Principy návrhu - RISC

Principy návrhu

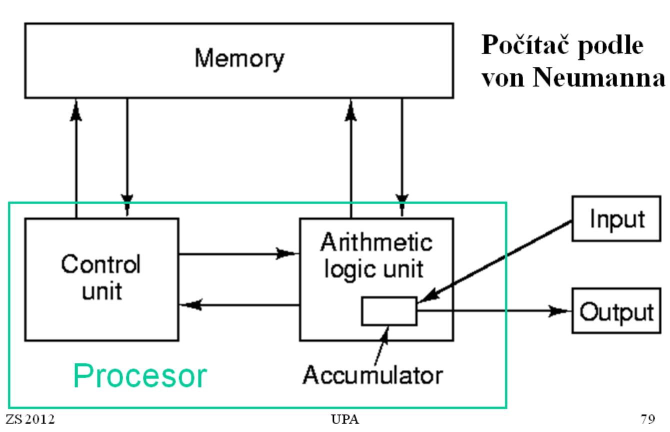
CISC versus RISC

RISC:

- Instrukce jsou přímo prováděny hardwarem
- Maximální průchodnost instrukcí (ILP)
- Jednoduché instrukce (snadné dekódování)
- Přístup do paměti jen instrukcemi load/store
- Velké množství registrů
- Pipelining

Von Neumann – organizace počítače, organizace CPU

Organizace počítače



Jednotky KB, MB, GB

Opakování: Některé základní definice

- Kilobyte – 2^{10} nebo1024 bytů
- Megabyte – 2^{20} nebo 1048576 bytů
- Gigabyte – 2^{30} nebo 1073741824 bytů
- Terabyte – 2^{40} nebo 1099511627776 bytů
- Petabyte – 2^{50} nebo 1024 terabytů
- Exabyte – 2^{60} nebo 1024 petabytů

Pozn: Rozlišujte KB a kB, MB a mB, atd. !

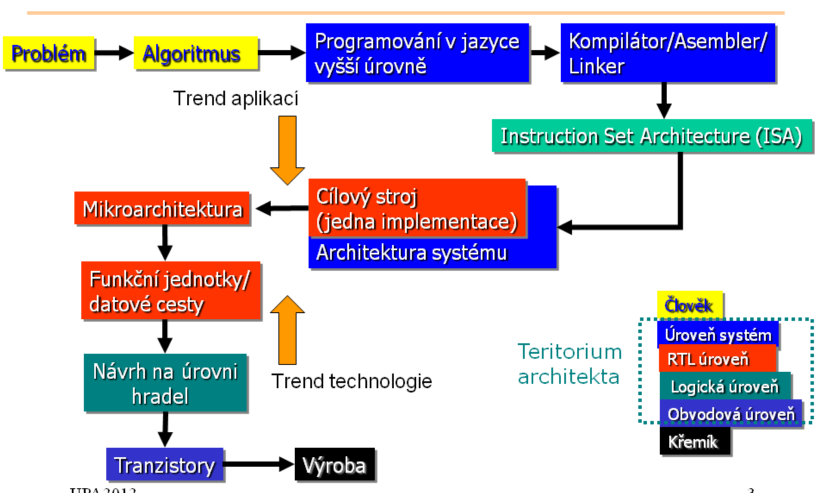
Princip abstrakce, Pružná data, koncepce programu v paměť, princip lokality, paralelní zpracování, kompilace vs. Interpretace, měření výkonu

Závěr - opakování

- < 13 týdnů ke studiu základních koncepcí v CS & CE
  - *Principy abstrakce*, použité ke stavbě systému po vrstvách
  - *„Pružná“ data*: program určuje interpretaci obsahu paměti
  - Koncepce *programu v paměti*: instrukce jsou také data
  - Princip *lokality*, využíván v paměťové hierarchii
  - Větší výkon využitím *paralelního zpracování* (pipeline)
  - *Kompilace* versus *interpretace*
  - Principy a problémy *měření výkonu*

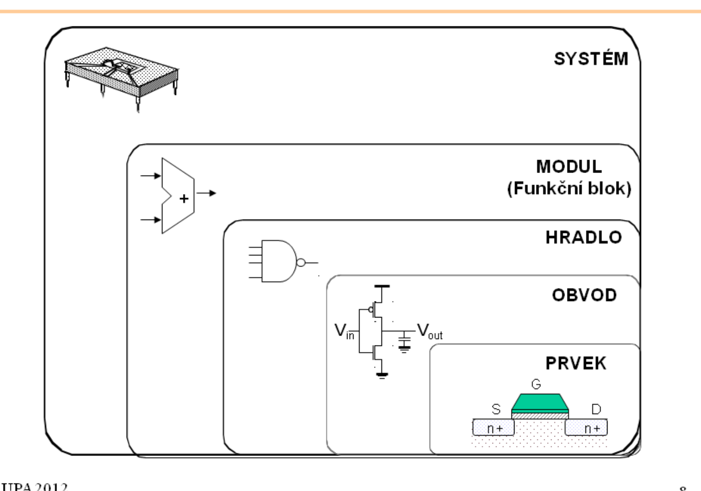
Rozklad problému

Rozklad problému



Návrh na úrovni abstrakce

Opakování: Návrh úrovní abstrakce



Základní metriky návrhu

Základní metriky návrhu

- Funkčnost
- Cena
  - Konstantní náklady - návrhové prostředky, infrastruktura
  - Variabilní náklady – cena vlastního obvodu, zapouzdření, testy
- Spolehlivost, robustnost
  - Odstup šumu
  - Šumová imunita
  - MTBF
- Výkonnost
  - Rychlost (zpoždění)
  - Spotřeba energie
- Doba potřebná pro uvedení na trh - „Time-to-market“

Cena integrovaného obvodu

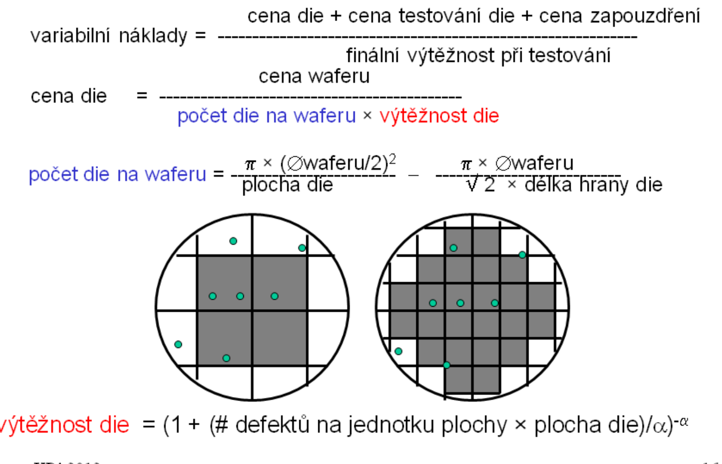
Cena integrovaného obvodu

- Konstantní náklady
  - Fixní náklady pro vytvoření návrhu
    - vlastní návrh
    - verifikace návrhu
    - generování masek
  - Ovlivněno složitostí návrhu a produktivitou návrhářů
  - Jsou více významné pro malé objemy výroby
- Variabilní náklady – úměrné objemu výroby
  - zpracování křemíku
    - úměrné také ploše čipu
  - zapouzdření
  - testování

$$\text{cena jednoho IC} = \text{variabilní náklady jednoho IC} + \frac{\text{konstantní náklady}}{\text{objem výroby}}$$

