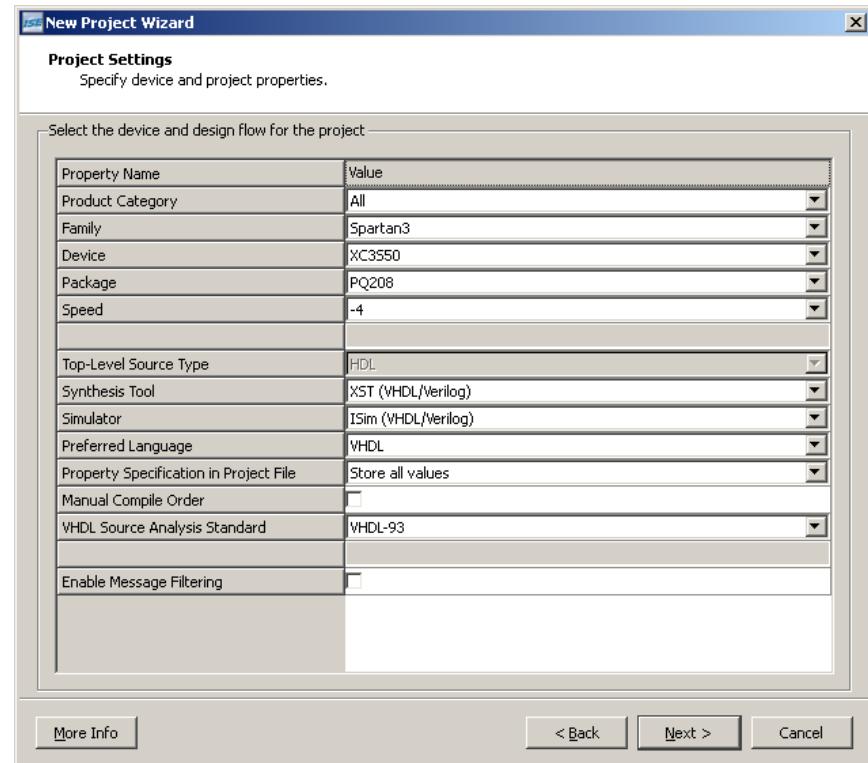
Typické části

- deklarace signálů potřebných pro simulaci
- instance simulované komponenty (pomocí entity nebo komponenty)
- generátor hodinového signálu
- proces řídicí signál rst a sledující odezvu

Xilinx ISIM tutorial

založení nového projektu

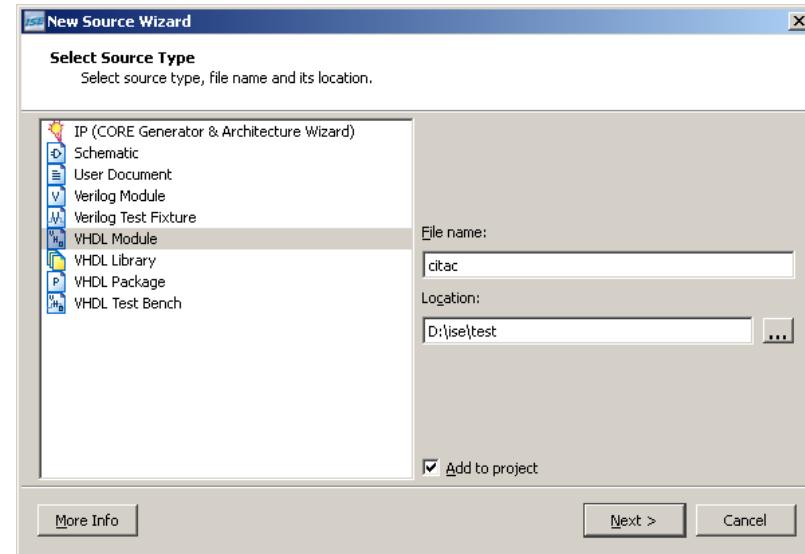
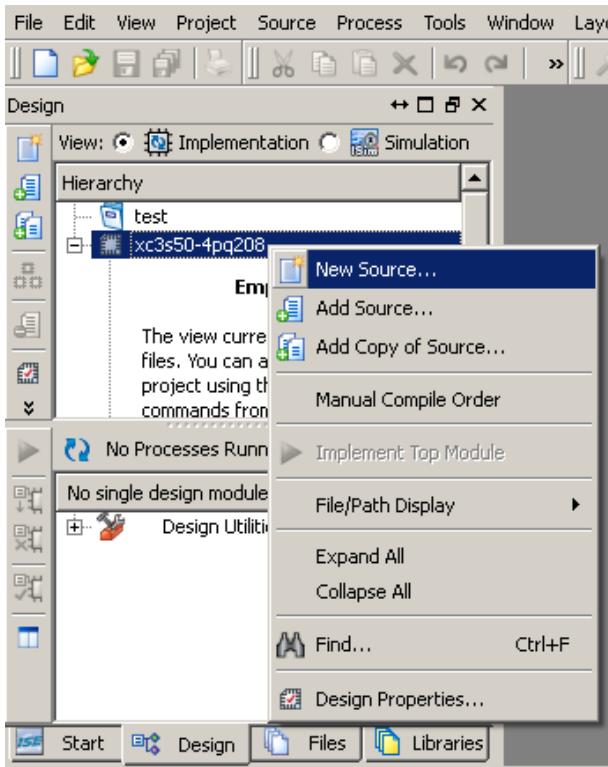
- Spustit Xilinx ISE Design Suite
- Založení nového projektu
 - *File > New Project*
 - zvolit konkrétní typ FPGA, např. XC3S50-PQ208 (Spartan-3 na FITkitu)



Xilinx ISIM tutorial

přidání zdrojových kódů

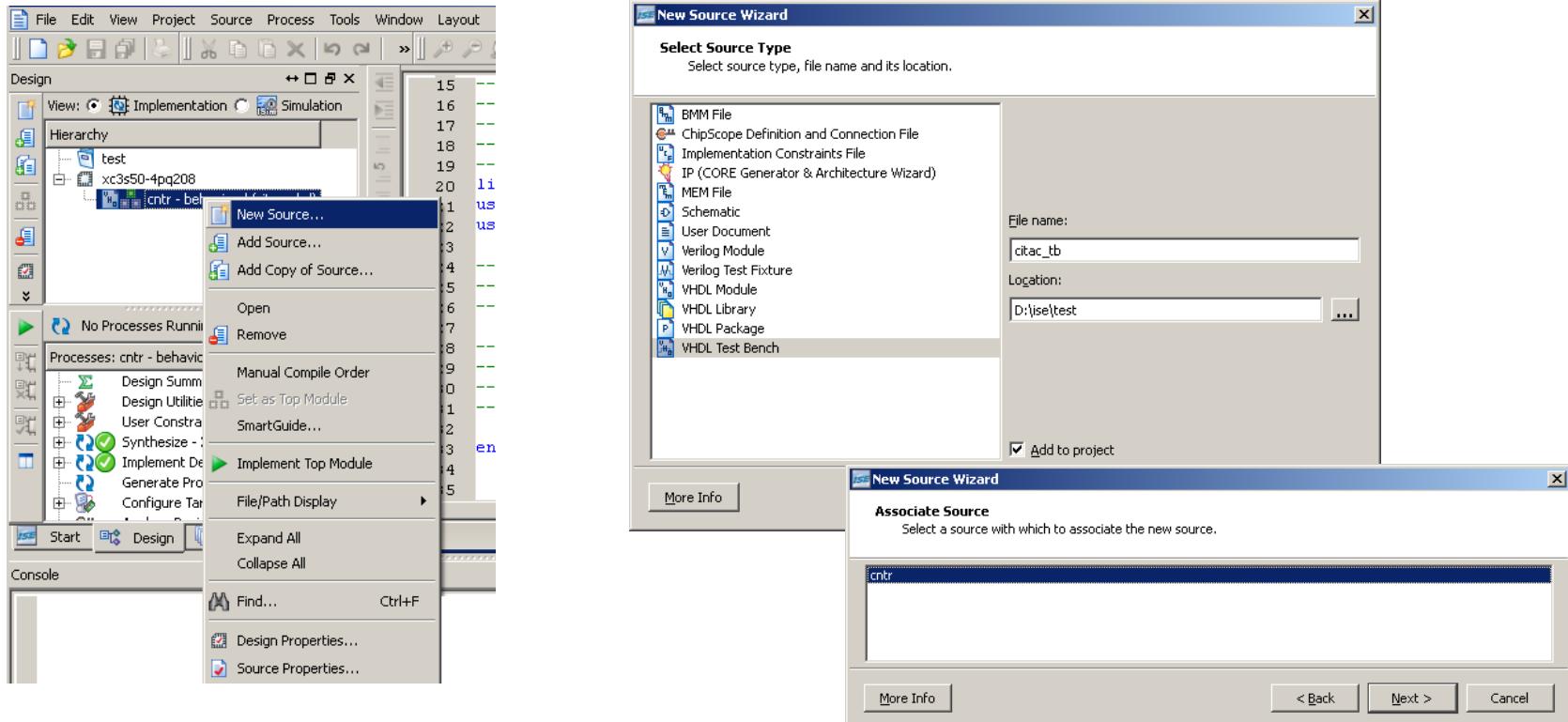
- Vytvoření zdrojového souboru s popisem čítače do projektu
 - kontextové menu *Hierarchy > New Source*
 - zvolit položku *VHDL Module*



Xilinx ISIM tutorial

vygenerování kostry testbench souboru

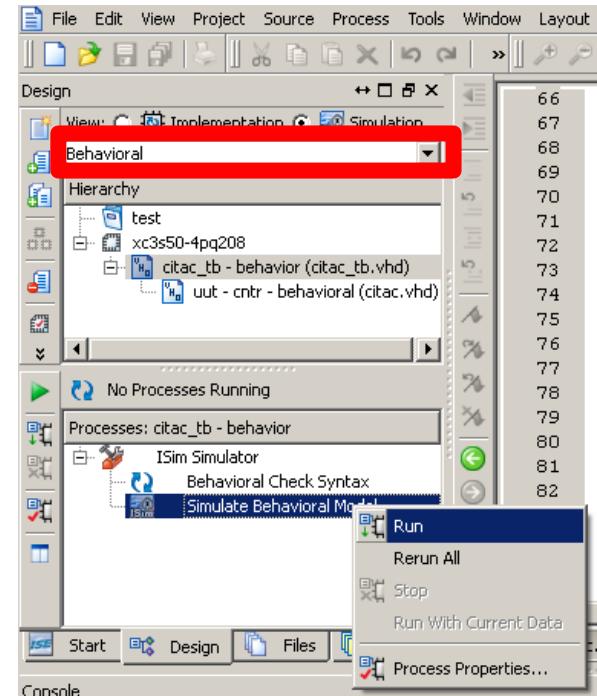
- Vytvoření zdrojového souboru obsahujícího test bench
 - kontextové menu *Hierarchy* > *New Source*
 - zvolit položku *VHDL Test Bench*
 - zvolit entitu, ke které se test bench vztahuje
 - nástroj **automaticky vygeneruje kostru kódu** pro zvolenou entitu



Xilinx ISIM tutorial

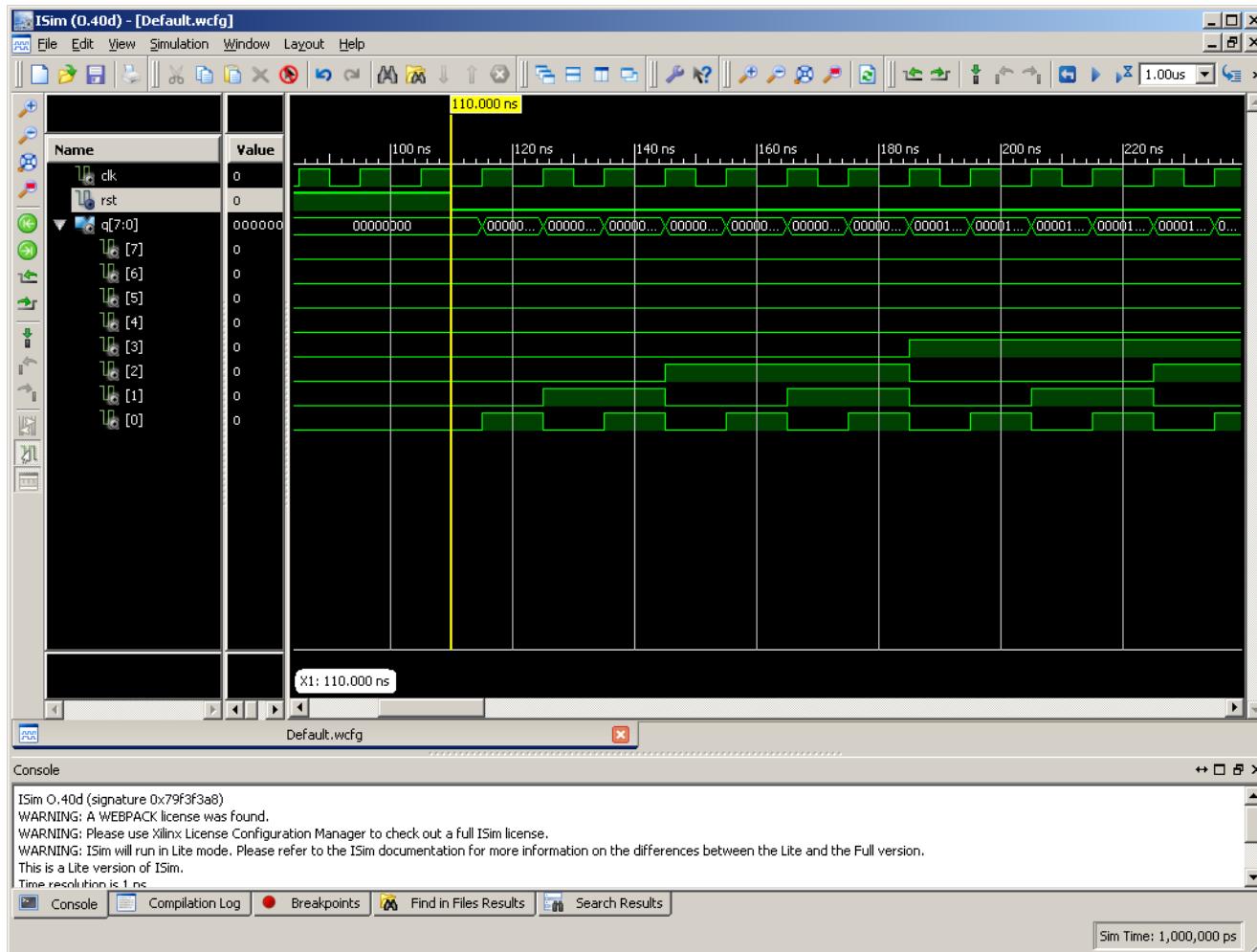
spuštění funkční simulace

- Simulace (funkční/behaviorální)
 - přepnoutí pohledu na *Simulation*
 - označit položku s názvem test bench entity (*citac_tb*)
 - ve vlastnostech procesu *Simulate Behavioral Model (Process Properties)* lze nastavit délku trvání simulace
 - spustit simulaci *Processes > Simulate Behavioral Model*



Xilinx ISIM tutorial

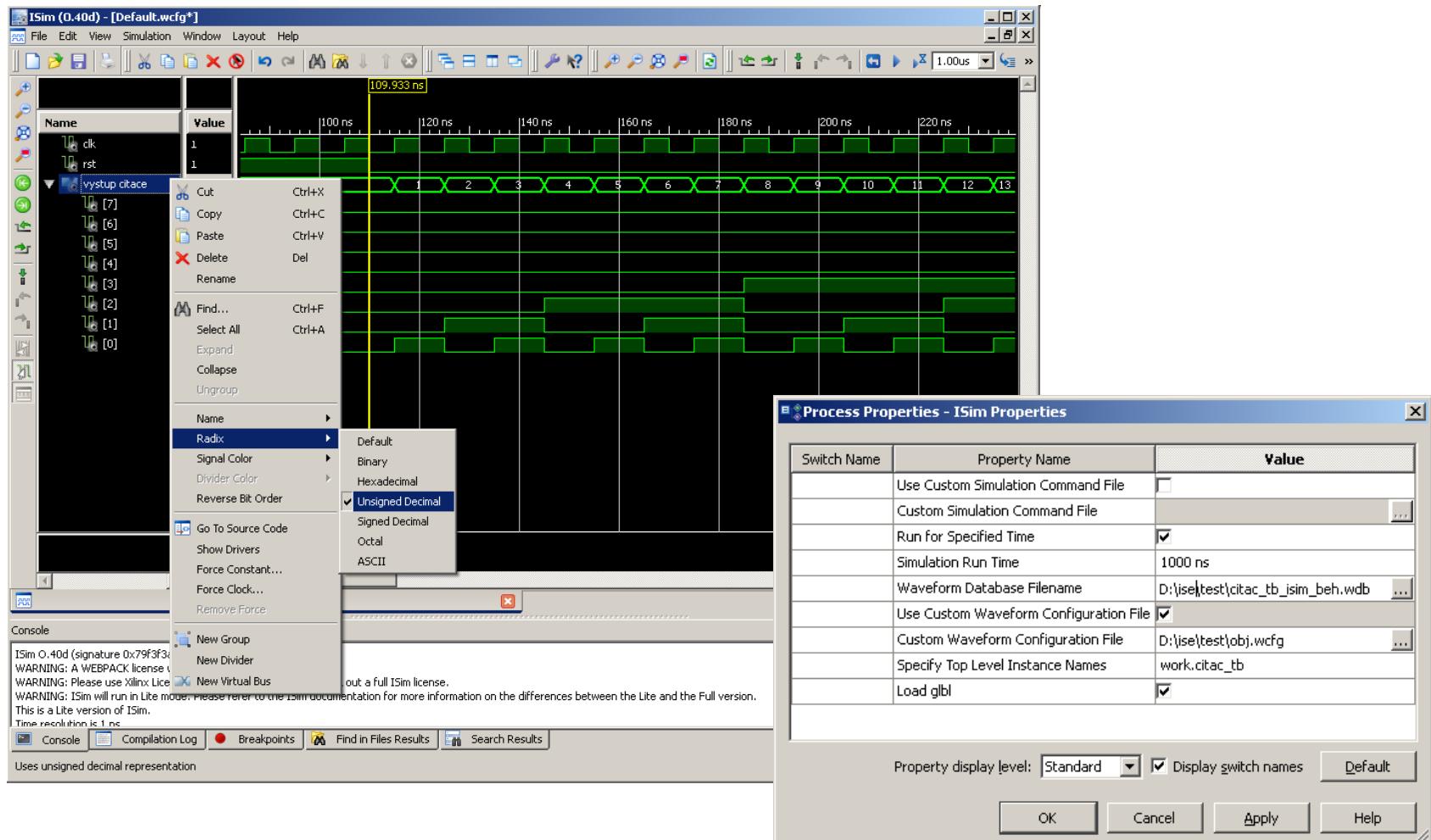
výsledek funkční simulace



V případě potřeby lze prodloužit simulaci příkazem run následovaným dobou (např. **run 1 ms**)

Xilinx ISIM tutorial

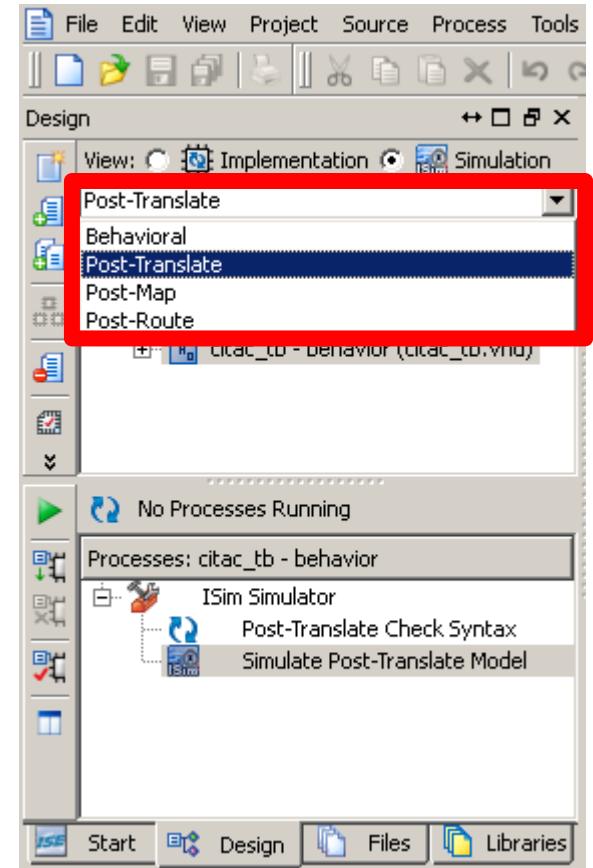
- Seznam signálů lze přizpůsobit dle potřeb, uložit (*File > Save As*) a nastavit jako výchozí pro simulaci



Xilinx ISIM tutorial

spuštění časové simulace

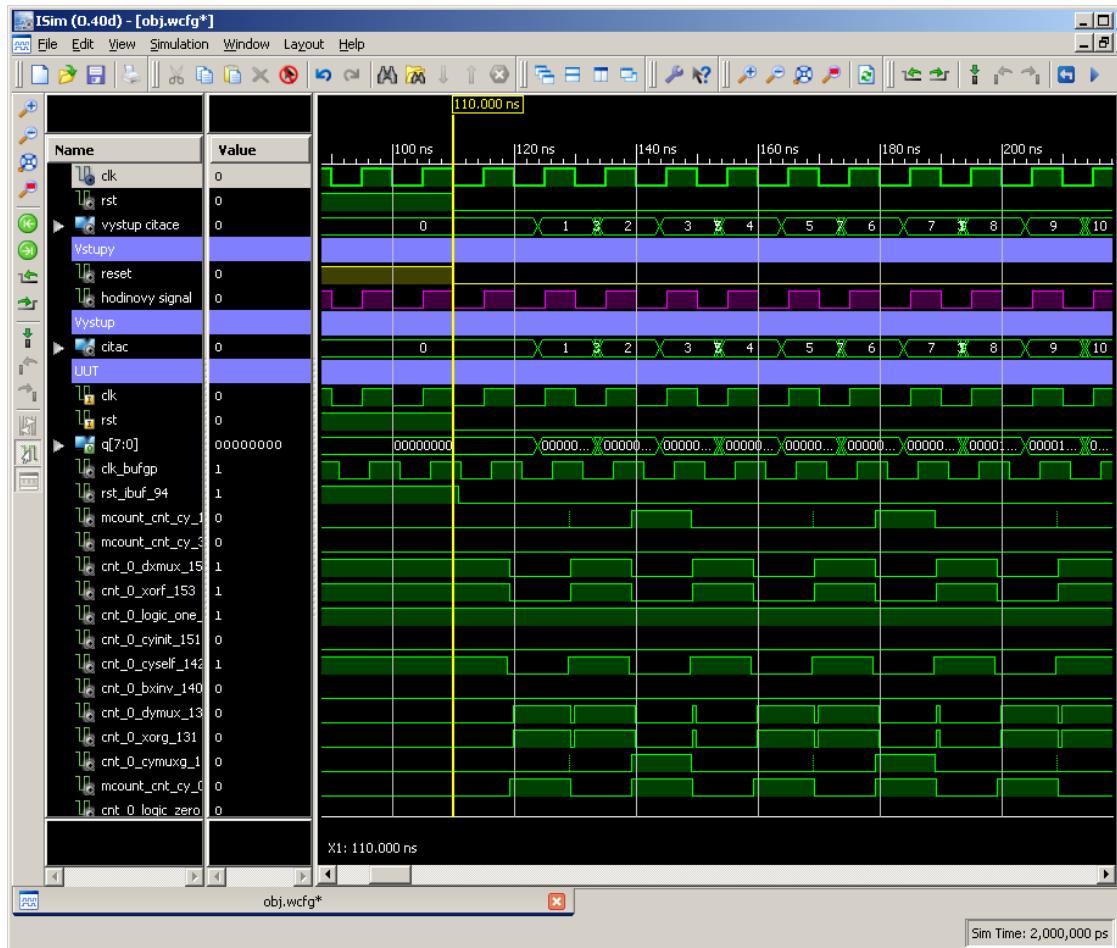
- Simulace (po syntéze, mapování)
 - přepnoutí pohledu na *Simulation*
 - zvolit typ simulace
 - *Behavioral* (funkční simulace)
 - *Post-Translate*
 - *Post-Map*
 - *Post-Route* (časová simulace)
 - označit položku s názvem test bench entity (*citac_tb*)
 - spustit simulaci, např. *Processes > Simulate Post-Translate Model*



Xilinx ISIM tutorial

výsledek časové simulace

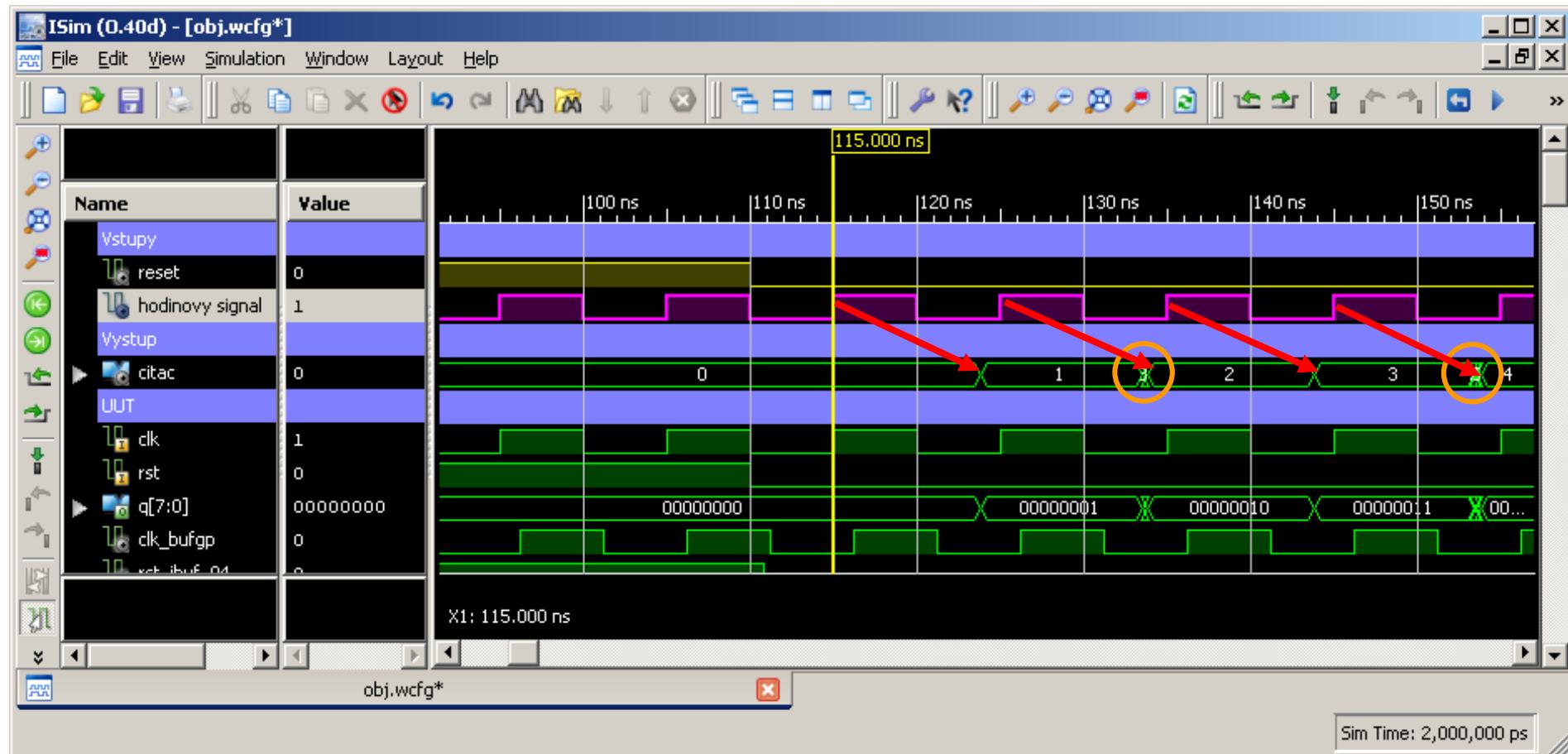
- Výsledek *Post-Route* simulace zohledňuje zpoždění komponent a vodičů (proto se nazývá časová simulace)



Xilinx ISIM tutorial

výsledek časové simulace (detail)

- Ke změně výstupu v praxi nikdy nedochází ihned po náběžné hraně, syntéza však musí garantovat, že ke změně dojde vždy před novou náběžnou hranou (viz maximální pracovní frekvence)



Xilinx ISIM tutorial

chybná konstrukce registru ve funkční a časové simulaci

Rozdíl funkční a časové simulace

(chybně popsaný čítač zneužívající chování sensitivity listu)

```
citac: process (RST, CLK)
begin
    if RST='1' then
        Q <= (others => '0');
    elsif (CLK = '1') then
        Q <= Q + 1;
    end if;
end process;
```

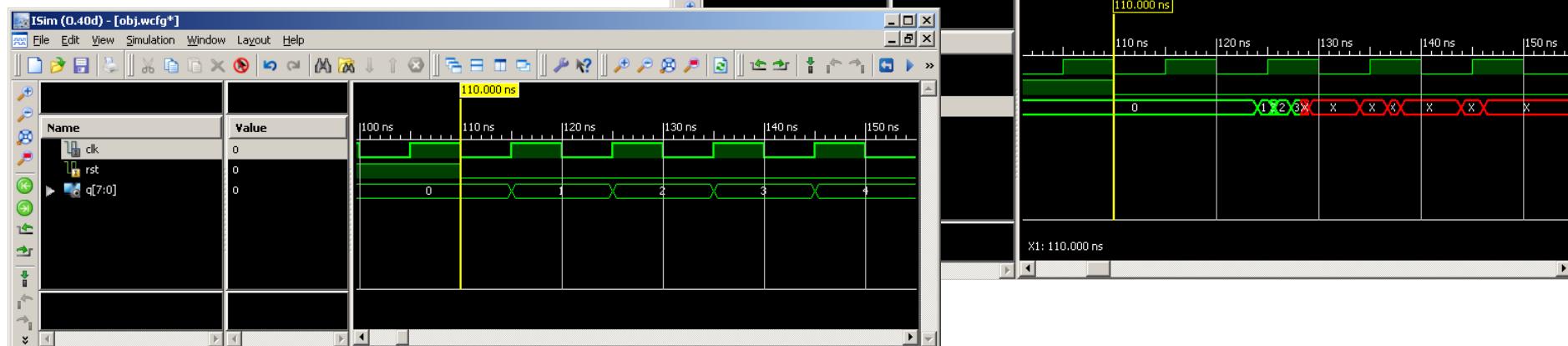
chybně (latch)

správně (registr)

```
elsif CLK'event and (CLK = '1') then
```

Behavioral Simulation

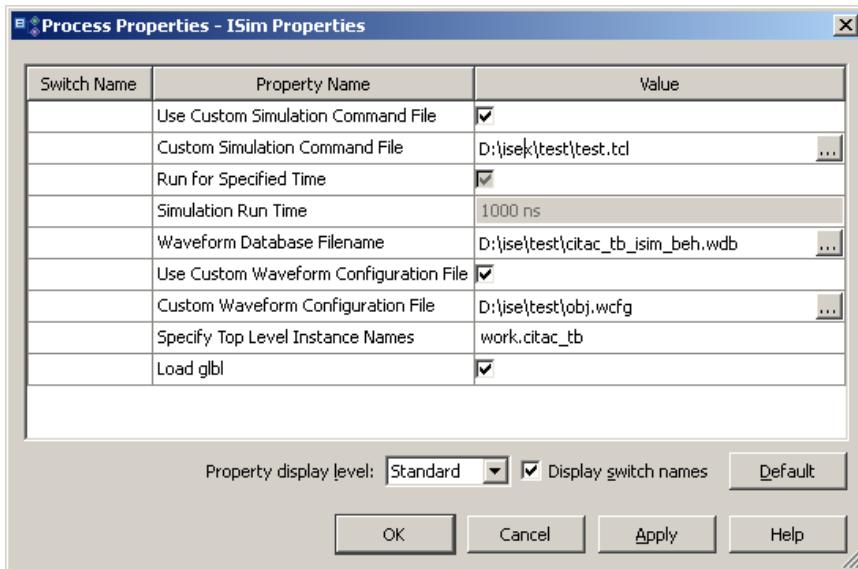
Post-Route Simulation



Xilinx ISIM tutorial

automatizace

- Uživatelské skripty (automatizace simulace)
 - ve vlastnostech procesu *Simulate Behavioral Model (Process Properties)* zvolit *Use Custom Simulation Command File* a vyplnit cestu k souboru obsahující simulační skript (kód v TCL)



Xilinx ISIM tutorial

automatizace

- Příklad skriptu

```
#zakladni signaly
divider add "Vstupy"
wave add /rst -name "reset" -color yellow
wave add /clk -name "hodinovy signal" -color magenta

divider add "Vystup"
wave add /q -radix unsigned -name "citac"

#vsechny signaly z UUT
divider add "UUT"
wave add /citac_tb/uut/

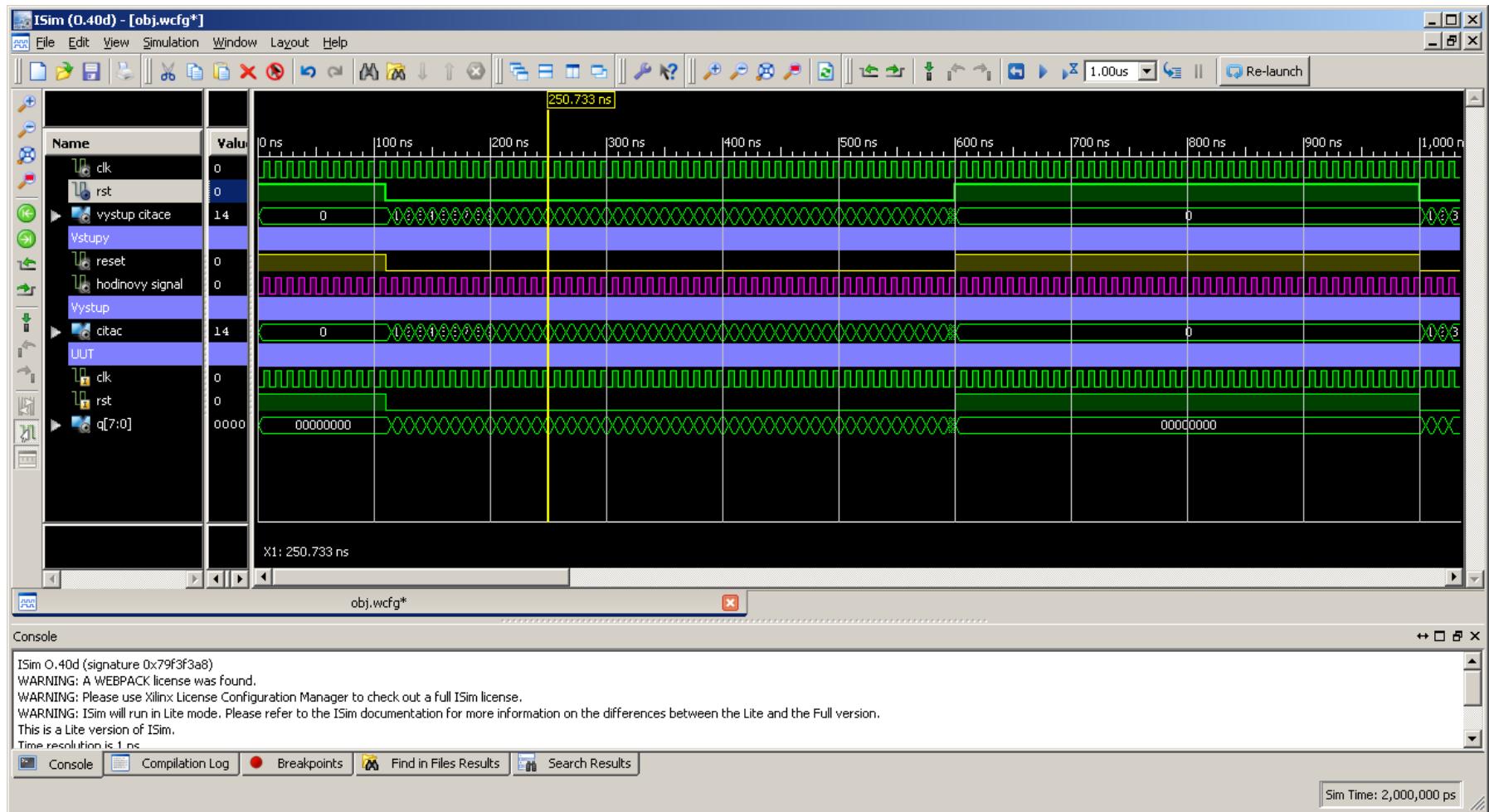
isim force add {/citac_tb/rst} 1 -radix bin -time 600 ns -cancel 1000 ns
run 2us
```

- Přehled podporovaných příkazů viz
http://www.xilinx.com/tools/feature/isim_qrg.pdf

Xilinx ISIM tutorial

automatizace

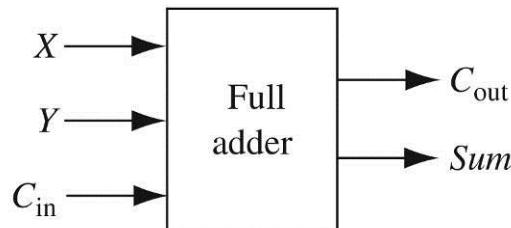
- Výstup z předchozího skriptu



Příklady popisu obvodů ve VHDL

Zdeněk Vašíček, 2015
vasicek@fit.vutbr.cz

Úplná sčítačka (dataflow popis)



```
entity FullAdder is
  port(X, Y, Cin: in bit;          --Inputs
       Cout, Sum: out bit);        --Outputs
end FullAdder;
```

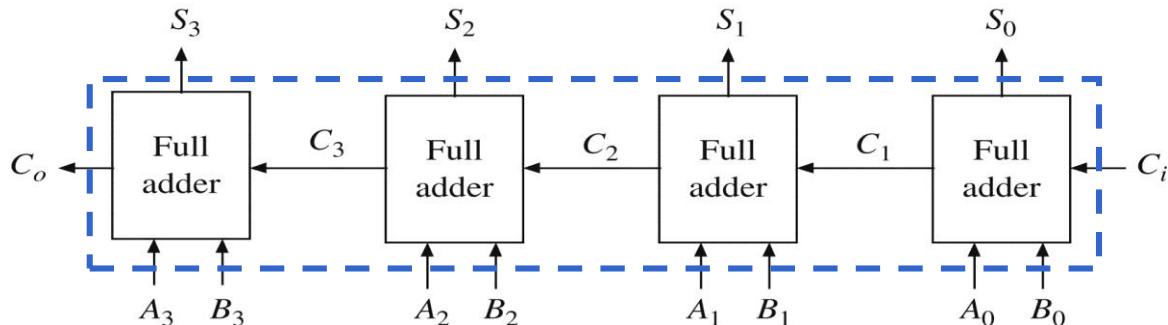
```
architecture dataflow of FullAdder is
begin

  Sum  <= X xor Y xor Cin after 1 ns;
  Cout <= '1' when (X and Y) else
            (X and Cin) or (Y and Cin)
            after 1 ns;

end architecture;
```

- využíváme pouze konstrukce přiřazení signálů
- lze uvést podmínu (when) – vhodné pro popis multiplexoru, dekodéru, ...
případně přiřadit zpoždění (after) – pro potřeby simulace

4-bitová sčítačka (strukturní popis)

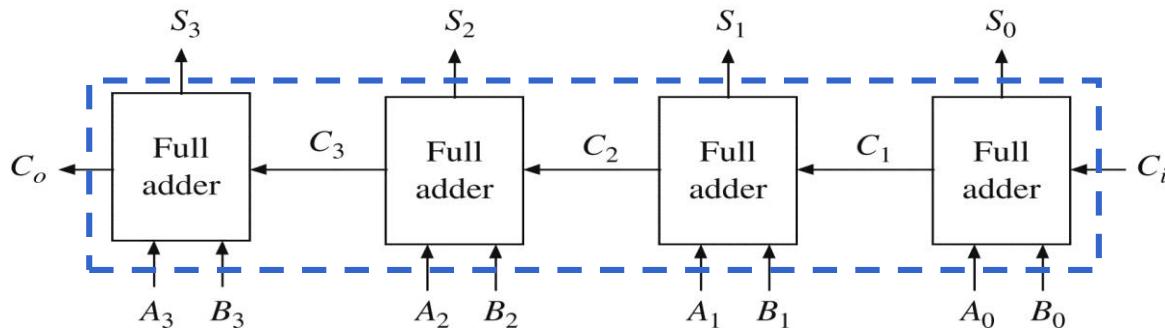


```
entity Adder4 is
  port(
    A, B: in std_logic_vector(3 downto 0);
    Ci: in std_logic;
    S: out std_logic_vector(3 downto 0);
    Co: out std_logic);
end Adder4;
```

```
architecture structure of Adder4 is
  component FullAdder
    port(X, Y, Cin: in std_logic; Cout, Sum: out std_logic);
  end component;
  signal C: std_logic_vector(3 downto 1);
begin
  FA0: FullAdder port map(A(0),B(0),Ci,C(1),S(0));
  FA1: FullAdder port map(A(1),B(1),C(1),C(2),S(1));
  FA2: FullAdder port map(A(2),B(2),C(2),C(3),S(2));
  FA3: FullAdder port map(X <= A(3),Y <= B(3),
                           Cin <= C(3),
                           Sum <= S(3), Cout <= Co);
end Structure;
```

- využití konstrukce **component** a **port map** (poziční ne/závislost)
- analogie konstrukce číslicových systémů ze součástek – propojovaní komponent (součástek) signály (dráty)
- komponenta jako blackbox – pracovat může více vývojářů současně

4-bitová sčítačka (behaviorální popis)



```
entity Adder4 is
  port(
    A, B: in std_logic_vector(3 downto 0);
    Ci: in std_logic;
    S: out std_logic_vector(3 downto 0);
    Co: out std_logic);
end Adder4;
```

```
architecture behavioral of Adder4 is
begin
  process (A,B,Ci)
  variable t: std_logic_vector
    (4 downto 0);
  begin
    t := ('0'&A) + ('0'&B) + Ci;
    S <= t(3 downto 0);
    Co <= t(4);
  end process;
end behavioral;
```

```
use IEEE.std_logic_unsigned.all;
```

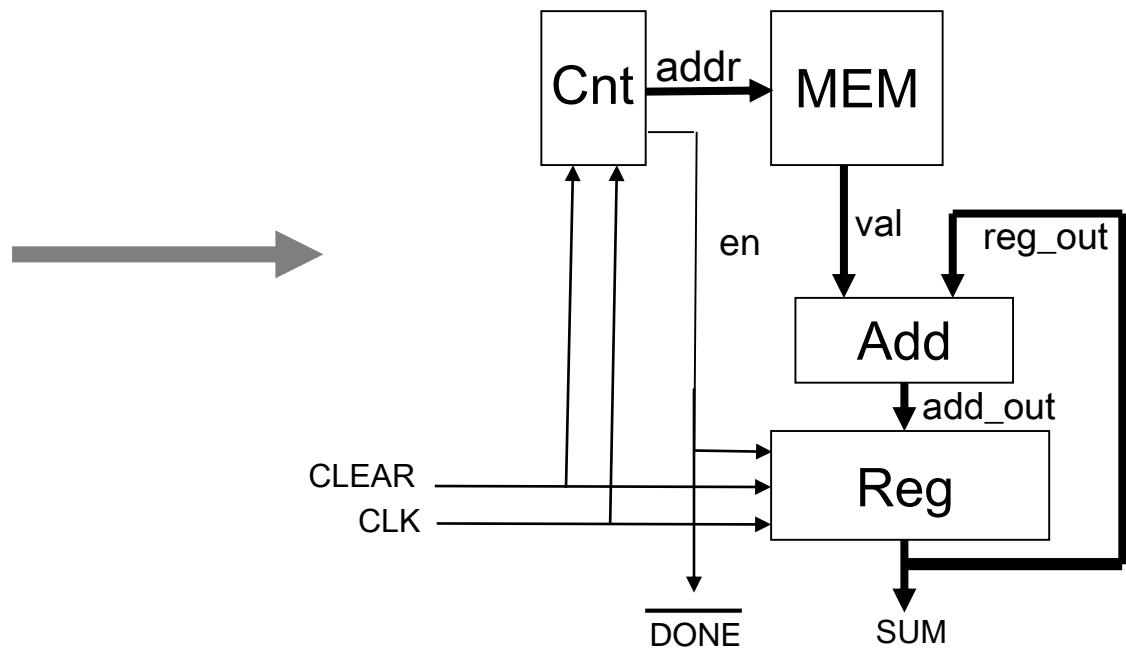
- využití procesů
- pozor na sensitivity list
(v případě kombinační funkce musí být uveden seznam všech signálů ze kterých je odvozen výsledek)

Návrh obvodu realizujícího iterační součet posloupnosti

- Cílem je navrhnut primitivní obvod, který je schopen vypočítat součet hodnot N (např. 8) prvků umístěných v paměti.
- Praktické použití (po rozšíření): kontrola a generování CRC součtu

```
const N = 8
char mem[N] = {1,...};
char i
char sum;

sum = 0
for (i=0;i<N;i++)
    sum += mem[i]
```



- Jak převedeme pseudo kód na HW implementaci?
Stačí umět převést sekvenci, selekci, iteraci.