Variabilní náklady – cena die, počet die na waferu, výtěžnost die (počítání die)

Variabilní náklady

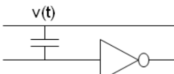


Spolehlivost robustnost – kapacitní induktivní (indukční) vazba, šum

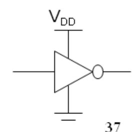
Spolehlivost a robustnost

Šum v digitálních integrovaných obvodech

- Šum – nežádoucí změny napětí a proudu na logických uzlech
- Mezi dvěma vodiči, umístěnými v těsné blízkosti
  - kapacitní vazba
    - změna napětí na jednom vodiči ovlivňuje signál sousedního vodiče
    - přeslechy
  - induktivní vazba
    - změna proudu v jednom vodiči ovlivňuje signál sousedního vodiče
- Šum napájení a zemí
  - může ovlivnit signálové úrovně v hradle



i(t)



UPA 2012 37

Směrovost – klíčové metriky

Směrovost

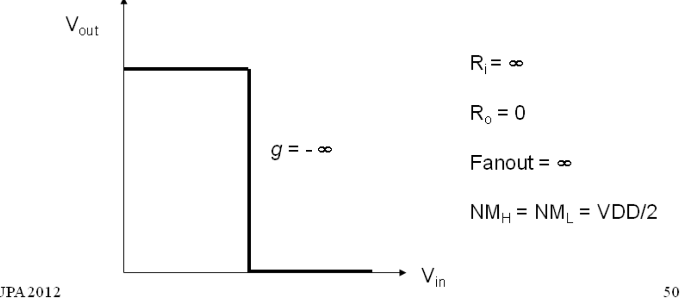
- Hradlo musí být *jednosměrné*: změny výstupní úrovně nesmí ovlivňovat žádný vstup toho samého obvodu
  - V reálných obvodech je *úplná* jednosměrnost iluze (např. již kvůli zmíněným kapacitním vazbám mezi vstupem a výstupem)
- Klíčové metriky: *výstupní impedance* budiče a *vstupní impedance* vstupu
  - ideálně, výstupní impedance budiče je nulová a
  - vstupní impedance vstupu je nekonečná

UPA 2012 47

Ideální invertor – ideální hradlo

Ideální invertor

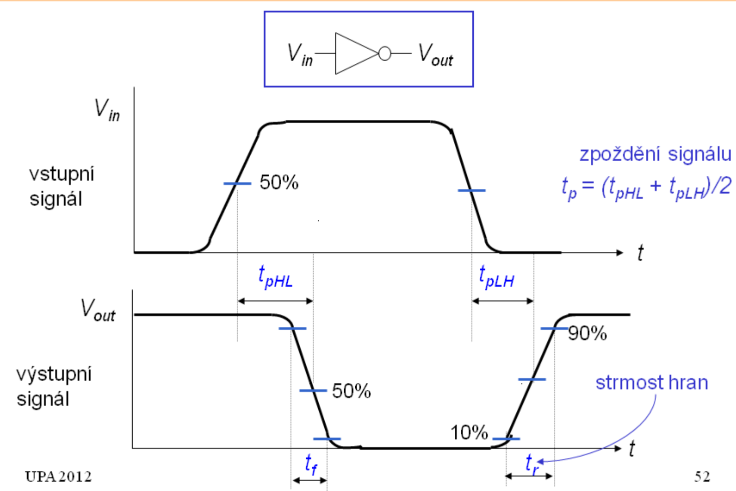
- Ideální hradlo by mělo mít
  - nekonečný zisk v přechodové oblasti
  - přepínací úroveň umístěnou ve středu logického zdvihu
  - horní a dolní pásy logických úrovní stejné a rovné polovině zdvihu
  - nulovou výstupní a nekonečnou vstupní impedanci



UPA 2012 50

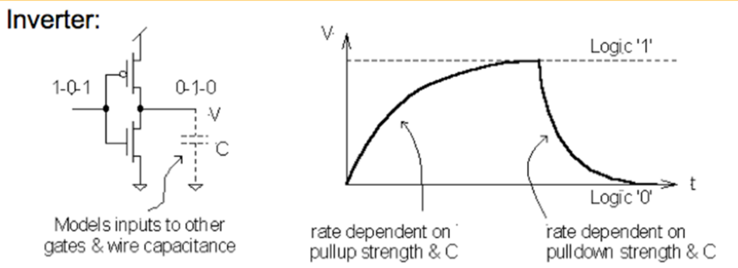
Definice zpoždění

Definice zpoždění



Logický zisk

Logický zisk



"Logický zisk": Počet vstupů hradel buzených výstupem hradla.

Buzení více hradel zpomaluje přechod signálu. Buzení vodičů zpomaluje přechod signálu.

Pravdivostní tabulka

Operace Booleovy algebry

- 0 & 1: jediné hodnoty pro proměnné a funkce
  B = {0,1} se nazývají *Booleovská čísla*
- Funkce NOT : $f(A) = \begin{cases} 1 \text{ platí-li } A = 0 \\ 0 \text{ platí-li } A = 1 \end{cases}$
- Pravdivostní tabulky
  - Úplně definují Booleovskou funkci
  - n proměnných => 2^n řádek v pravdivostní tabulce => 2^{2^n} různých funkcí, protože 2^n řádek mohu vyplnit právě tolika způsoby
    - Př.: Existuje 16 Booleovských funkcí dvou proměnných
  - Zkrácený zápis: uvedení řádek s nenulovým výstupem

Booleova algebra – zákony, distribuční, DeMorgan, Absorpční, Asociační, Distributivní, Komutativní

Booleova Algebra

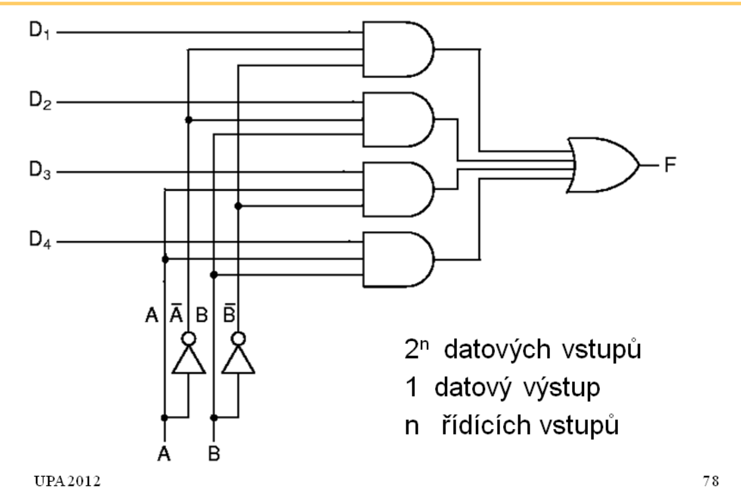
- Základní operátory: OR (součet), AND (součin), NOT
- Zákony Booleovy algebry:

Name	AND form	OR form
Identity law	1A = A	0 + A = A
Null law	0A = 0	1 + A = 1
Idempotent law	AA = A	A + A = A
Inverse law	$A\overline{A} = 0$	$A + \overline{A} = 1$
Commutative law	AB = BA	A + B = B + A
Associative law	(AB)C = A(BC)	(A + B) + C = A + (B + C)
Distributive law	A + BC = (A + B)(A + C)	A(B + C) = AB + AC
Absorption law	A(A + B) = A	A + AB = A
De Morgan's law	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$

UPA 2012 74

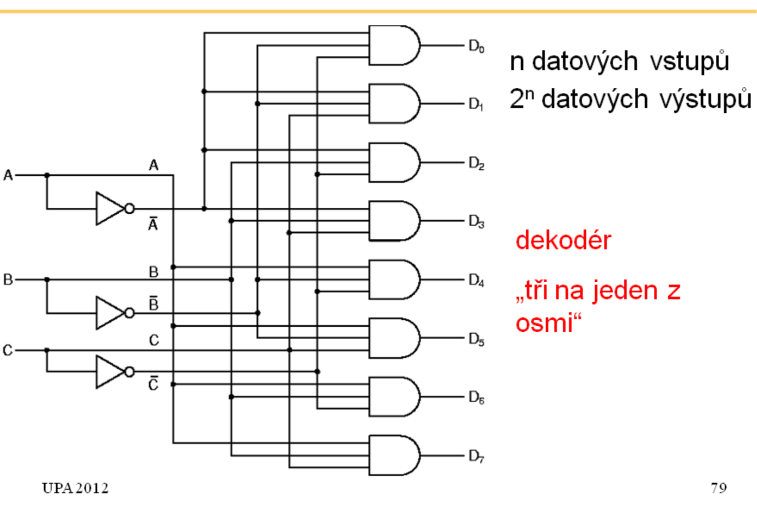
Multiplexor

Multiplexer



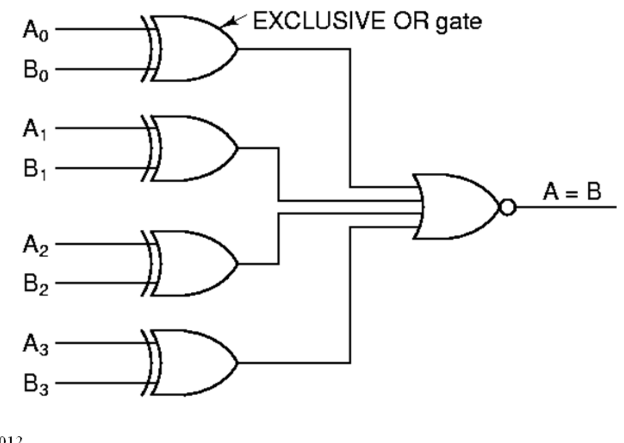
Dekodér

Dekodér



Komparátor

Komparátor (logický!!!)



UPA 2012 80

Výkon počítače – doba výpočtu, doba odezvy, propustnost

Výkon počítače

- Hlavním kritériem uživatele může být *doba odezvy* (nebo *doba výpočtu*) – kolik času uplyne od startu do ukončení úlohy?
  - Důležité pro jednotlivé uživatele.
- Šéf výpočetního střediska se naopak bude asi zajímat nejvíce o *propustnost* – kolik celkové práce vykoná počítač za určitou dobu?
  - Důležité pro šéfa výpočetního centra.
- Co je tedy vlastně výkon?
- Potřebujeme prostředky pro měření výkonu výpočetních prostředků od systémů třídy „embeddeded“ přes stolní počítače až po výkonné servery.

ZS 2012 UPA 5

Hodnocení systému se sdílením času – timesharing, doba CPU, doba CPU uživatele, doba CPU systému

Hodnocení systémů se sdílením času?

- Většina moderních počítačů dnes pracuje v režimu sdílení času - *timesharing* (uživatel zpracovává současně více úloh).
- Systém je většinou navržen tak, aby se maximalizovala propustnost. Nesleduje se tedy minimální doba výpočtu jednotlivého programu.
- *Doba CPU* – doba, kterou procesor věnuje zpracování jedné úlohy a nezahrnuje I/O aktivity, ani čas, který procesor věnuje zpracování jiných úloh.
- *Doba CPU* se dále dělí na *dobu CPU uživatele* a *dobu CPU systému*. Doba CPU uživatele je čas procesoru, věnovaný jen dané úloze, doba CPU systému se vztahuje na dobu, kdy procesor provádí akce operačního systému, vázající se na provádění uživatelské úlohy.

ZS 2012 UPA 7

Koncepce hodnocení výkonnosti – vyhodnocení výkonu procesoru, relativní výkon

Koncepce hodnocení výkonnosti

- Hodnocení výkonu počítače je založeno na:
  - Propustnosti
  - Době výpočtu (*execution time, elapsed time*) ← Doba od startu do ukončení. (nezapočítává se doba I/O a zpracování jiných programů)
- Hodnocení výkonu komponent a vrstev
- Vyhodnocení výkonu procesoru
  - Založeno na době výpočtu *(execution time)* **programu:**

$výkon_X = 1/\ doba\ výpočtu_X$

Relativní výkon: $n = výkon_X / výkon_Y$

$n > 1$ => *X je n krát **rychlejší než** Y*

- Terminologie
  - **Zlepšit** výkon: = **zvýšit** výkon
  - **Zlepšit** dobu výpočtu: = **zkrátit** dobu výpočtu

ZS 2012 UPA 9

Výpočet výkonu CPU – CPU time, IC, CPI

Výpočet výkonu CPU

- **CPU time** = počet cyklů CPU * doba cyklu
  = počet cyklů CPU / frekvence hodin
  - počet cyklů CPU = IC * CPI
    - IC: počet instrukcí (počet instrukcí na program)
    - CPI: střední hodnota počtu cyklů na instrukci
- **CPU time = IC * CPI * doba cyklu**

$$\frac{\text{sekundy}}{\text{program}} = \frac{\text{instrukce}}{\text{program}} * \frac{\text{cykly hodin}}{\text{instrukce}} * \frac{\text{sekundy}}{\text{cykly hodin}}$$

- **Cykly hodin CPU** = $\Sigma_i (CPI_i * IC_i)$
  - IC_i: počet instrukcí třídy *i*
  - CPI_i: cykly, které jsou třeba k provedení instrukcí třídy i

ZS 2012 UPA 12

Příklad počítání doby výpočtu

Příklad

- Program proběhne za 10 sekund na 2 GHz procesoru. Návrhář chce postavit počítač, který provede stejný program za 6 sekund zvýšením hodinové frekvence.
- Vlivem změny časových poměrů ale naroste také střední hodnota CPI 1.2 krát.
- Jaká frekvence hodin má být použita ?

$$\frac{10}{6} = \frac{IC * CPI / (2\ GHz)}{IC * 1.2\ CPI / (X\ GHz)}$$

(stávající doba výpočtu)
(cílová doba výpočtu)

Řešením pro X získáme ***X = 4 GHz***

ZS 2012 UPA 17

Třídy instrukcí

Třídy instrukcí

- Instrukce lze s určitým zjednodušením rozdělit do *instrukčních tříd* – instrukce se společnými výkonnostními charakteristikami (počtem cyklů/instrukci).
- Například instrukce, které přistupují do paměti obvykle vyžadují větší počet cyklů než jednoduchá instrukce součtu.
- Jestliže pro určitý procesor existuje *n* instrukčních tříd, potom můžeme určit počet cyklů hodin, které vyžaduje zpracování specifického programu …

$$\text{\# hodinových cyklů CPU} = \sum_{i=1}^{n} (CPI_i \times C_i)$$

C_i # instrukcí třídy i, které byly provedeny

CPI_i střední počet cyklů na instrukci (pro třídu i)

N # instrukčních tříd

ZS 2012 UPA 18

Příklad třídy instrukcí

Příklad – třídy instrukcí

Porovnání dvou kódových sekvencí kompilátoru

Třída instrukcí i	CPI_i pro třídu instrukcí i
A	1
B	2
C	3

Kódová sekvence	Počet instrukcí (IC_i) pro třídu instrukcí		
	A	B	C
1	2	1	2
2	4	1	1

- Která kódová sekvence provede největší počet instrukcí?
- Která je rychlejší? $S_1 = 2 . 1 + 1 . 2 + 2 . 3 = 10$
  $S_2 = 4 . 1 + 1 . 2 + 1 . 3 = 9$

Druhá sekvence instrukcí je rychlejší, i když jich obsahuje více.