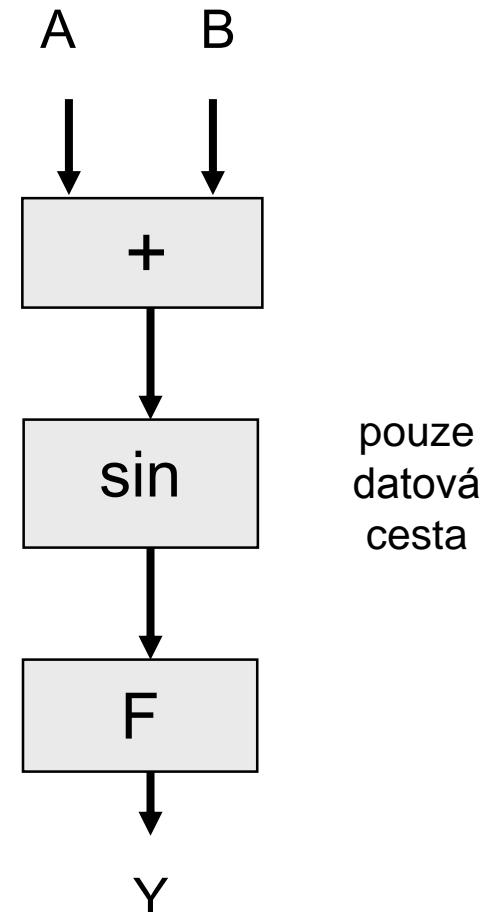
Implementace základních konstrukcí v SW a HW

sekvence (posloupnost operací)

Sekvence příkazů

```
A = B + C  
D = sin(A)  
Y = F(D)
```

(3 kroky)



Implementace základních konstrukcí v SW a HW

selekcce (větvení výpočtu)

Podmíněné vykonávání

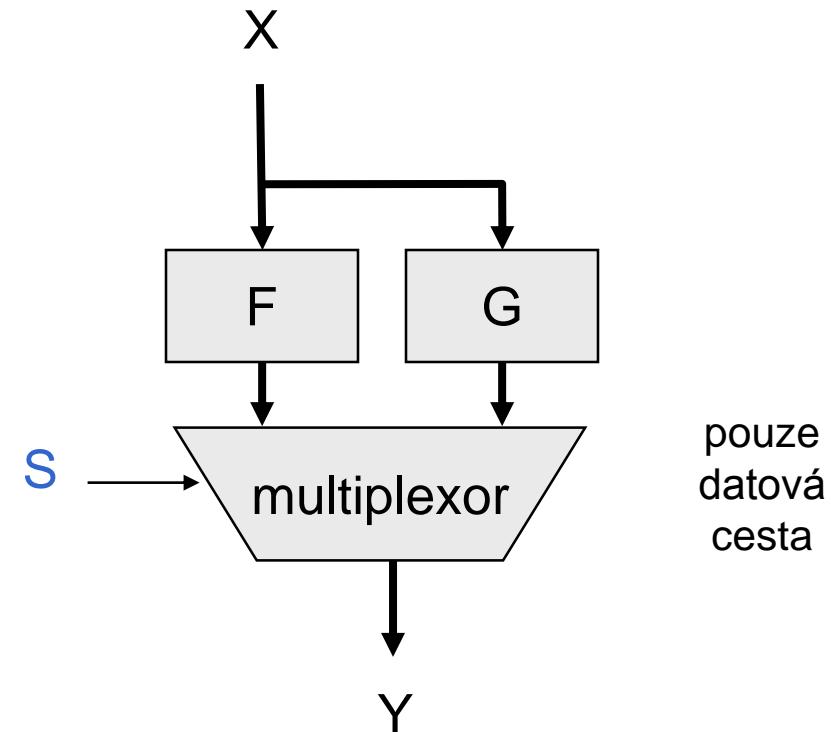
```
If (S == 0):  
    Y = F(X)  
else:  
    Y = G(X)
```

(3/4 instrukce CPU)

vs.

```
Y = G(X)  
If (S == 0):  
    Y = F(X)
```

(3/4 instrukce CPU)



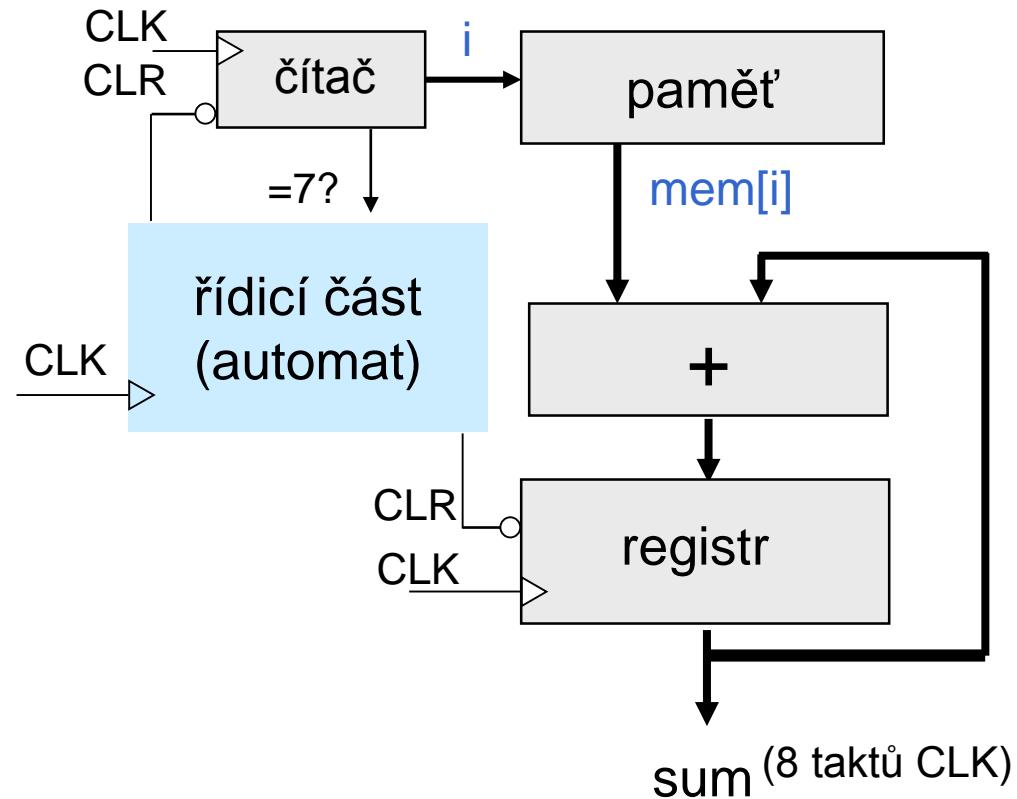
Implementace základních konstrukcí v SW a HW

iterace (opakování výpočtu)

Iterativní zpracování (smyčky)

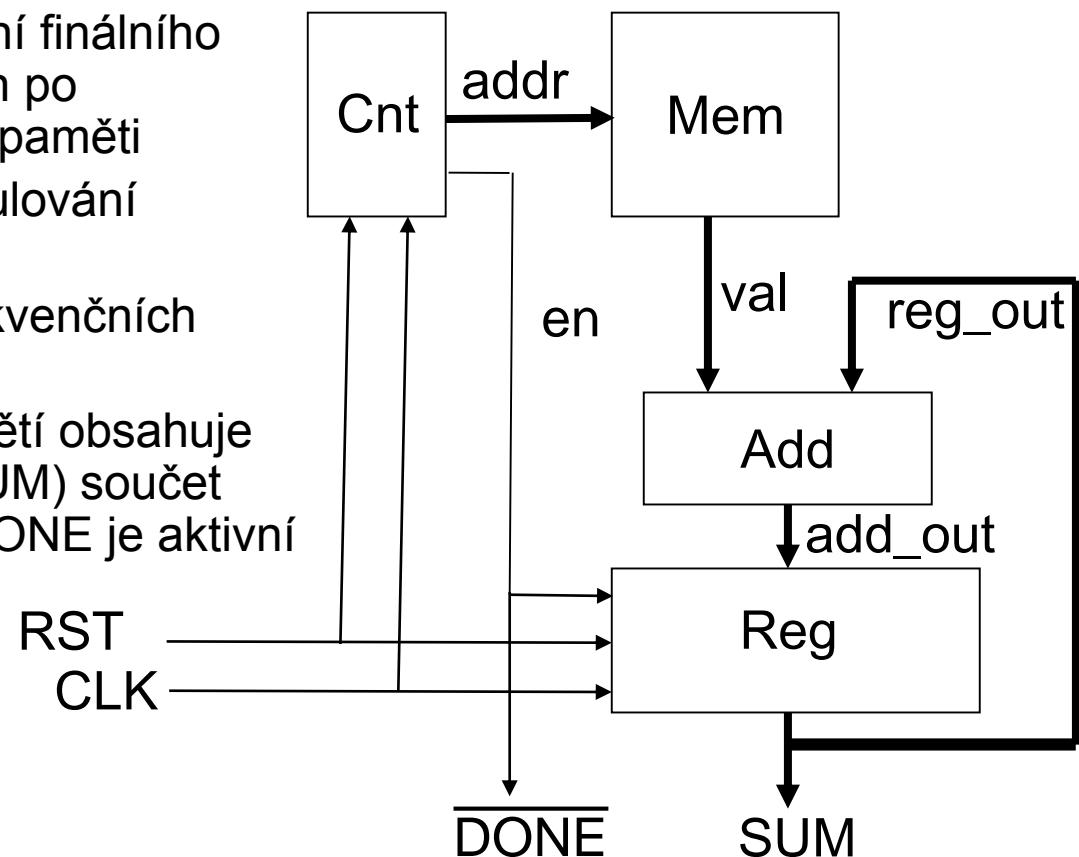
```
sum = 0  
i = 0  
while (i < 8):  
    sum = sum + mem[i]  
    i += 1
```

(cca 40 instrukcí CPU)



Obvod realizující iterační součet posloupnosti

- Čítač (Cnt) generuje adresy celého rozsahu paměti (Mem), zastaví se u adresy 7
- Hodnoty (val) z paměti jsou sčítány sčítáčkou (Add) a mezivýsledek uchován v registru (Reg)
- Signál EN, zabraňující přepsání finálního výsledku v Reg, je deaktivován po vygenerování nejvyšší adresy paměti
- Signál RST v log. 1 způsobí nulování registru a čítače
- CLK synchronizuje činnost sekvenčních obvodů – čítače a registru
- Po kompletním průchodu pamětí obsahuje Reg (a jeho výstup – signál SUM) součet všech jejích hodnot a signál DONE je aktivní (v log. 0)



Sekvenční sčítačka s využitím komponent (strukturní popis)

```
entity Acc is
port (
    CLK: in std_logic;
    RST: in std_logic;
    SUM: out std_logic_vector(7 downto 0);
    DONE: out std_logic
);
end Acc;

architecture struct of Acc is

component Add is
port (
    A: in std_logic_vector(7 downto 0);
    B: in std_logic_vector(7 downto 0);
    RES: out std_logic_vector(7 downto 0)
);
end component;

component Cnt is
port (
    CLK: in std_logic;
    RST: in std_logic;
    ADDR: out std_logic_vector(2 downto 0);
    STOP: out std_logic
);
end component;

component Mem is
port (
    ADDR: in std_logic_vector(2 downto 0);
    VAL: out std_logic_vector(7 downto 0)
);
end component;
```

```
component Reg is
port (
    CLK: in std_logic;
    RST: in std_logic;
    EN: in std_logic;
    DIN: in std_logic_vector(7 downto 0);
    DOUT: out std_logic_vector(7 downto 0)
);
end component;

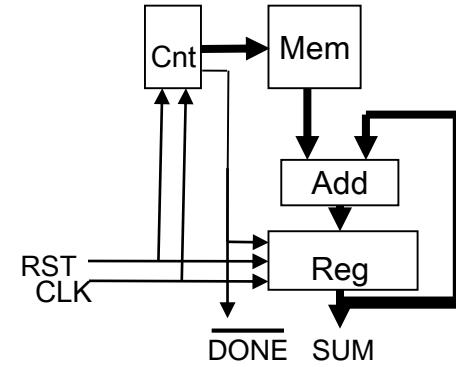
signal addr: std_logic_vector(2 downto 0);
signal val: std_logic_vector(7 downto 0);
signal add_out: std_logic_vector(7 downto 0);
signal reg_out: std_logic_vector(7 downto 0);
signal en: std_logic;

begin

add_inst: Add port map(A => val, B => reg_out, RES => add_out);
mem_inst: Mem port map(ADDR => addr, VAL => val);
cnt_inst: Cnt port map(CLK, RST, addr, en);
reg_inst: Reg port map(CLK, RST, en, add_out, reg_out);

SUM <= reg_out;
DONE <= en;

end struct;
```

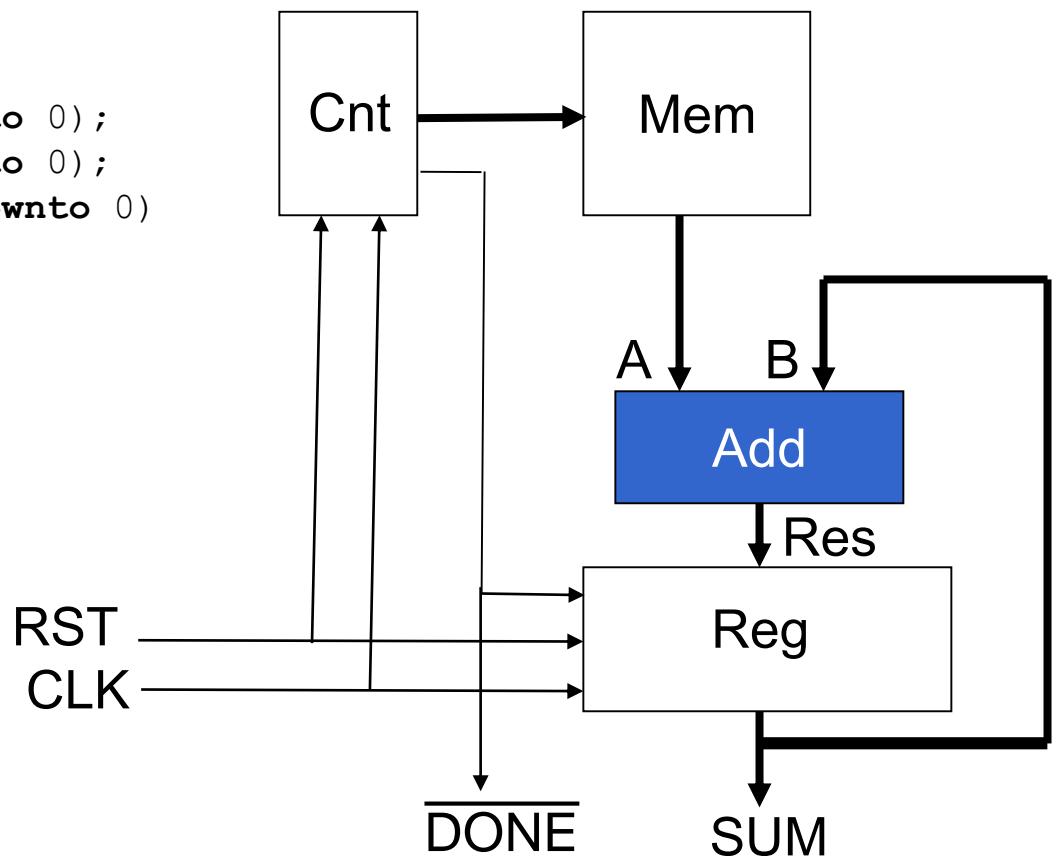


Model sčítáčky (komponenta Add)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity Add is
port (
    A: in std_logic_vector(7 downto 0);
    B: in std_logic_vector(7 downto 0);
    RES: out std_logic_vector(7 downto 0)
);
end Add;

architecture behav_add of Add is
begin
    RES <= A + B;
end behav_add;
```

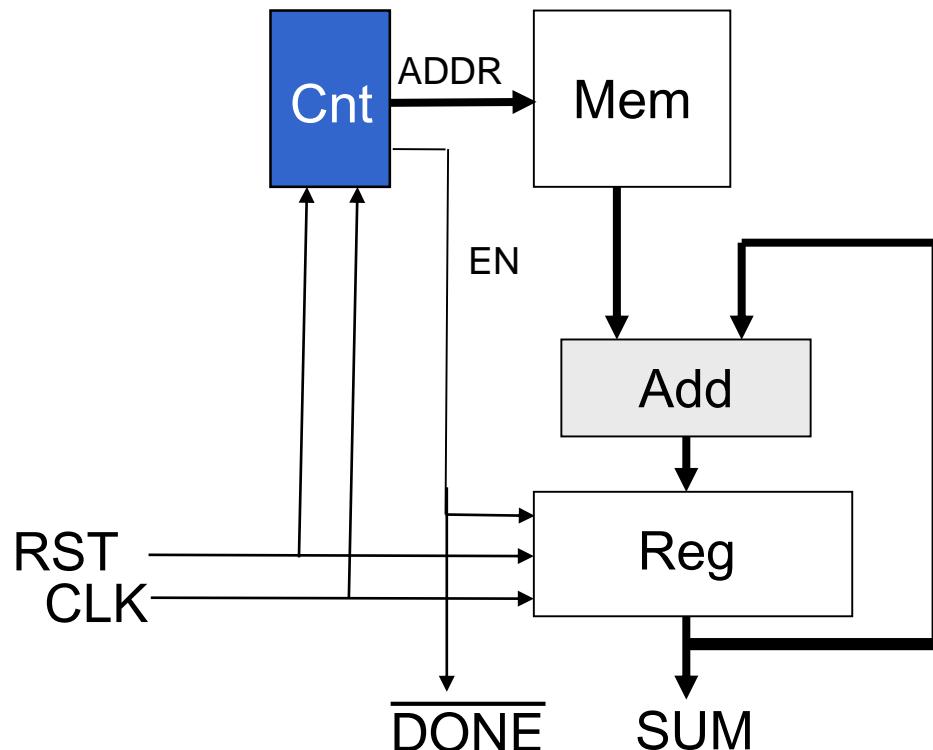


Model čítače (komponenta Cnt)

```
entity Cnt is
port (
    CLK: in std_logic;
    RST: in std_logic;
    ADDR: out std_logic_vector(2 downto 0);
    EN: out std_logic
);
end Cnt;
```

```
architecture behav of Cnt is
signal cnt: std_logic_vector(2 downto 0);
begin
    cnt_proc: process(CLK, RST)
    begin
        if RST = '1' then
            cnt <= "000";
            EN <= '1';
        elsif CLK'event and CLK = '1' then
            EN <= '1';
            if cnt = "111" then
                EN <= '0';
            else
                cnt <= cnt + 1;
            end if;
        end if;
    end process;

    ADDR <= cnt;
end behav;
```

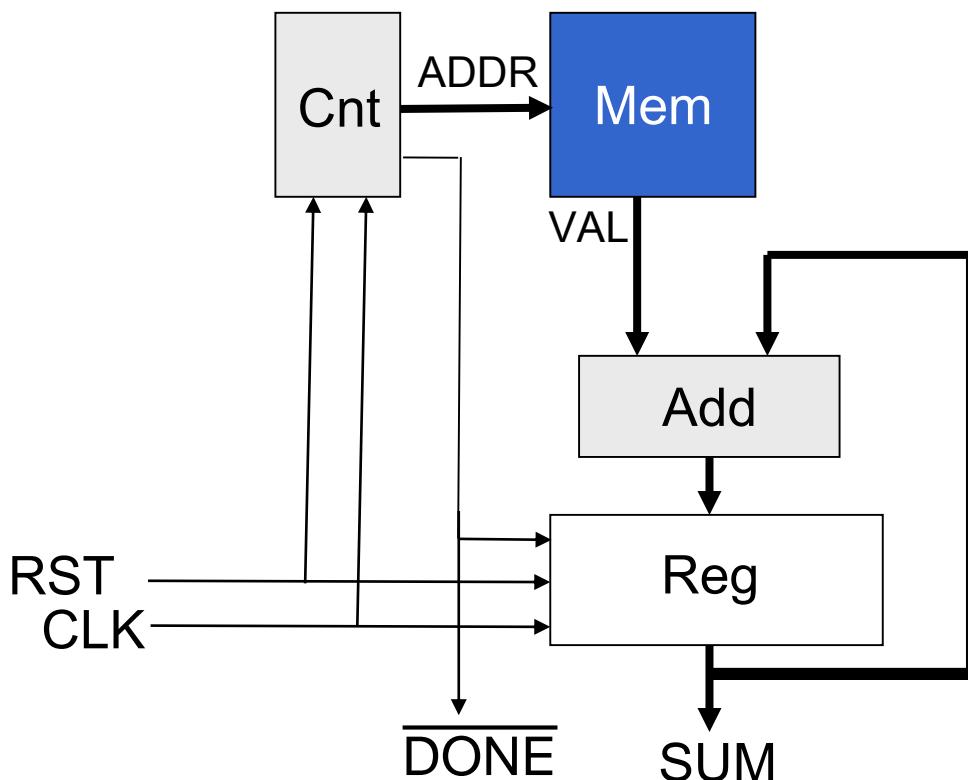


Model ROM paměti (komponenta Mem)

```
entity Mem is
port (
    ADDR: in std_logic_vector(2 downto 0);
    VAL: out std_logic_vector(7 downto 0)
);
end Mem;

architecture behav_mem of Mem is

begin
    mem: process (ADDR)
    begin
        case ADDR is
            when "000" => VAL <= "00000001";
            when "001" => VAL <= "00000010";
            when "010" => VAL <= "00000011";
            when "011" => VAL <= "00000111";
            when "100" => VAL <= "00001111";
            when "101" => VAL <= "00011111";
            when "110" => VAL <= "00111111";
            when "111" => VAL <= "01111111";
            when others => VAL <= "00000000";
        end case;
    end process;
end behav_mem;
```

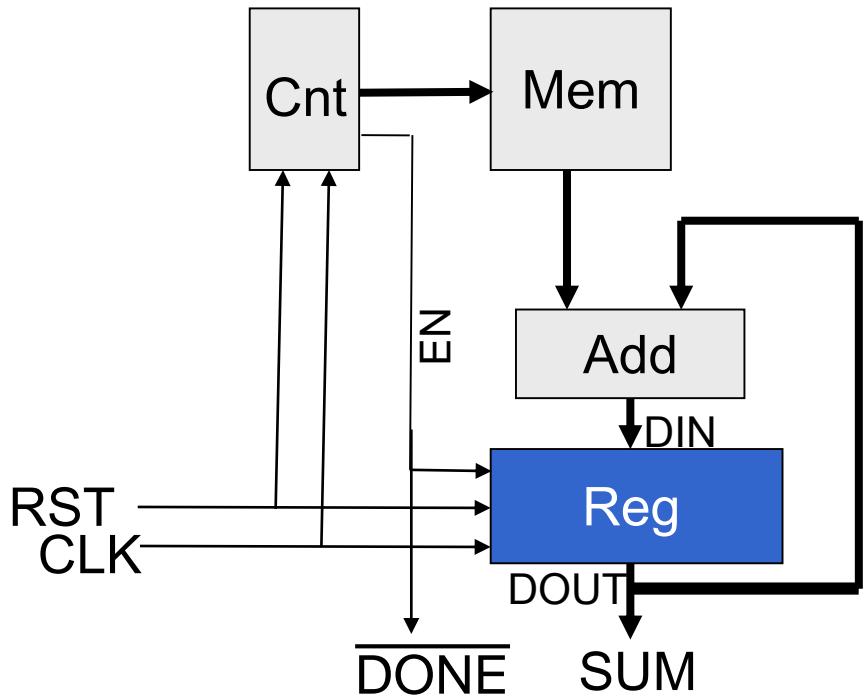


Model registru (komponenta Reg)

```
entity Reg is
port (
    CLK: in std_logic;
    RST: in std_logic;
    EN: in std_logic;
    DIN: in std_logic_vector(7 downto 0);
    DOUT: out std_logic_vector(7 downto 0)
);
end Reg;

architecture behav of Reg is

begin
    reg: process(CLK, RST)
    begin
        if RST = '1' then
            DOUT <= "00000000";
        elsif CLK'event and CLK = '1' then
            if EN='1' then
                DOUT <= DIN;
            end if;
        end if;
    end process;
end behav;
```



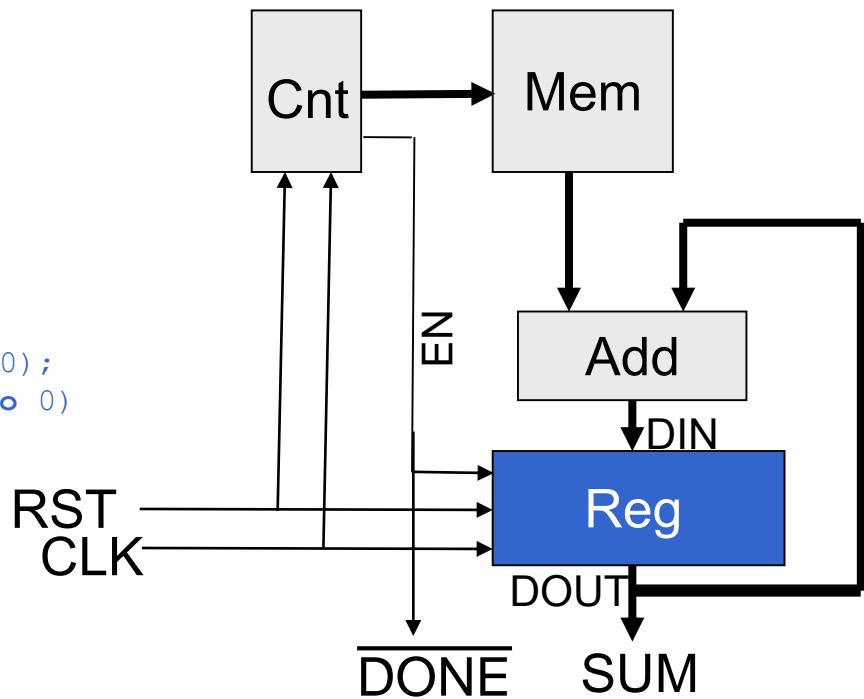
Model registru (komponenta Reg)

Příklad využití generických proměnných

```
entity Reg is
generic (
    DATA_WIDTH : integer := 8
)
port (
    CLK: in std_logic;
    RST: in std_logic;
    EN: in std_logic;
    DIN: in std_logic_vector(DATA_WIDTH-1 downto 0);
    DOUT: out std_logic_vector(DATA_WIDTH-1 downto 0)
);
end Reg;

architecture behav of Reg is

begin
    reg: process(CLK, RST)
    begin
        if RST = '1' then
            DOUT <= (others => '0');
        elsif CLK'event and CLK = '1' then
            if EN='1' then
                DOUT <= DIN;
            end if;
        end if;
    end process;
end behav;
```



Sekvenční sčítačka popsaná behaviorálně

```
entity Acc is
port (
    CLK: in std_logic;
    RST: in std_logic;
    SUM: out std_logic_vector(7 downto 0);
    DONE: out std_logic
);
end Acc;

architecture behav of Acc is

signal addr: std_logic_vector(2 downto 0);
signal val: std_logic_vector(7 downto 0);
signal add_out: std_logic_vector(7 downto 0);
signal reg_out: std_logic_vector(7 downto 0);
signal stop: std_logic;

begin

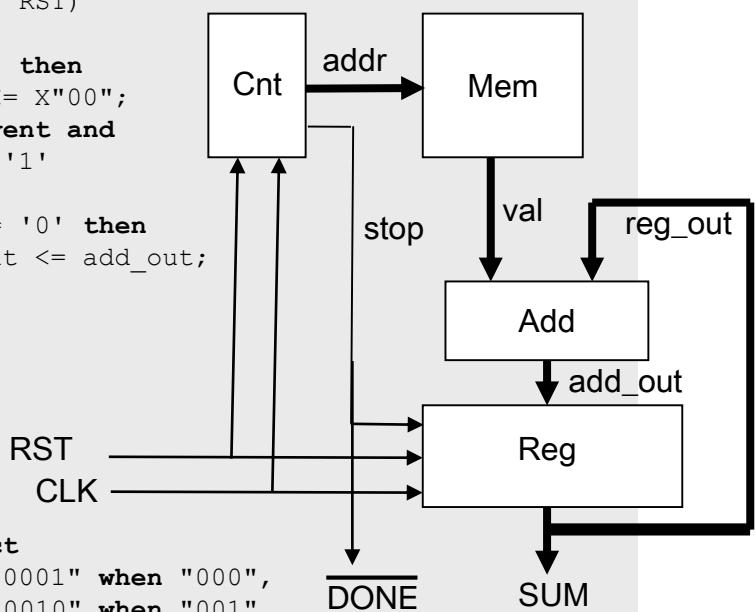
cnt: process(CLK, RST)
begin
    if RST = '1' then
        addr <= "000";
        stop <= '0';
    elsif clk'event and clk = '1' then
        if addr = "111" then
            stop <= '1';
        else
            addr <= addr + 1;
            stop <= '0';
        end if;
    end if;
end process;
```

```
reg: process(CLK, RST)
begin
    if RST = '1' then
        reg_out <= X"00";
    elsif clk'event and clk = '1'
    then
        if stop = '0' then
            reg_out <= add_out;
        end if;
    end if;
end process;

with addr select
    val <= "00000001" when "000",
    "00000010" when "001",
    "00000011" when "010",
    "00000011" when "011",
    "00001111" when "100",
    "00011111" when "101",
    "00111111" when "110",
    "01111111" when "111",
    "00000000" when others;
```

```
    add_out <= reg_out + val;
    SUM <= reg_out;
    DONE <= not stop;
```

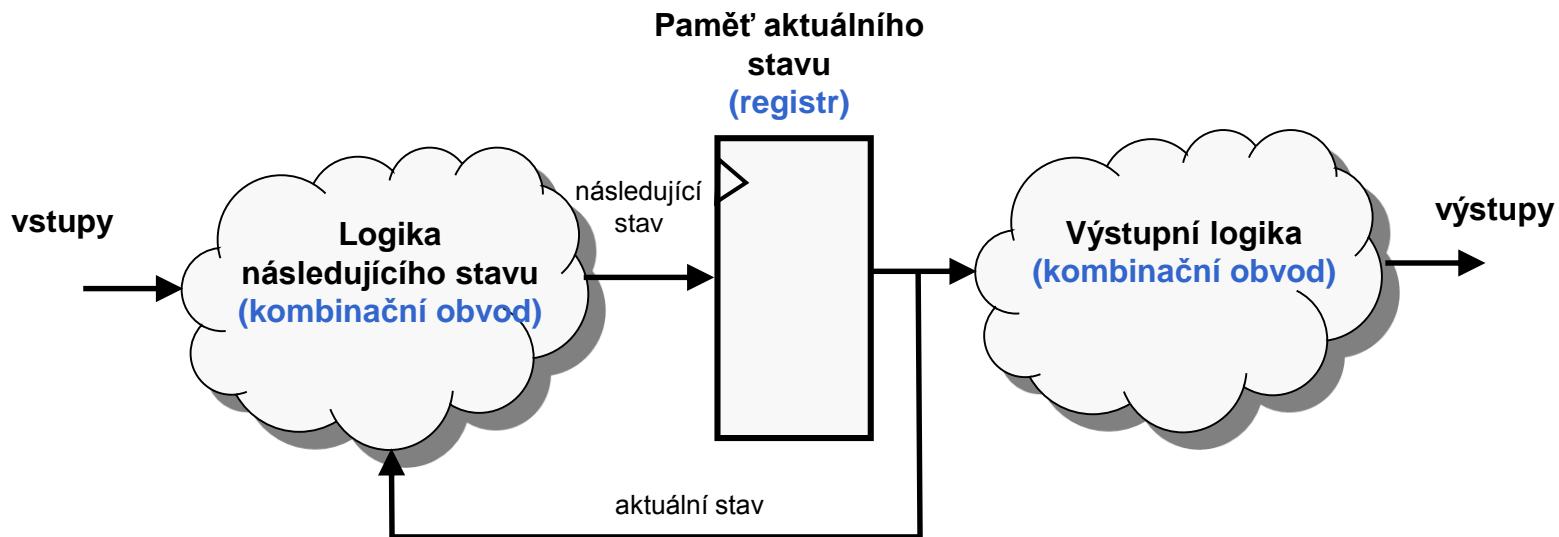
```
end behav;
```



Připomenutí principů modelování konečných automatů ve VHDL

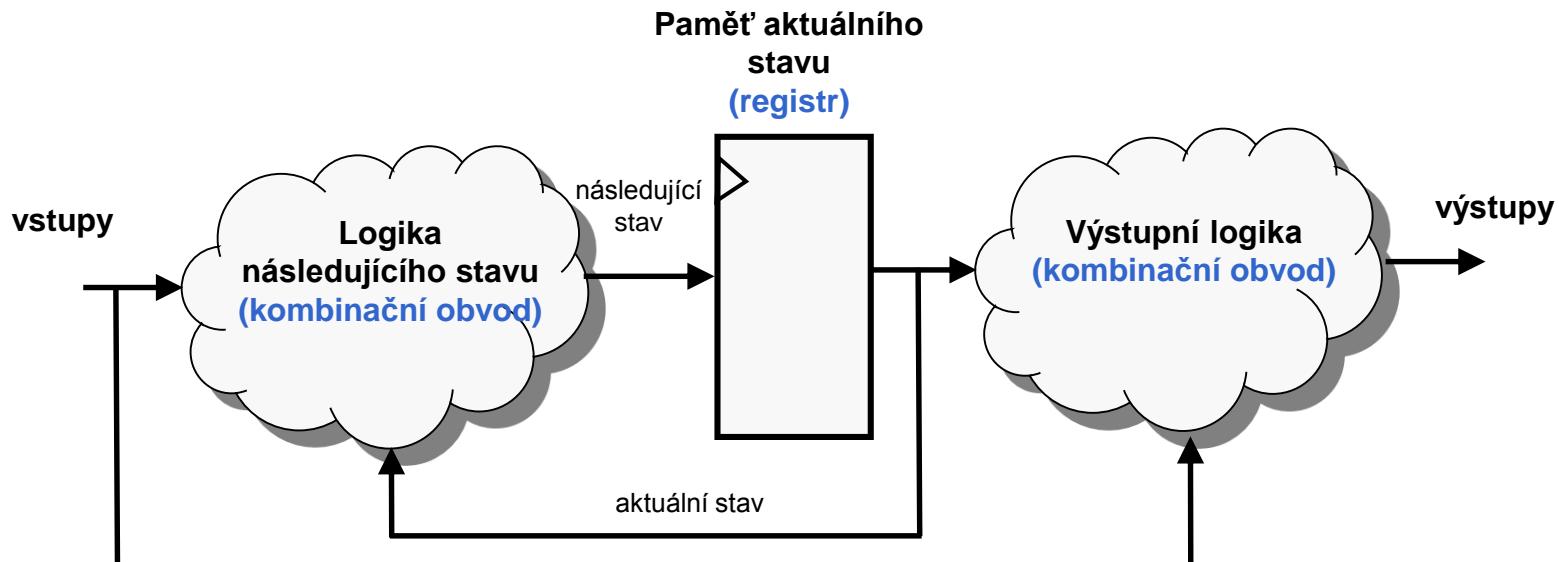
Typy konečných automatů

- Obvod mající paměť lze chápat jako konečný stavový automat (FSM)
- Návrh automatu vyžaduje
 - definovat stavy automatu, přechody mezi stavy a výstupní funkci
- Podle způsobu realizace výstupu rozlišujeme dva základní FSM
 - Moore (výstup je pouze funkcí stavu),
 - Mealy (výstup je funkcí stavu a vstupů)



Typy konečných automatů

- Obvod mající paměť lze chápat jako konečný stavový automat (FSM)
- Návrh automatu vyžaduje
 - definovat stavy automatu, přechody mezi stavy a výstupní funkci
- Podle způsobu realizace výstupu rozlišujeme dva základní FSM
 - Moore (výstup je pouze funkcí stavu),
 - Mealy (výstup je funkcí stavu a vstupů)

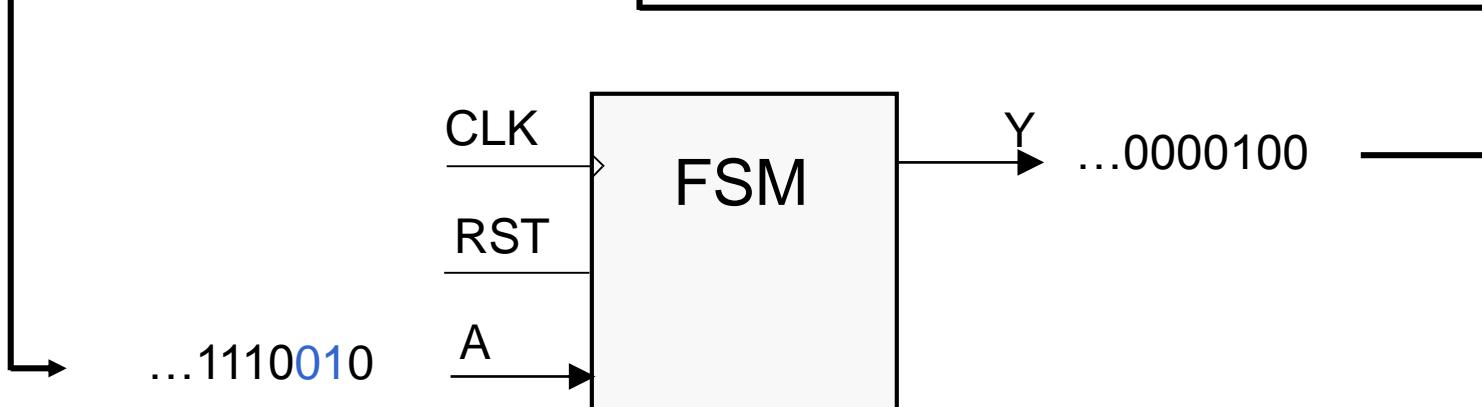


Implementace konečných automatů ve VHDL

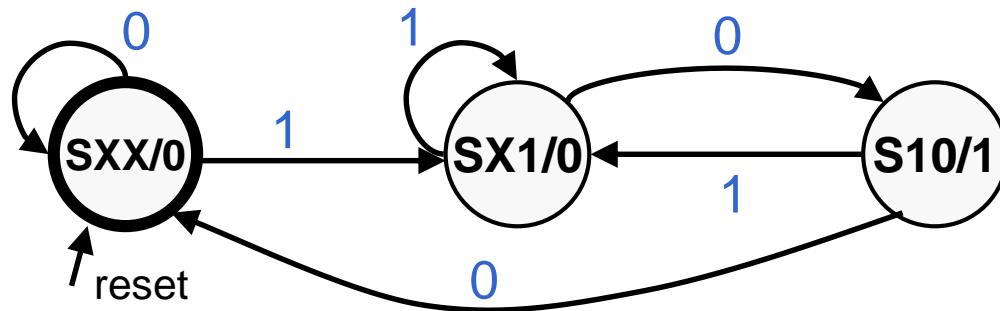
Příklad: detektor posloupnosti 10

- VHDL implementace konečných automatů
 - FSM lze ve VHDL popsat pomocí dvou nebo tří procesů
- Příklad FSM: detektor posloupnosti 10 v řetězci
 - výstupem je 1 pokud byla detekována posloupnost 10

— příklad vstupu: 01001110111101010011011
odpovídající výstup: → 00100001000010101000100



Detektor – Moore FSM – 2 procesy



Deklarace signálů

```
type FSMstate is (SXX, SX1, S10);
signal pstate : FSMstate;
signal nstate : FSMstate;
```

```
--Present State register
pstatereg: process(RST, CLK)
begin
    if (RST='1') then
        pstate <= SXX;
    elsif (CLK'event) and (CLK='1') then
        pstate <= nstate;
    end if;
end process;
```

```
--Next State logic + output logic
nstate_logic: process(pstate, A)
begin
    nstate <= SXX;
    Y <= '0';
    case pstate is
        when SXX =>
            if (A = '1') then
                nstate <= SX1;
            end if;
        when SX1 =>
            nstate <= SX1;
            if (A = '0') then
                nstate <= S10;
            end if;
        when S10 =>
            nstate <= SX1;
            Y <= '1';
            if (A = '0') then
                nstate <= SXX;
            end if;
        when others => null;
    end case;
end process;
```

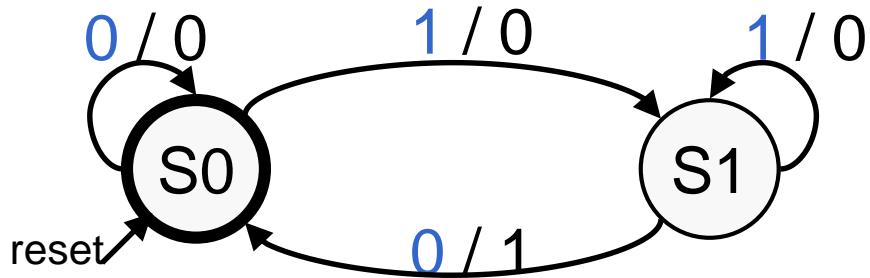
Detektor – Moore FSM – 3 procesy

```
--Present State register
pstatereg: process(RST, CLK)
begin
    if (RST='1') then
        pstate <= SXX;
    elsif (CLK'event) and (CLK='1') then
        pstate <= nstate;
    end if;
end process;
```

```
--Output logic
output_logic: process(pstate)
begin
    Y <= '0';
    case pstate is
        when S10 =>
            Y <= '1';
        when others => null;
    end case;
end process;
```

```
--Next State logic
nstate_logic: process(pstate, A)
begin
    nstate <= SXX;
    case pstate is
        when SXX =>
            if (A = '1') then
                nstate <= SX1;
            end if;
        when SX1 =>
            nstate <= SX1;
            if (A = '0') then
                nstate <= S10;
            end if;
        when S10 =>
            nstate <= SX1;
            if (A = '0') then
                nstate <= SXX;
            end if;
        when others => null;
    end case;
end process;
```

Detektor – Mealy FSM – 2 procesy



Deklarace signálů

```
type FSMstate is (S0, S1);
signal pstate : FSMstate;
signal nstate : FSMstate;
```

```
--Present State register
pstatereg: process(RST, CLK)
begin
    if (RST='1') then
        pstate <= S0;
    elsif (CLK'event) and (CLK='1') then
        pstate <= nstate;
    end if;
end process;
```

```
--Next State logic + output logic
nstate_logic: process(pstate, A)
begin
    nstate <= S0;
    Y <= '0';
    case pstate is
        when S0 =>
            if (A = '1') then
                nstate <= S1;
            end if;
        when S1 =>
            nstate <= S1;
            if (A = '0') then
                Y <= '1';
                nstate <= S0;
            end if;
        when others => null;
    end case;
end process;
```

Detektor – Mealy FSM – 3 procesy