ZS 2012 UPA 19

Co určuje výkon CPU

Co určuje výkon CPU

CPU time = počet instrukcí x CPI x doba_cyklu

	počet instrukcí	CPI	doba_cyklu
Algoritmus	X	X	
Programovací jazyk	X	X	
Kompilátor	X	X	
ISA	X	X	X
Organizace jádra		X	X
Technologie			X

ZS 2012 UPA 21

Příklad výpočtu výkonu CPU

Příklad

Operace	Četnost	CPI_i	Četnost x CPI_i			
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
			Σ = 2.2	1.6	2.0	1.95

- Jak se zrychlí počítač, zredukuje-li rychlejší cache dobu cyklu na dva takty?
  CPU time new = 1.6 x IC x CC proto 2.2/1.6 dává o 37.5% vyšší rychlost
- Jakého výsledku dosáhneme využitím predikce, která zkrátí cykl skoků na polovinu?
  CPU time new = 2.0 x IC x CC proto 2.2/2.0 dává o 10% vyšší rychlost
- Co kdyby bylo možno vykonat dvě ALU instrukce současně?
  CPU time new = 1.95 x IC x CC pak 2.2/1.95 a dostaneme o 12.8% vyšší rychlost

MIPS – výpočet MIPS

MIPS

- Kdykoliv se jednalo o měření výkonu počítače, bylo snahou použít čas jako měřítko výkonu. Velmi často to však vedlo ke špatným výsledkům a chybné interpretaci.
- Nejpopulárnějším měřítkem je počet instrukcí za jednotku času (měřeno v MIPS).
- MIPS je velmi jednoduchá koncepce … (příliš jednoduchá!)

$$MIPS = \frac{Počet_instrukcí}{Doba_výpočtu \times 10^6}$$

MIPS - příklad

MIPS - příklad

- Nechť má počítač následující třídy instrukcí…

Třída instrukcí	CPI pro tuto třídu instrukcí
Třída A	1 cykl / instrukci
Třída B	2 cykly / instrukci
Třída C	3 cykly / instrukci

- Program bude přeložen dvěma kompilátory:
  - **Kompilátor 1**: 5 instrukcí A, 1 instrukci B a 1 instrukci C
  - **Kompilátor 2**: 10 instrukcí A, 1 instrukci B a 1 instrukci C
- Celkový počet cyklů pro každou sekvenci…
  - **$Cykly_1$** = (5 x 1 + 1 x 2 + 1 x 3) x 10^9 = 10 x 10^9 cyklů
  - **$Cykly_2$** = (10 x 1 + 1 x 2 + 1 x 3) x 10^9 = 15 x 10^9 cyklů
- Předpokládejme, že počítač běží na 1 GHz, kód kompilátoru 1 zabere 10 sekund, kód kompilátoru 2 zabere 15 sekund. (Počet cyklů pro každou sekvenci instrukcí, dělený počtem cyklů za sekundu.)

MIPS – příklad (pokr.)

- Nyní můžeme počítat MIPS pro každou sekvenci instrukcí. MIPS určíme…

$$MIPS = \frac{Počet_instrukcí}{Doba_výpočtu \times 10^6}$$

- Pro každou posloupnost dostaneme …

$$MIPS_1 = \frac{(5+1+1)\times 10^9}{10\times 10^6} = 700 \qquad MIPS_2 = \frac{(10+1+1)\times 10^9}{15\times 10^6} = 750$$

- To znamená, že kompilátor 2 má vyšší ohodnocení MIPS, ale prakticky se sekvence 1 provádí rychleji. To je hlavní problém s použitím MIPS jako metriky pro výkon.

MFLOPS – vzorec výpočtu

MFLOPS

- Jiným alternativním měřítkem k době výpočtu jsou MFLOPS (*million floating-point operations per second)* Tato metrika se určí …

$$MFLOPS = \frac{\text{Počet FP operací v programu}}{\text{Doba výpočtu} \times 10^6}$$

- *Floating-point operace* - sem patří sčítání, odčítání násobení a dělení čísel v pohyblivé řádové čárce.
- Evidentně MFLOPS závisí na zpracovávaném programu. Například kompilátor (který skoro nepoužívá FP operace) by měl nulové ohodnocení na každém stroji.

MIPS a MFLOPS

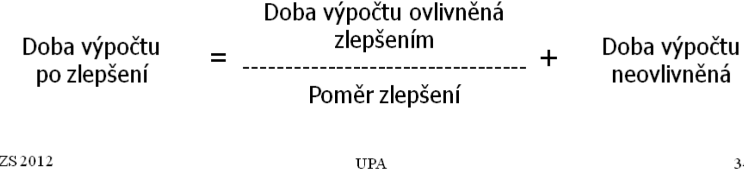
MIPS a MFLOPS

- Metriku MIPS nebo MFLOPS nelze zobecnit tak, aby se dala použít jako jediná pro hodnocení výkonu počítače.
  - Nejvyšší MIPS lze získat výběrem sekvence instrukcí, která minimalizuje dobu výpočtu (výběrem instrukcí, které minimalizují CPI).
  - Nejvyšší MFLOPS lze získat výběrem sekvence instrukcí, která bude obsahovat jen “rychlé” FP operace.
- Takové měřítko neříká o počítači nic, protože nikdo takový program nebude spouštět!

Amdahlův zákon – výpočet, zákon o klesajícím zisku, výpočet zrychlení, grafické znázornění

Amdahlův zákon

- Běžným problémem, který je spojen s měřením výkonu je myšlenka zlepšit určitý aspekt stroje, při čemž se očekává, že ve stejném poměru, ve kterém bude provedeno dílčí zlepšení se zvýší i celkový výkon stroje.
- Amdahlův zákon říká…

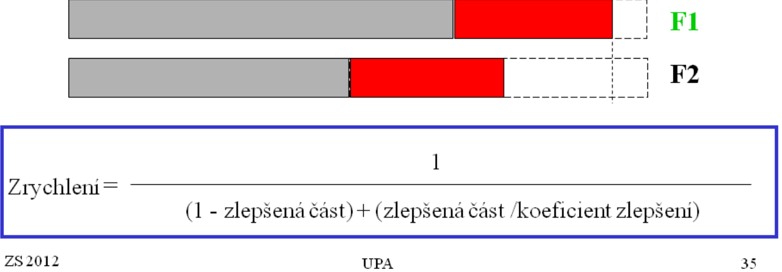


Amdahlův zákon

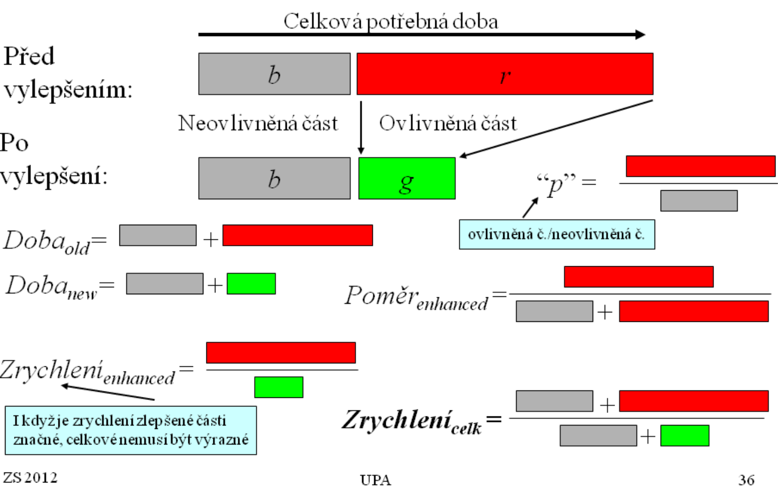
- Zákon o klesajícím zisku (návratnosti investic)

$$\text{Zrychlení} = \frac{\text{Doba výpočtu před zlepšením}}{\text{Doba výpočtu po zlepšení}}$$

Příklady: **F1**: zlepšení na 75% doby, **F2**: zlepšení na 50% doby



Grafické znázornění Amdahlova zákona



Benchmarky – druhy

Programy pro vyhodnocení výkonu

- Benchmarky měří různé stránky výkonu systému a jeho složek
- Ideální situace: známé programy (*workload*)
- Benchmarky
  - **Reálné programy**
  - Jádra
  - Hry-benchmarky
  - Syntetické benchmarky

- Risk: návrh může být podřízen požadavkům benchmarku
  - (částečné) řešení: použití **reálných programů**
    - Inženýrské nebo vědeckotechnické aplikační programy
    - Softwarové vývojové prostředky
    - Provádění transakcí
    - Aplikační programy (Office, ...)

Hodnocení kritéria výkonu – reprodukovatelnost, shrnut výkonových parametrů, aritmetický, vážený aritmetický, harmonický průměr

Kritéria hodnocení výkonu

- **Reprodukovatelnost**
  - Zahrnuje konfiguraci hardware / software
  - Podmínky vyhodnocovacího procesu
- **Shrnutí výkonových parametrů**
  - *Celkový čas*: čas CPU + čas I/O + ostatní doby
  - *Aritmetický průměr*: $AM = (1/n) * \Sigma\ exec_time_i$
  - *Vážený aritmetický průměr*: $WM = \Sigma\ w_i * exec_time_i$
  - *Harmonický průměr*: $HM = n / \Sigma\ (1/rate_i)$
  - *Geometrický průměr*: $GM = (\Pi\ exec_time_ratio_i)^{1/n}$

$$\frac{GM(X_i)}{GM(Y_i)} = GM\left(\frac{X_i}{Y_i}\right)$$

Podstatné !!!

Celková doba výpočtu

Přístup 1: Celková doba výpočtu

- Nejjednodušším přístupem je porovnat celkovou dobu výpočtu pro všechny programy ve zkušební úloze.
- Tímto postupem dostaneme …

$$\frac{Výkon_B}{Výkon_A} = \frac{DobaVýpočtu_A}{DobaVýpočtu_B} = \frac{1001}{110} = 9.1$$

- Pro uvedenou skladbu programů dostaneme, že B je 9.1 krát rychlejší než A pro programy 1 a 2 dohromady.

Poznámka: poměr 9.1x platí POUZE pro uvedené dva programy, každý spuštěný pouze jednou. O poměru výkonu počítačů A a B nelze říci vlastně nic.

Aritmetický střední hodnota

Přístup 2: Aritmetická střední hodnota

- Jinou možností jak vyhodnotit benchmark je určení střední doby výpočtu.
- Tímto způsobem dostaneme…

$$AM = \frac{1}{n}\sum_{i=1}^{n} Doba_i$$

$$AM_A = \frac{1}{2}\sum_{i=1}^{2} Doba_i = \frac{1}{2}(1001) = 500.5 \qquad AM_B = \frac{1}{2}\sum_{i=1}^{2} Doba_i = \frac{1}{2}(110) = 55$$

- Pro uvedenou kombinaci programů dostaneme, že střední doba výpočtu pro B je 9.1 krát rychlejší než pro A, což souhlasí s předchozím výpočtem celkové doby výpočtu.

Vážená aritmetický střední hodnota

Přístup 3: Vážená aritmetická střední hodnota

- Aritmetická střední hodnota předpokládá, že počet spuštění každého programu je stejný (přispívají ke střední hodnotě stejnou měrou). Není-li to pravda, lze každému programu přiřadit váhu.
- Tak dostaneme …

$$WAM = \frac{1}{n}\sum_{i=1}^{n} w_i * Doba_i$$

- Jestliže program 1 představuje 20% celkové zátěže a program 2 80%, bude váhový koeficient programu 1 roven 0.2 a váhový koeficient programu 2 bude 0.8.

Chyby a omyly

Chyby a omyly

- Ignorování Amdahlova zákona
- Použití MIPS jako metriky pro výkon
- Použití aritmetického průměru (AM) normalizovaných dob CPU (ratios)
- Užití hardwarově nezávislých metrik
  - Velikost kódu jako měřítko rychlosti
- Syntetické benchmarky predikují výkon
  - Nerespektují chování reálných programů
- Geometrický průměr poměrných dob CPU je proporcionální celkové době výpočtu [Nikoliv!!]

Vyberte nejlepší metodu, ideální benchmark, zvyšování výkonu pro ISA, spolehlivé měřítko výkonu

Závěr

- Různá kritéria
  - čas CPU, čas I/O, ostatní doby, celkový čas
- Vyberte nejlepší metodu prezentace
  - Aritmetická stř. hodnota pro dobu výpočtu
  - Geometrická stř. hodnota pro poměrný výkon
- Nezapomeňte na Amdahlův zákon
  - Velké náklady **<==>** a často jen malá zlepšení
  - Je třeba zahrnout i náklady na vývoj

Závěr (pokr.)

- Výkonnost je vázána na konkrétní programy !
- Čas CPU: jediný adekvátní parametr pro měření výkonu
- Pro danou ISA lze výkon zvyšovat:
  - nárůstem hodinové frekvence (bez zhoršení CPI)
  - zlepšením organizace procesoru, která snižuje CPI
  - zlepšením kompilátoru, které snižuje CPI a/nebo IC
- Vaše úlohy (workload) = Váš *ideální benchmark*
- Nesmíte vždy věřit všemu co se píše, ani v technické oblasti!

Závěr (pokr.)