```
--Present State register  
pstatereg: process(RST, CLK)  
begin  
    if (RST='1') then  
        pstate <= S0;  
    elsif (CLK'event) and (CLK='1') then  
        pstate <= nstate;  
    end if;  
end process;
```

```
--Output logic  
output_logic: process(pstate, A)  
begin  
    Y <= '0';  
    if (pstate = S1) and (A = 0) then  
        Y <= '1';  
    end if;  
end process;
```

```
--Next State logic + output logic  
  
nstate_logic: process(pstate, A)  
begin  
    nstate <= S0;  
    case pstate is  
        when S0 =>  
            if (A = '1') then  
                nstate <= S1;  
            end if;  
        when S1 =>  
            nstate <= S1;  
            if (A = '0') then  
                nstate <= S0;  
            end if;  
        when others => null;  
    end case;  
end process;
```

Pozor na sensitivity list u výstupní logiky

Detektor – Mealy FSM – 3 procesy

```
--Present State register  
pstatereg: process(RST, CLK)  
begin  
    if (RST='1') then  
        pstate <= S0;  
    elsif (CLK'event) and (CLK='1') then  
        pstate <= nstate;  
    end if;  
end process;
```

Výstupní logika nemusí být nutně popsána pomocí procesu

```
--Output logic  
Y <= '1' when (pstate = S1) and (A = 0) else  
    '0';
```

```
--Next State logic + output logic  
nstate_logic: process(pstate, A)  
begin  
    nstate <= S0;  
    case pstate is  
        when S0 =>  
            if (A = '1') then  
                nstate <= S1;  
            end if;  
        when S1 =>  
            nstate <= S1;  
        if (A = '0') then  
            nstate <= S0;  
        end if;  
        when others => null;  
    end case;  
end process;
```

Implementace procesoru ve VHDL

INP - cvičení 3

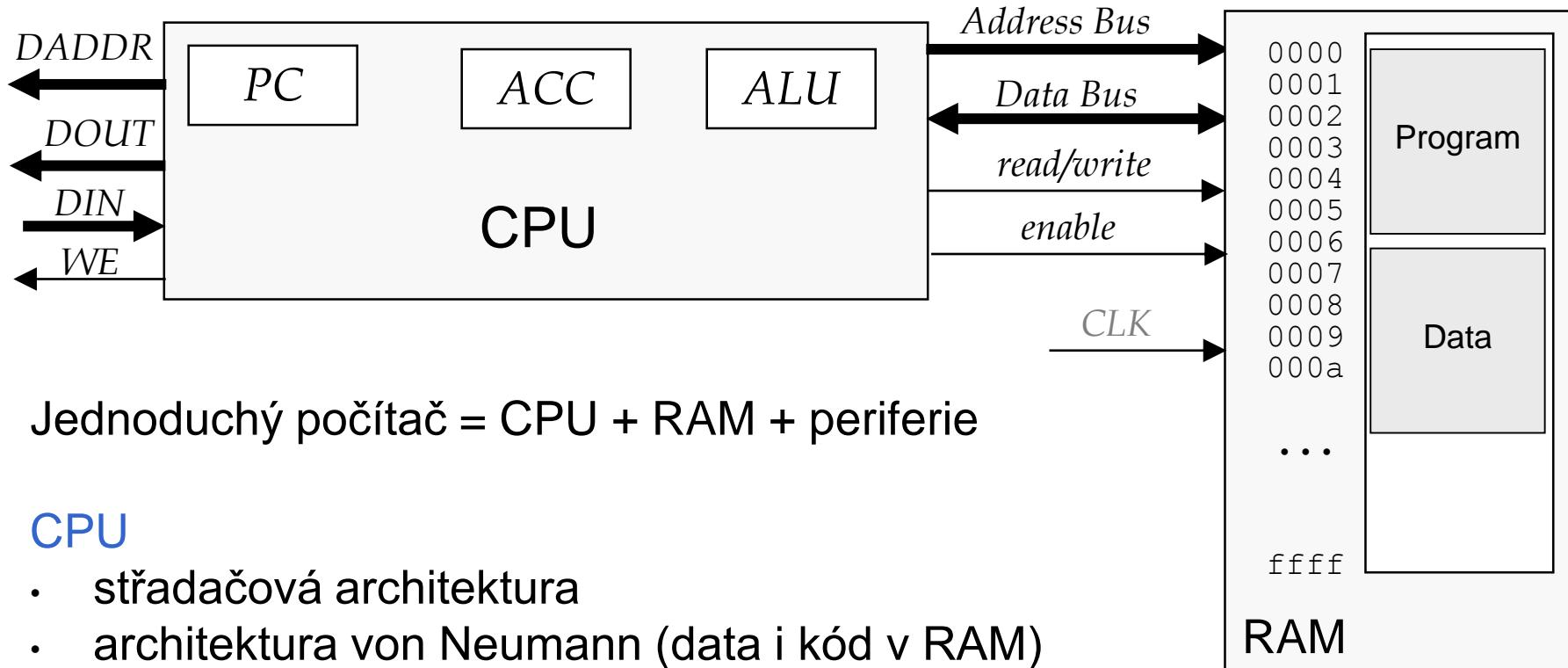
Zdeněk Vašíček, 2015
vasicek@fit.vutbr.cz

Implementace procesoru ve VHDL

1. Volba architektury
 - registrová, střadačová, zásobníková
 - Harward, Von Neumann
 - specifikace datové šířky
2. Návrh instrukční sady s ohledem na HW implementaci
3. Návrh blokového schema datové cesty
4. Přepis blokového schema do VHDL, implementace konečného automatu dle požadavků instrukční sady
 - v případě jednodušších procesorů obvykle jedna komponenta, jeden process

Realizace jednoduchého počítače

Zadání



Jednoduchý počítač = CPU + RAM + periferie

CPU

- střadačová architektura
- architektura von Neumann (data i kód v RAM)
- vstupně-výstupní 16-bitové rozhraní

RAM

- synchronní paměť s organizací N x 16 bitů
- komunikace přes sběrnici (šetření zdroji)

Sada instrukcí

Zadání

Operační znak	Instrukce	Popis
0000	halt	zastav provádění programu
0001	negate	vytvoř dvojkový doplněk z ACC
0002	accdec	sniž hodnotu ACC o jedna
0003	accinc	zvyš hodnotu ACC o jedna
000F	nop	prázdná operace
01xx	outp	zapiš hodnotu ACC na port s adresou xx
02xx	inp	načti do ACC hodnotu z portu s adresou xx
1xxx	mload	nahrej do ACC hodnotu xxx
2xxx	dload	nahrej do ACC hodnotu z adresy xxx
3xxx	iload	nahrej do ACC hodnotu, která je uložena na adrese, kterou definuje obsah buňky xxx
4xxx	dstore	ulož hodnotu z ACC na adresu xxx
5xxx	istore	ulož hodnotu z ACC na adresu, která je určena hodnotou paměťové buňky s adresou xxx

Sada instrukcí

Zadání

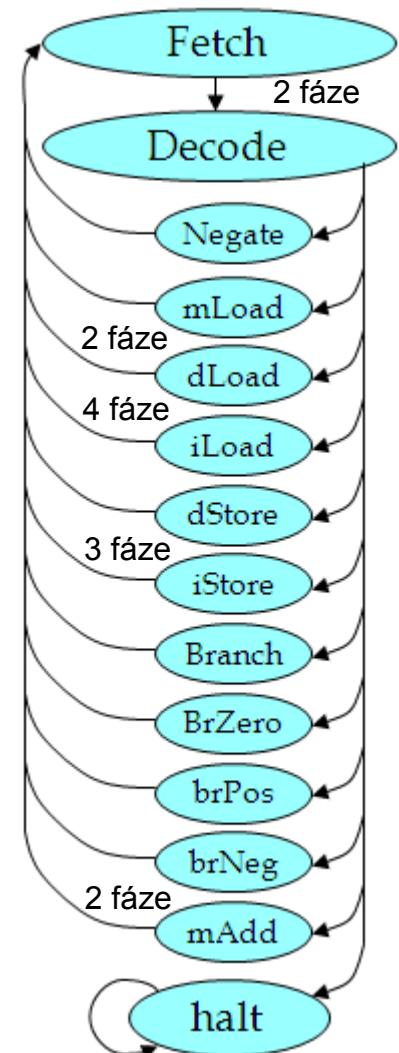
Operační znak	Instrukce	Popis
6xxx	branch	změň PC na xxx
7xxx	brzero	změň PC na xxx jestliže ACC = 0
8xxx	brpos	změň PC na xxx jestliže ACC > 0
9xxx	brneg	změň PC na xxx jestliže ACC < 0
Axxx	madd	přičti k ACC obsah paměťové buňky na adresu xxx
Fxxx	ijump	nepřímý skok na adresu uloženou na adresce xxx

Pomocí této poměrně strohé instrukční sady lze implementovat libovolný algoritmus.

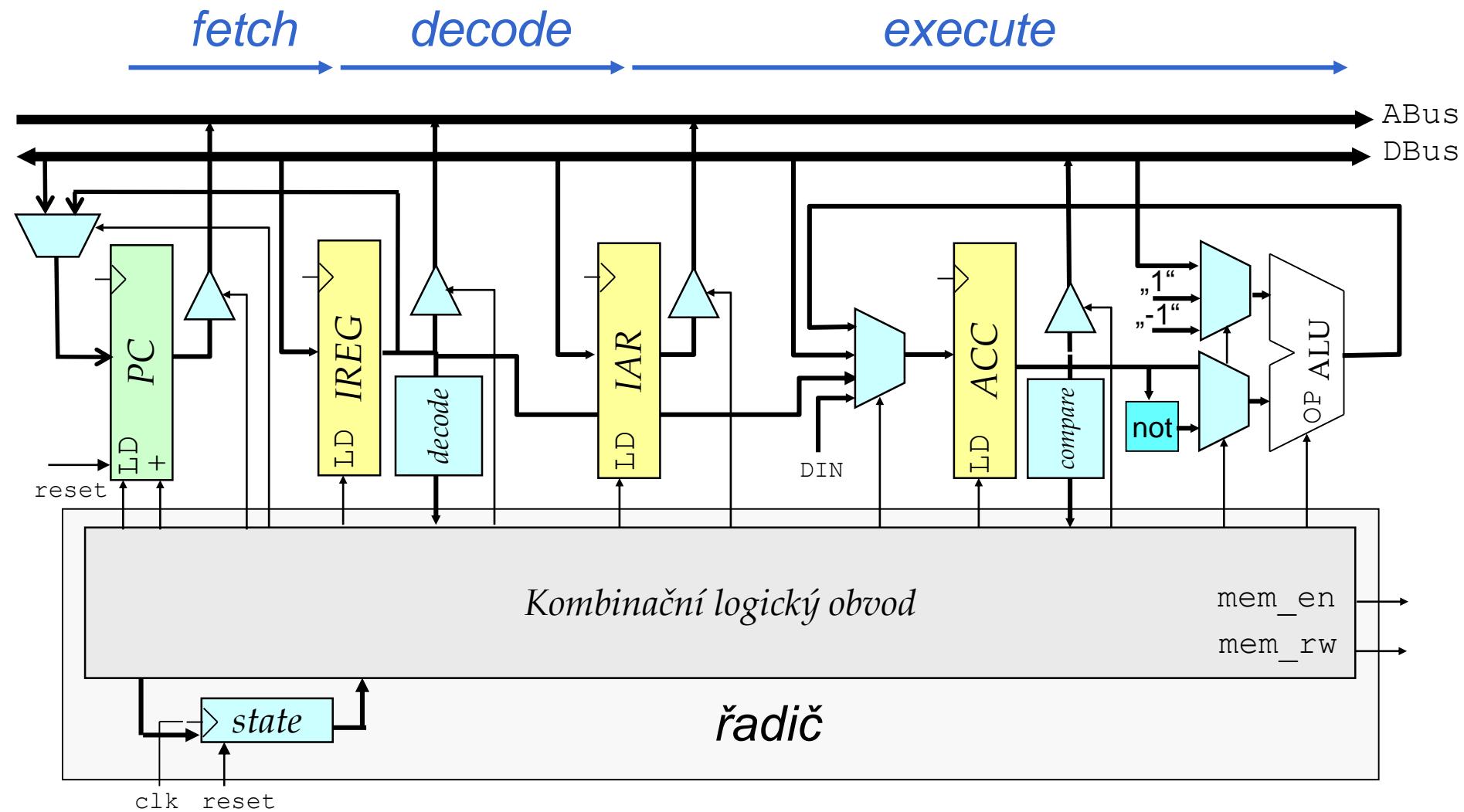
Instrukční cyklus - FSM

Procesor neustále vykonává tyto operace

- **Načtení instrukce** (Instruction fetch)
 - *PC* je použit pro čtení slova z paměti
 - *PC* je inkrementován, zápis slova do instrukčního registru
- **Dekódování instrukce** (Instruction decode)
 - podle nejvyšších 4,8,12,16 bitů se určí, co se bude dělat
 - aktivují se příslušné obvody
- **Provedení instrukce** (Instruction execution)
 - načtení dalších potřebných slov
 - zápis do paměti
 - modifikace *PC*, *ACC* apod.
 - může trvat různý počet taktů dle typu instrukce

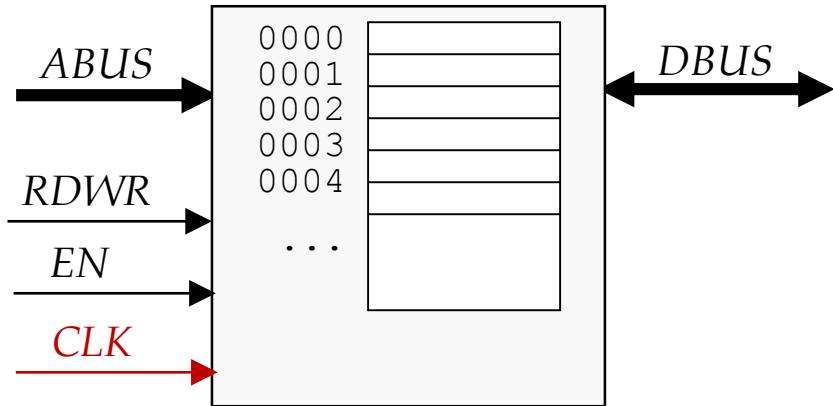


Architektura procesoru a jeho činnost



RAM

- RAM (*random access memory*)
 - když $EN = 1$ a $RDWR = 1$, $DBUS$ obsahuje hodnotu uloženou na pozici, kterou definuje $ABUS$
 - když $EN = 1$ a $RDWR = 0$, hodnota na $DBUS$ přepíše data uložená na adrese vystavené na $ABUS$
 - jinak je na $DBUS$ stav vysoké impedance
- Pro implementaci budeme uvažovat **synchronní** paměť RAM, protože lze na rozdíl od asynchronní varianty efektivně implementovat v FPGA
- VHDL implementace paměti RAM
 - **efektivní implementace:** nesmí mít RESET, musí být synchronní
 - možnosti zápisu: process (signal, shared variable) nebo strukturní popis využívající vestavěné blokové synchronní paměti BRAM



VHDL model synchronní paměti RAM

```
entity ram is
  port (
    CLK, EN, RDWR: in STD_LOGIC;
    ABUS: in STD_LOGIC_VECTOR(15 downto 0);
    DBUS: inout STD_LOGIC_VECTOR(15 downto 0)
  );
end ram;
```

```
type t_ram is array (0 to 2**10-1) of std_logic_vector (15 downto 0);
signal ram: t_ram := (x"0201",x"0003", x"0101", others=>x"0000");

DBUS <= dout when EN = '1' else (others => 'Z');

process (CLK)
begin
  if (CLK'event) and (CLK = '1') then
    if (EN = '1') then
      if (RDWR = '0') then
        ram(conv_integer(ABUS(9 downto 0))) <= DBUS;
      end if;
      dout <= ram(conv_integer(ABUS(9 downto 0)));
    end if;
  end if;
end process;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

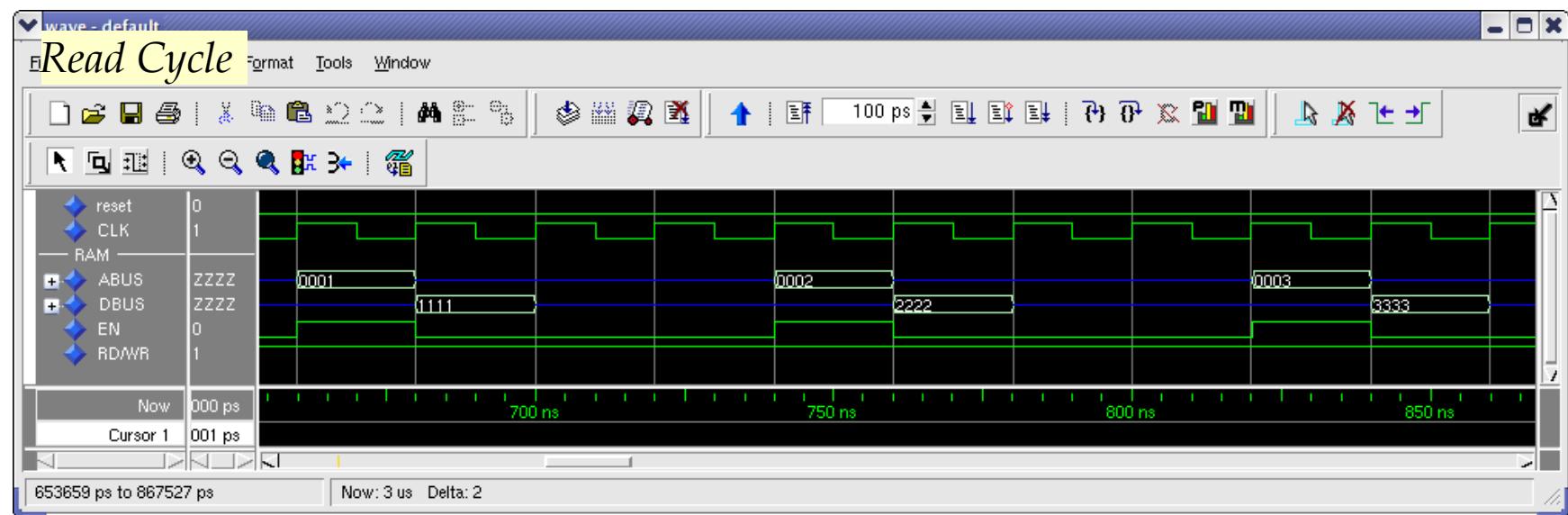
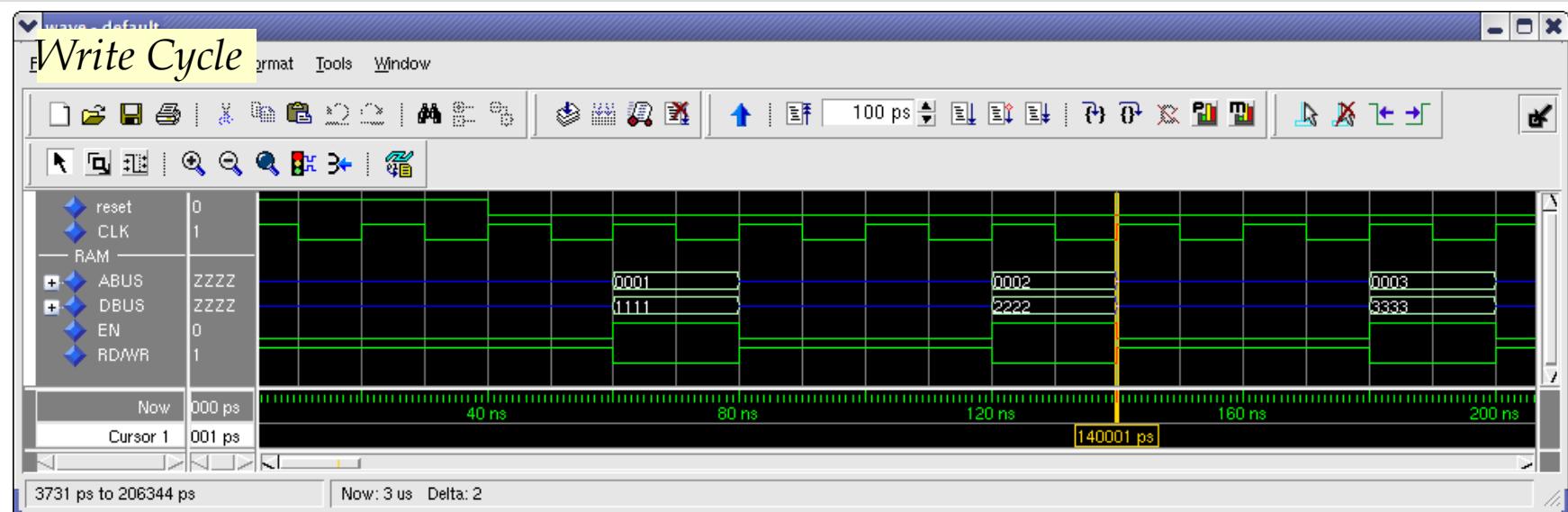
Paměť jako pole 16b slov

Inicializace obsahu paměti (není nutné)

Zápis na pozici určenou adresou

čtení z pozice určené adresou

Simulace a časování synchronní RAM



Model CPU ve VHDL

Entita

```
entity cpu is
  port (
    RESET : in std_logic;
    CLK   : in std_logic;
    CE    : in std_logic;

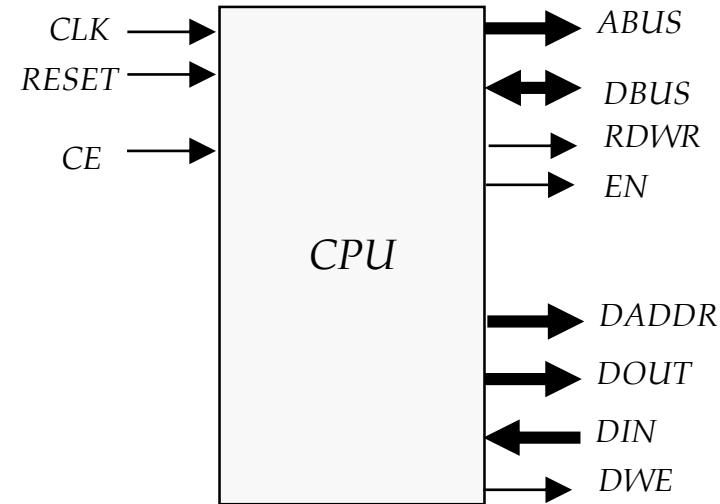
    -- BUS
    ABUS  : out std_logic_vector(15 downto 0);
    DBUS  : inout std_logic_vector(15 downto 0);

    EN     : out std_logic;
    RDWR  : out std_logic

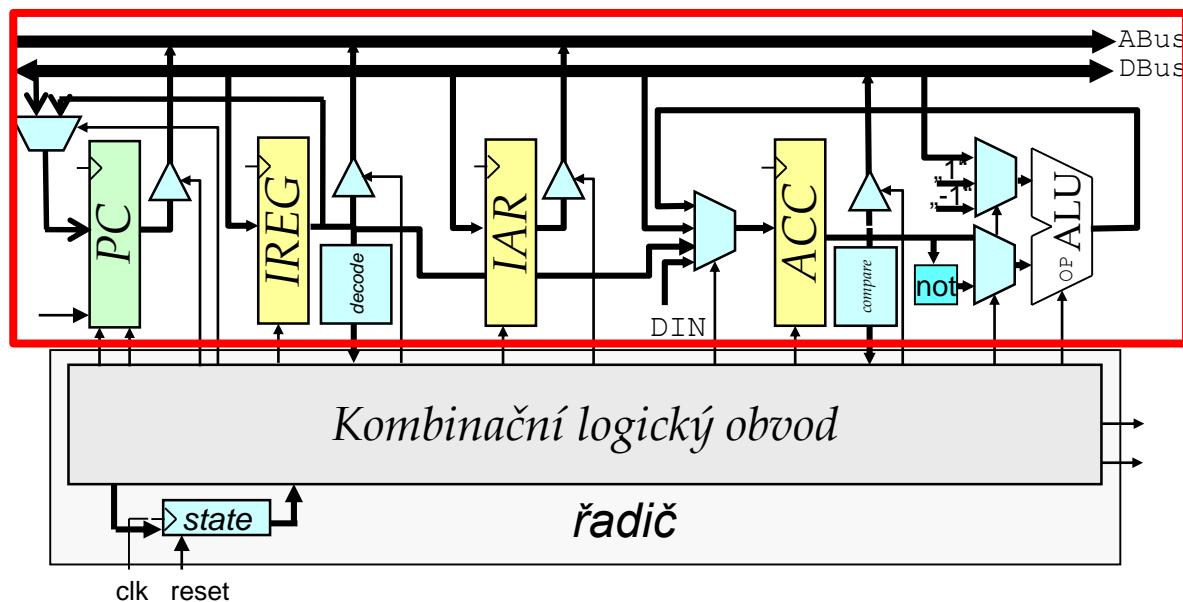
    -- I/O port
    DADDR : out std_logic_vector(7 downto 0);
    DOUT  : out std_logic_vector(15 downto 0);
    DIN   : in  std_logic_vector(15 downto 0);
    DWE   : out std_logic;
  );
end cpu;
```

Asynchronní nulování

Povolení činnosti (chip enable)



Implementace datové cesty



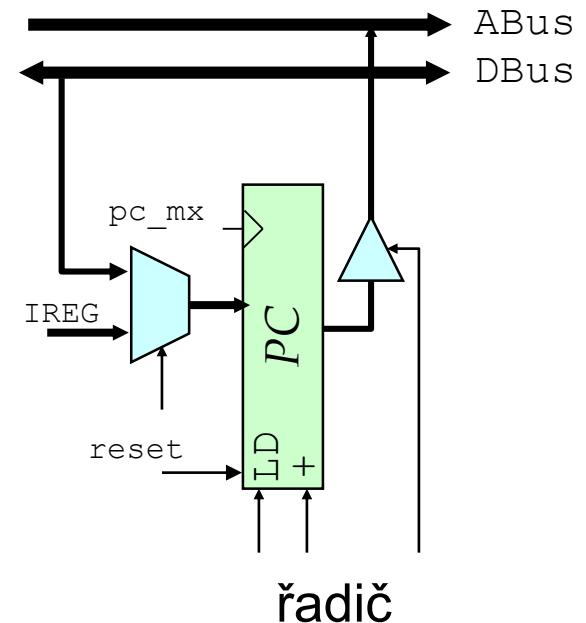
Programový čítač (PC)

```
signal pc_reg : std_logic_vector(15 downto 0);
signal pc_ld : std_logic;
signal pc_inc : std_logic;

-- Program counter PC
pc_cntr: process (RESET, CLK)
begin
    if (RESET='1') then
        pc_reg <= (others=>'0');
    elsif (CLK'event) and (CLK='1') then
        if (pc_ld='1') then
            pc_reg <= pc_mx;
        elsif (pc_inc='1') then
            pc_reg <= pc_reg + 1;
        end if;
    end if;
end process;

pc_mx <= "0000" & ireg_reg(11 downto 0) when
            pc_mx_sel="00" else DBUS

-- Tristate driver
ABUS <= pc_reg when (pc_abus = '1')
            else (others => 'Z');
```



Hodnotu čítače lze

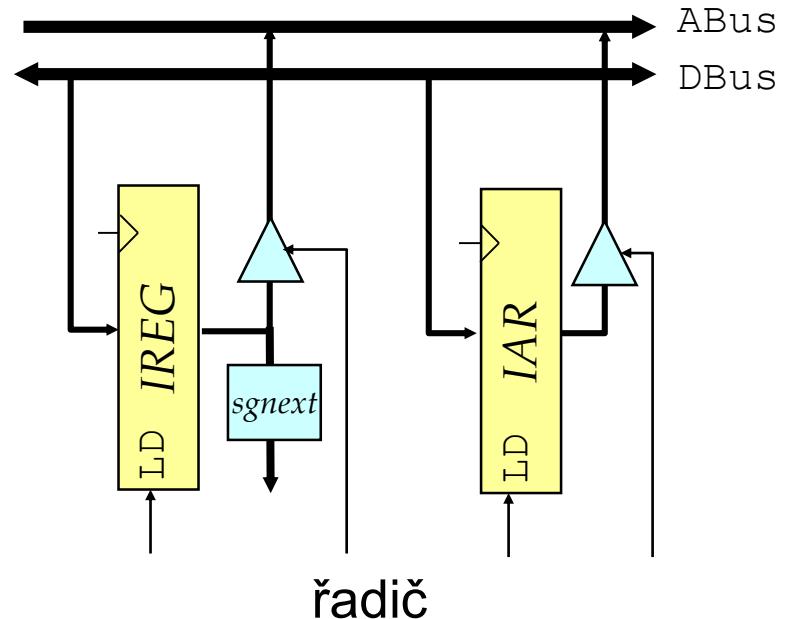
- inkrementovat
- přepsat hodnotou na DBUS (instrukce `ijump`)
- přepsat hodnotou v IREG (skoky ne/podmíněné)

Instrukční a adresový registr (IREG, IAR)

```
signal ireg_reg : std_logic_vector(15 downto 0);
signal ireg_ld  : std_logic;

-- Instruction register IREG
ireg: process (RESET, CLK)
begin
    if (RESET='1') then
        ireg_reg <= (others=>'0');
    elsif (CLK'event) and (CLK='1') then
        if (ireg_ld='1') then
            ireg_reg <= DBUS;
        end if;
    end if;
end process;

-- Indirect address register IAR
iar: process (RESET, CLK)
begin
    if (RESET='1') then
        iar_reg <= (others=>'0');
    elsif (CLK'event) and (CLK='1') then
        if (iar_ld='1') then
            iar_reg <= DBUS;
        end if;
    end if;
end process;
```



```
-- Tristate driver
ABUS <= "0000" & ireg_reg(11 downto 0)
when (ireg_abus = '1')
else (others=>'Z');

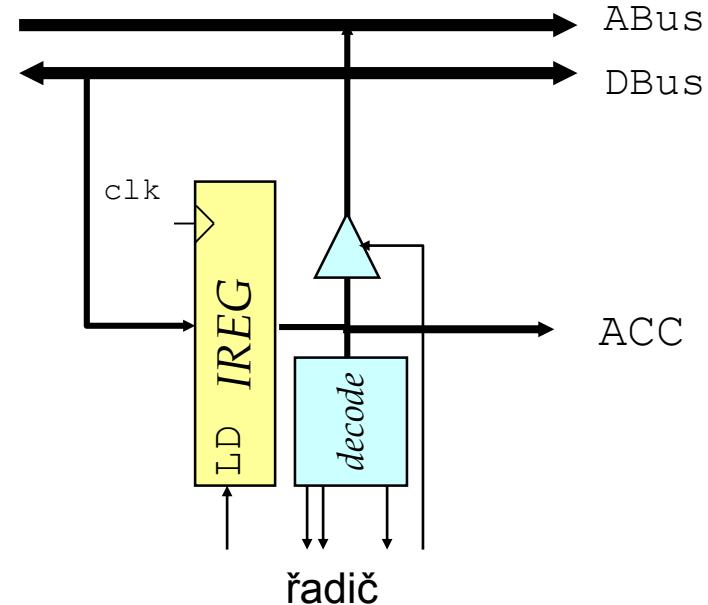
ABUS <= iar_reg when (iar_abus = '1')
else (others=>'Z');

--IREG sign extension
ireg_sgnnext <= "1111" when ireg_reg(11)='1'
else "0000";
```

Instrukční dekodér

```
type inst_type is (halt, negate, mload, dload, iload,
                    dstore, istore, branch, brzero, ... );
signal ireg_dec : inst_type;

--Instruction decoder
process (ireg)
begin
    case (ireg(15 downto 12)) is
        when X"0" =>
            case (ireg_reg(11 downto 8)) is
                when X"0" =>
                    case (ireg_reg(3 downto 0)) is
                        when X"0"      => ireg_dec <= halt;
                        when X"1"      => ireg_dec <= negate;
                        ...
                        when others   => ireg_dec <= halt;
                    end case;
                when X"1" => ireg_dec <= outp;
                when X"2" => ireg_dec <= inp;
                when others   => ireg_dec <= halt;
            end case;
        when X"1" => ireg_dec <= mload;
        when X"2" => ireg_dec <= dload;
        ...
        when others   => ireg_dec <= halt;
    end case;
end process;
```



Operační znak	Instrukce
0000	halt
0001	negate
...	...
01xx	outp
02xx	inp
1xxx	mload
2xxx	dload
...	...
Fxxx	ijump

Akumulátor (ACC)

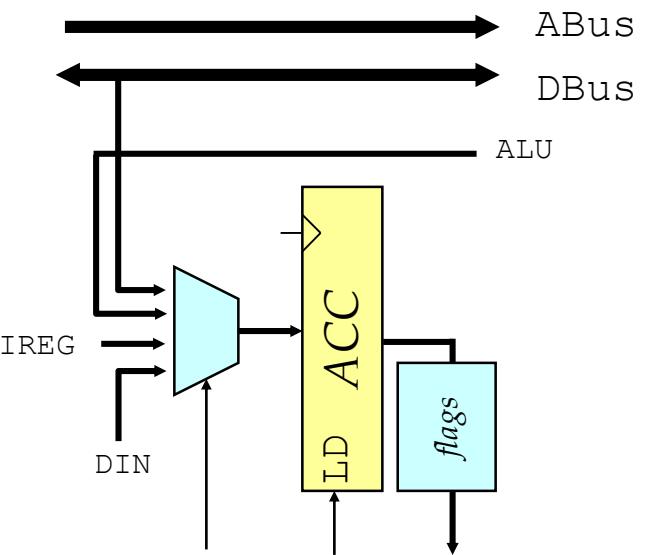
```

signal acc_reg      : std_logic_vector(15 downto 0);
signal acc_ld       : std_logic;
signal acc_mx       : std_logic_vector(15 downto 0);
signal accmx_sel   : std_logic_vector(1 downto 0);
signal acc_zero     : std_logic;
signal acc_neg      : std_logic;
signal acc_pos      : std_logic;

-- ACC data multiplexor
with accmx_sel select
acc_mx <= ireg_sgnnext & ireg_reg(11 downto 0)
    when "00",
        DBUS    when "01",
        DIN     when "10",
        alu_out when others;

-- Accumulator register ACC
accreg: process(RESET, CLK)
begin
    if (RESET='1') then
        acc_reg <= (others>'0');
    elsif (CLK'event) and (CLK='1') then
        if (acc_ld='1') then
            acc_reg <= acc_mx;
        end if;
    end if;
end process;

```



```

-- ACC flags comparator
acc_zero <= '1' when (acc_reg = X"0000")
            else '0';
acc_neg   <= '1' when (acc_reg(15) = '1')
            else '0';
acc_pos   <= '1' when (acc_reg(15) = '0')
            and (acc_zero='0')
            else '0';

```

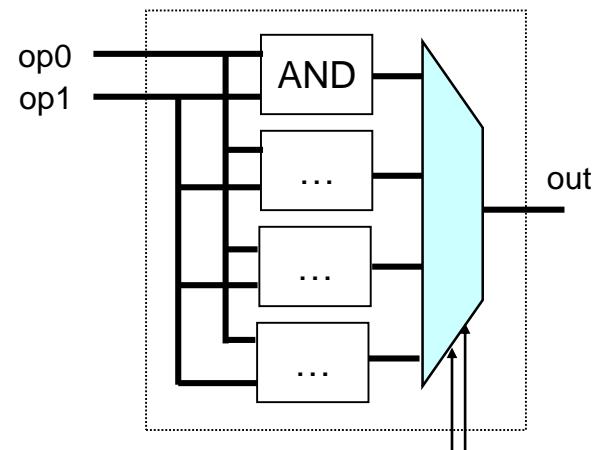
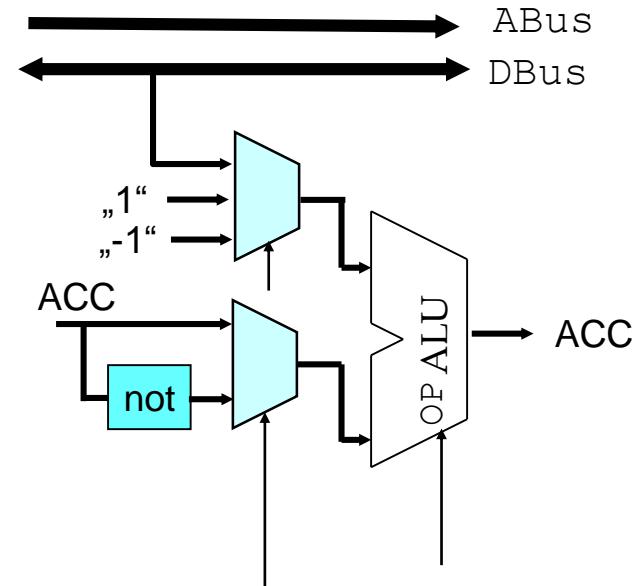
Aritmeticko logická jednotka (ALU)

```
type aluoper_type is (alu_add, alu_and, ... );
signal alu_oper : aluoper_type;
signal alu_op0, alu_op1 : std_logic_vector(15 downto 0);
signal alu_out : std_logic_vector(15 downto 0);
signal alu_mx1_sel : std_logic_vector(1 downto 0);
signal alu_mx2_sel : std_logic;

-- Operands multiplexors
alu_op0 <= DBUS when alu_mx1_sel="00"
    X"0001" when alu_mx1_sel="01"
    else X"1111";

alu_op1 <= acc_reg when alu_mx2_sel='0'
    else (not acc_reg);

-- ALU
with alu_oper select
    alu_out <= alu_op0 + alu_op1 when alu_add,
        alu_op0 and alu_op1 when alu_and,
        ...
        alu_op1 when others;
```



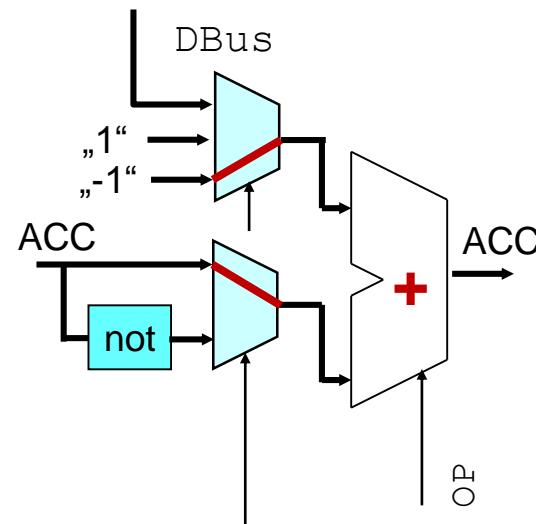
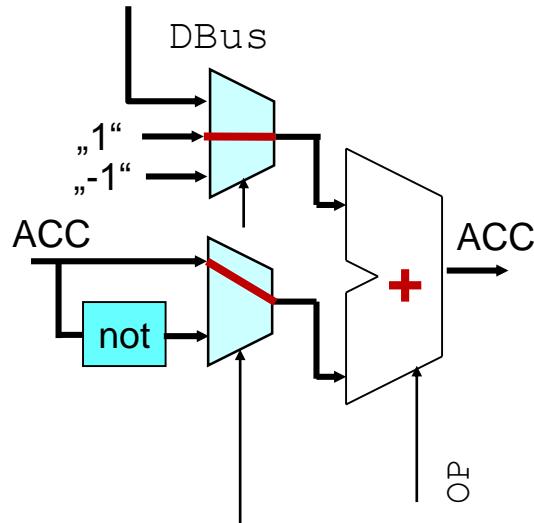
Navržená ALU je schopna

- inkrementovat a dekrementovat obsah ACC
 - vytvořit jedničkový (negace) a dvojkový doplněk k ACC
 - provádět operace: add, sub, and, ... s ACC a hodnotou na DBUS
- Přepsání ACC hodnotou na DBUS je implementováno v ACC

Aritmeticko logická jednotka (ALU)

Podporované operace nad ACC

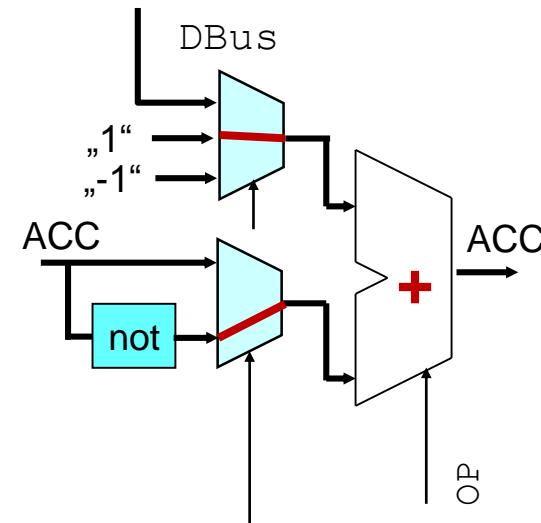
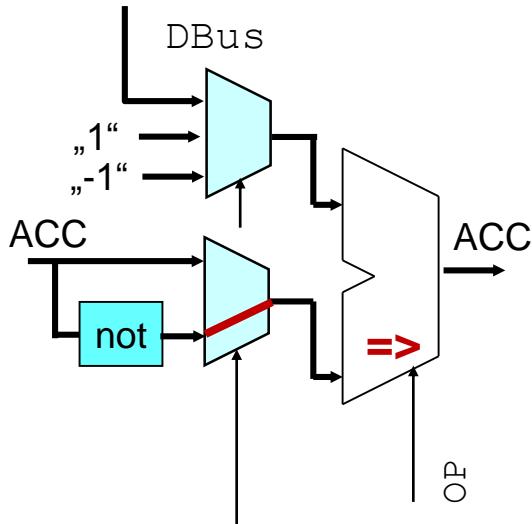
- Inkrementace obsahu ACC
 $(ACC \leq ACC + 1)$
- Dekrementace obsahu ACC
 $(ACC \leq ACC - 1)$



Aritmeticko logická jednotka (ALU)

Podporované operace nad ACC

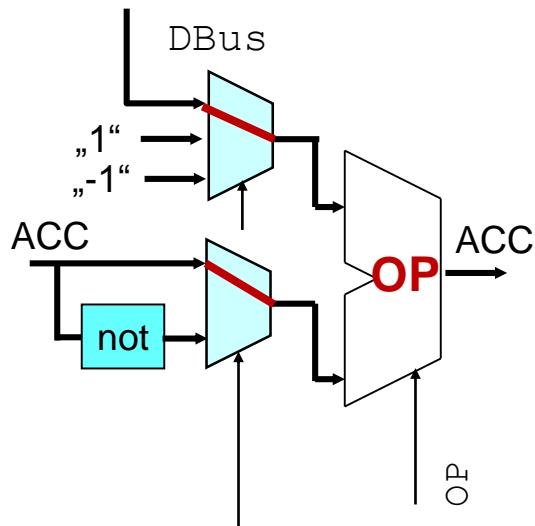
- Negace ACC
(jedničkový doplněk)
($ACC \leq \text{not } ACC$)
- Inverze ACC
(dvojkový doplněk)
($ACC \leq \neg ACC$)



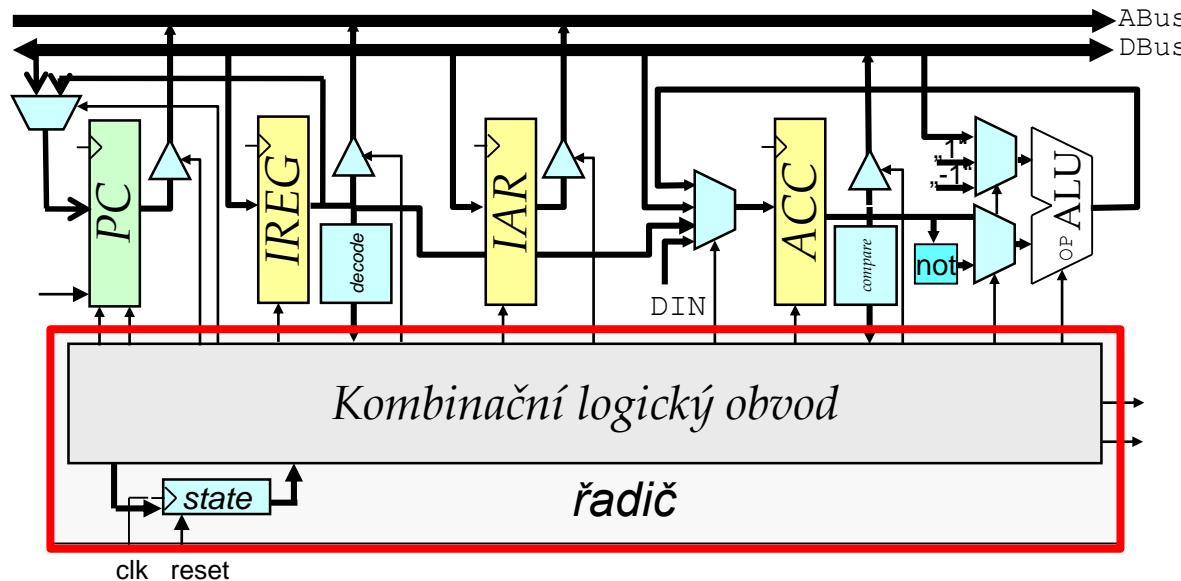
Aritmeticko logická jednotka (ALU)

Podporované operace nad ACC

- Operace s DBUS a ACC dle možností ALU
(ACC <= DBUS op ACC)

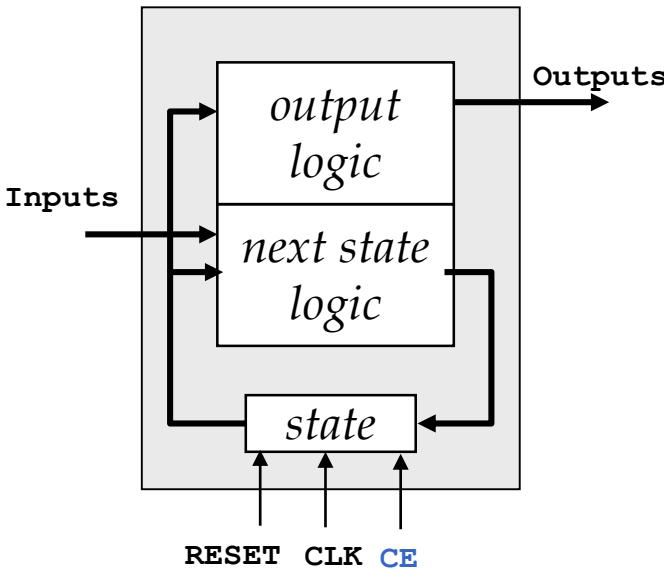


Implementace řídící cesty



Řadič

FSM



deklarace signálů

```
type fsm_state is (sidle, sfetch0, sfetch1, sdecode,  
sbranch, shalt, snop, smload, snegate, smadd0,  
smadd1, sdload0, sdload1, sdstore, siload0,  
siload1, siload2, siload3, sistore0, sistore1,  
sistore2, saccdec, saccinc, sijump0, sijump1,  
soutp, sinp);  
  
signal pstate : fsm_state;  
signal nstate : fsm_state;
```

registr aktuálního stavu

```
--FSM present state  
fsm_pstate: process (RESET, CLK)  
begin  
    if (RESET='1') then  
        pstate <= sidle;  
    elsif (CLK'event) and (CLK='1') then  
        if (CE = '1') then  
            pstate <= nstate;  
        end if;  
    end if;  
end process;
```

Řadič – 1/3

logika následujícího stavu a výstupní logika

```
--FSM next state logic,  
  output logic  
nsl: process (pstate,  
    ireg_dec, acc_zero,  
    acc_pos, acc_neg)  
begin  
  -- INIT  
  EN <= '0';  
  DWE <= '0';  
  RDWR <= '1';  
  ireg_ld <= '0';  
  ireg_abus <= '0';  
  pc_inc <= '0';  
  pc_ld <= '0';  
  pc_abus <= '0';  
  pc_mx_sel <= "00";  
  iar_ld <= '0';  
  iar_abus <= '0';  
  acc_mx_sel <= "00";  
  acc_ld <= '0';  
  alu_mx1_sel <= "01";  
  alu_mx2_sel <= '0';  
  alu_oper <= alu_add;  
  dbus_sel <= '0';
```

```
case pstate is  
  -- IDLE  
  when sidle =>  
    nstate <= sfetch0;  
  
  -- INSTRUCTION FETCH  
  when sfetch0 =>  
    nstate <= sfetch1;  
    pc_abus <= '1';  
    EN <= '1';  
  
  when sfetch1 =>  
    ireg_ld <= '1';  
    nstate <= sdecode;  
  
  -- INSTRUCTION DECODE  
  when sdecode =>  
    case ireg_dec is  
      when halt =>  
        nstate <= halt;  
      when nop =>  
        nstate <= snop;  
      when branch =>  
        nstate <= sbranch;  
      when brzero =>  
        if (acc_zero='1') then  
          nstate <= sbranch;  
        else  
          nstate <= snop;  
        end if;
```

```
when brpos =>  
  if (acc_pos='1') then  
    nstate <= sbranch;  
  else  
    nstate <= snop;  
  end if;  
when brneg =>  
  if (acc_neg='1') then  
    nstate <= sbranch;  
  else  
    nstate <= snop;  
  end if;  
when accdec =>  
  nstate <= saccdec;  
when accinc =>  
  nstate <= saccinc;  
when add =>  
  nstate <= sadd0;  
when dload =>  
  nstate <= sdload0;  
when dstore =>  
  nstate <= sdstore;  
when iload =>  
  nstate <= siload0;  
when istore =>  
  nstate <= sistore0;  
...  
when others =>  
  nstate <= shalt;  
end case;
```

Q: proč je při vykonávání instrukce brzero, brpos, brneg přechod do stavu snop?