- Přesné měření výkonu počítače je vlastně obtížné !
- Jediným spolehlivým měřítkem je doba výpočtu. Všechny ostatní metriky jsou nepřesné. Buď nerespektují dobu výpočtu a nebo se soustřeďují jen na málo významný aspekt výkonnosti a chybně jej interpretují jako celkový výkon.
- Benchmarky musí respektovat předpokládanou zátěž. Jedině tak jsou použitelné.
- Vážený aritmetický průměr dob výpočtu vybraných benchmarků lze použít pro odhad předpokládaného výkonu pro danou zátěž. Výsledky lze pak použít pro porovnání různých strojů.

Architektura, Architektura počítače

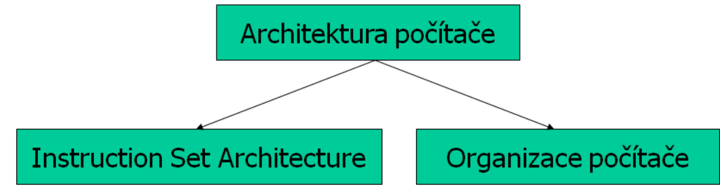
Definice

- Co je "architektura?"
  - "The art or science or building…the art or practice of designing and building structures…" (Webster Dictionary)
  - "včetně plánu, návrhu, konstrukce a dekorace…" (American College Dictionary)

- Co je "architektura počítače?"
  - "… architekturou rozumíme strukturu modulů, jak jsou organizovány v počítačovém systému …" (H. Stone, 1987)
  - "Architektura počítače je interface mezi strojem a softwarem" (Andris Padges, architekt IBM 360/370)

Hlavní oblasti počítače – rozdělení

Dvě hlavní oblasti



- Architekturu počítače můžeme rozdělit do dvou oblastí:
  - ISA (Instruction Set Architecture)
  - Organizace počítače

ISA – instruction set architecture, definice, příklad ISA MIPS R3000

ISA (Instruction Set Architecture)

- ISA (Instruction Set Architecture) je definována jako:

  Atributy [počítačového] systému z hlediska programátora, to znamená strukturu a funkční chování, na rozdíl od organizace a řízení datových toků, logického návrhu a fyzické implementace. (Amdahl, Blaaw a Brooks 1964)

- ISA zahrnuje takové volby a rozhodnutí, jako například:
  - Datové typy & struktury (kódování & reprezentace)
  - Instrukční soubor
  - Instrukční formáty
  - Adresní módy a přístup k datům a instrukce
  - Podmínky výjimek

ISA

- Instrukční soubor lze chápat jako interface mezi softwarem a hardwarem – představuje abstraktní formu hardwaru. Odděluje celou složitost implementačních detailů od software.
- Příklady různých ISA zahrnují…
  - Intel IA-32 (x86)
  - Intel IA-64
  - DEC Alpha
  - MIPS
  - Sparc (a UltraSparc)
- Například softwarový vývojář vyvíjí software pro ISA, např. IA-32. Aktuální hardwarová implementace není rozhodující a může se lišit systém od systému, pokud je zachována ISA.

(Z toho důvodu může tentýž program být zpracováván na Pentiu i na Pentiu 4, což jsou z hlediska stavby velmi rozdílné procesory)

Příklad: ISA procesoru MIPS R3000

Registry (zde 35)

R0 - R31
PC
HI
LO

- Šest hlavních typů instrukcí
  - Čtení/zápis (Load/Store)
  - Aritmetické
  - Skoky a větvení
  - Floating Point operace
  - Správa paměti (Memory Management)
  - Speciální
- Tři formáty instrukcí – všechny o šíři 32 bitů …

opcode	rs	rt	rd	sa	funct
opcode	rs	rt	immediate value		
opcode	jump or branch target				

Organizace počítače

Organizace počítače

- Organizace počítače zahrnuje implementační detaily – jak vlastně hardware doopravdy pracuje…
  - Vlastnosti a výkonové charakteristiky funkčních jednotek (jako např. registrů, ALU, posuv. jednotek, logických jednotek atd.)
  - Propojení těchto komponent
  - Informační toky mezi komponentami
  - Řízení toku informací
  - Kombinace funkčních jednotek pro realizaci ISA
  - Popis RTL (Register Transfer Level)

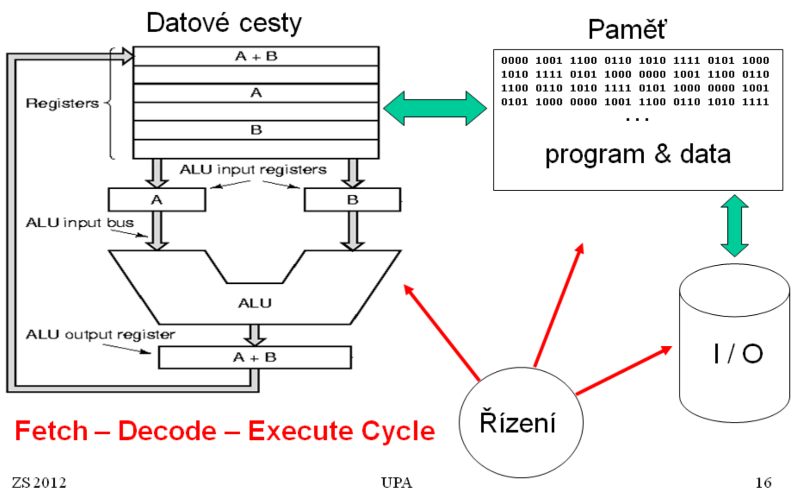
Úrovně ISA

Přehled úrovní ISA

- *ISA: jak se počítač jeví programátoru na úrovni strojního jazyka (kompilátor)*
  - Paměťový model
  - Instrukce
    - Formáty
    - Typy (aritmetické, logické, přesuny dat a řízení výpočtu)
    - Režimy (jádro a uživatel)
  - Operandy
    - Registry
    - Datové typy
    - Adresování
- Dokumenty s formální definicí ISA
  - V9 SPARC a JVM

Výpočet programu

Výpočet programu



Strojový jazyk MIPS – typy instrukcí

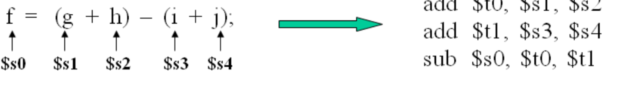
Strojový jazyk MIPS

- Aritmetické instrukce
  - add, sub, mult, div
- Logické instrukce
  - and, or, ssl (shift left), srl (shift right)
- Přesuny dat
  - lw (load), sw (store), lui (load upper immediate)
- Větvení
  - Podmíněné skoky: beq, bne, slt (set on les than)
  - Nepodmíněné skoky: j, jr (jump register), jal (jump and link)

Operandy – registry

Operandy

- Operand nemůže být libovolná proměnná (jako v C)
  - princip KISS – odstraňuje nedostatky CISC
- Registry (Keep It Small and Simple)
  - Omezený počet (32 32-bitových registrů u MIPS)
    - **Zásada_2: Menší je rychlejší**
  - Pojmenování: čísla nebo *jména*
    - $8 - $15 => $t0 - $t7 (vztahuje se na dočasnou proměnnou)
    - $16 - $22 => $s0 - $s8 (vztahuje se na proměnnou z C)
    - Jména zvýší srozumitelnost vašeho kódu



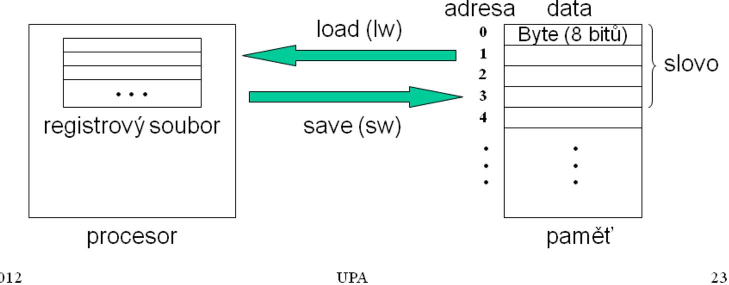
Registry - konvence

Jméno	Číslo registru	Použití	Vyhrazen při call
$zero	0	the constant value 0	n.a.
$at	1	reserved for the assembler	n.a.
$v0-$v1	2-3	value for results and expressions	no
$a0-$a3	4-7	arguments (procedures/functions)	yes
$t0-$t7	8-15	temporaries	no
$s0-$s7	16-23	saved	yes
$t8-$t9	24-25	more temporaries	no
$k0-$k1	26-27	reserved for the operating system	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

Paměť, paměťový model, zarovnání slov, load/store, word 4B

Paměť

- Obsahuje instrukce a data programu
  - Instrukce jsou čteny *automaticky* řadičem
  - Data jsou přesouvána explicitně tam a zpět mezi pamětí a procesorem
- Instrukce přesunu dat

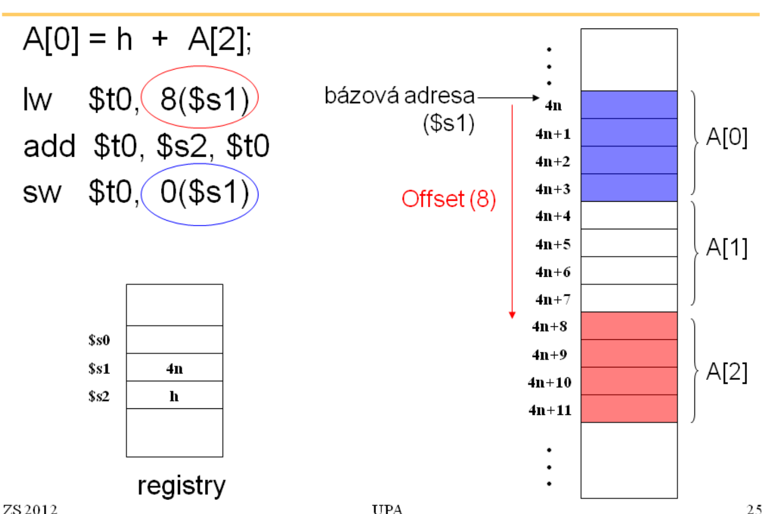


Paměťový model

- Paměť je adresovatelná po bytech (1 byte = 8 bitů)
- Load/Store - jediný přístup k datům v paměti
- Jednotka pro přenos: *word* (4 byty)
  - M[0], M[4], M[4n], …., M[4,294,967,292]
- Slova musí být **zarovnána** !!!
  - Slovo začíná na adresách 0, 4, … 4n
- Adresa je dlouhá 32 bitů
  - 2^{32} bytů nebo 2^{30} slov

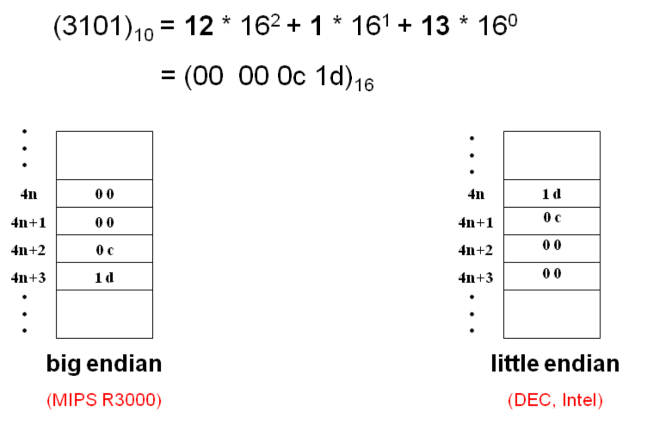
Když A[2] tak začíná na offsetu 2*4 tzn. 8 protože word jsou 4 byty a jedno políčko v poli má délku 4B

Load / Store



Malý velký Endian – Malý nižší řády na nižších adresách, velký nižší řády na vyšších adresách

„Big“ a „Little“ Endian



ISA MIPS přehled

Přehled

- ISA: představuje interface pro hardware / software
  - Principy návrhu, využití
- Instrukce MIPS
  - Aritmetické: add/sub $t0, $s0, $s1
  - Přenos dat: lw/sw $t1, 8($s1) (znamená load/store-word)
- Operandy musí být registry
  - 32 32-bitových registrů
  - $t0 - $t7 („dočasné“) mají adresy $8 - $15
  - $s0 - $s7 („ukládané“) mají adresy $16 - $23
- Paměť: velké, jednorozměrné pole bytů M[2^{32}]
  - Adresa v paměti je index do toho pole bytů
  - Zarovnaná slova: M[0], M[4], M[8], ... M[4,294,967,292]
  - Big/little endian – pořádek bytů ve slově

ZS 2012 UPA 27

Strojový jazyk, 3 (tři) formáty instrukcí R I J, zásada 3, formát R, formát I, formát J

Strojový jazyk

- Všechny instrukce mají stejnou délku (32 bitů)
- Zásada_3: Dobrý návrh vyžaduje dobré kompromisy
  - Stejná délka nebo stejný formát
- Tři různé formáty instrukcí
  - R: formát aritmetických instrukcí
  - I: formát pro přesuny dat, větvení, immediate
  - J: formát skokové instrukce
- add $t0, $s1, $s2
  - 32 bitů ve strojní instrukci
  - Pole pro:
    - Operaci (add)
    - Operandy ($s1, $s2, $t0)

```
101011010010100000000010010110000
000000100100100001000000000100000
100011010010100000000010010110000
```

```
lw   $t0, 1200($t1)
add  $t0, $s2, $t0
sw   $t0, 1200($t1)
```

```
A[300] = h + A[300];
```

ZS 2012 UPA 28

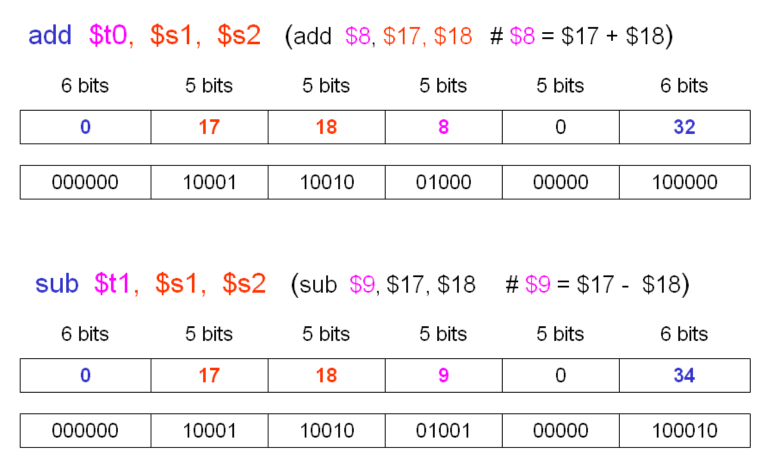
Formáty instrukcí

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
I:	op	rs	rt	address / immediate		
J:	op	target address				

op: základní operace (opcode)
rs: první operand - registr
rt: druhý operand - registr
rd: registr pro uložení výsledku
shamt: délka posuvu
funct: vybírá specifickou variantu operačního kódu (funkční kód)
address: offset pro instrukce typu load/store (+/-2^{15})
immediate: konstanty pro instrukce s přímými operandy („immediate op.“)