Řadič – 2/3

logika následujícího stavu a výstupní logika

```
-- HALT
when shalt =>
    nstate <= shalt;

-- BRANCH
when sbranch =>
    nstate <= sfetch0;
    pc_ld <= '1';

-- NOP
when snop =>
    nstate <= sfetch0;
    pc_inc <= '1';

-- LOAD IMMEDIATE
when smload =>
    nstate <= sfetch0;
    acc_mx_sel <= "00";
    acc_ld <= '1';
    pc_inc <= '1';

-- NEGATE
when snegate =>
    nstate <= sfetch0;
    acc_mx_sel <= "11";
    alu_oper <= alu_add;
    alu_mx1_sel <= "01";
    alu_mx2_sel <= '1';
    acc_ld <= '1';
    pc_inc <= '1';
```

```
-- ACC DEC
when saccdec =>
    nstate <= sfetch0;
    acc_mx_sel <= "11";
    alu_oper <= alu_add;
    alu_mx1_sel <= "11";
    alu_mx2_sel <= '0';
    acc_ld <= '1';
    pc_inc <= '1';

-- ACC INC
when saccinc =>
    nstate <= sfetch0;
    acc_mx_sel <= "11";
    alu_oper <= alu_add;
    alu_mx1_sel <= "01";
    alu_mx2_sel <= '0';
    acc_ld <= '1';
    pc_inc <= '1';

-- ADD
when smadd0 => --phase 0
    nstate <= smadd1;
    ireg_abus <= '1';
    EN <= '1';
```

```
when smadd1 => --phase 1
    nstate <= sfetch0;
    alu_oper <= alu_add;
    acc_mx_sel <= "11";
    alu_mx1_sel <= "00";
    alu_mx2_sel <= '0';
    acc_ld <= '1';
    pc_inc <= '1';

-- LOAD DIRECT
when sdload0 => --phase 0
    nstate <= sdload1;
    ireg_abus <= '1';
    EN <= '1';

when sdload1 => --phase 1
    nstate <= sfetch0;
    acc_mx_sel <= "01";
    acc_ld <= '1';
    pc_inc <= '1';

-- STORE DIRECT
when sdstore =>
    nstate <= sfetch0;
    dbus_sel <= '1';
    ireg_abus <= '1';
    EN <= '1';
    RDWR <= '0';
    pc_inc <= '1';
```

Řadič – 3/3

logika následujícího stavu a výstupní logika

```
-- LOAD INDIRECT
when siload0 => --phase 0
    nstate <= siload1;
    ireg_abus <= '1';
    EN <= '1';

when siload1 => --phase 1
    nstate <= siload2;
    iar_ld <= '1';

when siload2 => --phase 2
    nstate <= siload3;
    iar_abus <= '1';
    EN <= '1';

when siload3 => --phase 3
    nstate <= sfetch0;
    acc_mx_sel <= "01";
    acc_ld <= '1';
    pc_inc <= '1';
```

```
-- STORE INDIRECT
when sistore0 => --phase 0
    nstate <= sistore1;
    ireg_abus <= '1';
    EN <= '1';

when sistore1 => --phase 1
    nstate <= sistore2;
    iar_ld <= '1';

when sistore2 => --phase 2
    nstate <= sfetch0;
    iar_abus <= '1';
    dbus_sel <= '1';
    EN <= '1';
    RDWR <= '0';
    pc_inc <= '1';
```

```
-- INDIRECT JUMP
when sijump0 => --phase 0
    nstate <= sijump1;
    ireg_abus <= '1';
    EN <= '1';

when sijump1 => --phase 1
    nstate <= sfetch0;
    iar_ld <= '1';
    pc_ld <= '1';
    pc_mx_sel <= "11";

-- PORT OUTPUT
when soutp =>
    nstate <= sfetch0;
    DWE <= '1';
    pc_inc <= '1';

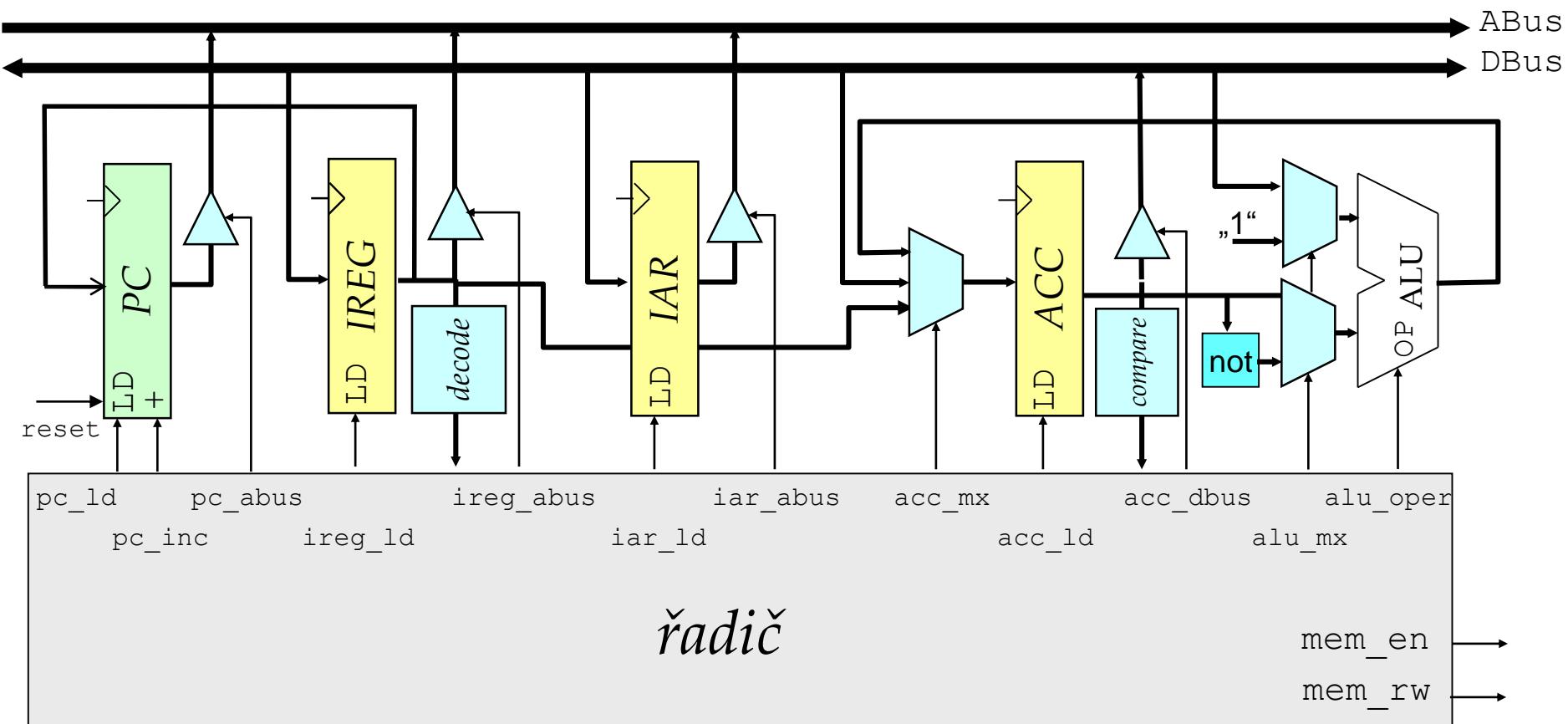
-- PORT INPUT
when sinp =>
    nstate <= sfetch0;
    acc_mx_sel <= "10";
    acc_ld <= '1';
    pc_inc <= '1';

when others =>
    null;

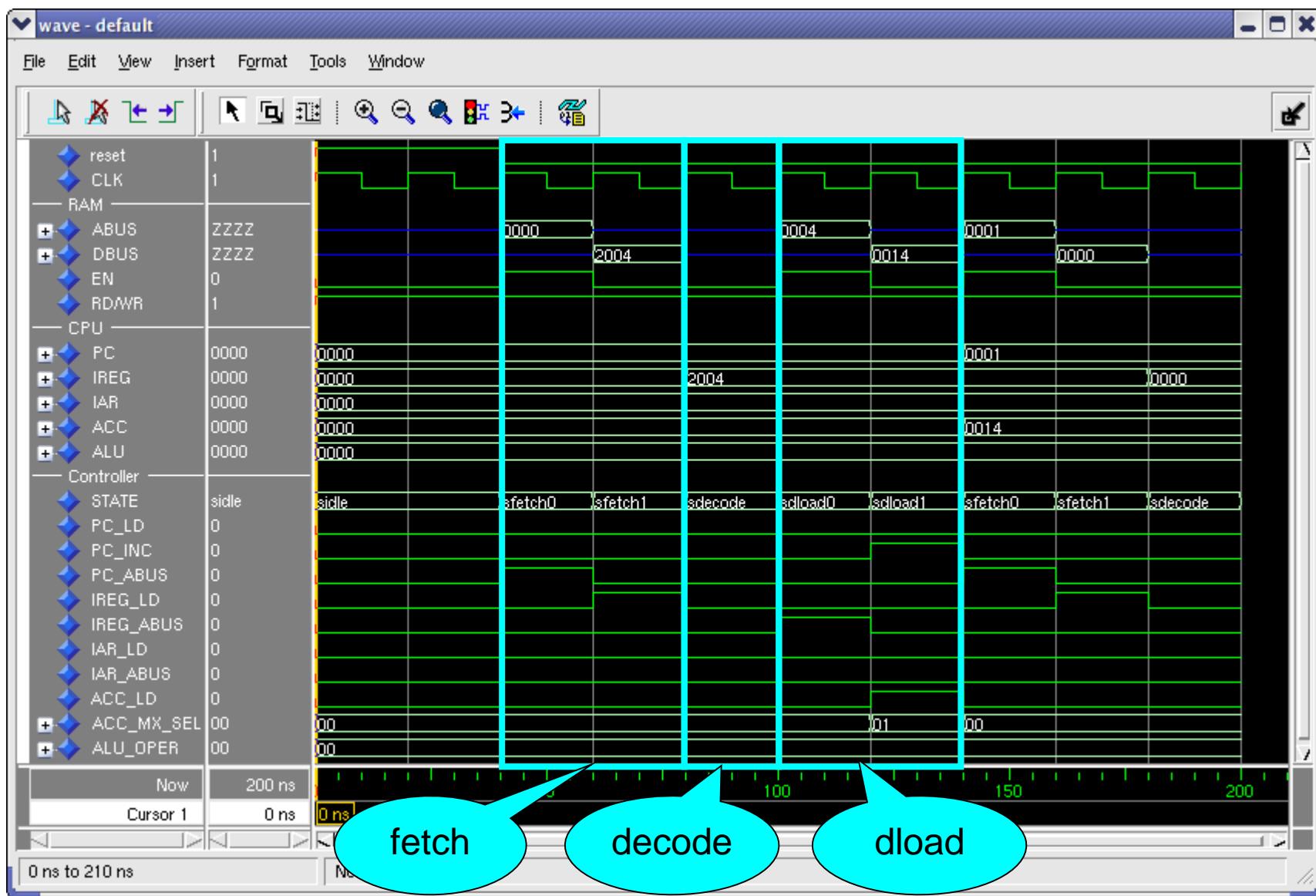
end case;
end process;
```

Q: proč vykonání instrukce iload trvá 4 a istore jen 3 takty?

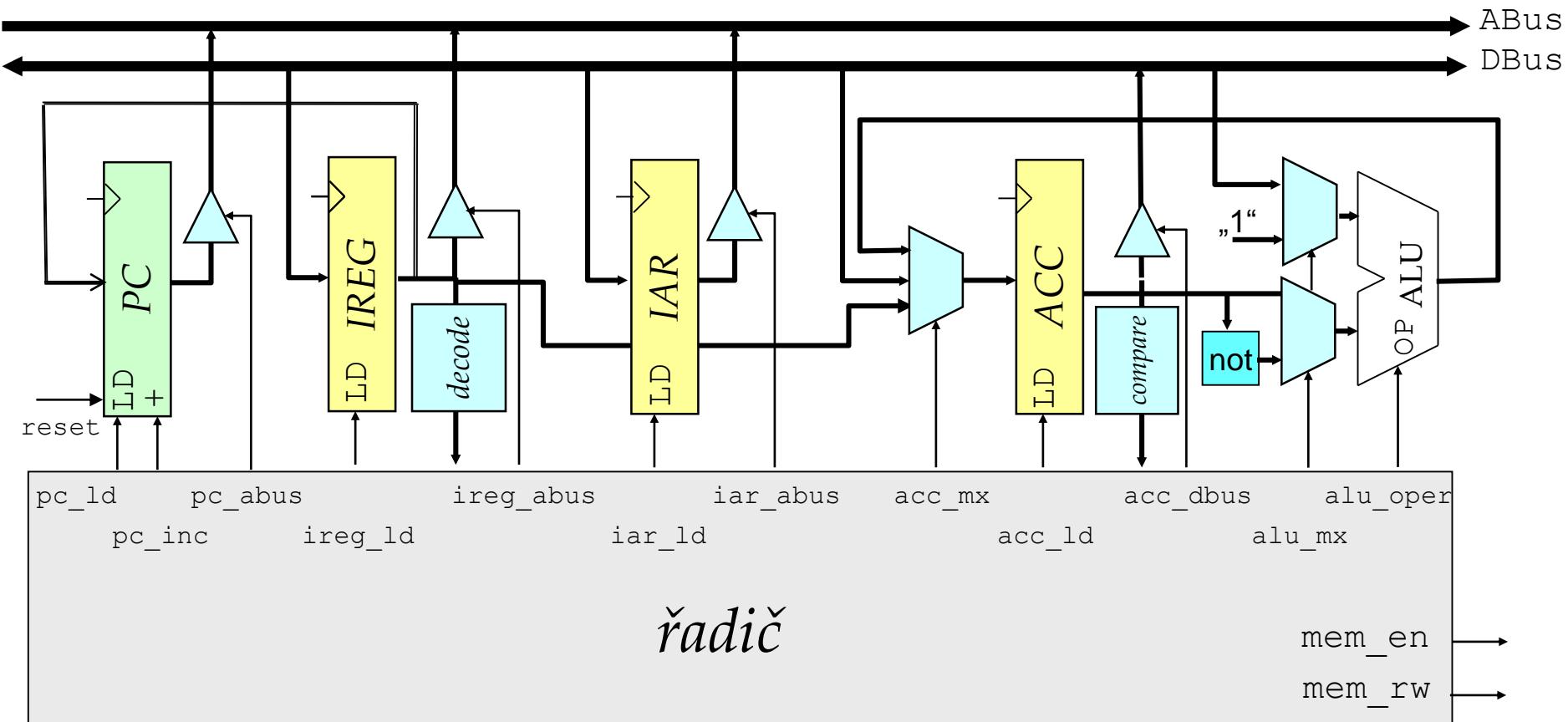
Příklad: DLOAD acc, m[04]



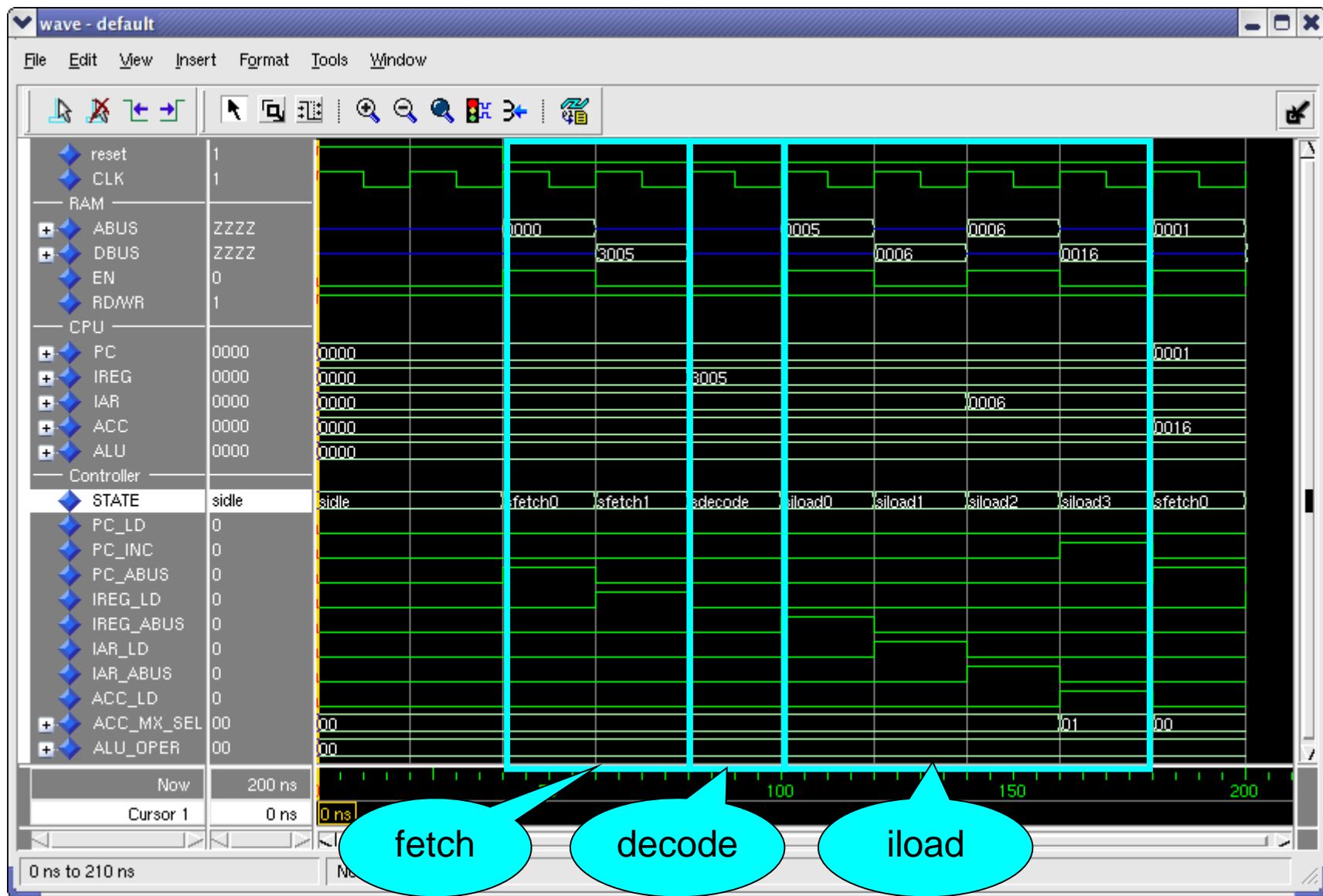
Příklad: DLOAD acc, m[04]



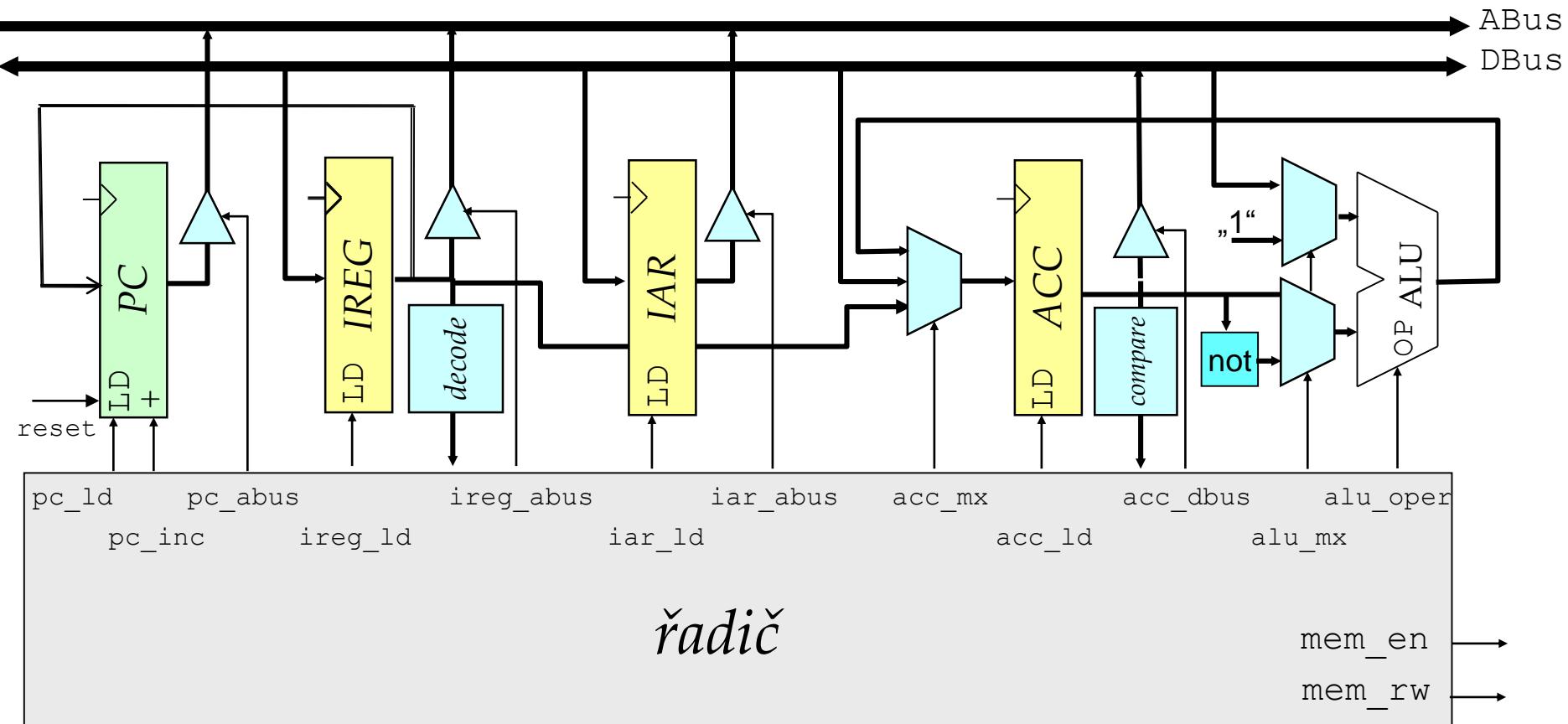
Příklad: ILOAD acc, *m[05]



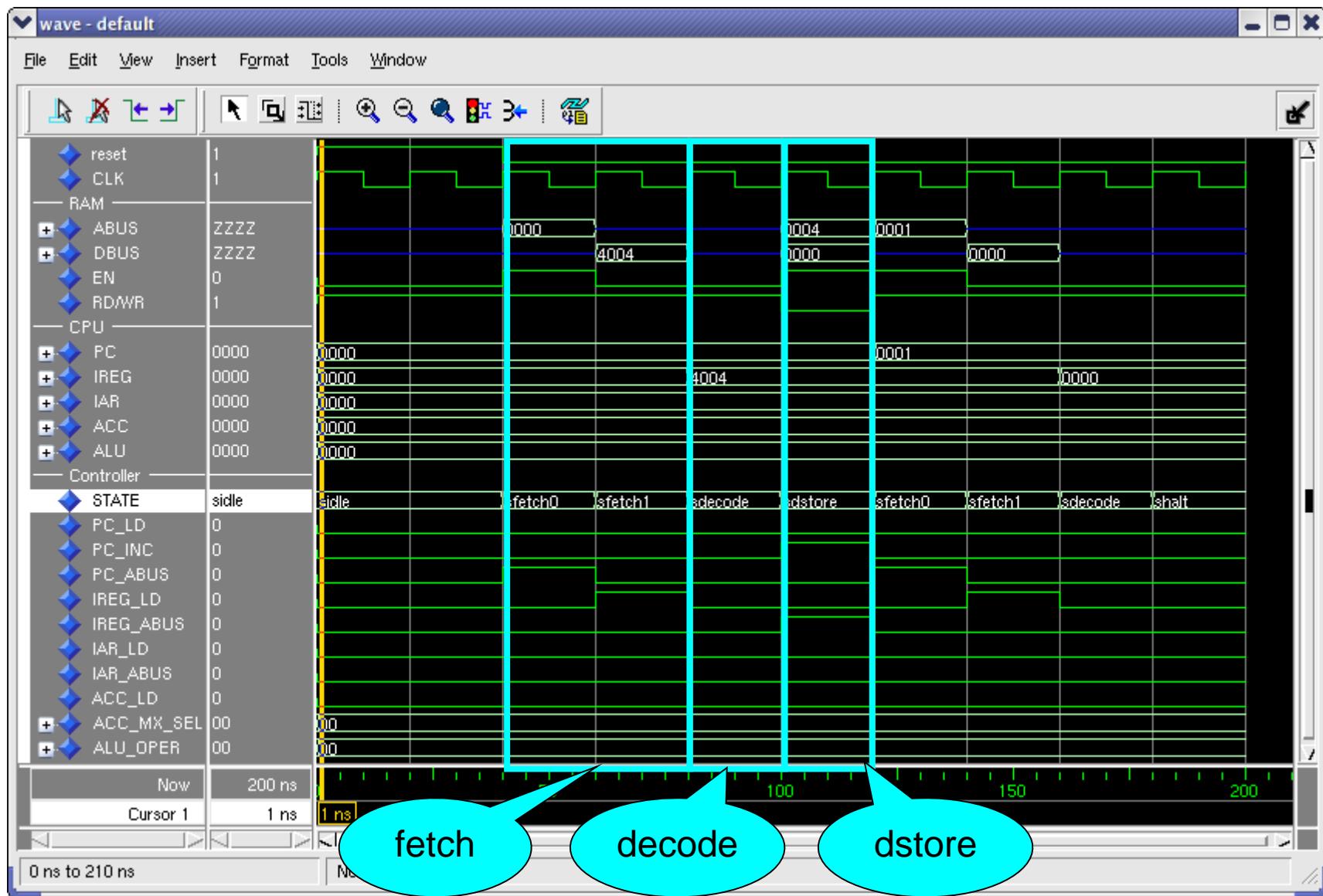
Příklad: ILOAD acc, *m[05]



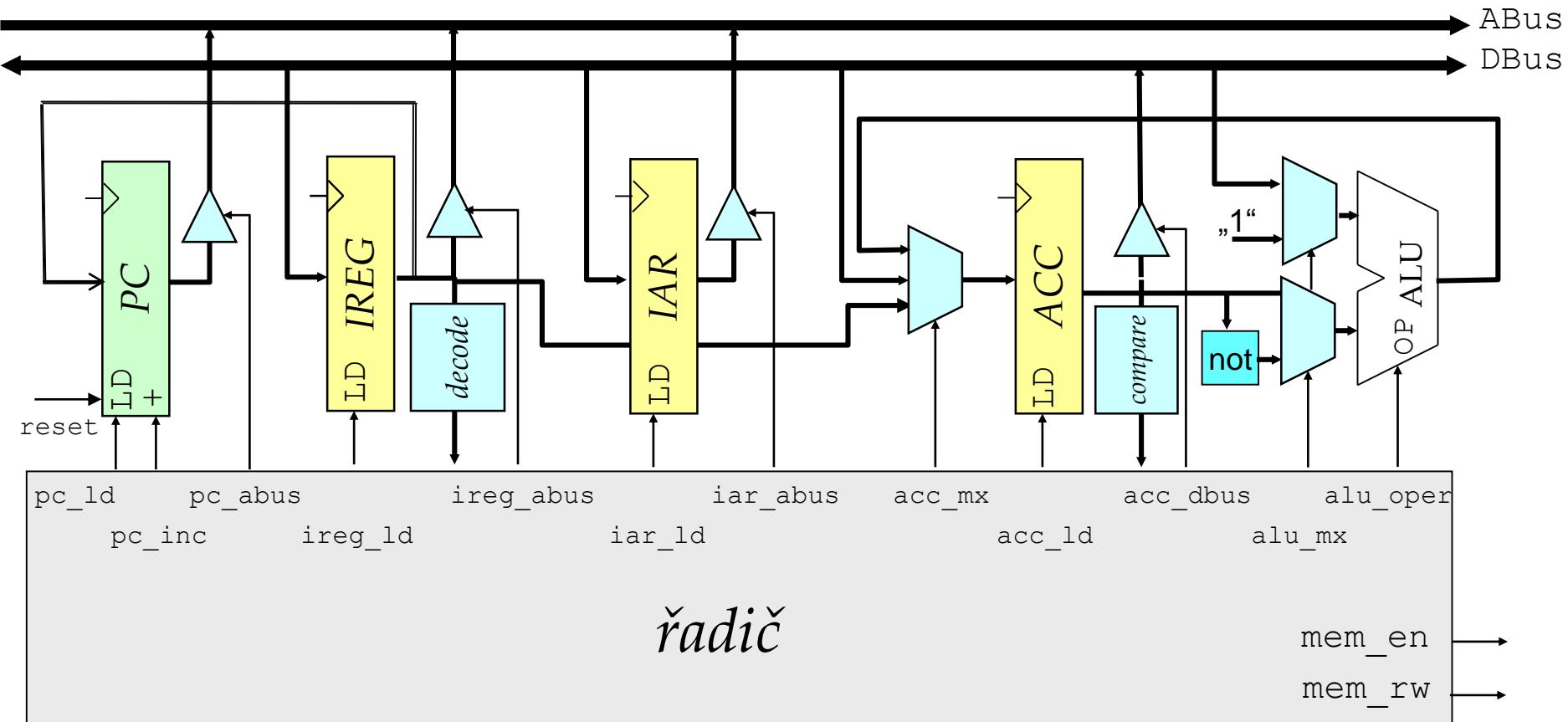
Příklad: DSTORE m[04], acc



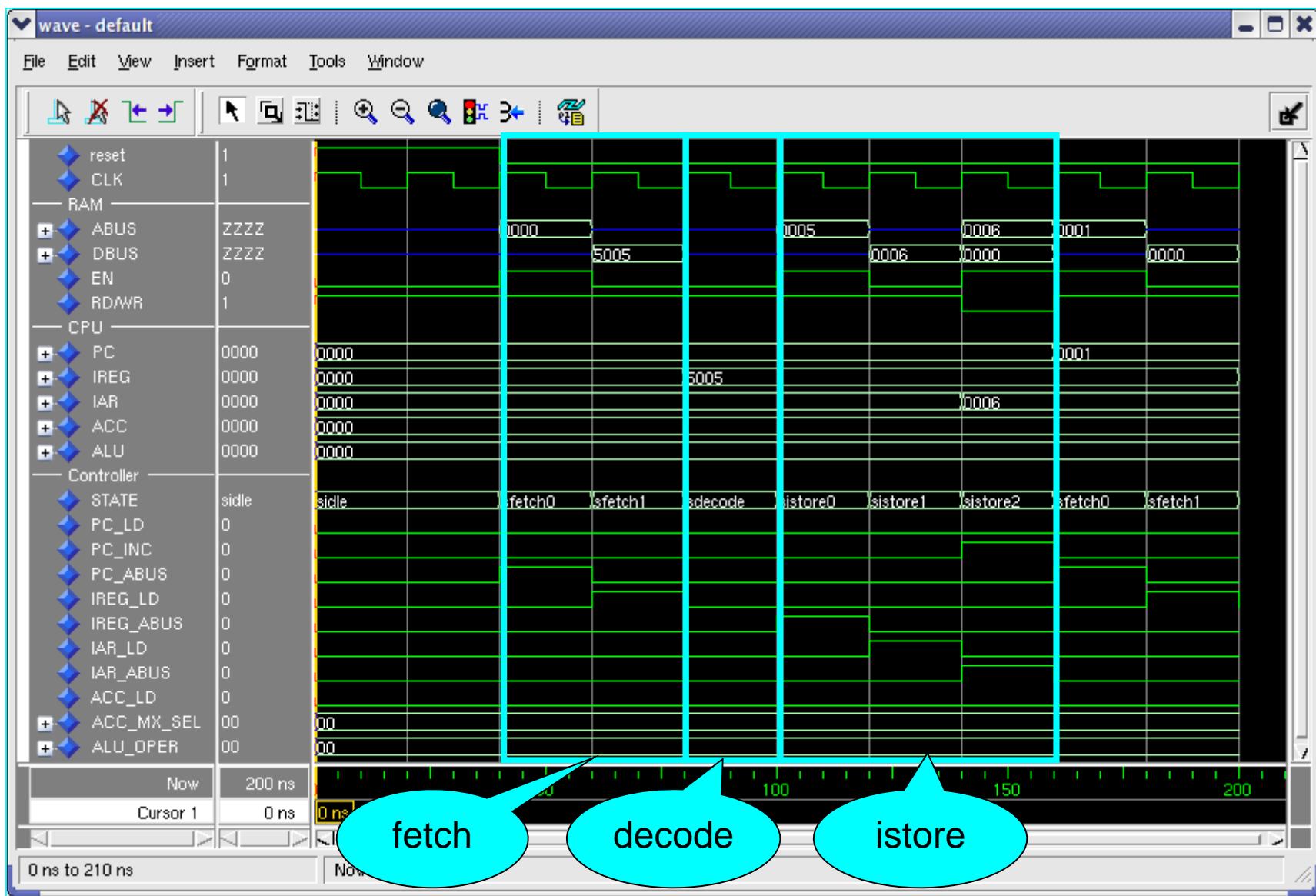
Příklad: DSTORE m[04], acc



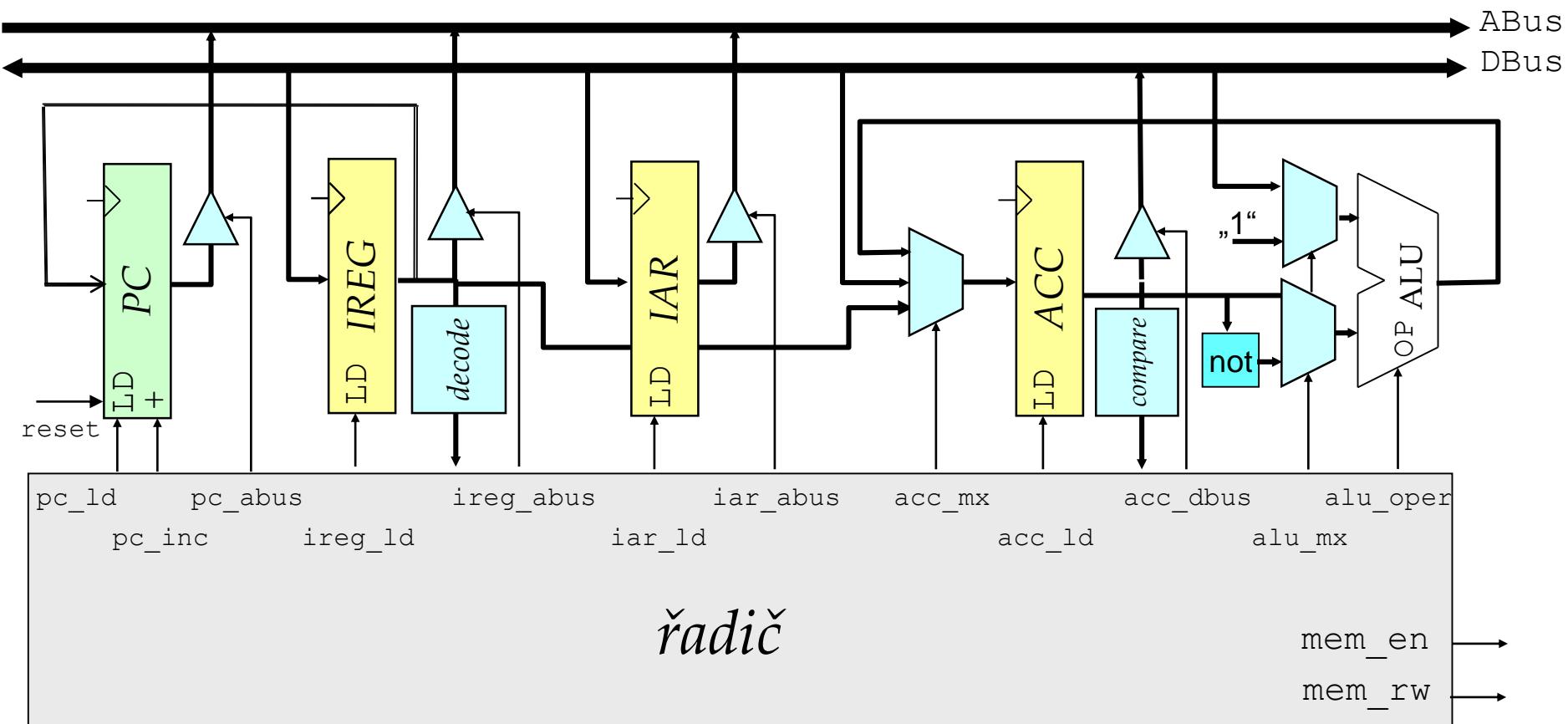
Příklad: ISTORE *m[05], acc



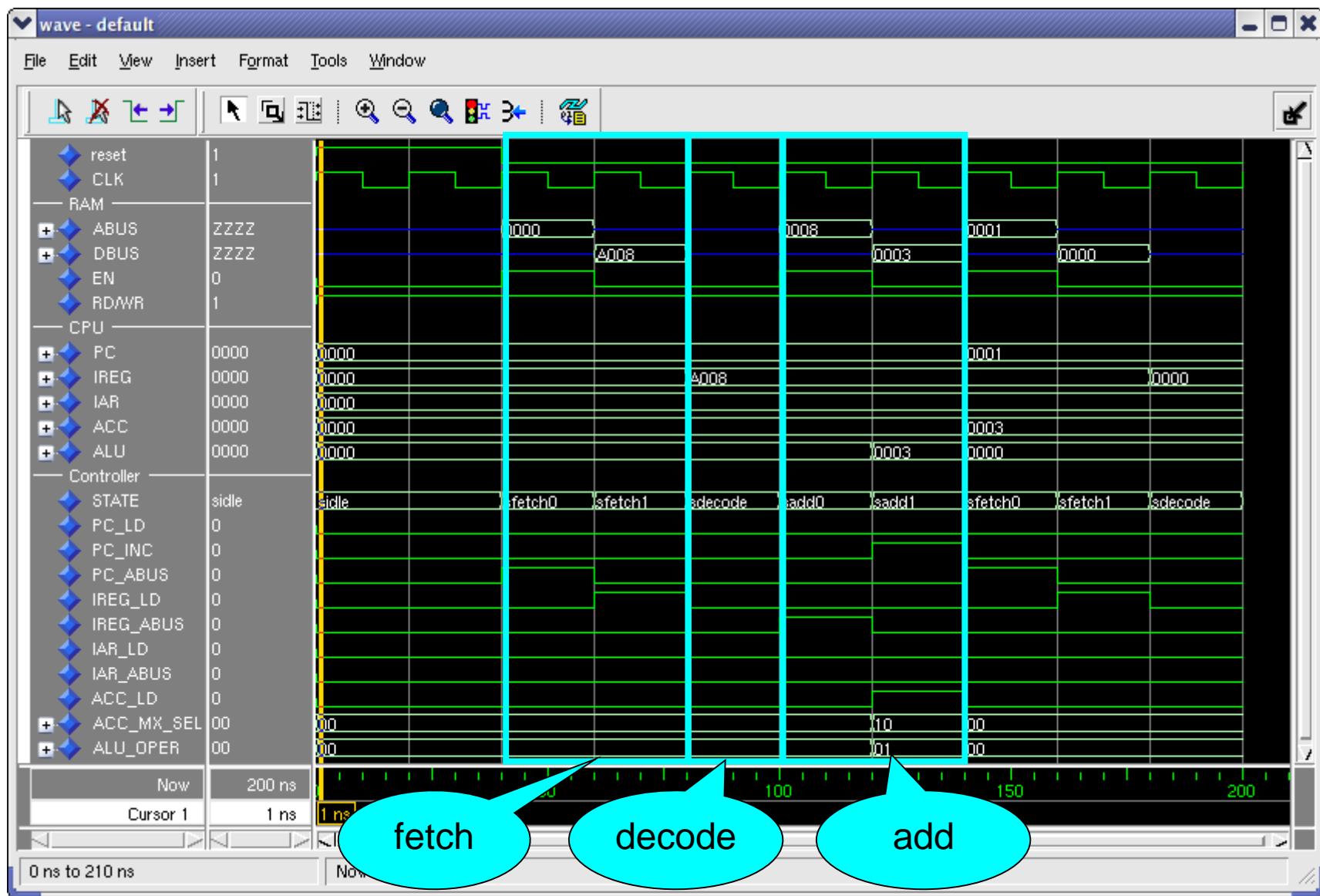
Příklad: ISTORE *m[05], acc



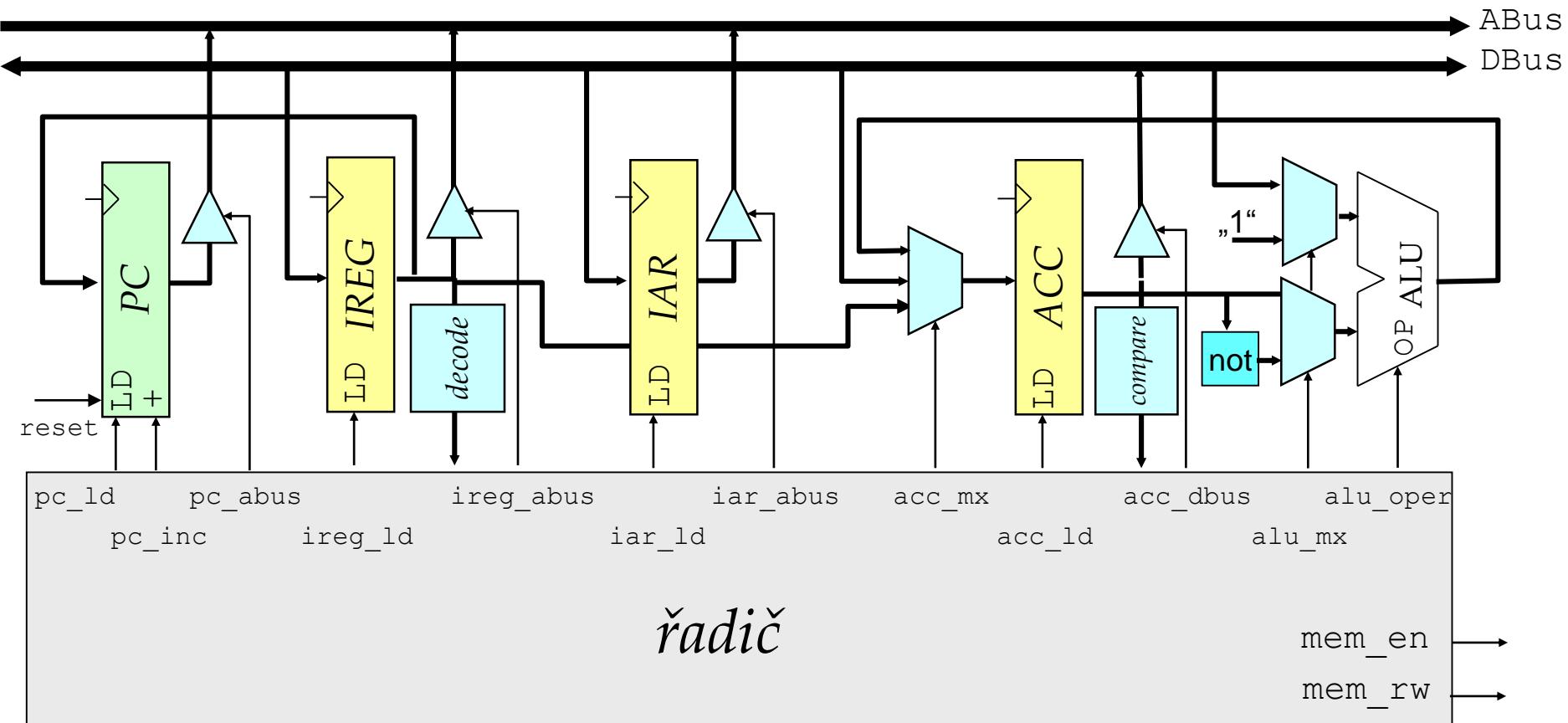
Příklad: ADD acc, m[08]



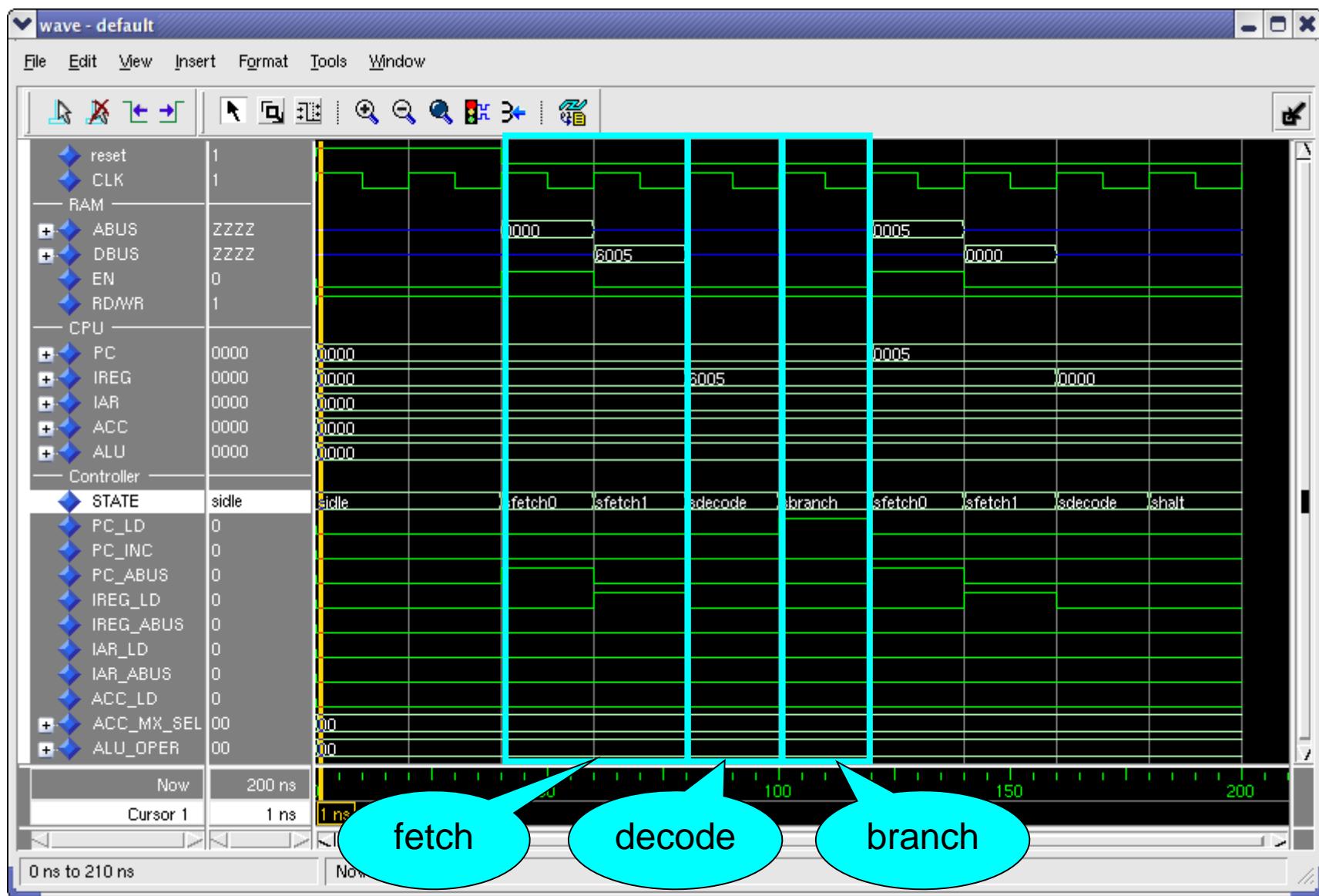
Příklad: ADD acc, m[08]



Příklad: BRANCH 05



Příklad: BRANCH 05

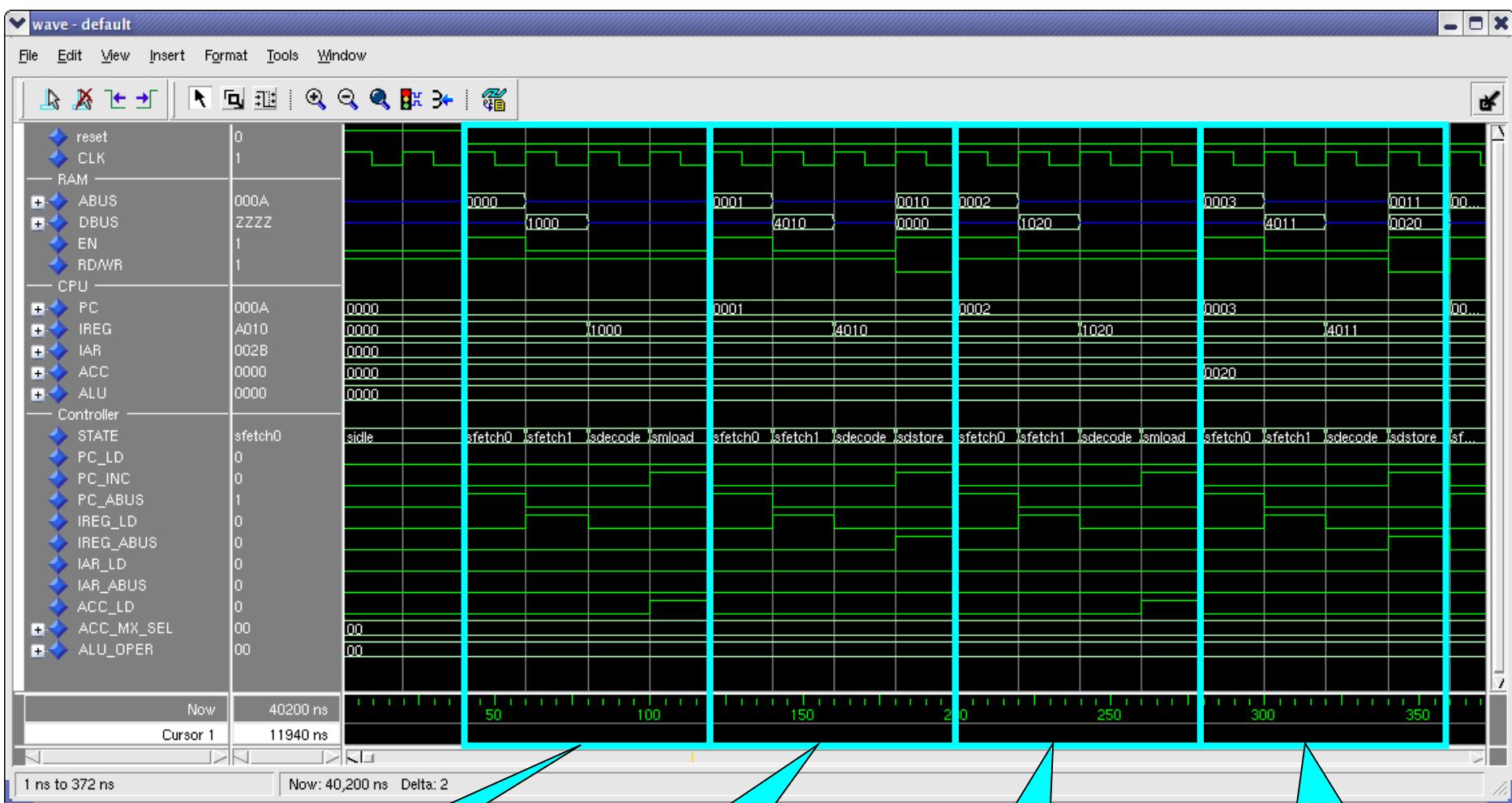


Příklad programu

Sečti hodnoty uložené na adresách 20-2f a zapiš výsledek na adresu 10.

<u>Adresa</u>	<u>Instrukce</u>	<u>Komentář</u>
0000 (start)	1000 mload 0000	ACC <= 0
0001	4010 dstore M[0010]	M[0010] <= ACC
0002	1020 mload 0020	ACC <= 0020
0003	4011 dstore M[0011]	M[0011] <= ACC
0004 (loop)	1030 mload 0030	ACC <= 0030
0005	0001 negate	ACC <= -ACC
0006	a011 add M[0011]	ACC <= ACC + M[0011]
0007	700f if 0 branch 000f	goto end if zero
0008	3011 iload *M[0011]	ACC <= *M[0011]
0009	a010 add M[0010]	ACC <= ACC + M[0010]
000a	4010 dstore M[0010]	M[0010] <= ACC
000b	1001 mload 0001	ACC <= 1
000c	a011 add M[0011]	ACC <= M[0011]+ACC
000d	4011 dstore M[0011]	M[0011] <= ACC
000e	6004 branch 0004	goto loop
000f (end)	0000 halt	halt
0010		Store sum here
0011		Pointer to "next" value

Simulace programu



mload 0000

dstore M[0010]

mload 0020

dstore M[0011]

Kódy

INP - cvičení 4 / 5

Zdeněk Vašíček, 2015
vasicek@fit.vutbr.cz

Obsah

- Huffmanovo kódování
 - bezetrátový kód pro odstranění redundance
- Hammingův kód
 - kód pro zabezpečení dat proti chybě
- Kód zbytkových tříd
 - kód pro efektivní realizaci aritmetických operací

Základní pojmy

- zdrojová abeceda
 - množina symbolů, které se mohou vyskytovat ve zprávě (abeceda zpráv)
- kódová abeceda
 - množina symbolů, které se mohou vyskytovat v zakódované zprávě (abeceda kódových slov)
- kódovací předpis
 - návod, jak transformovat slova ze zdrojové do kódové abecedy a naopak (nutné respektovat počet znaků, po kterých je zdrojová zpráva zpracovávána)
- kodové slovo versus nekodové slovo
 - taková kombinace znaků kódové abecedy, která se vyskytuje/nevyskytuje v zakódované zprávě
- Hammingova vzdálenost
 - nejmenší počet znaků kódové abecedy v nichž se dvojice kódových slov liší (počítáno přes všechna kódová slova)
 - užitečná metrika pro zabezpečovací kódy (2 – SED, 3 – SEC, 4 – SEC, DED, 5 – DEC)
- Míry kódů (redundance, teoretická optimální délka kódu, ...)

Příklad kódů a jejich vlastnosti

Příklad:

- zdrojová abeceda {0,1}
- kódová abeceda {L,H}
- kódovací předpis:
 $0 \rightarrow LH, 1 \rightarrow HL$
- počet znaků po kterých se zpracovává zpráva: []
- kódová slova: []
- nekódová slova: []
- Hammingova vzdálenost:
=> SED - kód schopný detekovat chybu v jednom znaku
- Použití: přenos dat v Ethernetu
(Manchester kódování)

Příklad:

- zdrojová abeceda {0,1}
- kódová abeceda {L,H}
- kódovací předpis:
 $0 \rightarrow LLL, 1 \rightarrow HHH$
- počet znaků po kterých se čte zpráva: []
- kódová slova: []
- nekódová slova: []
- Hammingova vzdálenost: []
=> SEC – kód schopný detekovat a opravit chybu v jednom znaku
- Použití: zabezpečení přenosu proti chybě (neefektivní – redundance [] %)

Zabezpečení pomocí parity

Příklad:

- zdrojová abeceda {0,1}
- kódová abeceda {0,1}
- kódovací předpis:

0000 → 00000, 0001 → 00101, 0010 → 00110, 0011 → 00011,
0100 → 01100, 0101 → 01001, 0110 → 01010, 0111 → 01111,
1000 → 10100, 1001 → 10001, 1010 → 10010, 1011 → 10111,
1100 → 11000, 1101 → 11101, 1110 → 11110, 1111 → 11011,

- počet znaků po kterých se zpráva zpracovává:
- varianta kódu: sudá/lichá parita
- kódová slova:
- nekódová slova:
- Hammingova vzdálenost:
- zabezpečovací schopnosti:

Q: Jak se zabezpečí zpráva, která má délku 10 bitů?

Huffmanovo kódování

- nejefektivnější prefixový kód pro odstranění redundance
- pro sestrojení kódovacího předpisu je zapotřebí znát četnosti jednotlivých kódovaných symbolů
- použití:
 - kódování instrukcí,
 - součást mnoha technik pro kompresi obrazu, zvuku a dokumentů (MPG, JPEG, MP3, ZIP, 7z, ...)

Huffmanovo kódování - příklad

- kódování po blocích velikosti $M=128$ B, zdrojová abeceda {A...J}, kódovaná zpráva

```
AECACCAICCHDCACAHDCAFCCCHBGCGECIHFCHCHBCADCJIHC  
HCDHCGBDCCGHCGJCAHHJCHCEEECIHCEBCHHCCCACDDCHGHHA  
HHCHCGDHIEDCCCDCHHCHAHHICAHHHEHA
```

- Analýzou zprávy získáme četnost výskytu jednotlivých symbolů zprávy.

symbol	počet výskytů (m_i)	četnost výskytu (f_i)
A	14	0.109 (7/64)
B	4	0.031 (1/32)
C	42	0.328 (21/64)
D	10	0.078 (5/64)
E	8	0.063 (1/16)
F	2	0.016 (1/64)
G	7	0.055 (7/128)
H	32	0.250 (1/4)
I	6	0.047 (3/64)
J	3	0.023 (3/128)

Konstrukce kódových slov

1. Jednotlivé symboly prohlásíme za uzly (listy) stromu. Postupně po dvojcích spojujeme uzly s nejmenším ohodnocením až zkonstruujeme Huffmanův strom.
2. **Systemicky** ohodnotíme hrany stromu (např. hrana vedoucí do uzlu s menším ohodnocením 0, jinak 1).
3. Cesta z vrcholu až k listu tvoří kódové slovo symbolu odpovídajícího danému listu.

Huffmanovo kódování - příklad

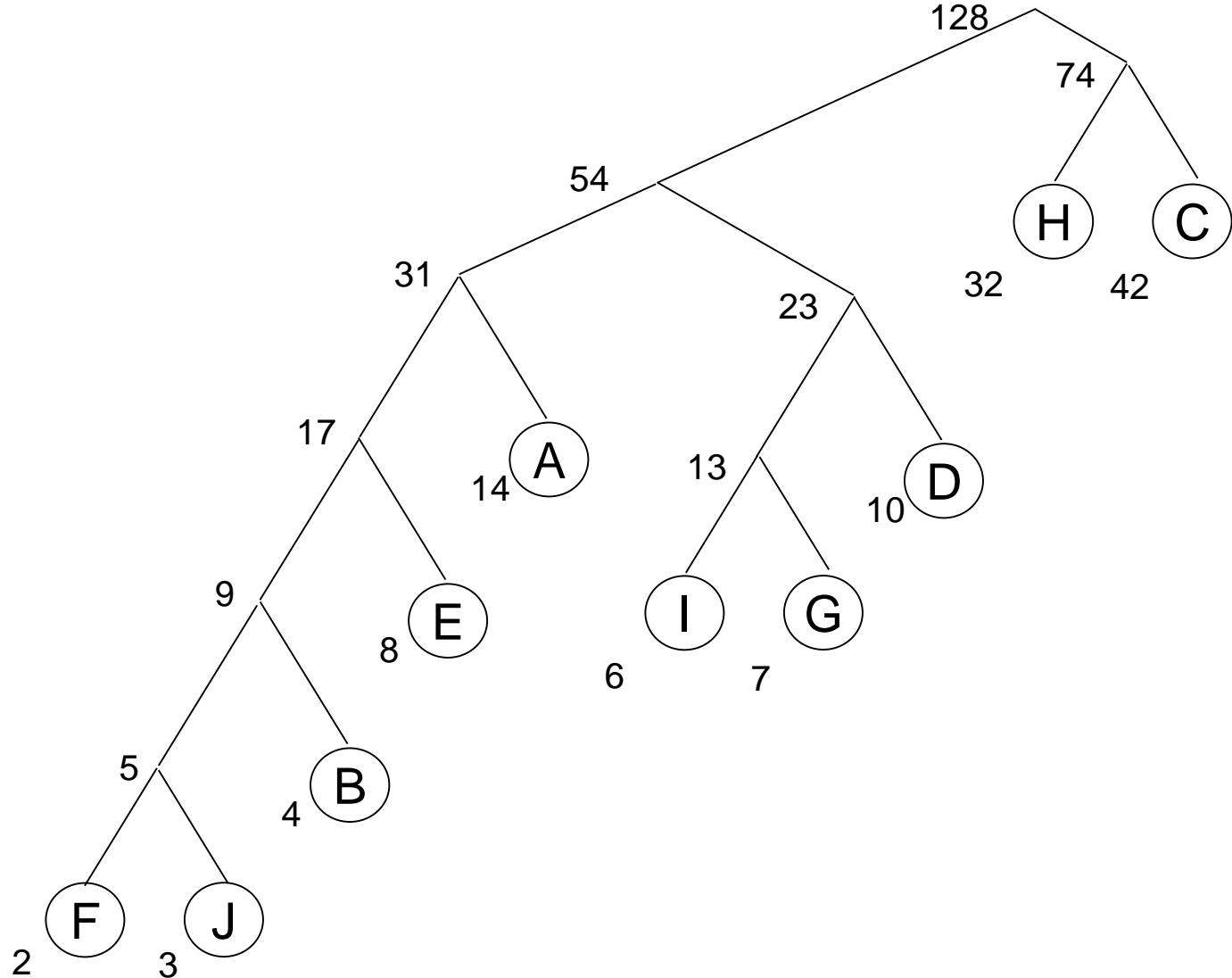
Q: Sestrojte Huffmanův kód na základě znalosti počtu výskytů jednotlivých symbolů

symbol	počet výskytů
C	42
H	32
A	14
D	10
E	8
G	7
I	6
B	4
J	3
F	2

Huffmanovo kódování - příklad

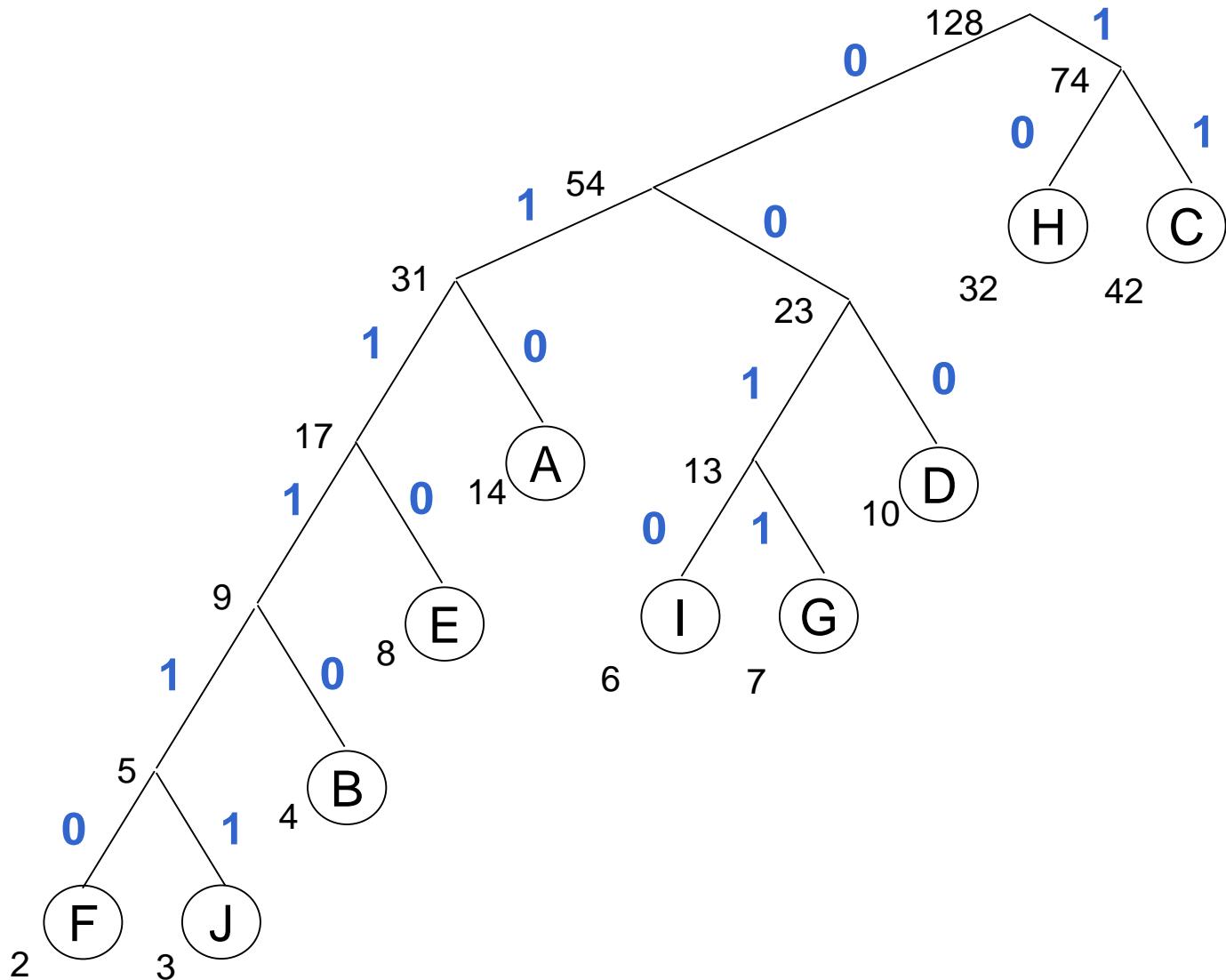
Q: Sestrojte Huffmanův kód na základě znalosti počtu výskytů jednotlivých symbolů

symbol	počet výskytů
C	42
H	32
A	14
D	10
E	8
G	7
I	6
B	4
J	3
F	2



Huffmanovo kódování - příklad

Q: Sestrojte Huffmanův kód na základě znalosti počtu výskytů jednotlivých symbolů



Huffmanovo kódování - příklad

Tabulka symbolů a odpovídajících kódových slov (častěji se vyskytující symboly mají přiřazeno kratší kódové slovo)

symbol	počet výskytů (m_i)	kódové slovo	L_i
A	14	010	3
B	4	01110	5
C	42	11	2
D	10	000	3
E	8	0110	4
F	2	011110	6
G	7	0011	4
H	32	10	2
I	6	0010	4
J	3	011111	6

$$L_{dyn} = \sum_{i=1}^N L_i f_i = \frac{1}{M} \sum_{i=1}^N L_i m_i$$

$$f_i = \frac{m_i}{M}$$

$$L_{opt} = - \sum_{i=1}^N f_i \log_2 f_i =$$

$$= \log_2 M - \frac{1}{M} \sum_{i=1}^N m_i \log_2 m_i$$

$$R = \frac{L_{dyn} - L_{opt}}{L_{dyn}}$$

M=128 (délka původní zprávy)
N=10 (počet kódových slov)

Q: Jak vypadá prvních 16 bitů zakódované zprávy?

Q: Dekódujte zprávu 01110010011110

Q: Jaká je střední dyn. délka kódu, teor. optimální délka kódu, redundance kódu?

Hammingův kód (HK)

- nejfektivnější kód (minimální možná redundance) pro zabezpečení dat proti výskytu chyby v jednom symbolu s možností chybu opravit (SEC kód)
- kódové slovo obsahuje kontrolní bity, jejichž umístění je dáno pozicí bitu ve slově
 - kontrolní bity jsou tvořeny pomocí funkce XOR z informačních bitů (vhodně zvolené pokrytí informačních bitů sadou parit)
 - je-li index pozice bitu ve slově mocnina 2, jedná se o bit kontrolní (C); v opačném případě se jedná o bit informační (I)
- Hammingův kód je separovatelný a má Hammingovu vzdálenost 3

Paritní bit

- Paritní bit je typ kódu, který přidává redundantní bit k datovému slovu obsahující informaci o počtu jedničkových bitů ve slově.
- Paritní bit je určen k jednoduché detekci chyby ve slově (speciální případ 1-bitového CRC, polynom $x+1$).
- Rozlišujeme dva typy: lichá a sudá parita

8bitová data	kódové slovo s paritním bitem	
	sudá parita	lichá parita
00000000	000000000	100000000
10010001	110010001	010010001
11001001	011001001	111001001
11111111	011111111	111111111

- Výpočet: funkce XOR

Hammingův kód (HK)

- před použitím je nutné definovat počet bitů kódového slova
 - $HK(n,k)$ nebo $Hamming(n,k)$ je běžně používané označení, kde
 - n je délka kódového slova (počet bitů),
 - k je počet informačních bitů a
 - platí, že $k = n - m$, kde m je počet kontrolních bitů a $n = 2^m - 1$
 - příklad používaných kódů
 - $HK(7,4)$ – 3 kontrolní byty (právě 3 byty jsou zapotřebí k zakódování 8 kombinací)
 - $HK(15,11)$ – 4 kontrolní byty (právě 4 byty jsou zapotřebí k zakódování 16 kombinací)
 - $HK(127,120)$ – 7 kontrolních bitů
 - čím větší n , tím je redundance kódu menší (poměr k/n příznivější)

Hammingův kód – sestavení kódu HK(7,4)

- je-li index pozice bitu mocnina 2, jedná se o bit kontrolní, jinak datový (informační)

pozice bitu	1	2	3	4	5	6	7
význam bitu	C1	C2	I3	C4	I5	I6	I7

- pokrytí kontrolními (paritními) bity

	C1	C2	I3	C4	I5	I6	I7
C1	X		X		X		X
C2		X	X			X	X
C4				X	X	X	X

- Generující rovnice (pro výpočet kontrolních bitů):

$$C_1 = I_3 \text{ xor } I_5 \text{ xor } I_7$$

$$C_2 = I_3 \text{ xor } I_6 \text{ xor } I_7$$

$$C_4 = I_5 \text{ xor } I_6 \text{ xor } I_7$$

Hammingův kód - příklad

Zakódujte v HK(7,4) slovo 1101 (MSB vlevo)

pozice bitu	7	6	5	4	3	2	1
význam bitu	I7	I6	I5	C4	I3	C2	C1

pozice bitu	7	6	5	4	3	2	1
hodnota bitu	1	1	0	C4	1	C2	C1

pozice bitu	7	6	5	4	3	2	1
hodnota bitu	1	1	0	0	1	1	0

$$C1 = I3 \text{ xor } I5 \text{ xor } I7$$

$$C2 = I3 \text{ xor } I6 \text{ xor } I7$$

$$C4 = I5 \text{ xor } I6 \text{ xor } I7$$

Pozn: operace xor v tomto případě odpovídá sčítání modulo 2

Hammingův kód – detekce a oprava chyby

- Chybu v jednom bitu lze detektovat na základě hodnoty tzv. chybového syndromu S , který určuje pozici chyby ve slově.

$$S_1 = C'_1 \text{ xor } I'_3 \text{ xor } I'_5 \text{ xor } I'_7$$

$$S_2 = C'_2 \text{ xor } I'_3 \text{ xor } I'_6 \text{ xor } I'_7$$

$$S_4 = C'_4 \text{ xor } I'_5 \text{ xor } I'_6 \text{ xor } I'_7$$

- Jednotlivé bity syndromu testují, zda-li odpovídající paritní bity jsou správné. Pokud nikoliv, tak vzhledem k rozložení paritních bitů získáme pozici chyby v přijatém slově.

Hammingův kód - příklad

Dekódujte v HK(7,4) přijaté slovo 1001100 a 1001000 (MSB vlevo)

pozice bitu	7	6	5	4	3	2	1
hodnota bitu	1	0	0	1	1	0	0

$S_1 = 0$ $S_2 = 0$ $S_4 = 0$ $S = 000$ přenos proběhl bez chyby nebo nastal větší počet chyb

pozice bitu	7	6	5	4	3	2	1
hodnota bitu	1	0	0	1	1	0	0

$S_1 = 1$ $S_2 = 1$ $S_4 = 0$ $S = 011$ bit na pozici 3 má chybou hodnotu, opravíme na opačnou

$$S_1 = C'_1 \text{ xor } l'_3 \text{ xor } l'_5 \text{ xor } l'_7 \quad S_2 = C'_2 \text{ xor } l'_3 \text{ xor } l'_6 \text{ xor } l'_7 \quad S_4 = C'_4 \text{ xor } l'_5 \text{ xor } l'_6 \text{ xor } l'_7$$

Rozšířený Hammingův kód

- Přidáním paritního bitu přes všechny byty $HK(n,k)$ lze rozšířit Hammingův kód na kód typu SEC-DED, který je schopen detekovat dvojchybu ($d=4$), označovaný jako $HK(n+1,k)$
- Příklad pro $HK(8,4)$:
 - přidáme kontrolní paritní bit C_0 :
$$C_0 = C_1 \text{ xor } C_2 \text{ xor } I_3 \text{ xor } C_4 \text{ xor } I_5 \text{ xor } I_6 \text{ xor } I_7$$
 - obdobně rozšíříme syndrom chyby:
$$S_0 = C'_0 \text{ xor } C'_1 \text{ xor } C'_2 \text{ xor } I'_3 \text{ xor } C'_4 \text{ xor } I'_5 \text{ xor } I'_6 \text{ xor } I'_7$$
 - rozšíříme syndrom o příznak R
$$R = S_1 \text{ or } S_2 \text{ or } S_4$$
 - Chyby klasifikujeme následovně

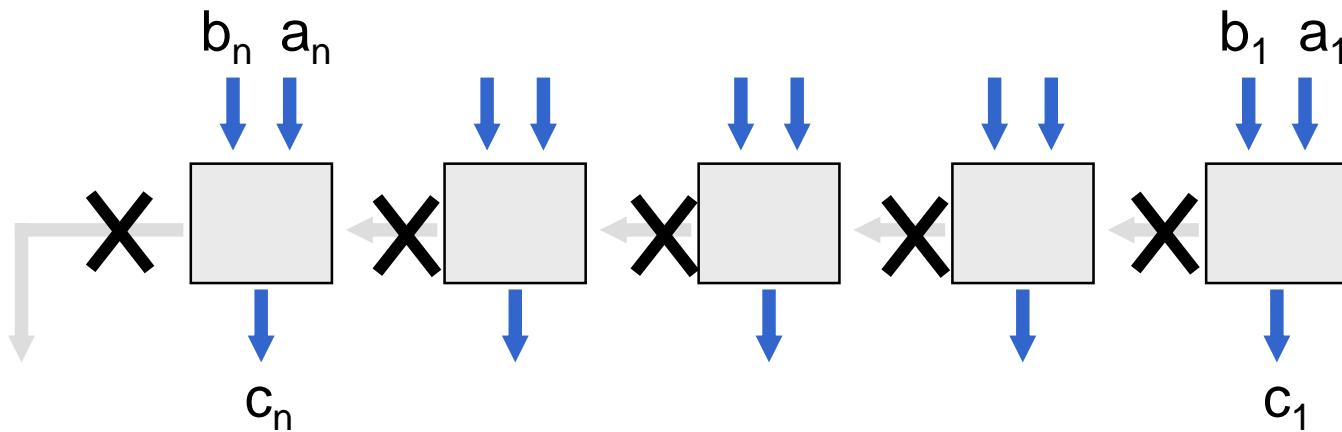
R	S_0	Význam
0	0	Bez chyby
0	1	Neopravitelná chyba (porucha hlídce, vícenásobná chyba)
1	0	Neopravitelná 2-chyba, 4-chyba, atd.
1	1	Opravitelná 1-chyba

Hammingův kód

- Q: Dekódujte v HK(7,4) přijaté slovo 1001010 obsahující dvojchybu (původní hodnota byla 1001100).
- Q: Dekódujte v HK(15,11) slovo 110111010101110
- Q: Co se stane nastane-li v HK(7,4) dvojchyba a co v HK(15,11)?
- Q: Kolik kontrolních bitů je zapotřebí na vytvoření HK má-li informační slovo délku 31 bitů?
- Q: Zakódujte slovo 1010 v rozšířeném HK(8,4).
- Q: Dekódujte slovo 10101010 v rozšířeném HK(8,4).

Kód zbytkových tříd (KZT, RNS)

- Kód zbytkových tříd je systém reprezentace dat, pomocí kterého je možné realizovat sčítání, odčítání a násobení bez přenosů, což má významný dopad na výkonnost zejména v případě mnohabitových operandů (např. v kryptografii RSA algoritmus používá 2048bitů)
- Princip:



$$c_i = (a_i \text{ op } b_i) \bmod m_i$$

- Známé problémy: obecné dělení, test na přetečení, porovnání, test na znaménko jsou obtížně realizovatelné operace

Volba modulů RNS

- Nejprve je nutné vhodně zvolit jednotlivé moduly dle požadovaného dynamického rozsahu
- Dynamický rozsah je roven **nejmenšímu společnému násobku (NSD) modulů.**
- Příklady:

Moduly	Moduly rozložené na prvočísla	Dynamický rozsah (NSD)	
(3,6)	(3, $2 \cdot 3$)	$2 \cdot 3 = 6$	
(6)	($2 \cdot 3$)	$2 \cdot 3 = 6$	} totožný rozsah
(3,7,9)	(3, 7, $3 \cdot 3$)	$3 \cdot 3 \cdot 7 = 63$	
(7, 9)	(7, $3 \cdot 3$)	$3 \cdot 3 \cdot 7 = 63$	} totožný rozsah

- Pro efektivní využití reprezentace je vhodné mít největší společný dělitel všech modulů (po dvojcích) roven jedné (tzn. navzájem nesoudělná čísla, ideálně **prvočísla**).

Převod do RNS

- Máme-li definované vhodné moduly $\{m_1, \dots, m_N\}$, pak číslo X v binárním tvaru lze v RNS reprezentovat jako $\{x_1, \dots, x_N\}$, kde $x_i = X \bmod m_i$
- Příklady:

$$X = 7 \rightarrow \langle 1 | 1 | 2 \rangle_{RNS \ (2|3|5)}$$

$$\begin{array}{l} 2 \cdot 3 + 1 = 7 \quad 7 \bmod 2 = 1 \\ 3 \cdot 2 + 1 = 7 \quad 7 \bmod 3 = 1 \\ 5 \cdot 1 + 2 = 7 \quad 7 \bmod 5 = 2 \end{array}$$

$$X = 70 \rightarrow \langle 1 | 0 | 0 \rangle_{RNS \ (3|5|7)}$$

$$\begin{array}{l} 3 \cdot 23 + 1 = 70 \quad 70 \bmod 3 = 1 \\ 5 \cdot 14 + 0 = 70 \quad 70 \bmod 5 = 0 \\ 7 \cdot 10 + 0 = 70 \quad 70 \bmod 7 = 0 \end{array}$$

Převod z RNS zpět

- Zpětný převod je výpočetně mnohem náročnější. V praxi se využívá buď výpočet založený na algoritmu CRT (Chinese Remainder Theorem) nebo MRS (Mixed Radix System).
- Máme-li číslo $\{x_1, \dots, x_N\}$, moduly $\{m_1, \dots, m_N\}$ a $M = m_1 m_2 \dots m_N$, pak X získáme jako

$$X = \sum_{i=1}^N x_i M_i N_i \bmod M = (x_1 M_1 N_1 + \dots + x_N M_N N_N) \bmod M$$

kde M_i odpovídá součinu hodnot všech modulů kromě i -tého a $N_i = M_i^{-1} \bmod m_i$ je **multiplikativní inverze** M_i v Z_{m_i}

- Multiplikativní inverze x na tělese Z_{m_i} je takové číslo x^{-1} , pro které platí, že $x \cdot x^{-1} \equiv 1$

Příklady:

V případě, že $M_i = 15$ a $m_i = 2$ musí být splněno, že $15 * N_i \bmod 2 = 1$, což splňuje $N_i = 1$

V případě, že $M_i = 35$ a $m_i = 3$ musí být splněno, že $35 * N_i \bmod 3 = 1$, což splňuje $N_i = 2$

Převod z RNS zpět

- Zpětný převod je výpočetně mnohem náročnější. V praxi se využívá buď výpočet založený na algoritmu CRT (Chinese Remainder Theorem) nebo MRS (Mixed Radix System).
- Máme-li číslo $\{x_1, \dots, x_N\}$, moduly $\{m_1, \dots, m_N\}$ a $M = m_1 m_2 \dots m_N$, pak X získáme jako

$$X = \sum_{i=1}^N x_i M_i N_i \bmod M = (x_1 M_1 N_1 + \dots + x_N M_N N_N) \bmod M$$

kde M_i odpovídá součinu hodnot všech modulů kromě i -tého a
 $N_i = M_i^{-1} \bmod m_i$ je **množstevní inverz** M_i v Z_{m_i}

- Příklady:

$$(1 | 1 | 2)_{RNS \ (2|3|5)} \rightarrow X = (1 \cdot (3 \cdot 5) \cdot 1 + 1 \cdot (2 \cdot 5) \cdot 1 + 2 \cdot (2 \cdot 3) \cdot 1) \bmod (2 \cdot 3 \cdot 5) = 7$$

$$(1 | 0 | 0)_{RNS \ (3|5|7)} \rightarrow X = (1 \cdot (5 \cdot 7) \cdot 2 + 0 \cdot (3 \cdot 7) \cdot 1 + 0 \cdot (3 \cdot 5) \cdot 1) \bmod (3 \cdot 5 \cdot 7) = 70$$

Zakódování čísel a aritmetické operace sčítání

No	m_1	m_2	m_3
0	2	3	5
1	1	1	1
2	0	2	2
3	1	0	3
4	0	1	4
5	1	2	0
6	0	0	1
7	1	1	2
8	0	2	3
9	1	0	4
10	0	1	0
11	1	2	1
12	0	0	2
15	1	0	0
.....			
29	1	2	4
30	0	0	0
Opakuje se			
$30 = 2 \cdot 3 \cdot 5$			

$$c_i = (a_i + b_i) \bmod m_i \quad \text{bez přenosu!}$$

$$\begin{array}{r}
 & 1 & 0 & 3 & (3) \\
 & +0 & 1 & 4 & (4) \\
 \hline
 & 1 & 1 & 2 &
 \end{array}$$

$$(1|1|2)_{RNS\ (2|3|5)} \rightarrow X = (1 \cdot 15 + 1 \cdot 10 + 2 \cdot 6) \bmod 30 = 7$$

$$\begin{array}{r}
 & 1 & 0 & 4 & (9) \\
 & +1 & 2 & 1 & (11) \\
 \hline
 & 0 & 2 & 0 &
 \end{array}$$

$$(0|2|0)_{RNS\ (2|3|5)} \rightarrow X = (0 \cdot 15 + 2 \cdot 10 + 0 \cdot 6) \bmod 30 = 20$$

Zakódování čísel a aritmetické operace odčítání

No	m_1	m_2	m_3
0	2	3	5
1	0	0	0
2	1	1	1
3	0	2	2
4	1	0	3
5	0	1	4
6	1	2	0
7	0	0	1
8	1	1	2
9	0	2	3
10	1	0	4
11	0	1	0
12	1	2	1
.....	0	0	2
27	1	0	2
28	0	1	3
29	1	2	4
30	0	0	0
Opakuje se			
$30 = 2 \cdot 3 \cdot 5$			

$$c_i = (a_i - b_i) \bmod m_i$$

$$\begin{array}{r} 0 \ 0 \ 1 \\ -1 \ 2 \ 0 \end{array} \quad (6)$$

$$\underline{1 \ 1 \ 1} \quad (5)$$

$$1 \ 1 \ 1$$

$$(1 \mid 1 \mid 1)_{RNS \ (2|3|5)} \rightarrow X = (1 \cdot 15 + 1 \cdot 10 + 1 \cdot 6) \bmod 30 = 1$$

$$\begin{array}{r} 1 \ 0 \ 3 \\ -0 \ 0 \ 1 \end{array} \quad (3)$$

$$\underline{-0 \ 0 \ 1} \quad (6)$$

$$1 \ 0 \ 2$$

V této reprezentaci může být výsledkem pouze kladné číslo! (tj $30 - 3$)

$$(1 \mid 0 \mid 2)_{RNS \ (2|3|5)} \rightarrow X = (1 \cdot 15 + 0 \cdot 10 + 2 \cdot 6) \bmod 30 = 27$$

Zakódování čísel a aritmetické operace násobení

No	m_1	m_2	m_3
0	2	3	5
1	1	1	1
2	0	2	2
3	1	0	3
4	0	1	4
5	1	2	0
6	0	0	1
7	1	1	2
8	0	2	3
9	1	0	4
10	0	1	0
11	1	2	1
12	0	0	2
13	1	1	3
14	0	2	4
.....			
29	1	2	4
30	0	0	0
Opakuje se			
30 = 2*3*5			

$$c_i = (a_i * b_i) \bmod m_i$$

$$\begin{array}{r} 0 \\ *1 \end{array} \quad \begin{array}{r} 1 \\ 2 \end{array} \quad \begin{array}{r} 4 \\ 0 \end{array} \quad (4)$$

$$\begin{array}{r} *1 \\ \hline 2 \end{array} \quad 0 \quad (5)$$

$$\begin{array}{r} 0 \\ 0 \\ 1 \end{array}$$

$$(0 | 2 | 0)_{RNS \ (2|3|5)} \rightarrow X = (0 \cdot 15 + 2 \cdot 10 + 0 \cdot 6) \bmod 30 = 20$$

$$\begin{array}{r} 1 \\ *1 \end{array} \quad \begin{array}{r} 0 \\ 0 \end{array} \quad \begin{array}{r} 3 \\ 0 \end{array} \quad (3) \quad (15)$$

Výsledkem operace je
 $(3*15) \bmod 30!$

$$\begin{array}{r} 1 \\ 0 \\ 0 \end{array}$$

$$(1 | 0 | 0)_{RNS \ (2|3|5)} \rightarrow X = (1 \cdot 15 + 0 \cdot 10 + 0 \cdot 6) \bmod 30 = 15$$

Kód zbytkových tříd

- Q: Jaký je dynamický rozsah systému RNS(7|15)?
- Q: Jaký je dynamický rozsah systému RNS(3|6)?
- Q: Převeďte číslo 120 do systému RNS(11|15).
- Q: Převeďte číslo $(1|2|3)_{RNS(3|5|7)}$ na číslo v desítkové soustavě.
- Q: Sečtěte čísla 5 a 8 v RNS(3|6) a výsledek převeďte na číslo v desítkové soustavě.
- Q: Z množiny $\{3,5,6,7,12,18\}$ zvolte tři moduly tak, aby výsledný RNS systém měl největší možný dynamický rozsah.

Aritmetické a logické operace (posuvy a rotace)

INP - cvičení 5

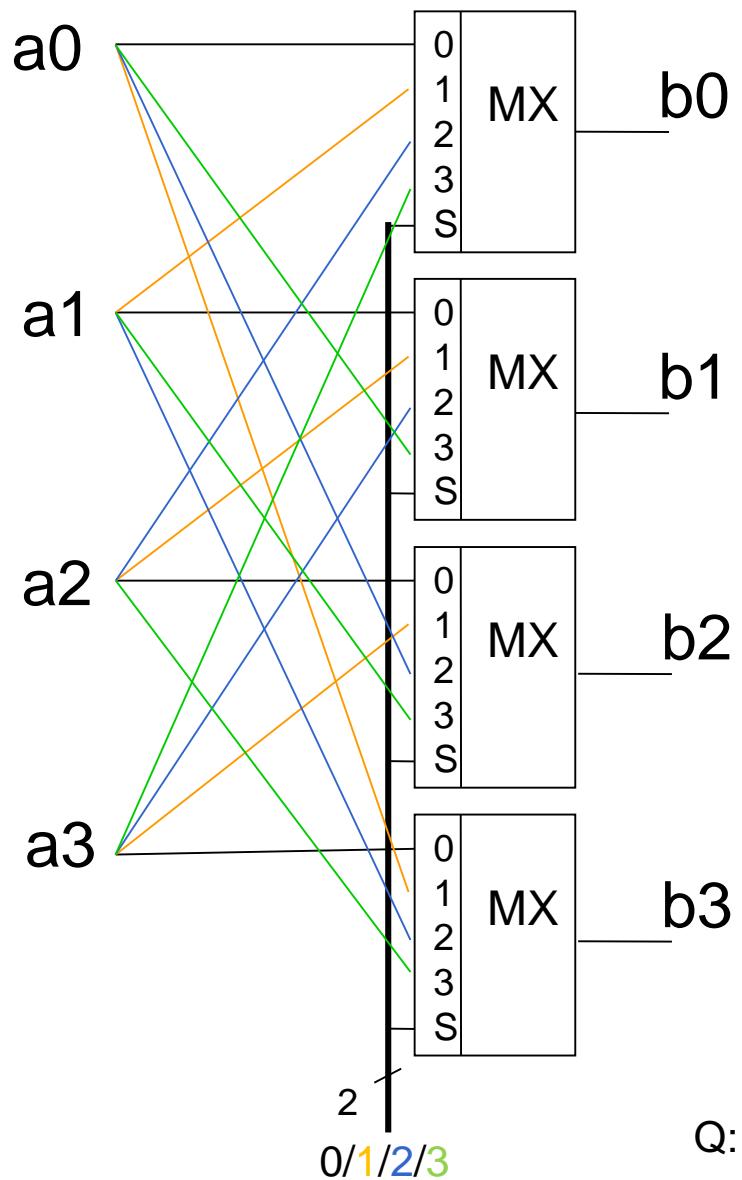
Zdeněk Vašíček, 2015
vasicek@fit.vutbr.cz

Rotace a posuvy

- Bitové **rotace a posuvy** se implementují výhradně pomocí pomocí tzv. **Barrel Shifter** obvodu, což je kombinační obvod, který je schopen vyčíslit výsledek operace během jediného taktu.
- Počet taktů při použití posuvného registru by závisel na počtu bitů o kolik posuv realizujeme.
- Barrel Shifter je obvod složený z vhodně uspořádaných multiplexorů, používá se označení $BS(n_1, n_2, \dots, n_N)$, kde N odpovídá počtu stupňů (zpoždění) a n_i šířce multiplexorů v jednotlivých stupních.
- $BS(B)$, kde B je počet bitů operandu odpovídá B multiplexorům s B-bitovým vstupem (velmi drahé).

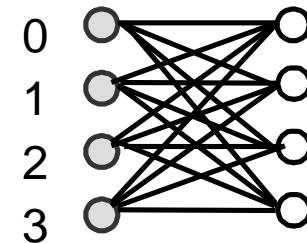
4-bitový válcový posouvač pro rotace vpravo

nejjjednodušší varianta - BS(4)



b_3	b_2	b_1	b_0
↓	↓	↓	↓
a_3	a_2	a_1	a_0
0 bit			
a_0	a_3	a_2	a_1
1 bit			
a_1	a_0	a_3	a_2
2 bit			
a_2	a_1	a_0	a_3
3 bit			

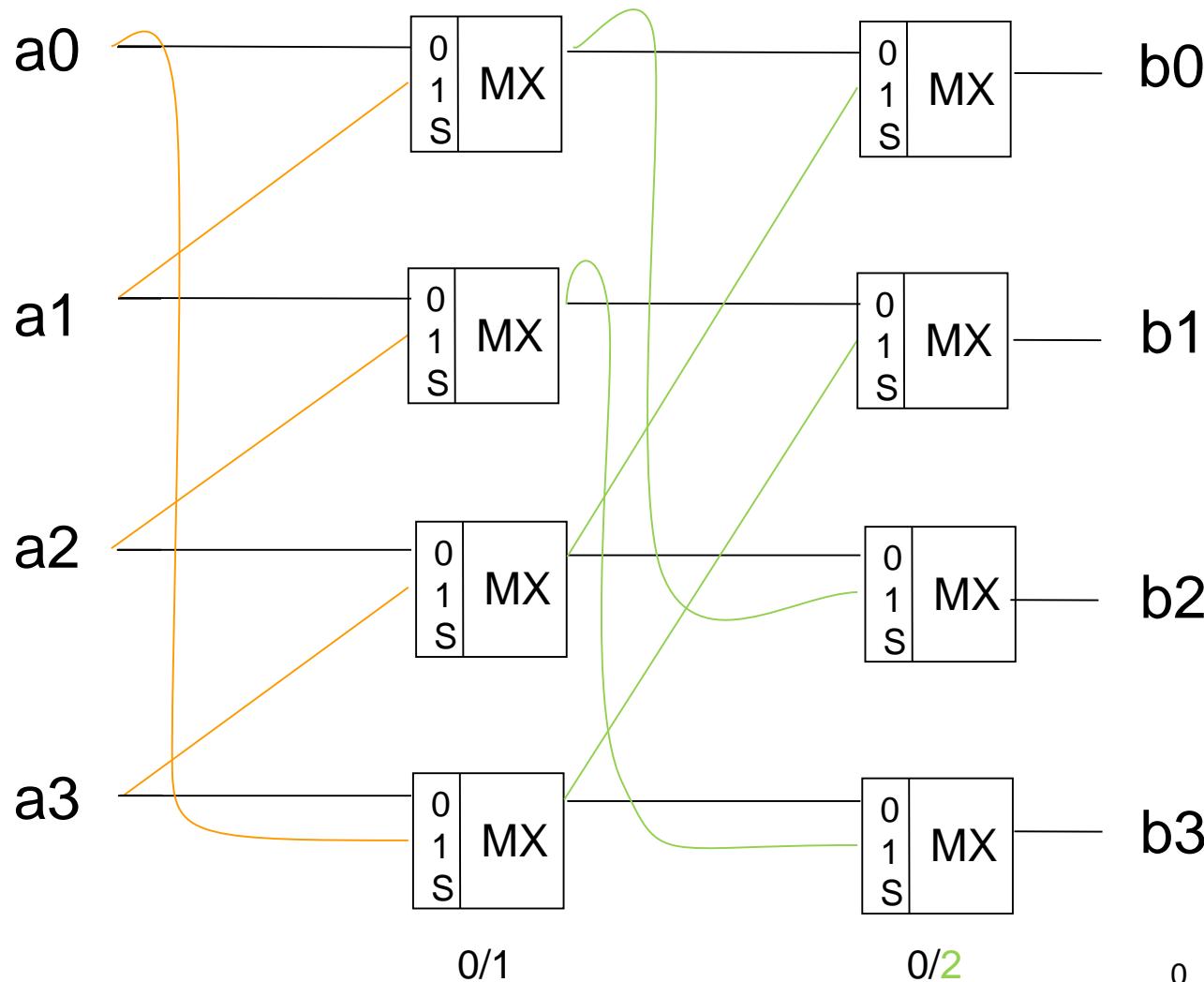
Schematické znázornění:



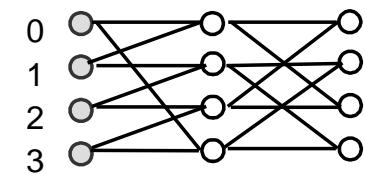
Q: Jak by se realizoval BS pro posuv nikoliv rotaci?

4-bitový válcový posouvač pro rotace vpravo

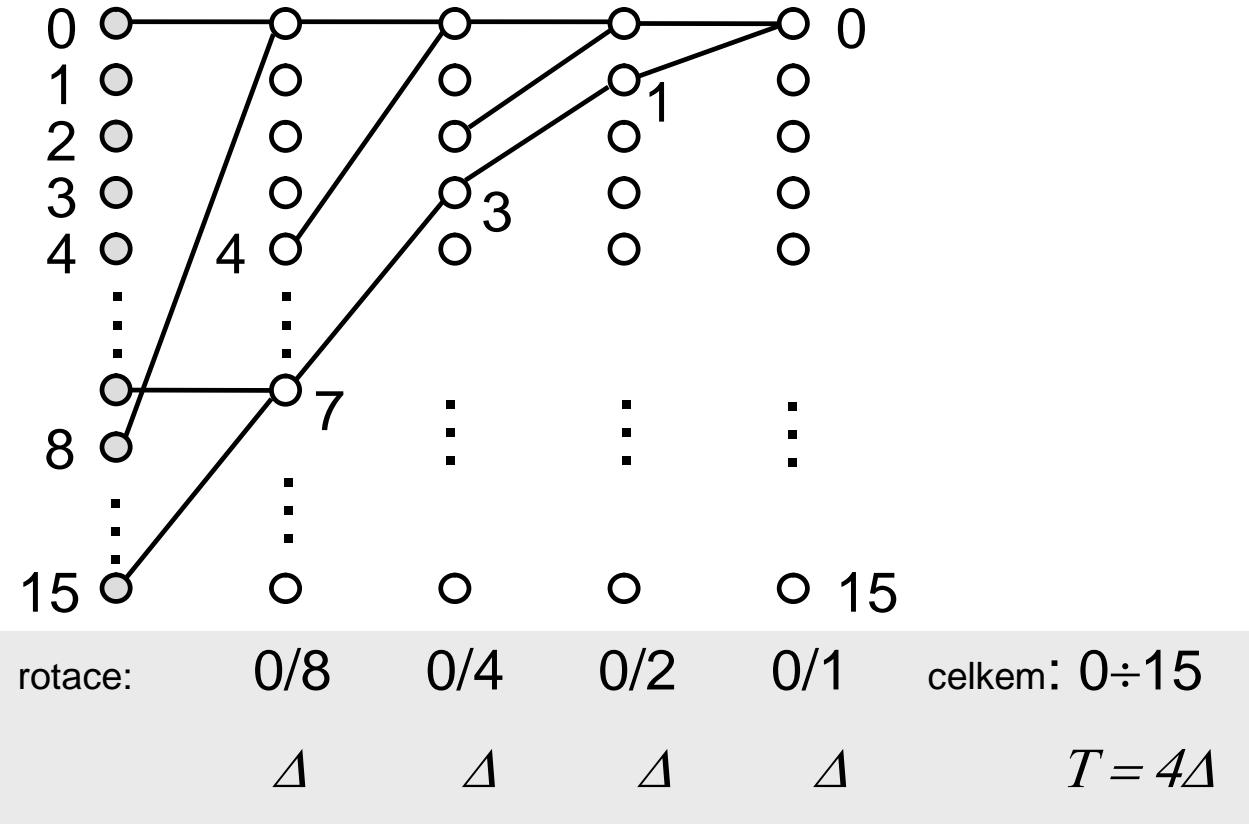
varianta BS(2, 2)



Q: Jak by se realizoval BS pro posuv nikoliv rotaci?



16-bitový válcový posouvač pro rotace vpravo BS(2,2,2,2)

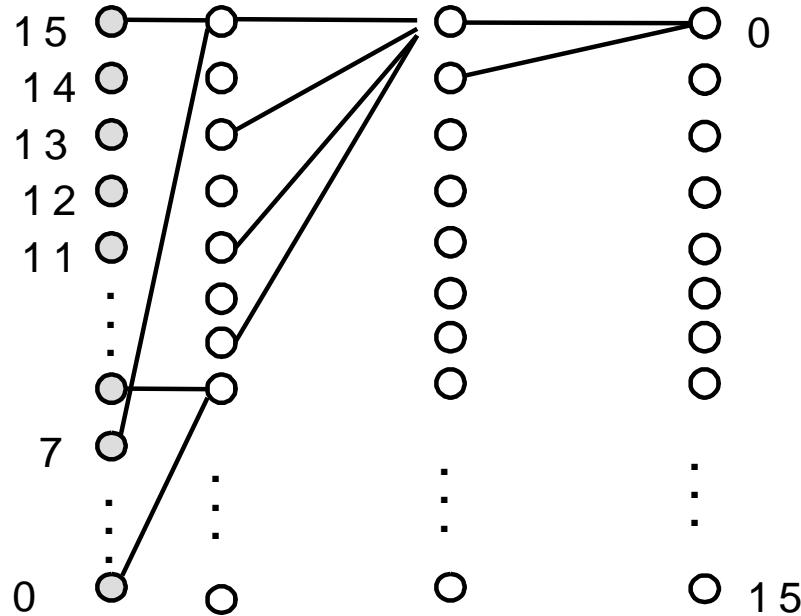


Q: Jak bude vypadat BS(2,4,2)?

Q: Jak by se implementoval BS pro rotaci vlevo?

16-bitový válcový posouvač pro rotace vlevo

BS(2,4,2)



rotace:

0/8

0/2/4/6

0/1

celkem: $0 \div 15$

Δ

Δ

Δ

$T = 3\Delta$

W-bitový válcový posouvač ve VHDL

BS(2,...,2) rotace doprava

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.math_pack.all;

entity BS is
  generic ( W : natural := 16 );
  port (
    DIN   : in  std_logic_vector( W-1 downto 0 );
    DOUT  : out std_logic_vector( W-1 downto 0 );
    SHIFT : in  std_logic_vector( log2(W)-2 downto 0 )
  );
end BS;

architecture rtl of BS is
  constant STEPS : natural := SHIFT'length;
  type tarray is array(STEPS downto 0) of std_logic_vector(W-1 downto 0);
  signal level, unshifted, shifted : tarray;
begin
  level(0) <= DIN;
  DOUT <= level(STEPS);

  levelgen: for i in 1 to STEPS generate
    unshifted(i) <= level(i-1);
    shifted(i) <= level(i-1)(2** (STEPS-i)-1 downto 0) &
                  level(i-1)(W-1 downto 2** (STEPS-i));
    level(i) <= shifted(i) when SHIFT(STEPS-i) = '1' else unshifted(i);
  end generate;
end rtl;
```

i	shifted	shift
1	(7...0) & (15...8)	0/8
2	(3...0) & (15...4)	0/4
3	(1...0) & (15...2)	0/2
4	(0...0) & (15...1)	0/1

W-bitový válcový posouvač ve VHDL BS(2,...,2)

```
architecture rtl of BS is
constant STEPS : natural := SHIFT'length;
type tarray is array(STEPS downto 0) of std_logic_vector(W-1 downto 0);
signal level, unshifted, shifted : tarray;
signal zeros: std_logic_vector(W-1 downto 0);
begin
level(0) <= DIN;
DOUT <= level(STEPS);
zeros <= (others => '0');

levelgen: for i in 1 to STEPS generate
    unshifted(i) <= level(i-1);

    --ROTACE DOPRAVA
    shifted(i) <= level(i-1)(2** (STEPS-i)-1 downto 0) &
                  level(i-1)(W-1 downto 2** (STEPS-i));
    --ROTACE DOLEVA
    shifted(i) <= level(i-1)(W - 2** (STEPS-i)-1 downto 0) &
                  level(i-1)(W-1 downto W-2** (STEPS-i));
    --POSUN DOPRAVA
    shifted(i) <= zeros(2** (STEPS-i)-1 downto 0) &
                  level(i-1)(W-1 downto 2** (STEPS-i));
    --POSUN DOLEVA
    shifted(i) <= level(i-1)(W - 2** (STEPS-i)-1 downto 0) &
                  zeros(W-1 downto W-2** (STEPS-i));

    level(i) <= shifted(i) when SHIFT(STEPS-i) = '1' else unshifted(i);
end generate;
end rtl;
```

Implementace funkce log2

```
library IEEE;
use IEEE.std_logic_1164.all;

package math_pack is
    function log2(constant a : integer) return integer;
end math_pack;

package body math_pack is

    function log2(constant a : integer) return integer is
        variable bits, b : integer;
    begin
        bits := 0;
        b := 1;
        while (b <= a) loop
            b := b * 2;
            bits := bits + 1;
        end loop;

        return bits;
    end function;
end math_pack;
```

- Tento kód se nesyntizuje (je použit během fáze odvozování hodnot parametrů) a tudíž může obsahovat while.

Testbench – 32 bitů

```
entity bs_tb is
end bs_tb;

architecture behv of bs_tb is
constant DW : natural := 32;
component BS is
    generic ( W : natural := DW );
    port ( ... )
end component;

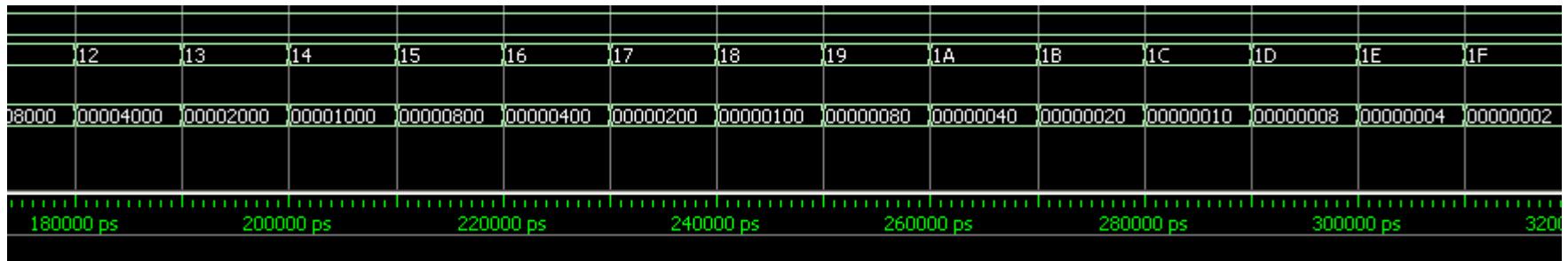
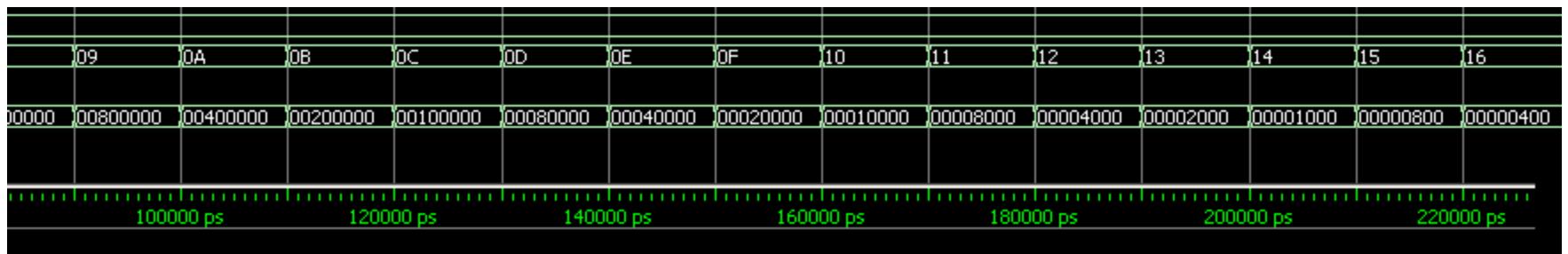
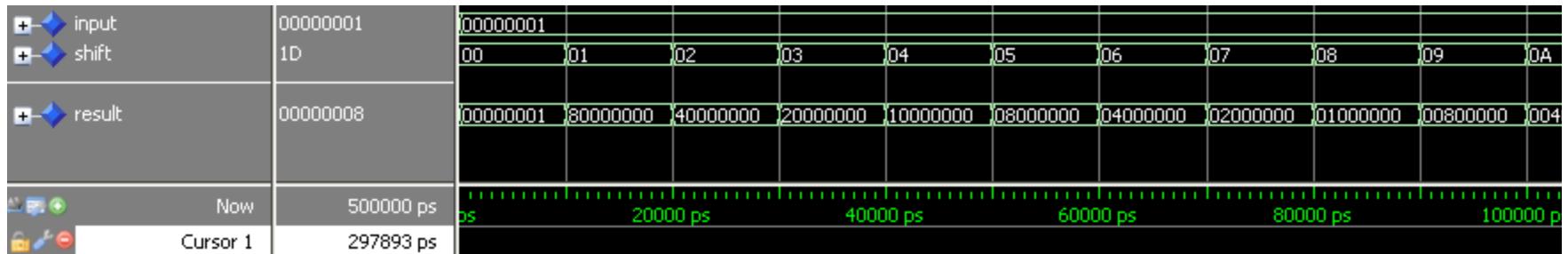
signal input: std_logic_vector(DW-1 downto 0) := (others => '0');
signal result: std_logic_vector(DW-1 downto 0);
signal shift: std_logic_vector(log2(DW)-2 downto 0) := (others => '0');

begin
    uut: BS port map(DIN => input, DOUT => result, SHIFT=>shift);

process
begin
    for i in 0 to DW-1 loop
        input <= conv_std_logic_vector(1, input'length);
        shift <= conv_std_logic_vector(i, shift'length);
        wait for 10 ns;
--        assert to_bitvector(result) = (to_bitvector(input) sll i);
--        assert to_bitvector(result) = (to_bitvector(input) srl i);
--        assert to_bitvector(result) = (to_bitvector(input) rol i);
        assert to_bitvector(result) = (to_bitvector(input) ror i);
    end loop;
    wait;
end process;

end behv;
```

Testbench – výsledek simulace – 32 bitů



Aritmetické a logické operace (násobení)

INP - cvičení 6

Zdeněk Vašíček, 2015
vasicek@fit.vutbr.cz

Násobení

- Existuje řada přístupů (Ripple Carry Array, Carry Save Array, Wallace tree, LUT), které se v zásadě liší
 - v **ploše**, kterou násobička na čipu zabírá a
 - v **rychlosti** (zpoždění násobičky).
- Operace násobení se v HW realizuje jako postupné přičítání vhodně posunutého násobence na základě znalosti jednotlivých bitů násobitele.

Násobení 8b čísel bez znaménka v dvojkové soustavě

01010010	(násobenec)	82
\times	01111101	(násobitel)
<hr/>	01010010	\times 125
00000000		<hr/>
0101001000		410
01010010000		1640
010100100000		<hr/>
0101001000000		8200
00000000000000		<hr/>
001010000001010	(součin)	10250

Výsledný součin musí obsahovat $8+8 = 16$ bitů

Násobení 8b čísel se znaménkem v dvojkové soustavě

- Naivní implementace – zapamatování znaménka a převod na kladná čísla (nepoužívá se).
- Lze použít předchozí algoritmus je však nutné rozšířit (na šířku součinu) / šířit znaménko.

$$\begin{array}{rcl} & \begin{array}{c} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{array} & 10101110 \quad (\text{násobenec}) \\ \times & \begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array} & 01111101 \quad (\text{násobitel}) \\ \hline & \begin{array}{c} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{array} & 10101110 \\ & 0 & 0000000000000000 \\ & 1 & 1111110101011000 \\ & 1 & 1111101010111000 \\ & 1 & 111101011100000 \\ & 1 & 110101110000000 \\ & 0 & 0000000000000000 \\ \hline & 1 & 10101111110110 \quad (\text{součin}) \end{array} \quad \begin{array}{r} -82 \\ \times 125 \\ \hline 410 \\ 1640 \\ 8200 \\ -10250 \\ \hline \end{array}$$

Optimalizace z pohledu zpoždění

- Násobení sestává z řízeného přičítání dílčích součinů (násobitele) k průběžnému výsledku. Pokud je cílem proces zrychlit, musíme
 - buď urychlit operaci přičtení dílčího součinu nebo
 - počet operací přičtení redukovat (použitím jiné reprezentace).
- Motivace: hodnotu $15_{10} = 1111_2$ lze reprezentovat také jako rozdíl $16 - 1 = 10000_2 - 00001_2$
- Při použití předchozího algoritmu se pro výpočet součinu 7×15 musí vyčíslit výraz $0111_2 + 01110_2 + 011100_2 + 0111000_2$ avšak při použití této reprezentace stačí vyčíslit výraz $01110000_2 - 0111_2$. Lze tedy ušetřit 50% operací.

Boothův algoritmus

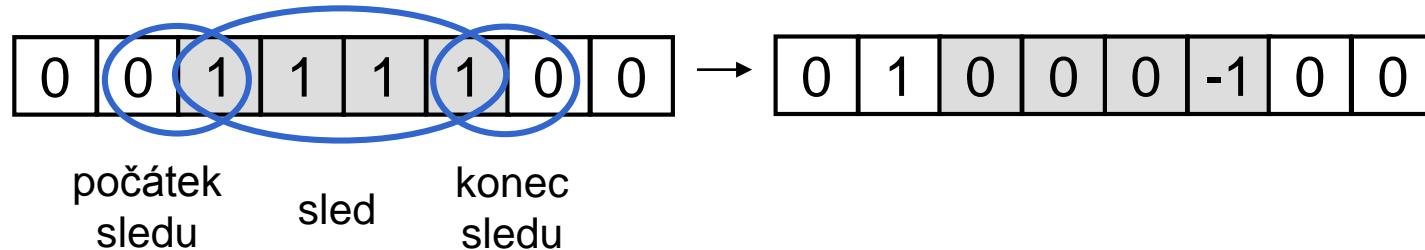
- Boothův algoritmus je algoritmus určený k redukci počtu operací, který využívá mimo operaci přičtení násobence také jeho odečtení.
- Princip: detekovat a nahradit řadu po sobě následujících jedniček (sled jedniček) v násobiteli a tím redukovat počet operací.

0	0	1	1	1	1	0	0	
→								
0	1	0	0	0	0	-1	0	0

- Existuje řada variant, základní verze je Boothovo překódování radix 2 využívající znalosti bitu a jeho pravého souseda (je-li MSB vlevo).

Boothův algoritmus (radix 2) – tvorba kódu

- Lze detekovat sled jedniček pomocí hodnoty bitu a jeho souseda?



- Detekci můžeme provádět na základě znalosti sousedního bitu

hodnota překódovávaného bitu	hodnota sousedního bitu (bit vpravo)	kód	popis situace
0	0 / neexistuje		
0	1		
1	1		
1	0 / neexistuje		

- Neefektivita: za sled je považována i osamocená jednička (010), která je nahrazena dvěma operacemi (1-10). Řešení: použít vyšší radix

Násobení 8b čísel – Boothův algoritmus – radix 2

$$82 \times 125 = 01010010 \times 01111101$$

125:

0	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---

 →

1	0	0	0	0	-1	1	-1
---	---	---	---	---	----	---	----

$$\begin{array}{r} 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0 \\ \times \quad 1\ 0\ 0\ 0\ 0-1\ 1-1 \\ \hline 1111111110101110 \\ 0000000010100100 \\ 1111111010111000 \\ \hline 0010100100000000 \\ \hline 0010100000001010 \end{array} \quad \begin{array}{l} (82) \\ (125) \\ (-82) \\ (+82 \ll 1) \\ (-82 \ll 2) \\ (+82 \ll 7) \end{array}$$

Počet operací se redukoval z 6 na 4

Kontrola: $82 \times 125 = 82 \times (128 + -4 + 2 + -1) = -82 + 82 \times 2 + -82 \times 4 + 82 \times 128$

Boothův algoritmus (radix 4) – tvorba kódu

- Umožníme-li používat dvojnásobnou zápornou a kladnou hodnotu násobence, můžeme dále redukovat počet operací (vyrobit v HW požadované násobky není obtížné).
- **Kód vyššího řádu lze vytvořit z kódu radix 2 přiřazením vah 2^i jednotlivým pozicím.**

hodnota překódovávaných bitů	hodnota sousedního bitu (bit vpravo)	kód radix 2	kód radix 4
00	0	0 0	0
00	1	0 1	1
01	0	1-1	1
01	1	1 0	2
10	0	-1 0	-2
10	1	-1 1	-1
11	0	0-1	-1
11	1	0 0	0

- Rychlá kontrola - nejvyšší bit určuje znaménko (1 – záporné, 0 – kladné)

Násobení 8b čísel – Boothův algoritmus – radix 4

$$82 \times 125 = 01010010 \times 01111101$$

125:

0	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---

 →

1	0	0	0	0	-1	1	-1
2^1	2^0	2^1	2^0	2^1	2^0	2^1	2^0

 radix 2 →

	2	0	-1	1
--	---	---	----	---

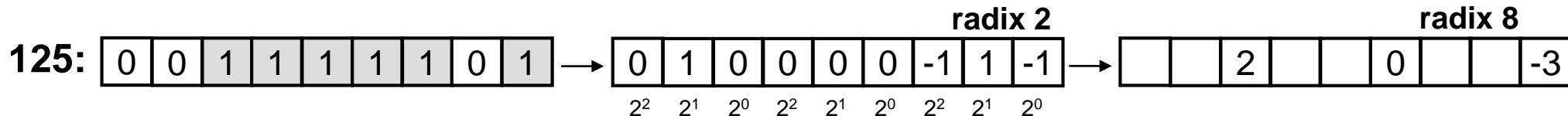
 radix 4

$$\begin{array}{r} 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \quad (82) \\ \times \quad \underline{2 \ 0 \ -1 \ 1} \quad (125) \\ \hline 0000000001010010 \quad (+82) \\ 1111111010111000 \quad (-82 \ll 2) \\ 0010100100000000 \quad (+164 \ll 6) \\ \hline 0010100000001010 \end{array}$$

- **Pozor** na důslednou práci se znaménkem a potřebný počet bitů pro zakódování dvojnásobku násobence. Jednotlivé dílčí součiny je zapotřebí posouvat o 2 bity vlevo!
- Počet operací se redukoval z 6 na 3

Násobení 8b čísel – Boothův algoritmus – radix 8

$$82 \times 125 = 01010010 \times 01111101$$



$$\begin{array}{r} 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0 \\ \times \quad 2\ \quad 0\ \quad -3 \\ \hline 1111111100001010 \\ 0010100100000000 \\ \hline 0010100000001010 \end{array} \quad \begin{array}{l} (82) \\ (125) \\ (-246) \\ (+164 \ll 6) \end{array}$$

- Počet bitů musí být násobkem 3 (přechod na 9-bitová čísla)
- Počet operací se redukoval z 6 na 2.

Násobení 8b čísel – Boothův algoritmus – radix 8

$$-82 \times -125 = 110101110 \times 110000011$$

-125:

1	1	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---

 →

0	-1	0	0	0	0	1	0	-1
---	----	---	---	---	---	---	---	----

 →

		-2			0			3
--	--	----	--	--	---	--	--	---

 radix 2 radix 8

$$\begin{array}{r} 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ \times & \quad -2 & \quad 0 & \quad 3 \\ \hline 1111111100001010 & (-246) \\ 0010100100000000 \\ \hline 0010100000001010 \end{array} \quad \begin{array}{l} (-82) \\ (-125) \\ (+164 \ll 6) \end{array}$$

- Počet bitů musí být násobkem 3 (přechod na 9-bitová čísla)
- Počet operací se redukoval z 4 na 2 (viz počet jedničkových bitů násobitele).

Boothův algoritmus – jiný přístup

Postup podle obecného Boothova algoritmu:

1. Zopakujeme nejvyšší bit skupiny (skupina je k -tice bitů, k dáno radixem)
2. K takto získané hodnotě přičteme hodnotu nejvyššího bitu skupiny vpravo od překódovávané skupiny
3. Výsledná hodnota je binárním vyjádřením relativní číslice

Překódujte číslo -121 na 9 bitech pomocí radix 8 ($2^k = 8$, $k = 3$)

-121 =	<table border="1"><tr><td>1</td><td>1</td><td>0</td></tr></table>	1	1	0	<table border="1"><tr><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	<table border="1"><tr><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1			
1	1	0													
0	0	0													
1	1	1													
	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>	1	1	1	0	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1
1	1	1	0												
0	0	0	0												
1	1	1	1												
	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>	1	1	1	0	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	1	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1
1	1	1	0												
0	0	0	1												
1	1	1	1												
	-2	1	-1												

Ověření správnosti výsledku:

$$-121 = -2 \cdot 2^6 + 1 \cdot 2^3 - 1 \cdot 2^0 = -2 \cdot 64 + 1 \cdot 8 - 1 \cdot 1$$

Boothův algoritmus

- Q: Překódujte číslo -257 na 12 bitech do Boothova kódu radix 8.
- Q: Vynásobte čísla -81×-78 s využitím Boothova algoritmu.
- Q: Kolik operací ušetříme při násobení čísel -81×-78 na 8 bitech zavedením Boothova překódování radix 4?

Boothovo překódování s radixem 4 ve VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity RECODER is
port (
    DIN  : in  std_logic_vector(15 downto 0);
    BIT3 : in  std_logic_vector( 2 downto 0);
    DOUT : out std_logic_vector(16 downto 0));
end RECODER;

architecture RTL of RECODER is
constant N : integer := 16;
subtype bitn1 is std_logic_vector(N downto 0);

function COMP2 (D : in bitn1) return bitn1 is
variable Dn : bitn1;
begin
    Dn := not D;
    return (Dn+1);
end COMP2;
```

Funkce ve VHDL vracející
dvojkový doplněk čísla D

Boothovo překódování s radixem 4 ve VHDL (2)

```
begin
  process (DIN, BIT3)
  begin
    case BIT3 is
      when "001" | "010" =>
        DOUT <= DIN(N-1) & DIN;
      when "101" | "110" =>
        DOUT <= COMP2(DIN(N-1) & DIN);
      when "100" =>
        DOUT <= COMP2(DIN & '0');
      when "011" =>
        DOUT <= DIN & '0';
      when others =>
        DOUT <= (others => '0');
    end case;
  end process;
end RTL;
```

překódovávaný blok	kód
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	2
1 0 0	-2
1 0 1	-1
1 1 0	-1
1 1 1	0

Aritmetické a logické operace (dělení)

INP - cvičení 7

Zdeněk Vašíček, 2015
vasicek@fit.vutbr.cz

Možnosti realizace operace dělení

- Algoritmy s lineární konvergencí
 - s návratem (restaurací) k nezápornému zbytku
 - bez návratu (restaurace) k nezápornému zbytku
 - SRT
- Algoritmy s kvadratickou konvergencí
 - iterační algoritmy (Newtonův iterační algoritmus, Goldschmidtův iterační algoritmus)

Dělení (čísla bez znaménka)

- Hledáme: podíl a zbytek po dělení D/d
- Platí:

$$D = Q \cdot d + R$$

D – dělenec ($2N$ bitů)

d – dělitel (N bitů)

Q – podíl (N bitů)

R – zbytek (N bitů)

- x86 intrukce DIV:
 - dělenec 64b - DX:AX, dělitel – 32b reg, Q – AX, R - DX
- Pozor na rozsahy D a d ($8589934592 / 1$)
- Klasický přístup:
 - 1) zarovnání dělitele podle dělence (posun o $N-1$ bitů)
 - 2) snažíme se odečíst posunutý dělitel ($2^i d$ – posuv o i bitů vlevo) od průběžného zbytku R_i

Dělení 53:7

(posuv d, R_i v pevné poloze, N=4)

	00110101	R0 = D	D = 53 = 0011 0101 ₂
0 ×	00111000	$q_3 \cdot 2^3 \cdot d$	$d = 7 = 0111_2$

	00110101	R1	
	00110101	R1	
-1 ×	00011100	$q_2 \cdot 2^2 \cdot d$	$q_2 = 1$

	00011001	R2	
	00011001	R2	
-1 ×	00001110	$q_1 \cdot 2^1 \cdot d$	$q_1 = 1$

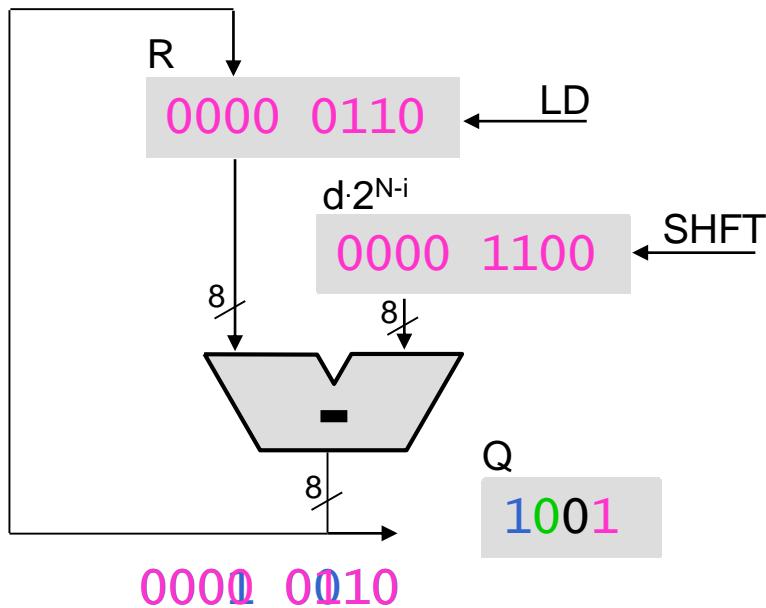
	00001011	R3	
	00001011	R3	
-1 ×	00000111	$q_0 \cdot 2^0 \cdot d$	$q_0 = 1$

	00000100	R=0100	Q=0111

Ukázka HW realizace sekvenční 4b děličky

Základní varianta využívající 8-bitovou odčítačku (lze využít i sčítačku).
Jedná se o tzv. děličku s návratem k (restaurací) nezápornému zbytku

$$01110010 \text{ (114)} : 1100 \text{ (12)} = 1001 \text{ (9)}$$



Algoritmus:

$$R_0 = D$$
$$d = 2^{N-1}d$$

```
for i = 1 to N
    if  $R_{i-1} - d \geq 0$ 
         $Q_{N-i} = 1$ 
         $R_i = R_{i-1} - d$ 
    else
         $Q_{N-i} = 0$ 
         $d = d / 2$ 
```

N – počet bitů, R – průběžný zbytek,
d – dělitel, Q - podíl

Vyhodnocení

- HW realizace takto implementovaného dělení by vyžadovala uchovávat hodnotu dělitele na $2n$ bitech a používat $2n$ bitovou odčítačku / sčítacíku. Dále by byl zapotřebí n -bitový posuvný registr pro uchování podílu.
- Necháme-li d v pevné poloze a budeme-li posouvat R (pomocí posuvného registru), odstraníme výše uvedené nedostatky.

Dělení 53:7

(d v pevné poloze, posuv R_i , N=3)

	1101010	$R_0 = 2D$	$D = 53 = 110\ 101_2$
-1 ×	0111000	$q_2 \cdot 2^3 \cdot d$	$d = 7 = 111_2$

	0110010	R_1	
	1100100	$2 \cdot R_1$	
-1 ×	0111000	$q_1 \cdot 2^3 \cdot d$	$q_1 = 1$

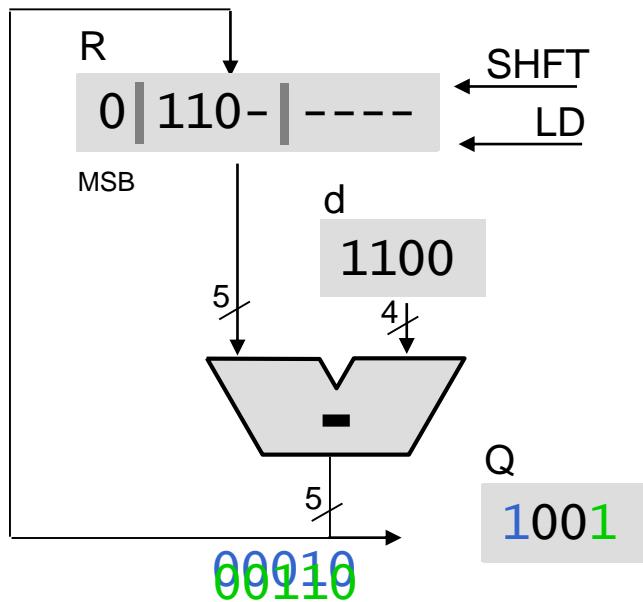
	0101100	R_2	
	1011000	$2 \cdot R_2$	
-1 ×	0111000	$q_0 \cdot 2^3 \cdot d$	$q_0 = 1$

	0100000	$R = 100$	$Q = 111$

Ukázka HW realizace sekvenční 4b děličky

Optimalizovaná verze využívající 5-bitovou odčítačku (lze využít i sčítačku)

$$01110010 \text{ (114)} : 1100 \text{ (12)} = 1001 \text{ (9)}$$



Algoritmus:

$$R_0 = 2 D$$

for $i = 1$ to N

$$\text{if } R_{i-1} - d 2^N \geq 0$$

$$Q_{N-i} = 1$$

$$R_i = R_{i-1} - d 2^N$$

else

$$Q_{N-i} = 0$$

$$R_i = 2 R_{i-1}$$

N – počet bitů, R – průběžný zbytek,
 d – dělitel, Q - podíl

Pro uchování Q lze použít spodní bity registru uchovávajícího R .

Dělení 53:7 s restaurací s využitím sčítáčky

(d v pevné poloze, posuv R_i , $N=4$, vždy odečteme d)

$$\begin{array}{r} 0 \ 01101010 \\ +1 \ 10010000 \\ \hline \end{array} \quad R_0 = 2 \cdot D - d \cdot 2^4$$

$$D = 53 = 0011 \ 0101_2$$

$$\begin{array}{r} 1 \ 11111010 \\ \hline \end{array} \quad < 0 \quad q_3=0$$

$$\begin{aligned} d &= 7 = 0 \ 0111_2 \\ -d &= -7 = 1 \ 1001_2 \end{aligned}$$

$$\begin{array}{r} 01101010 \\ \hline \end{array} \quad R_1 = R_0$$

$$\begin{array}{r} 0 \ 11010100 \\ +1 \ 10010000 \\ \hline \end{array} \quad R_1 = 2 \cdot R_1 - d \cdot 2^4$$

$$\begin{array}{r} 0 \ 10110000 \\ +1 \ 10010000 \\ \hline \end{array} \quad R_3 = 2 \cdot R_3 - d \cdot 2^4$$

$$\begin{array}{r} 0 \ 01100100 \\ \hline \end{array} \quad > 0 \quad q_2=1$$

$$\begin{array}{r} 0 \ 01000000 \\ \hline \end{array} \quad > 0 \quad q_0=1$$

$$\begin{array}{r} 01100100 \\ \hline \end{array} \quad R_2 = 2 \cdot R_1 - d \cdot 2^4$$

$$\begin{array}{r} 01000000 \\ \hline \end{array} \quad R$$

$$R=0100 \quad Q=0111$$

$$\begin{array}{r} 0 \ 11001000 \\ +1 \ 10010000 \\ \hline \end{array} \quad R_2 = 2 \cdot R_2 - d \cdot 2^4$$

$$\begin{array}{r} 0 \ 01011000 \\ \hline \end{array} \quad > 0 \quad q_1=1$$

$$\begin{array}{r} 01011000 \\ \hline \end{array} \quad R_3 = 2 \cdot R_2 - d \cdot 2^4$$

Dělení bez návratu k nezápornému zbytku

- Nevýhodou děličky s návratem k nezápornému zbytku (s restaurací) je značné zpoždění
- V jednom taktu je zapotřebí
 - posunout obsah registru R
 - odečíst posunuté d od dvojnásobku R
 - dle výsledku zjistit bit q
 - dle výsledku přepsat obsah registru R
- Při dělení bez návratu k nezápornému zbytku
 - předpokládáme, že nejvyšší bit Q je 1
 - nemusíme čekat na výsledek a lze ihned rozhodnout pouze na základě znaménka R (nikoliv rozdílu)
 - je nutná korekce v případě chybného předpokladu

Algoritmus

$$R_0 = R - d$$

for $i = 1$ to N

if $R_{i-1} \geq 0$

$$Q_{N-1-i} = 1$$

$$R_i = 2 \cdot R_{i-1} - d$$

else

$$Q_{N-1-i} = 0$$

$$R_i = 2 \cdot R_{i-1} + d$$

if $R_N \geq 0$

$$Q_0 = 1$$

else

$$Q_0 = 0$$

$$R_N = R_N + d$$

Dělení 53:7 bez restaurace s využitím sčítáčky

(d v pevné poloze, posuv R_i , N=4)

$$\begin{array}{r}
 0 \ 00110101 \quad D \\
 +1 \ 10010000 \quad -d \cdot 2^4 \\
 \hline
 1 \ 11000101 \quad R_0 \quad R_0 < 0
 \end{array}
 \qquad
 \begin{array}{l}
 D = 53 = 0011\ 0101_2 \\
 d = 7 = 0\ 0111_2 \\
 -d = -7 = 1\ 1001_2
 \end{array}$$

$$\begin{array}{r}
 1 \ 10001010 \quad 2R_0 \\
 +0 \ 01110000 \quad +d \cdot 2^4 \\
 \hline
 1 \ 11111010 \quad R_1 \quad R_1 < 0 \quad q_3=0
 \end{array}
 \qquad
 \begin{array}{r}
 0 \ 10110000 \quad 2R_3 \\
 +1 \ 10010000 \quad -d \cdot 2^4 \\
 \hline
 0 \ 01000000 \quad R=R_4 \quad R_4 > 0 \quad q_0=1
 \end{array}$$

$$\begin{array}{r}
 1 \ 11110100 \quad 2R_1 \\
 +0 \ 01110000 \quad +d \cdot 2^4 \\
 \hline
 0 \ 01100100 \quad R_2 \quad R_2 > 0 \quad q_2=1
 \end{array}
 \qquad
 \begin{array}{l}
 R=0100 \quad Q=0111
 \end{array}$$

$$\begin{array}{r}
 0 \ 11001000 \quad 2R_2 \\
 +1 \ 10010000 \quad -d \cdot 2^4 \\
 \hline
 0 \ 01011000 \quad R_3 \quad R_3 > 0 \quad q_1=1
 \end{array}$$

Dělení 129:12 bez restaurace s využitím sčítáky

(d v pevné poloze, posuv R_i , N=4)

0	10000001	D	$D = 129 = 1000\ 0001_2$
+1	01000000	$-d \cdot 2^4$	

1	11000001	R_0	$R_0 < 0$
			$d = 12 = 0\ 1100_2$
			$-d = -12 = 1\ 0100_2$
=====			

1	10000010	$2R_0$	0	10010000	$2R_3$
+0	11000000	$+d \cdot 2^4$	+1	01000000	$-d \cdot 2^4$
-----			-----		
0	01000010	R_1	$R_1 > 0$	q3=1	$R=R_4$
1	11010000				$R_4 < 0$
0	10000100	$2R_1$	+0	1100	korekce záporného zbytku
+1	01000000	$-d \cdot 2^4$	-----		
-----			1001		
1	11000100	R_2	$R_2 < 0$	q2=0	$R=1001$
1	10001000	$2R_2$	Q=1010		
+0	11000000	$+d \cdot 2^4$			

0	01001000	R_3	$R_3 > 0$	q1=1	

Algoritmus SRT

- Podobně jako v případě násobení existují algoritmy, které jsou a) schopny redukovat počet operací dělení zpracováním více bitů najednou a b) schopny pracovat s čísly se znaménkem.
- Nejjednodušším přístupem je základní varianta algoritmu SRT (radix 2) pracující po jednom bitu a určující následující **operaci dle nejvyšších 3 bitů** (včetně znaménkového bitu).

Poznámka:

- Při dělení čísel se znaménkem požadujeme, aby zbytek byl vždy kladný (je zapotřebí provádět korekci).

Příklad SRT: $29:7 = 4$ zbytek 1

$29=00011101$, $7=0\ 111$, $-7=1\ 001$

0	00011101		$q_4 = 0$
+1	0011101x	<-	$q_3 = 1$
	<u>1001</u>	-d	
	1100101x		
-1	100101xx	<-	$q_2 = -1$
	<u>0111</u>	+d	
	000001xx		
0	00001xxx	<-	$q_1 = 0$
0	0001xxxx	<-	$q_0 = 0$

zbytek 1

$$\begin{aligned} Q &= 1 \ -1 \ 0 \ 0 = \\ &\quad 2^3 \ 2^2 \ 2^1 \ 2^0 \\ &= 8-4 = 4 \end{aligned}$$

Ri	d>0		d<0	
	bit podílu	operace	bit podílu	operace
000 111	0	žádná	0	žádná
001 010 011	1	-d	-1	+d
101 110 100	-1	+d	1	-d

Dle tabulky se provede přičtení nebo odečtení dělitele a určí hodnota bitu podílu. Následuje posunutí vlevo (s výjimkou posledního kroku).

Příklad SRT: $39:6 = 6$ zbytek 3

$$39=00100111, \ 6=0\ 110, \ -6=1\ 010$$

1	<u>001</u> 00111	
	<u>1010</u> -d	
	11000111	
-1	<u>100</u> 0111x <-	
	<u>0110</u> +d	
	1110111x	
-1	<u>110</u> 111xx <-	
	<u>0110</u> +d	
	001111xx	
1	<u>011</u> 11xxx <-	
	<u>1010</u> -d	
	00011xxx	
1	<u>001</u> 1xxxx <-	
	<u>1010</u> -d	
	1101xxxx záporný zbytek	
	<u>0110</u> +d (korekce)	
	<u>0011</u> zbytek 3	

$$Q = 1 \ -1 \ -1 \ 1 \ 1 =$$

$$16-8-4+2+1=7$$

$$\text{po korekci } 7-1 = 6$$

	d>0		d<0	
Ri	bit podílu	operace	bit podílu	operace
000 111	0	žádná	0	žádná
001 010 011	1	-d	-1	+d
101 110 100	-1	+d	1	-d

Dle tabulky se provede přičtení nebo odečtení dělitele a určí hodnota bitu podílu. Následuje posunutí vlevo (s výjimkou posledního kroku).

V případě záporného zbytku je zapotřebí provést korekci na kladný a snížit Q o 1 (záporný zbytek kladnou hodnotu zmenšoval).

Příklad SRT: $41 : -6 = -6$ zbytek 5

$$41=00101001, \ 6=0\ 110, \ -6=1\ 010$$

-1 00101001
 1010 +d
 11001001
 1 1001001x <-
 0110 -d
 1111001x
 0 111001xx <-
 1 11001xxx <-
 0110 -d
 00101xxx
 -1 0101xxxx <-
 1010 +d
 1111xxxx záporný zbytek
 0110 +d (korekce)
 0101 zbytek 5

$$\begin{aligned}
 Q &= -1\ 1\ 0\ 1\ -1 = \\
 -16+8+0+2-1 &= -7 \\
 \text{po korekci} \quad -7+1 &= -6
 \end{aligned}$$

	d>0		d<0	
Ri	bit podílu	operace	bit podílu	operace
000 111	0	žádná	0	žádná
001 010 011	1	-d	-1	+d
101 110 100	-1	+d	1	-d

Dle tabulky se provede přičtení nebo odečtení dělitele a určí hodnota bitu podílu. Následuje posunutí vlevo (s výjimkou posledního kroku).

V případě záporného zbytku je zapotřebí provést korekci na kladný a **zvýšit** Q o 1 (záporný zbytek zápornou hodnotu zvětšoval).

Příklad SRT: $-39:-6 = 7$ zbytek 3

$$-39=11011001, \ 6=0\ 110, \ -6=1\ 010$$

1 **110**11001
0110 -d
 00111001
 -1 **011**1001x <-
1010 +d
 0001001x
 -1 **001**001xx <-
1010 +d
 110001xx
 1 **100**01xxx <-
0110 -d
 11101xxx
 1 **110**1xxxx <-
0110 -d
 0011 zbytek 3

	d>0		d<0	
Ri	bit podílu	operace	bit podílu	operace
000 111	0	žádná	0	žádná
001 010 011	1	-d	-1	+d
101 110 100	-1	+d	1	-d

Dle tabulky se provede přičtení nebo odečtení dělítelé a určí hodnota bitu podílu. Následuje posunutí vlevo (s výjimkou posledního kroku).

$$\begin{aligned}
 Q &= 1 \ -1 \ -1 \ 1 \ 1 = \\
 16-8-4+2+1 &= 7
 \end{aligned}$$

Příklad SRT: $65:7 = 9$ zbytek 2

(musíme přejít na 5bitů)

$$65=0001000001, \ 7=0\ 0111, \ -7=1\ 1001$$

0	000 1000001	
1	001 000001x	<-
	<u>11001</u>	-d
	111010001x	
-1	110 10001xx	<-
	<u>00111</u>	+d
	00001001xx	
0	000 1001xxx	<-
1	001 001xxxx	<-
	<u>11001</u>	-d
	111011xxxx	
-1	110 11xxxxx	<-
	<u>00111</u>	+d
	00010	zbytek 2

	d>0		d<0	
Ri	bit podílu	operace	bit podílu	operace
000 111	0	žádná	0	žádná
001 010 011	1	-d	-1	+d
101 110 100	-1	+d	1	-d

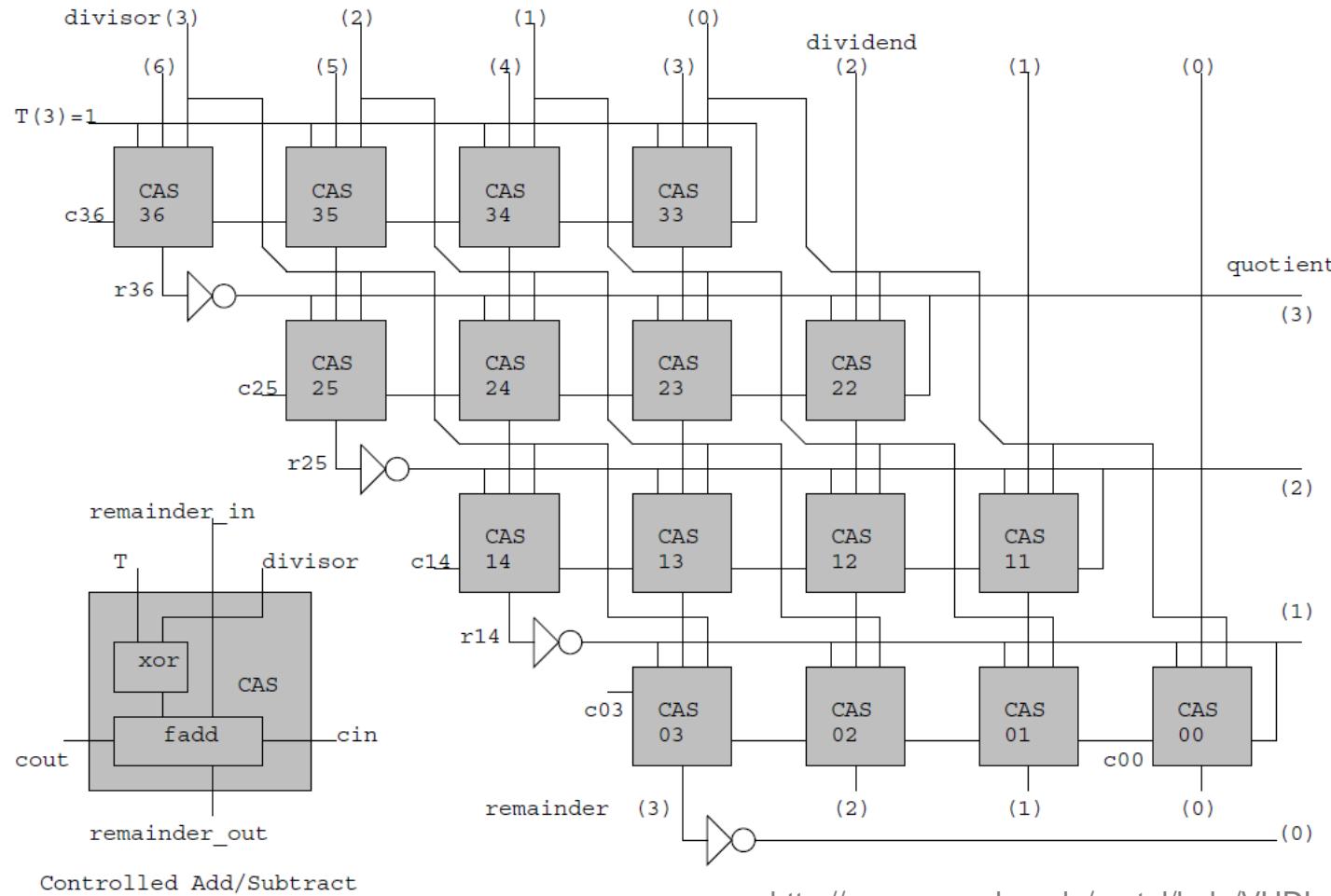
Dle tabulky se provede přičtení nebo odečtení dělitele a určí hodnota bitu podílu. Následuje posunutí vlevo (s výjimkou posledního kroku).

$$Q = 0\ 1\ -1\ 0\ 1\ -1 = 16-8+2-1 = 9$$

Úkoly

- Pomocí algoritmu bez návratu k nezápornému zbytku vypočítejte
 - $103 : 15$ (4-bitový dělitel)
 - $55 : 11$ (4-bitový dělitel)
 - $8 : 3$ (3-bitový dělitel)
- Pomocí algoritmu SRT s radixem 2 vypočítejte
 - $-63 : -3$ (3-bitový dělitel)
 - $41 : 5$ (3-bitový dělitel)
 - $22 : -3$ (3-bitový dělitel)
 - $61 : 8$ (4-bitový dělitel)
- Jaká je nejmenší možná hodnota dělitele, kterým lze v SRT vydělit bezchybně číslo 21 v případě použití 3-bitového dělitele.

Princip realizace kombinační 4b děličky (bez návratu k nezápornému zbytku)



- Základní stavební prvek: konfigurovatelná sčítka/odčítka

Iterační algoritmy

INP - cvičení 8

Zdeněk Vašíček, 2015
vasicek@fit.vutbr.cz

Obsah

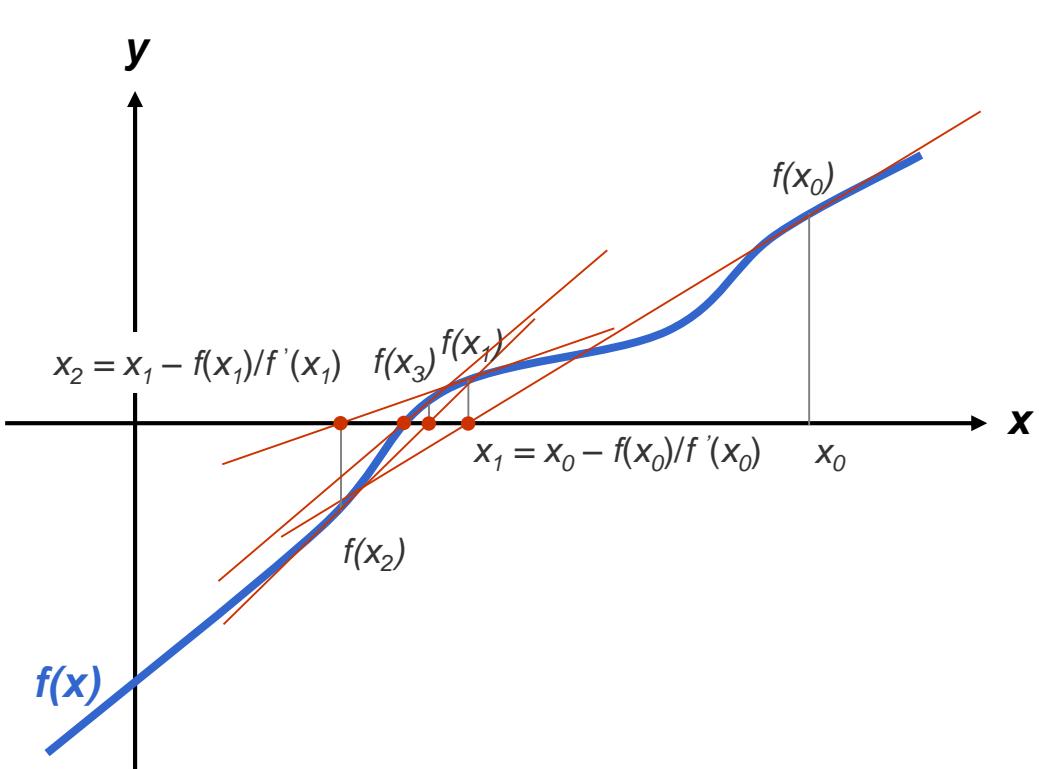
- Newtonův iterační algoritmus
 - dělení, výpočet odmocniny
- Goldschmidtův iterační algoritmus
 - dělení
- CORDIC
 - HW implementace

Newtonova iterační metoda

- Newton–Raphsonova metoda (metoda tečen) je **iterační metoda** určená k nalezení kořenů reálné funkce (tj. bodů v nichž je funkční hodnota funkce rovna nule).
- Zpřesnění počátečního odhadu je provedeno na základě znalosti **derivace** zadané funkce (spojitost funkce, monotonní funkce, ...).
- Použití:
 - nalezení minima či maxima funkce (pracuje nad $f'(x)$)
 - dělení reálných čísel (využívá se operace násobení)
 - výpočet druhé odmocniny reálných čísel
- Problémy:
 - divergence / pomalá konvergence při nevhodně zvoleném počátečním odhadu
 - nediferencovatelné funkce

Newtonova iterační metoda - princip

Iterativní hledání bodu x_k pro který platí $f(x_k) = 0$ (průsečík funkce $f(x)$ s osou Y)



$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Odvození iteračního vzorce vychází ze znalosti rovnice tečny v bodě (x_0, y_0)

$$y - y_0 = k(x - x_0), \quad f(x_0) = y_0, \quad k = f'(x_0)$$

Newtonova iterační metoda – dělení

Máme-li dělit číslem b , zvolíme

$$f(x) = 1/x - b$$

Pokud $f(x) = 0$, pak $1/x = b$ a tedy $x = 1/b$

Operaci dělení a/b nahradíme násobením $a * (1/b)$

Odvození iteračního vzorce:

$f'(x) = -1/x^2$, dosazením do $x_{i+1} = x_i - f(x_i)/f'(x_i)$ dostaneme

$$x_{i+1} = x_i - (1/x_i - b)/(-1/x_i^2) = x_i + x_i - bx_i^2 = x_i(2 - bx_i)$$

$$x_{i+1} = x_i(2 - bx_i)$$

Postup výpočtu a/b :

1. Zarovnáme (normalizujeme) b tak, aby platilo $1 < b' < 2$, tedy $b' = b * 2^{-k}$
2. Pomocí tabulky odhadů zvolíme první odhad x_0
3. Postupně provádíme výpočty x_{i+1} dokud nedostaneme řešení s požadovanou přesností (viz dále)
4. Výsledek n -té iterace (x_n) vynásobíme číslem a , součin posuneme o odpovídající počet bitů (vynásobíme 2^{-k})

Pomocí Newtonovy iterační metody vypočítejte hodnotu výrazu $1/20$, $x_0 = 1.0$

$$\underline{b} = 20 = 10100_2$$

normalizace b:

$$b = 2^{-4}\underline{b} = 1.0100_2 = 1.25 \text{ (posun desetinné čárky o 4 bity vlevo)}$$

$$x_0 = 1.0$$

$$x_1 = x_0(2 - b^*x_0) = 1.000000 (2 - 1.25 * 1.000000) = 0.750000$$

$$x_2 = x_1(2 - b^*x_1) = 0.750000 (2 - 1.25 * 0.750000) = 0.796875$$

$$x_3 = x_2(2 - b^*x_2) = 0.796875 (2 - 1.25 * 0.796875) = 0.799988$$

$$x_4 = x_3(2 - b^*x_3) = 0.799988 (2 - 1.25 * 0.799988) = 0.800000$$

$$x_5 = x_4(2 - b^*x_4) = 0.800000 (2 - 1.25 * 0.800000) = 0.800000$$

$$x_6 = x_5(2 - b^*x_5) = 0.800000 (2 - 1.25 * 0.800000) = 0.800000$$

Výsledek po 6 iteračních krocích:

$$1/\underline{b} = 1/20 = 0.800000 * 2^{-4} = 0.05$$

Pomocí Newtonovy iterační metody vypočítejte hodnotu výrazu $1/20$, $x_0 = 0.125$

$$\underline{b} = 20 = 10100_2$$

normalizace b:

$$b = 2^{-4}\underline{b} = 1.0100_2 = 1.25$$

$$x_0 = 0.125$$

$$x_1 = x_0(2 - b^*x_0) = 0.125000 (2 - 1.25 * 0.125000) = 0.230469$$

$$x_2 = x_1(2 - b^*x_1) = 0.230469 (2 - 1.25 * 0.230469) = 0.394543$$

$$x_3 = x_2(2 - b^*x_2) = 0.394543 (2 - 1.25 * 0.394543) = 0.594505$$

$$x_4 = x_3(2 - b^*x_3) = 0.594505 (2 - 1.25 * 0.594505) = 0.747215$$

$$x_5 = x_4(2 - b^*x_4) = 0.747215 (2 - 1.25 * 0.747215) = 0.796517$$

$$x_6 = x_5(2 - b^*x_5) = 0.796517 (2 - 1.25 * 0.796517) = 0.799985$$

$$x_7 = x_6(2 - b^*x_6) = 0.799985 (2 - 1.25 * 0.799985) = 0.800000$$

$$x_8 = x_7(2 - b^*x_7) = 0.800000 (2 - 1.25 * 0.800000) = 0.800000$$

Výsledek po 8 iteračních krocích:

$$1/\underline{b} = 1/20 = 0.800000 * 2^{-4} = 0.05$$

Určení prvního odhadu

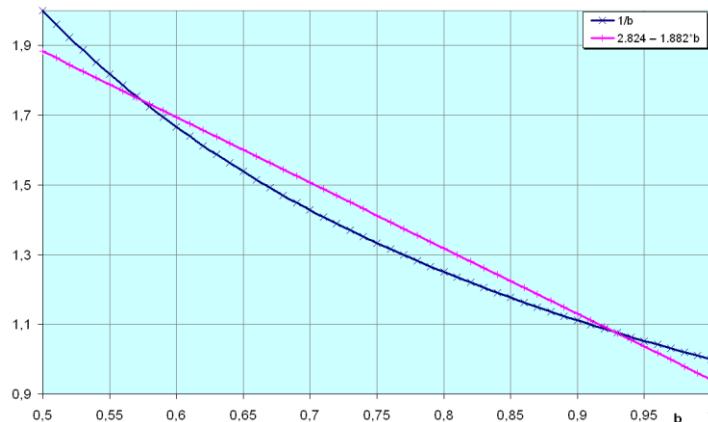
- První odhad je možné určit pomocí lineární approximace.

$$1/b \sim x_0 = k^*b + q$$

- Pokud $0.5 < b < 1$, pak hodnoty $q = 48/17$ a $k = -32/17$ poskytují odhad s minimální maximální absolutní chybou.

$$x_0 = 2.824 - 1.882^*b$$

- V případe použití tohoto odhadu je zaručena kvadratická konvergence.



Pomocí Newtonovy iterační metody vypočítejte hodnotu výrazu $1/20$

$$\underline{b} = 20 = 10100_2$$

normalizace $0.5 < b < 1$:

$$b = \underline{b} * 2^{-5} = 0.10100_2 = 0.625 \text{ (posun des. čárky o } 5\text{b vlevo)}$$

určení odhadu x_0 :

$$x_0 = 2.824 - 1.882 * b = 2.824 - 1.882 * 0.625 = 1.64775$$

$$x_1 = \underline{x}_0(2 - b * x_0) = 1.64775 (2 - 0.625 * 1.64775) = 1.59857$$

$$x_2 = x_1(2 - b * x_1) = 1.59857 (2 - 0.625 * 1.59857) = 1.60000$$

$$x_3 = x_2(2 - b * x_2) = 1.60000 (2 - 0.625 * 1.60000) = 1.60000$$

Výsledek po 3 iteračních krocích:

$$1/\underline{b} = 1/20 = 1.6000 * 2^{-5} = 0.05$$

Pomocí Newtonovy iterační metody vypočítejte hodnotu výrazu $13/8$

$$\underline{b} = 8 = 1000_2$$

normalizace $0.5 < b < 1$:

$$b = \underline{b} * 2^{-4} = 0.1000_2 = 0.5 \text{ (posun des. čárky o 4b vlevo)}$$

určení odhadu x_0 :

$$x_0 = 2.824 - 1.882 * b = 2.824 - 1.882 * 0.5 = 1.88300$$

$$x_1 = \underline{x}_0(2 - b * \underline{x}_0) = 1.88300 (2 - 0.5 * 1.88300) = 1.99316$$

$$x_2 = x_1(2 - b * x_1) = 1.99316 (2 - 0.5 * 1.99316) = 1.99998$$

$$x_3 = x_2(2 - b * x_2) = 1.99998 (2 - 0.5 * 1.99998) = 2.00000$$

Výsledek po 3 iteračních krocích:

$$a/b = 13/8 = 13 * 2.0000 * 2^{-4} = 1.625$$

Newtonova iterační metoda – odmocnina

Máme-li vypočítat odmocninu čísla S , můžeme zvolit

$$f(x) = x^2 - S$$

Pokud $f(x) = 0$, pak $x^2 = S$ a tedy $x = \sqrt{S}$

Odvození iteračního vzorce:

$f'(x) = 2x$, dosazením do $x_{i+1} = x_i - f(x_i)/f'(x_i)$ dostaneme

$$x_{i+1} = x_i - (x_i^2 - S)/2x_i = (x_i^2 + S) / 2x_i$$

$$x_{i+1} = (x_i^2 + S) / 2x_i$$

Odvozený iterační vzorec však obsahuje operaci dělení proměnnou hodnotou, což je velmi drahá operace.

Newtonova iterační metoda – odmocnina

Efektivnější iterační předpis získáme, zvolíme-li

$$f(x) = 1/x^2 - S = x^2 - S$$

Pokud $f(x) = 0$, pak $x^2 = S$ a tedy $x = \sqrt{S}$ (tzv. *reciproká odmocnina*)

Odvození iteračního vzorce:

$$\begin{aligned} f'(x) &= -2x^{-3}, \text{ dosazením do } x_{i+1} = x_i - f(x_i)/f'(x_i) \text{ dostaneme} \\ x_{i+1} &= x_i - (x_i^{-2} - S)/(-2x_i^{-3}) = x_i + (x_i - x_i^3 S) / 2 = (3x_i - x_i^3 S) / 2 \\ x_{i+1} &= x_i(3 - x_i^2 S) / 2 \end{aligned}$$

Zjednodušený postup výpočtu \sqrt{S} :

- Zvolíme první odhad x_0
 - Postupně provádíme výpočty x_{i+1} dokud nedostaneme řešení s požadovanou přesností
1. Výsledek n -té iterace (x_n) vynásobíme číslem S , čímž získáme \sqrt{S}

Pomocí Newtonovy iterační metody vypočítejte hodnotu výrazu $\sqrt{324}$, $x_0 = 0.01$

$x_0 = 0.01$ ($\sqrt{100} = 10 < \sqrt{324} < \sqrt{10000} = 100$)

$$x_1 = x_0(3 - x_0^2 S)/2 = 0.0100000 (3 - 0.000100000 * 324)/2 = 0.0148380$$

$$x_2 = x_1(3 - x_1^2 S)/2 = 0.0148380 (3 - 0.000220166 * 324)/2 = 0.0217278$$

$$x_3 = x_2(3 - x_2^2 S)/2 = 0.0217278 (3 - 0.000472096 * 324)/2 = 0.0309299$$

$$x_4 = x_3(3 - x_3^2 S)/2 = 0.0309299 (3 - 0.000956661 * 324)/2 = 0.0416014$$

$$x_5 = x_4(3 - x_4^2 S)/2 = 0.0416014 (3 - 0.001730680 * 324)/2 = 0.0507383$$

$$x_6 = x_5(3 - x_5^2 S)/2 = 0.0507383 (3 - 0.002574380 * 324)/2 = 0.0549471$$

$$x_7 = x_6(3 - x_6^2 S)/2 = 0.0549471 (3 - 0.003019190 * 324)/2 = 0.0555456$$

$$x_8 = x_7(3 - x_7^2 S)/2 = 0.0555456 (3 - 0.003085310 * 324)/2 = 0.0555556$$

$$x_9 = x_8(3 - x_8^2 S)/2 = 0.0555556 (3 - 0.003086420 * 324)/2 = 0.0555556$$

Výsledek po 9 iteračních krocích:

$$\sqrt{324} = 324 * 0.0555555 = 18$$

Goldschmidtův iterační algoritmus - dělení

Princip:

- Goldschmidtův algoritmus se snaží nalézt podíl Q rozšiřováním zlomku N/D vhodnými zlomky tak, aby se ve jmenovateli získala hodnota blízká 1.

$$Q = \frac{N}{D} = \frac{N}{D} \frac{R_0}{R_0} \frac{R_1}{R_1} \frac{R_2}{R_2} \frac{R_3}{R_3} \dots \underset{1}{\approx} \frac{NR_0 R_1 R_2 R_3 \dots}{1}$$

- Po dostatečném počtu iterací je hodnota čitatele blízká hodnotě podílu

Goldschmidtův iterační algoritmus odvození iteračního předpisu

- Předpokládejme, že platí $1 \leq N, D < 2$ (normalizace N a D)
- Položíme $x_0=N, y_0=D$, náhledem do tabulky odhadneme R_0 tak, aby $y_0 R_0 \rightarrow 1$.
- Pro relativní chybu počátečního odhadu platí

$$\varepsilon = \frac{1 - y_0 R_0}{1} = 1 - y_0 R_0 \quad -1 < \varepsilon < 1$$

- Dále můžeme odvodit

$$\frac{x_1}{y_1} = \frac{x_0}{y_0} \frac{R_0}{R_0} = \frac{x_0}{y_0} \quad y_1 = y_0 R_0 \Rightarrow y_1 = 1 - \varepsilon$$

- Máme vynásobit y_1 tak, abychom se přiblížili k 1, zvolme $R_1=1+\varepsilon$

$$\frac{x_2}{y_2} = \frac{x_1}{y_1} \frac{R_1}{R_1} = \frac{x_1}{1-\varepsilon} \frac{1+\varepsilon}{1+\varepsilon} = \frac{x_1(1+\varepsilon)}{1-\varepsilon^2} \quad y_2 = 1 - \varepsilon^2$$
$$R_1 = 1 + \varepsilon = 2 - y_1$$

- Máme vynásobit y_2 tak, abychom se přiblížili k 1, zvolme $R_2=1+\varepsilon^2$

$$\frac{x_3}{y_3} = \frac{x_2}{y_2} \frac{R_2}{R_2} = \frac{x_2}{1-\varepsilon^2} \frac{1+\varepsilon^2}{1+\varepsilon^2} = \frac{x_2(1+\varepsilon^2)}{1-\varepsilon^4} \quad y_3 = 1 - \varepsilon^4$$
$$R_2 = 1 + \varepsilon^2 = 2 - y_2$$

Goldschmidtův iterační algoritmus

výpočet Q=N/D

Postup výpočtu

- Položíme $x_0 = N$, $y_0 = D$
- Odhadneme R_0 (např. náhled do tabulky)
- V první iteraci určíme $x_1 = x_0 R_0$, $y_1 = y_0 R_0$
- V dalších iteracích ($i = 1, \dots$) násobíme čitatel i jmenovatel koeficientem $R_i = 2 - y_i$. Násobením koeficientem R_i provádíme rozšiřování výrazem $1 + \varepsilon^{2i}$ (viz odvození).
- Po dostatečném počtu iterací je x_0 rovno podílu.

Odhad R_0

- Dostatečně dobrý odhad R_0 lze získat vyčíslením výrazu

$$R_0 = \operatorname{sgn}(y_0) 2^n (2 - \operatorname{sgn}(y_0) 2^n y_0)$$

což pro nezáporné číslo v rozsahu $<1,2$) odpovídá $R_0 = 2 - y_0$

Pomocí Goldschmidtova iteračního algoritmu vypočítejte hodnotu výrazu 3/22

$$\underline{N} = 3 = 0011_2 \quad \underline{D} = 22 = 10110_2$$

normalizace $1 \leq N, D < 2$:

$$N = \underline{N} * 2^{-1} = 1.1_2 = 1.500$$

$$D = \underline{D} * 2^{-4} = 1.011_2 = 1.375 \quad \text{výsledek musíme násobit } 2^{1-4} \text{ (posunout tečku o 3 bity doleva)}$$

$$R_0 = 2 - y_0 = 2 - 1.375 = 0.625 \quad (\text{prvotní odhad } R_0)$$

$$x_1 = x_0 * R_0 = 1.500 * 0.625 = 0.937500$$

$$y_1 = y_0 * R_0 = 1.375 * 0.625 = 0.859375$$

$$R_1 = 2 - y_1 = 2 - 0.859375 = 1.14063$$

$$x_2 = x_1 * R_1 = 0.937500 * 1.14063 = 1.069340$$

$$y_2 = y_1 * R_1 = 0.859375 * 1.14063 = 0.980225$$

$$R_2 = 2 - y_2 = 2 - 0.980225 = 1.01978$$

$$x_3 = x_2 * R_2 = 1.069340 * 1.01978 = 1.090480$$

$$y_3 = y_2 * R_2 = 0.980225 * 1.01978 = 0.999609$$

$$R_3 = 2 - y_3 = 2 - 0.999609 = 1.00039$$

$$x_4 = x_3 * R_3 = 1.090480 * 1.00039 = 1.09091$$

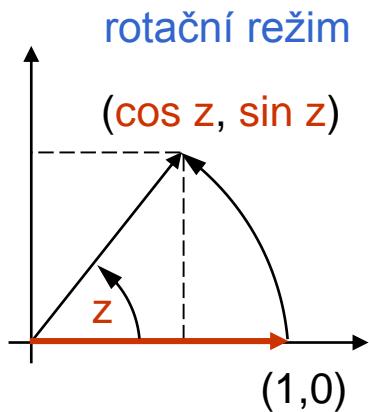
$$y_4 = y_3 * R_3 = 0.999609 * 1.00039 = 1.00000$$

Výsledek po 4 iteračních krocích:

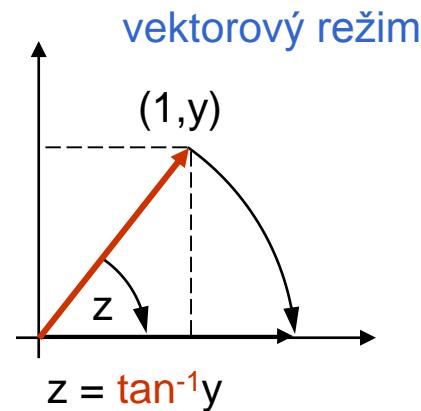
$$N/D = 3/22 = x_4 * 2^{-3} = 1.09091 * 0.125 = 0.1363636$$

CORDIC

- Metoda umožňující vypočítat řadu matematických funkcí vyčíslováním výrazu tvaru $a \pm b \cdot 2^i$ tj. pomocí součtů, rozdílů, posunů (řada operací realizována vyhledáváním v look-up tabulce).
- Umíme-li efektivně provést rotaci vektoru, můžeme snadno vypočítat \cos , \sin a \tan^{-1}



1. počátek v $(1,0)$,
2. rotace tak, aby úhel byl z ,
3. $x = \cos z$ a $y = \sin z$



1. počátek v $(1,y)$,
2. rotace o z tak, aby y bylo 0,
3. z odpovídá $\tan^{-1} y$

CORDIC – výpočet $\sin \alpha$ a $\cos \alpha$

- Položíme $x_0=1$, $y_0=0$ a $z_0=\alpha$
- Postupně přičítáme (odčítáme) od z_i úhel α_i dle i-tého řádku tabulky tak, abychom se přiblížili k nule. Počet iterací je dán požadovanou přesností. (mění se znaménko protože tan je funkce lichá).
- Hodnoty x_{i+1} a y_{i+1} vypočteme na základě nahlédnutí na i-tý řádek tabulky obsahující $\tan \alpha_i$ (mocnina dvou)

$$x_{i+1} = x_i - (+) y_i \tan \alpha_i = x_i - (+) y_i 2^{-i}$$

$$y_{i+1} = y_i + (-) x_i \tan \alpha_i = y_i + (-) x_i 2^{-i}$$

$$z_{i+1} = z_i + (-) \alpha_i$$

- Získané hodnoty x_i a y_i vynásobíme agregační konstantou.
- Hodnota x_i odpovídá $\cos \alpha$, hodnota y_i odpovídá $\sin \alpha$
- Optimalizace: Násobení agregační konstantou lze skrýt do x_0 .

CORDIC – tabulka konstant

i	$\alpha_i [^\circ]$	$\operatorname{tg} \alpha_i$	$\cos \alpha_i$
1	45,00	1	0,707
2	26,56	0,5	0,894
3	14,04	0,25	0,970
4	7,12	0,125	0,997
5	3,57	0,0625	0,998
6	1,789	0,03125	0,999
7	0,895	0,015625	0,9998
8	0,4476	0,0078125	0,9999

- Hodnoty α_i jsou voleny tak, aby $\tan \alpha_i$ byl roven mocnině dvou, což dovoluje nahradit násobení posuvem.
- Agregační konstanta je rovna součinu n po sobě následujících hodnot $\cos \alpha_i$, kde n odpovídá počtu iterací dle požadované přesnosti.
- Znaménko α_i nemusíme uvažovat, \cos je funkce sudá.

CORDIC (Python)

```
def cordic(alpha, iterations=16): #alpha v radianech

    alphatab = [math.atan(2**(-i)) for i in range(iterations)]
    K = reduce(lambda x,y: x*y, [math.cos(a) for a in alphatab])
    x, y, z = K, 0.0, float(alpha)

    for i in range(iterations):
        if z < 0:
            xn = x + y * 2**(-i)
            yn = y - x * 2**(-i)
            z += alphatab[i]
        else:
            xn = x - y * 2**(-i)
            yn = y + x * 2**(-i)
            z -= alphatab[i]
        x, y = xn,yn

    return (x, y) # cos(alpha), sin(alpha)
```

S využitím 16 kroků algoritmu CORDIC určete $\sin(\pi/4)$

$$x_0 = 1.0, y_0 = 0.0, z_0 = \pi/4 = 45^\circ$$

$z > 0$

$$\begin{aligned}x_1 &= x_0 - y_0 * 2^0 = 1.00000 - 0.00000 * 1 = 1 \\y_1 &= y_0 + x_0 * 2^0 = 0.00000 + 1.00000 * 1 = 1 \\z_1 &= z_0 - a_0 = 45 - 45 = 0\end{aligned}$$

$z > 0$

$$\begin{aligned}x_2 &= x_1 - y_1 * 2^{-1} = 1.00000 - 1.00000 * 0.5 = 0.5 \\y_2 &= y_1 + x_1 * 2^{-1} = 1.00000 + 1.00000 * 0.5 = 1.5 \\z_2 &= z_1 - a_1 = 0 - 26.5651 = -26.5651\end{aligned}$$

$z < 0$

$$\begin{aligned}x_3 &= x_2 + y_2 * 2^{-2} = 0.50000 + 1.50000 * 0.25 = 0.875 \\y_3 &= y_2 - x_2 * 2^{-2} = 1.50000 - 0.50000 * 0.25 = 1.375 \\z_3 &= z_2 + a_2 = -26.5651 + 14.0362 = -12.5288\end{aligned}$$

$z < 0$

$$\begin{aligned}x_4 &= x_3 + y_3 * 2^{-3} = 0.87500 + 1.37500 * 0.125 = 1.04688 \\y_4 &= y_3 - x_3 * 2^{-3} = 1.37500 - 0.87500 * 0.125 = 1.26563 \\z_4 &= z_3 + a_3 = -12.5288 + 7.12502 = -5.40379\end{aligned}$$

$z < 0$

$$\begin{aligned}x_5 &= x_4 + y_4 * 2^{-4} = 1.04688 + 1.26563 * 0.0625 = 1.12598 \\y_5 &= y_4 - x_4 * 2^{-4} = 1.26563 - 1.04688 * 0.0625 = 1.2002 \\z_5 &= z_4 + a_4 = -5.40379 + 3.57633 = -1.82746\end{aligned}$$

S využitím 16 kroků algoritmu CORDIC určete $\sin(\pi/4)$

$z < 0$ $x_6 = x_5 + y_5 \cdot 2^{-5} = 1.12598 + 1.20020 \cdot 0.03125 = 1.16348$
 $y_6 = y_5 - x_5 \cdot 2^{-5} = 1.20020 - 1.12598 \cdot 0.03125 = 1.16501$
 $z_6 = z_5 + a_5 = -1.82746 + 1.78991 = -0.0375464$

$z < 0$ $x_7 = x_6 + y_6 \cdot 2^{-6} = 1.16348 + 1.16501 \cdot 0.015625 = 1.18169$
 $y_7 = y_6 - x_6 \cdot 2^{-6} = 1.16501 - 1.16348 \cdot 0.015625 = 1.14683$
 $z_7 = z_6 + a_6 = -0.0375464 + 0.895174 = 0.857627$

...

$z < 0$ $x_{15} = x_{14} + y_{14} \cdot 2^{-14} = 1.16438 + 1.16449 \cdot 6.10352e-005 = 1.16445$
 $y_{15} = y_{14} - x_{14} \cdot 2^{-14} = 1.16449 - 1.16438 \cdot 6.10352e-005 = 1.16442$
 $z_{15} = z_{14} + a_{14} = -0.00263825 + 0.00349706 = 0.000858811$

$z > 0$ $x_{16} = x_{15} - y_{15} \cdot 2^{-15} = 1.16445 - 1.16442 \cdot 3.05176e-005 = 1.16442$
 $y_{16} = y_{15} + x_{15} \cdot 2^{-15} = 1.16442 + 1.16445 \cdot 3.05176e-005 = 1.16445$
 $z_{16} = z_{15} - a_{15} = 0.000858811 - 0.00174853 = -0.000889718$

$$\cos(z) = x_{16} * K = 1.16442 * 0.607253 = 0.707096$$
$$\sin(z) = y_{16} * K = 1.16445 * 0.607253 = 0.707118$$

S využitím CORDIC určete $\sin(\pi/4)$ modifikovaná verze, $x_0 = K$

$$x_0 = 0.607253, y_0 = 0.0, z_0 = \pi/4 = 45^\circ$$

$z > 0$

$$\begin{aligned} x_1 &= x_0 - y_0 * 2^0 = 0.60725 - 0.00000 * 1 = 0.607253 \\ y_1 &= y_0 + x_0 * 2^0 = 0.00000 + 0.60725 * 1 = 0.607253 \\ z_1 &= z_0 - a_0 = 45 - 45 = 0 \end{aligned}$$

$z > 0$

$$\begin{aligned} x_2 &= x_1 - y_1 * 2^{-1} = 0.60725 - 0.60725 * 0.5 = 0.303626 \\ y_2 &= y_1 + x_1 * 2^{-1} = 0.60725 + 0.60725 * 0.5 = 0.910879 \\ z_2 &= z_1 - a_1 = 0 - 26.5651 = -26.5651 \end{aligned}$$

$z < 0$

$$\begin{aligned} x_3 &= x_2 + y_2 * 2^{-2} = 0.30363 + 0.91088 * 0.25 = 0.531346 \\ y_3 &= y_2 - x_2 * 2^{-2} = 0.91088 - 0.30363 * 0.25 = 0.834973 \\ z_3 &= z_2 + a_2 = -26.5651 + 14.0362 = -12.5288 \end{aligned}$$

$z < 0$

$$\begin{aligned} x_4 &= x_3 + y_3 * 2^{-3} = 0.53135 + 0.83497 * 0.125 = 0.635718 \\ y_4 &= y_3 - x_3 * 2^{-3} = 0.83497 - 0.53135 * 0.125 = 0.768554 \\ z_4 &= z_3 + a_3 = -12.5288 + 7.12502 = -5.40379 \end{aligned}$$

$z < 0$

$$\begin{aligned} x_5 &= x_4 + y_4 * 2^{-4} = 0.63572 + 0.76855 * 0.0625 = 0.683753 \\ y_5 &= y_4 - x_4 * 2^{-4} = 0.76855 - 0.63572 * 0.0625 = 0.728822 \\ z_5 &= z_4 + a_4 = -5.40379 + 3.57633 = -1.82746 \end{aligned}$$

S využitím CORDIC určete $\sin(\pi/4)$ modifikovaná verze, $x_0 = K$

$z < 0$ $x_6 = x_5 + y_5 * 2^{-5} = 0.68375 + 0.72882 * 0.03125 = 0.706528$
 $y_6 = y_5 - x_5 * 2^{-5} = 0.72882 - 0.68375 * 0.03125 = 0.707455$
 $z_6 = z_5 + a_5 = -1.82746 + 1.78991 = -0.0375464$

$z < 0$ $x_7 = x_6 + y_6 * 2^{-6} = 0.70653 + 0.70745 * 0.015625 = 0.717582$
 $y_7 = y_6 - x_6 * 2^{-6} = 0.70745 - 0.70653 * 0.015625 = 0.696415$
 $z_7 = z_6 + a_6 = -0.0375464 + 0.895174 = 0.857627$

...

$z < 0$ $x_{15} = x_{14} + y_{14} * 2^{-14} = 0.70707 + 0.70714 * 6.10352e-005 = 0.707117$
 $y_{15} = y_{14} - x_{14} * 2^{-14} = 0.70714 - 0.70707 * 6.10352e-005 = 0.707096$
 $z_{15} = z_{14} + a_{14} = -0.00263825 + 0.00349706 = 0.000858811$

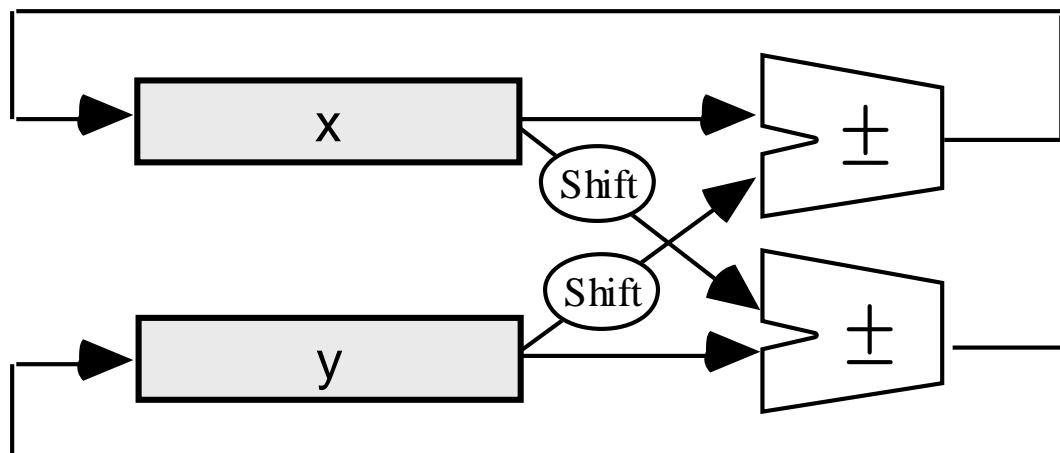
$z > 0$ $x_{16} = x_{15} - y_{15} * 2^{-15} = 0.70712 - 0.70710 * 3.05176e-005 = 0.707096$
 $y_{16} = y_{15} + x_{15} * 2^{-15} = 0.70710 + 0.70712 * 3.05176e-005 = 0.707118$
 $z_{16} = z_{15} - a_{15} = 0.000858811 - 0.00174853 = -0.000889718$

$$\cos(z) = x_{16} = 0.707096$$

$$\sin(z) = y_{16} = 0.707118$$

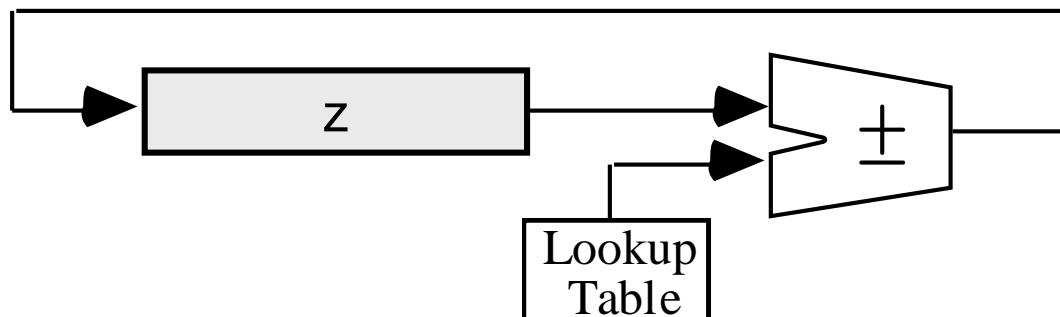
CORDIC v HW

3x Register 2x Barrel Shifter 3x Adder/Subtractor



$$\begin{aligned}
 x_{i+1} &= x_i - d_i y_i 2^{-i} \\
 y_{i+1} &= y_i + d_i x_i 2^{-i} \\
 z_{i+1} &= z_i - d_i \tan^{-1} 2^{-i} \\
 &= z^{(i)} - d_i T_i
 \end{aligned}$$

$$d_i = \{-1, 1\}$$



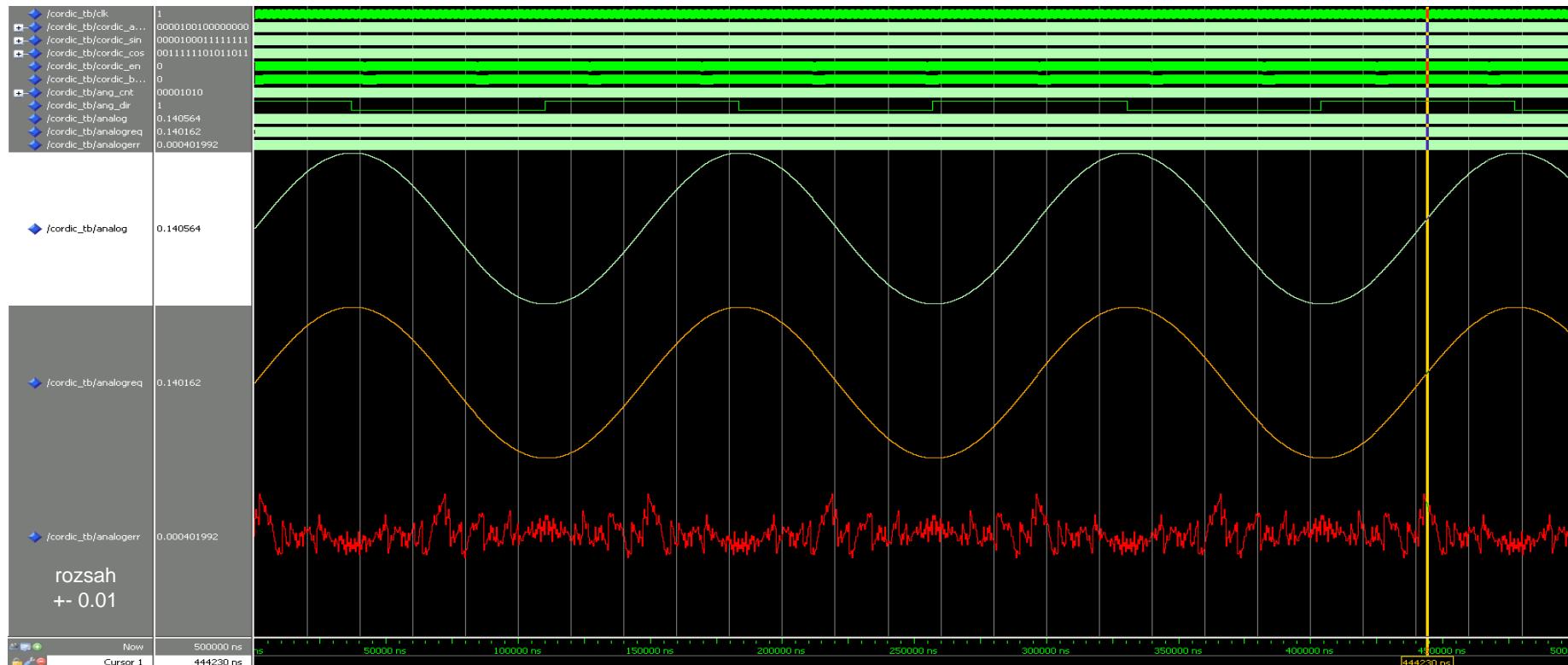
1x ROM

LUT má n položek pro n -bitovou přesnost

CORDIC v HW

(generátor sinusového signálu ve VHDL)

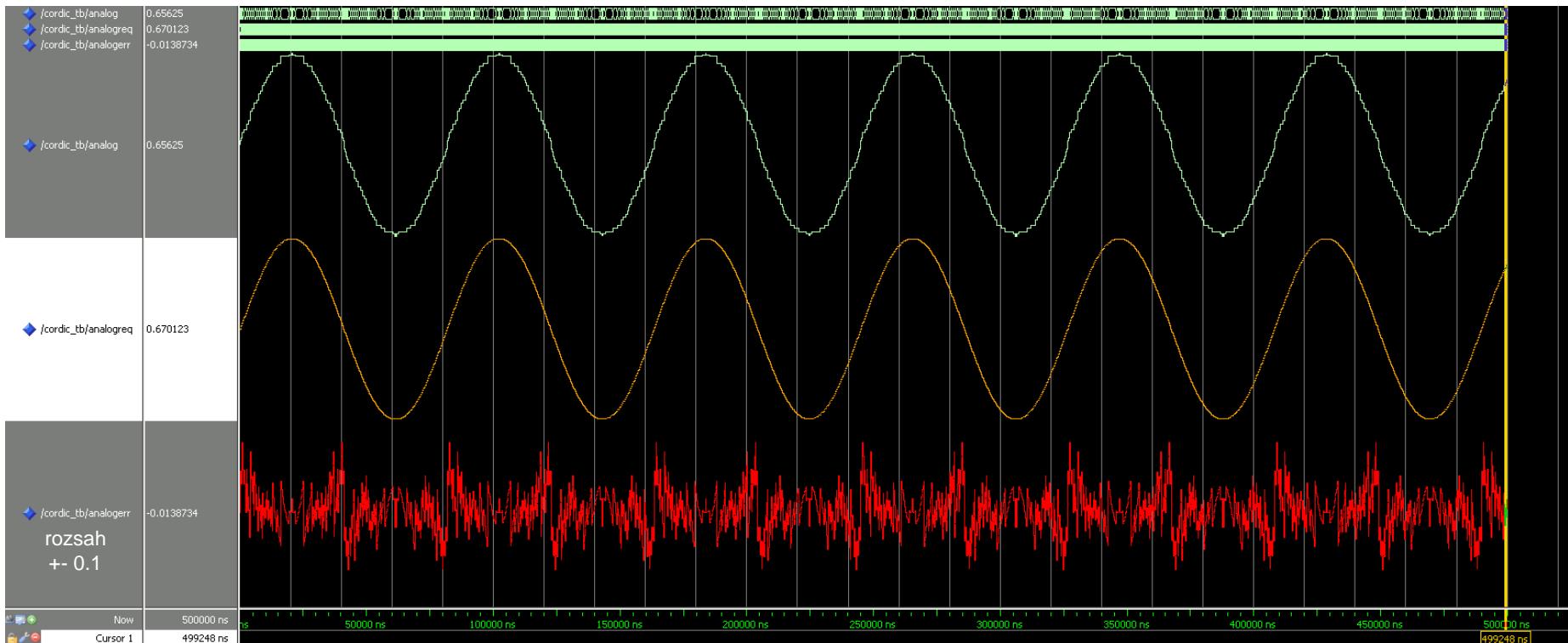
rozlišení 16 bitů, 16 kroků CORDIC



CORDIC v HW

(generátor sinusového signálu ve VHDL)

rozlišení 8 bitů, 8 kroků CORDIC



Úkoly

- S využitím Newtonovy iterační metody odvodte iterační vzorec pro výpočet třetí odmocniny.
- Jakou operaci lze realizovat, pokud použijeme Newtonovu iterační metodu a funkci $f(x) = (x-1+1/d)/(x-1)$, určete iterační předpis pro danou funkci.
- S využitím CORDIC vypočítejte hodnotu $\tan(20^\circ)$.
- Jaká je hodnota agregační konstanty pro 4, 8, 16 iterací CORDIC.
- Určete počet kroků potřebný k získání hodnoty výrazu $24/16$ použijeme-li Newtonovu iterační metodu a Goldschmidtův algoritmus.

Výkonnost, RVP a spolehlivost

INP - cvičení 9

Zdeněk Vašíček, 2015
vasicek@fit.vutbr.cz

Relativní výkonnost

- Zvolíme referenční počítač
- Zvolíme sadu testovacích úloh (tzv. benchmarky)
- Spočítáme relativní doby výpočtu úloh na daném počítači (tj. vzhledem k referenčnímu počítači)
- Z relativních dob můžeme určit a použít ke srovnání:
 - aritmetický průměr
 - vážený průměr
 - geometrický průměr
- Obvykle je používán **geometrický průměr** (viz sady testovacích úloh SPEC, Whetstone, Dhrystone, ...)

Sada testovacích úloh SPEC89

Benchmarks	Type	Application Description
001.gcc	INT ¹	GNU C compiler
008.espresso	INT	PLA optimizing tool
013.spice2g6	FP ²	Circuit simulation and analysis
015.doduc	FP	Monte Carlo simulation
020.nasa7	FP	Seven floating-point kernels
022.li	INT	LISP interpreter
023.eqntott	INT	Conversions of equations to truth table
030.matrix300	FP	Matrix solutions
042fpppp	FP	Quantum chemistry application
047.tomcatv	FP	Mesh generation application

INT = Integer intensive benchmark.

FP = Double-precision floating-point benchmark.

$$spec_{int\ 89} = \sqrt[4]{\prod_{i=1}^4 t_{r_i}} \quad spec_{fp\ 89} = \sqrt[6]{\prod_{i=1}^6 t_{r_i}}$$

Srovnání výkonnosti dvou systémů s využitím SPEC89

SPEC_{int89} (4 programy), SPEC_{fp89} (6 programů)

Počítač Úloha	VAX11/780 (s)	DEC3100 (s)	SPARC (s)	DEC3100 (relativní)	SPARC (relativní)
GCC	1482	145	138,9	10,22	10,67
Espresso	2266	194	254	11,68	8,92
Li	6206	480	689,5	12,93	9,08
Eqntott	1101	99	113,5	11,12	9,7
Spice	23951	2500	2875,5	9,58	8,33
Doduc	1863	208	374,1	8,96	4,98
NASA7	20093	1646	2308,2	12,21	8,71
Matrix300	4525	749	409,3	6,04	11,06
Fpppp	3038	292	387,2	10,4	7,85
Tomcatv	2649	260	469,8	10,19	5,64
Referenční počítač			SPEC-FX	11,45	9,57
			SPEC-FP	9,36	7,49

DEC3100 je výkonnější než SPARC

Sada testovacích úloh SPEC CPU2006

<http://www.spec.org/cpu2006/results/>

SPECINT 2006 (12 benchmarků)

Benchmark	Language	Application Area
400.perlbench	C	Programming Language
401.bzip2	C	Compression
403.gcc	C	C Compiler
429.mcf	C	Combinatorial Optimization
445.gobmk	C	Artificial Intelligence: Go
456.hmmer	C	Search Gene Sequence
458.sjeng	C	Artificial Intelligence: chess
462.libquantum	C	Physics / Quantum Computing
464.h264ref	C	Video Compression
471.omnetpp	C++	Discrete Event Simulation
473.astar	C++	Path-finding Algorithms
483.xalancbmk	C++	XML Processing

SPECFP 2006 (17 benchmarků)

Benchmark	Language	Application Area
410.bwaves	Fortran	Fluid Dynamics
416.gamess	Fortran	Quantum Chemistry.
433.milc	C	Physics / Quantum Chromodynamics
434.zeusmp	Fortran	Physics / CFD
435.gromacs	C, Fortran	Biochemistry / Molecular Dynamics
436.cactusADM	C, Fortran	Physics / General Relativity
437.leslie3d	Fortran	Fluid Dynamics
444.namd	C++	Biology / Molecular Dynamics
447.deallII	C++	Finite Element Analysis
450.soplex	C++	Linear Programming, Optimization
453.povray	C++	Image Ray-tracing
454.calculix	C, Fortran	Structural Mechanics
459.GemsFDTD	Fortran	Computational Electromagnetics
465.Tonto	Fortran	Quantum Chemistry
470.lbm	C	Fluid Dynamics
481.wrf	C, Fortran	Weather
482.sphinx3	C	Speech recognition

$$\text{SPEC}_{\text{int}2006} = \sqrt[12]{\prod_{i=1}^{12} t_{rel_i}} \quad t_{rel_i} = \frac{t_{ref_i}}{t_{act_i}}$$

$$\text{SPEC}_{\text{fp}2006} = \sqrt[17]{\prod_{i=1}^{17} t_{rel_i}} \quad t_{rel_i} = \frac{t_{ref_i}}{t_{act_i}}$$

Výhodnost koupě nového CPU

Procesor je využit z 50%, zbytek času čeká na I/O operace.
Cena procesoru tvoří 1/3 ceny počítače.

Je výhodné kupit 5x rychlejší CPU za 5x vyšší cenu?

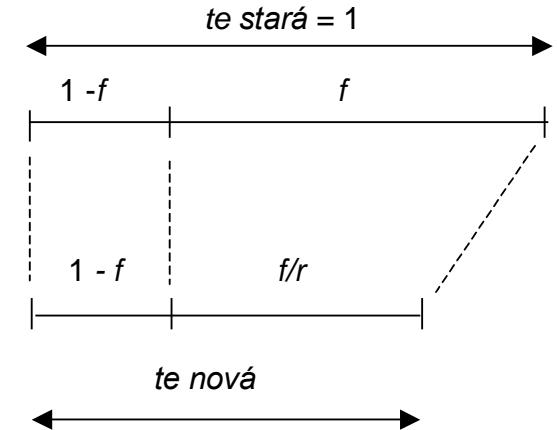
$$f = 0.5, \quad r = 5$$

zrychlení:

$$\begin{aligned} V_r &= 1 / ((1-f) + f/r) = 1 / ((1-0.5) + 0.5/5) \\ &= 1.66 \end{aligned}$$

násobek ceny počítače s novým CPU:

$$2/3 + 5 \cdot 1/3 = 2.33$$



Amdahlův zákon: celkové zrychlení je rovno
 $V_r = 1 / ((1 - f) + f/r)$

Počítač je 2.33x dražší, ale jen 1.66x výkonnější.
Rychlejší CPU není výhodné kupit.

Příkon CPU

Mějme procesor Athlon XP 1700+, který pracuje na frekvenci 1.466 GHz a napětí 1.75V. Jak se změní výkonnost a spotřeba, pokud procesor podtaktujeme na 1GHz (1.4 GHz, 1.266 GHz) a snížíme napájecí napětí na 1.15V (1.6, 1.45V).

Předpokládejme, že spotřebu na 1.75V lze approximovat vztahem
 $P = 20 + 0.03 * f$ v MHz

Pro příkon platí $P \sim C U^2 f$

$$P_{puv} \sim C 1.75^2 1.466$$

$$P_{novy} \sim C 1.15^2 1.000$$

$$P_{puv} = 20 + 0.03 * 1466 = 64 \text{ W}$$

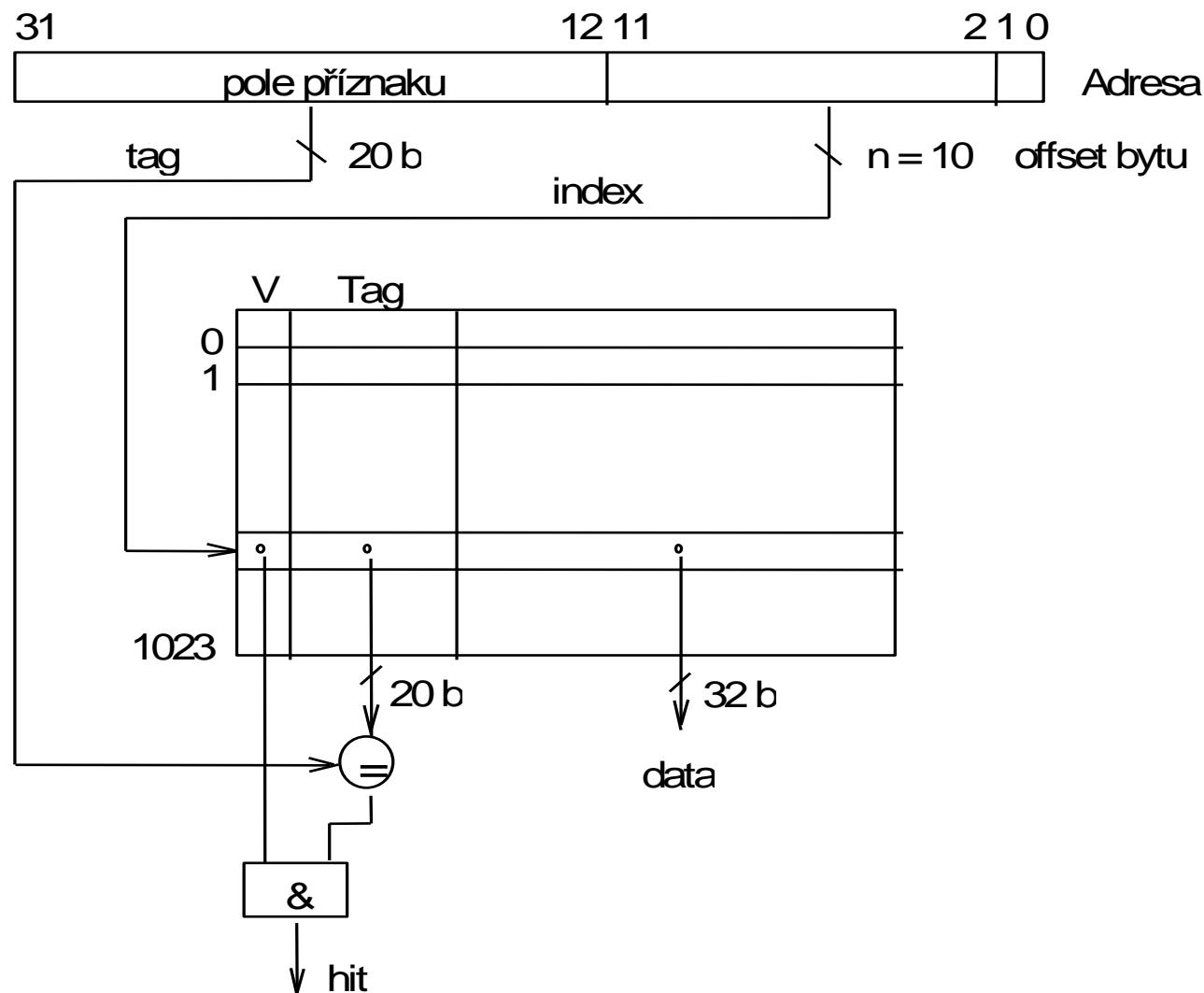
$$\begin{aligned} P_{novy}/P_{puv} &= 1.15^2 1.000 / 1.75^2 1.466 = 0.29 \\ \Rightarrow P_{novy} &= 64 * 0.29 = 18.56 \text{ W} \end{aligned}$$

$$f_{novy}/f_{puv} = 1.000 / 1.466 = 0.68$$

Spotřeba klesla o 71% zatímco výkon CPU pouze o 32%.

CACHE (RVP)

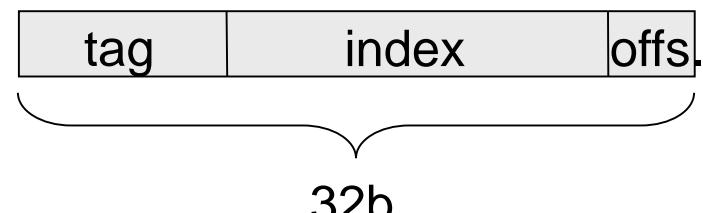
přehled koncepce s 32-bitovou adresou a přímým mapováním



Příklad 1

Zadání: Kolik kb paměti bude potřeba pro realizaci jednocestné RVP obsahující 1024 bloků? Uvažujte velikost bloku 64 bitů a 32-bitové adresování.

Řešení



Počet položek ... 1024

Počet bitů na jednu položku

Blok dat ... 64 bitů

Uložení tagu ... $32 - \log_2(1024) - \log_2(64/8) = 19$ bitů

Valid bit ... 1 bit

Celkem ... 84 bitů

Velikost RVP ... $1024 * 84 = 84$ kb

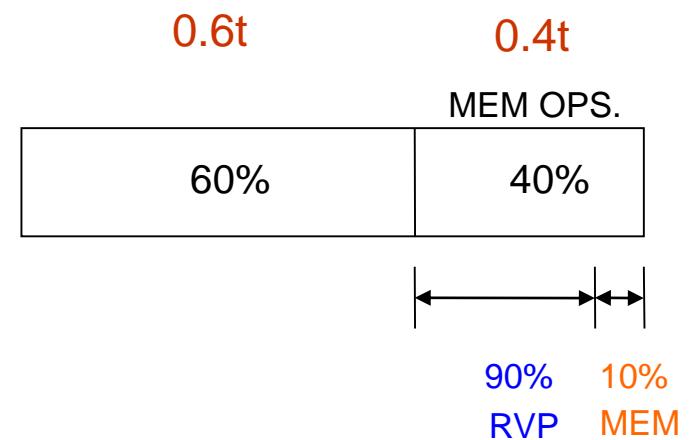
Příklad 2

Zadání: Paměťové operace tvoří 40 % času z celkové doby běhu programu. O kolik procent se zkrátí doba běhu programu při použití RVP, když $p_{hit} = 0,9$ a doba přístupu do RVP je rovna 1/10 doby přístupu do paměti?

$$t_{\text{puvodní}} = t$$

$$t_{\text{RVP}} = 0,6 \cdot t + 0,4 \cdot (0,1 \cdot t + 0,9 \cdot 0,1 \cdot t) = 0,676 \cdot t$$

$$t_{\text{puvodní}} / t_{\text{RVP}} = t / 0,676 \cdot t = 1,479$$



Doba běhu programu se zkrátila o 47,9 %.

Příklad 3

Zadání: Mezi procesor a paměť s dobou přístupu 500 ns byla vložena RVP s dobou přístupu 50 ns. Jaká musí být pravděpodobnost zásahu p_{hit} do RVP, aby bylo dosaženo alespoň dvojnásobné urychlení paměťových operací?



$$t_{\text{původní}} = t_1 \cdot n = 500 \cdot n$$

$$t_2 = 50 \text{ ns} \quad t_1 = 500 \text{ ns}$$

$$t_{\text{RVP}} = (t_2 \cdot p_{hit} + t_1 \cdot (1 - p_{hit})) \cdot n = ((t_2 - t_1) \cdot p_{hit} + t_1) \cdot n = (500 - 450 \cdot p_{hit}) \cdot n$$

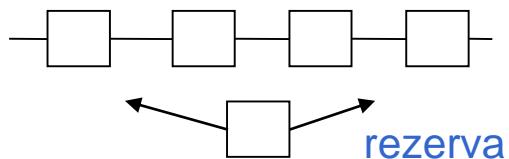
$$\frac{t_{\text{původní}}}{t_{\text{RVP}}} = 500 \cdot n / (500 - 450 \cdot p_{hit}) \cdot n = 500 / (500 - 450 \cdot p_{hit}) = 2$$

$$p_{hit} = 500 / 900 = 5/9 = 0,55$$

Pro dosažení dvojnásobného urychlení paměťových operací musí být pravděpodobnost zásahu do RVP alespoň 55 %.

Zálohování kol automobilu

Odvodte pravděpodobnost bezporuchové činnosti $R_C(t)$ kol automobilu, je-li zadána $R(t)$ jednoho kola.



$$R_C = \sum_{i=0}^1 \binom{5}{i} R^{5-i}(t) [1 - R(t)]^i$$

$$= \binom{5}{0} R^5(t) + \binom{5}{1} R^4(t) [1 - R(t)]$$

$$R(t)_{m \text{ z } n} = \sum_{i=0}^{n-m} \binom{n}{i} R^{n-i}(t) [1 - R(t)]^i$$

$$= 5R^4(t) - 4R^5(t)$$

kde n je počet všech modulů (5)

m je počet požadovaných (4)
fungujících modulů

i je počet přijatelných poruch
(0 nebo 1)

$$\binom{n}{i} = \frac{n!}{(n-i)!i!}$$

Zálohování kol automobilu

- Provozuschopné stavy:

- ABCDR

$$R^5(t)$$

- ABCD-

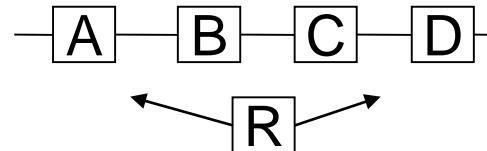
- ABC-R

- AB-DR

$$R^4(t)(1-R(t))$$

- A-CDR

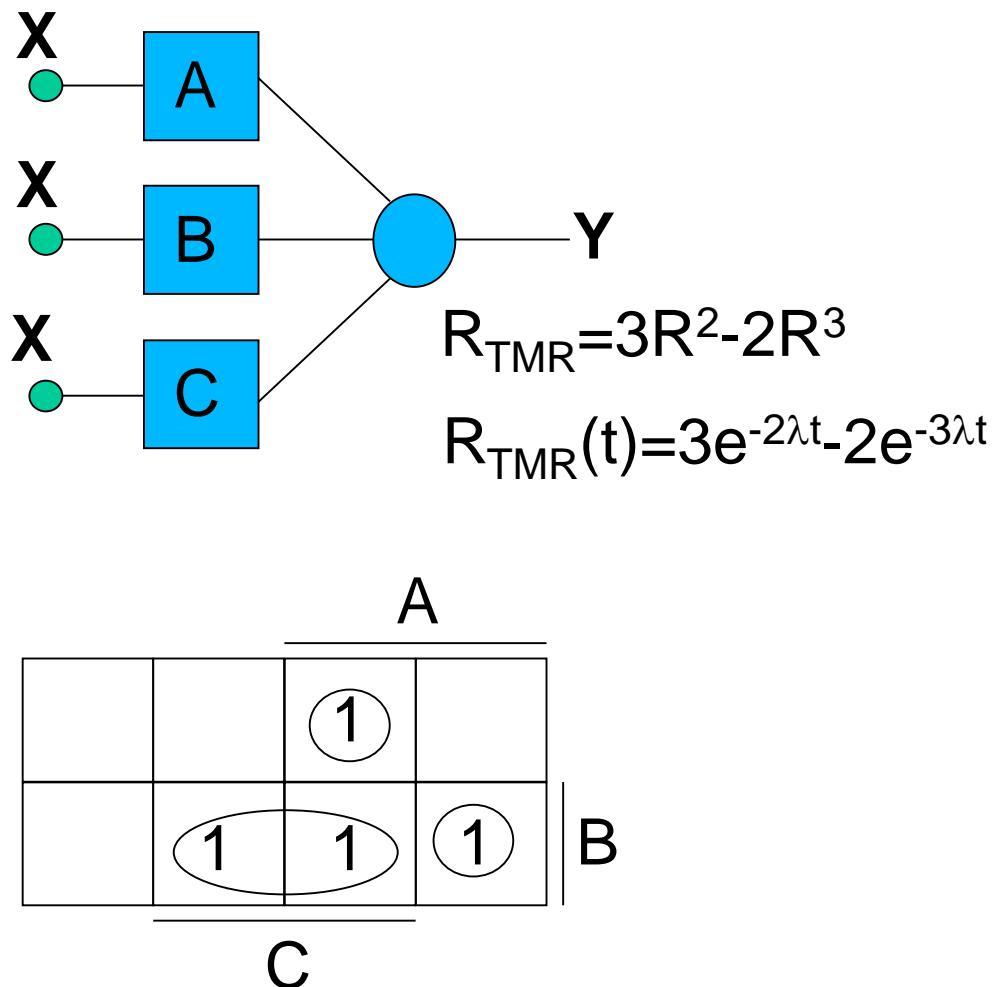
- -BCDR



- Pravděpodobnost bezporuchové činnosti $R_C(t)$:

Využití modifikované Kaurnaughovy mapy

Odvození R(t) pro TMR

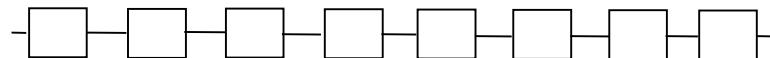


Module Outputs			Voter Output
A	B	C	
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Výpočet výrobní spolehlivosti (výtěžnosti)

EPROM 2764 bez zálohování

Předpokládejme, že 1KB paměť EPROM (2708) má pravděpodobnost bezporuchové činnosti $x = 0,9$. Jakou výtěžnost má výroba EPROM 2764 skládající se z osmi modulů 2708? Uvažujme, že přídavné obvody (adresové obvody, budiče, sběrnice, atd.) mají 7x větší plochu než-li paměťová matice.



$$R_M = x^8 = 0.9^8 = 0.43.$$

Přídavné obvody mají spolehlivost x^7 .

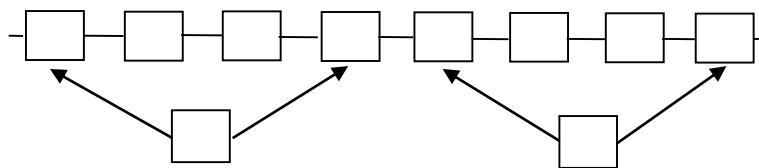
$$\text{Pak } R_C = x^8 \cdot x^7 = x^{15} = 0.20$$

To je nepřijatelně nízká výtěžnost (80% zmetkovitost), použijme tedy zálohování.

Výpočet výrobní spolehlivosti (výtěžnosti)

EPROM 2764 se zálohováním 4-1,4-1

Jaká bude výtěžnost výroby zavedením dvou tzv. vázaných rezerv?



Využijeme vztah pro zálohování 4 z 5.

$$R_C = \sum_{i=0}^1 \binom{5}{i} R^{5-i}(t) [1 - R(t)]^i = \binom{5}{0} R^5(t) + \binom{5}{1} R^4(t) [1 - R(t)]$$

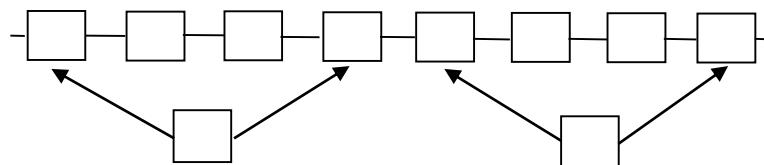
Po dosazení:

$$R_C = (x^5 + 5x^4[1-x])^2 x^7 = (5x^4 - 4x^5)^2 x^7 = 0,4$$

Výpočet výrobní spolehlivosti (výtěžnosti)

EPROM 2764 se zálohováním 4-1,4-1

Výpočet s využitím Karnaughovy mapy a **disjunktním** pokrytí.



Výčet provozuschopných stavů (5 modulů):

1111 1, 0111 1, 1011 1, 1101 1, 1110 1, 1111 0

A diagram consisting of a 6x8 grid of squares. The grid is bounded by thick black lines. Labels are placed around the grid:

- Label **B** is positioned above the top row of the grid.
- Label **A** is positioned above the top two rows of the grid.
- Label **E** is positioned to the left of the first column of the grid.
- Label **C** is positioned to the right of the last column of the grid.
- Label **D** is positioned below the bottom row of the grid.
- Label **D** is positioned below the bottom two rows of the grid.

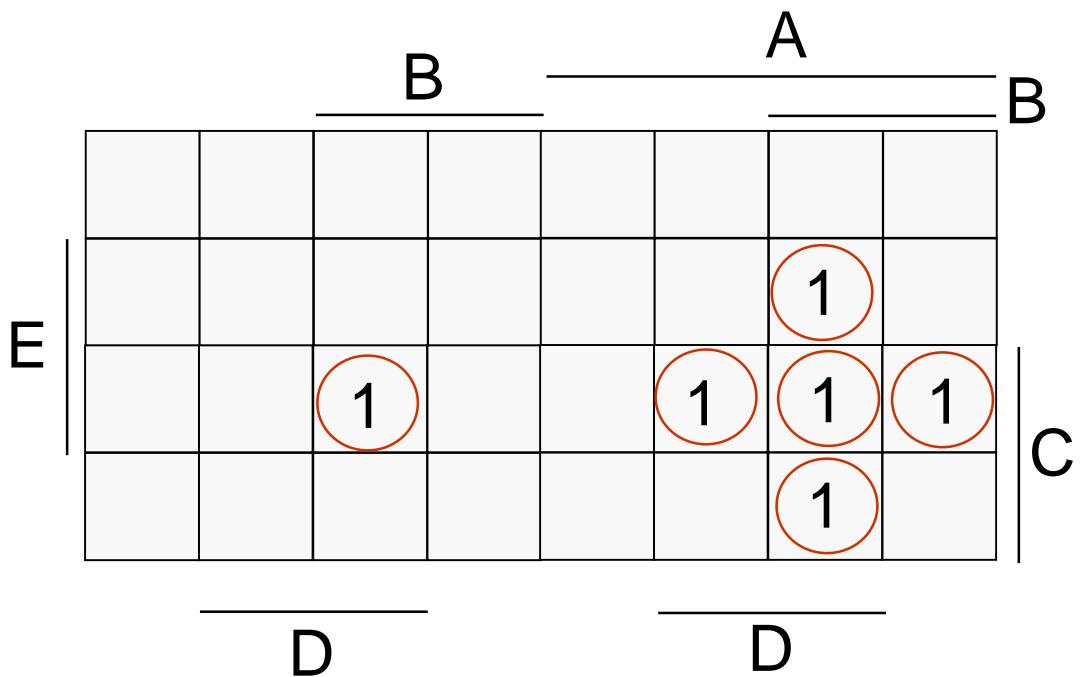
Specific cells in the grid are highlighted with red circles containing the number **1**:

- The cell at the intersection of the 4th column and the 4th row from the bottom-left is circled.
- The cell at the intersection of the 6th column and the 4th row from the bottom-left is circled.
- The cell at the intersection of the 7th column and the 4th row from the bottom-left is circled.
- The cell at the intersection of the 6th column and the 5th row from the bottom-left is circled.
- The cell at the intersection of the 7th column and the 5th row from the bottom-left is circled.
- The cell at the intersection of the 8th column and the 5th row from the bottom-left is circled.
- The cell at the intersection of the 6th column and the 6th row from the bottom-left is circled.
- The cell at the intersection of the 7th column and the 6th row from the bottom-left is circled.
- The cell at the intersection of the 8th column and the 6th row from the bottom-left is circled.

$$\begin{aligned}
 R_{4+1} &= ABDE + ABC(1-D)E + \\
 &\quad + ABCD(1-E) + A(1-B)CDE + \\
 &\quad + (1-A)BCDE \\
 &= x^4 + 4x^4(1-x) = 5x^4 - 4x^5
 \end{aligned}$$

Výpočet výrobní spolehlivosti (výtěžnosti)

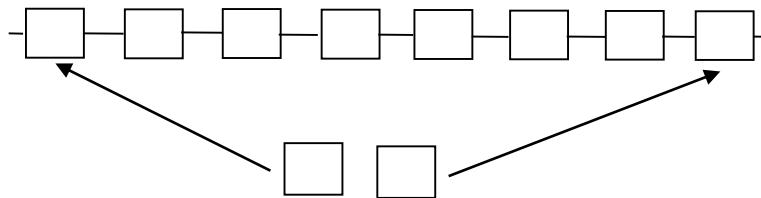
EPROM 2764 se zálohováním 4-1,4-1



Výpočet výrobní spolehlivosti (výtěžnosti)

EPROM 2764 se zálohováním 8-2

Jaká bude výtěžnost výroby zavedením dvou tzv. volných rezerv?



Využijeme vztah pro zálohování 8 z 10.

$$R(t)_{8z10} = \sum_{i=0}^2 \binom{10}{i} R^{10-i}(t) [1 - R(t)]^i$$
$$= \binom{10}{0} R^{10}(t) + \binom{10}{1} R^9(t) [1 - R(t)] + \binom{10}{2} R^8(t) [1 - R(t)]^2$$
$$\binom{n}{i} = \frac{n!}{(n-i)!i!}$$

$$R_C = (x^{10} + 10x^9[1-x] + 45x^8[1-x]^2)x^7 = 0,44$$