ZS 2012 UPA 29

Formát R



add $t0, $s1, $s2 (add $8, $17, $18 # $8 = $17 + $18)

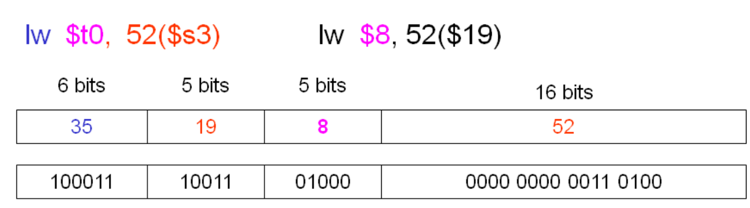
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

sub $t1, $s1, $s2 (sub $9, $17, $18 # $9 = $17 - $18)

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	9	0	34
000000	10001	10010	01001	00000	100010

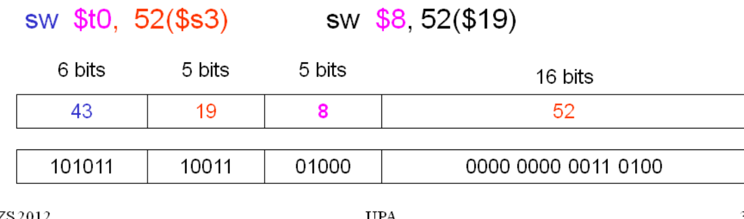
Formát I

lw $t0, 52($s3) lw $8, 52($19)



6 bits	5 bits	5 bits	16 bits
35	19	8	52
100011	10011	01000	0000 0000 0011 0100

sw $t0, 52($s3) sw $8, 52($19)

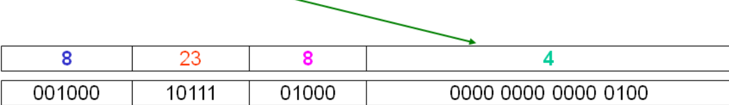


6 bits	5 bits	5 bits	16 bits
43	19	8	52
101011	10011	01000	0000 0000 0011 0100

Immediates – numerické konstanty, zásada 4

Immediates (numerické konstanty)

- Často se používají malé konstanty (50% všech operandů)
  - A = A + 5;
  - C = C – 1;
- Řešení
  - Uložíme typické konstanty do paměti a používáme je
  - Vytvoření HW registrů (např. **$0 nebo $zero**)
- Zásada_4: postavit sdílené sekce co nejrychlejší
- Instrukce MIPS pro konstanty (formát I)
  - addi $t0, $s7, 4 # $t0 = $s7 + 4



8	23	8	4
001000	10111	01000	0000 0000 0000 0100

Aritmetické přetečení – instrukce s přetečením/bez přetečení

Aritmetické přetečení

- Počítače mají omezenou délku slova (např. 32 bitů) a tím také omezenou přesnost

```
  15      1111
 + 3      0011
----     -----
  18     10010
```

- Některé jazyky detekují přetečení (Ada), jiné nikoliv (C)
- MIPS implementuje 2 typy aritmetických instrukcí:
  - Add, sub, and addi: způsobují přetečení
  - Addu, subu, and addiu: nezpůsobují přetečení
- Kompilátory MIPS C generují implicitně instrukce addu, subu, addiu

Logické instrukce,

Logické instrukce

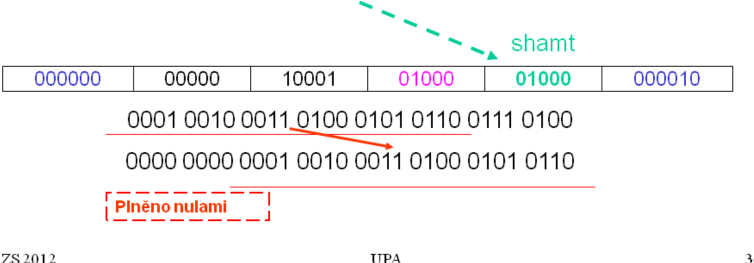
- Bitové operace
  - Obsah registrů je považován za pole o délce 32 bitů a nikoliv za jedno 32-bitové číslo
- Instrukce
  - and, or: 3 operandy jsou registry (formát R)
  - andi, ori: 3. argument je typu *immediate* (formát I)
- Příklad: maskování (andi $t0, $t0, 0xFFF)

```
1011 0110 1010 0100 0011 1101 1001 1010
0000 0000 0000 0000 0000 1111 1111 1111
---------------------------------------
0000 0000 0000 0000 0000 1101 1001 1010
```


Instrukce pro posuv

Instrukce pro posuv

- Posouvá všechny bity registru vlevo/vpravo
  - sll (shift left logical): uprázdněné pozice plní 0
  - srl (shift right logical): uprázdněné pozice plní 0
  - sra (shift right arithmetic): uprázdněné pozice plní znaménkem
- Příklad: srl $t0, $s1, 8 (formát R)



Instrukce násobení a dělení

Násobení a dělení

- Používají se speciální registry (hi, lo)
  - 32-bitů x 32-bitů = 64-bitů
- Mult $s0, $s1

000000	10000	10001	00000	00000	011000

  - hi: horní polovina součinu
  - lo: dolní polovina součinu
- Div $s0, $s1

000000	10000	10001	00000	00000	011010

  - hi: zbytek ($s0 % $s1)
  - lo: podíl ($s0 / $s1)
- Přesun výsledku do obecných registrů:
  - mfhi $s0
  - mflo $s1

000000	00000	00000	10000	00000	010000
000000	00000	00000	10001	00000	010010

Assembler vs strojový jazyk

Assembler vs. strojový jazyk

- Assembler umožňuje přehledný symbolický zápis
  - Mnohem snazší než psát samotná čísla
  - Prvý operand určen k uložení výsledku
  - Makroinstrukce
  - Návěští k identifikaci a pojmenování slov, která obsahují instrukce/data
- Strojní jazyk je základ
  - Operand pro výsledek nemusí být na prvém místě
  - Úsporný formát
- Assembler nahrazuje makroinstrukce (strojní move není!)
  - Move $t0, $t1 (add $t0, $t1, $zero)
- Pro určení výkonu je třeba započítávat reálné instrukce

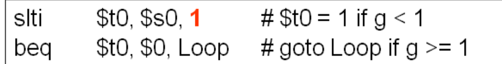
Instrukce porovnání a nastavení!!!

Porovnání

- Programy potřebují často testovat < and >
- Instrukce ***Set on less than***
- **slt** register1, register2, register3
  - register1 = (register2 < register3)? 1 : 0;
- Příklad: if (g < h) goto Less;

```
g: $s0        slt   $t0, $s0, $s1
h: $s1        bne   $t0, $0, Less
```

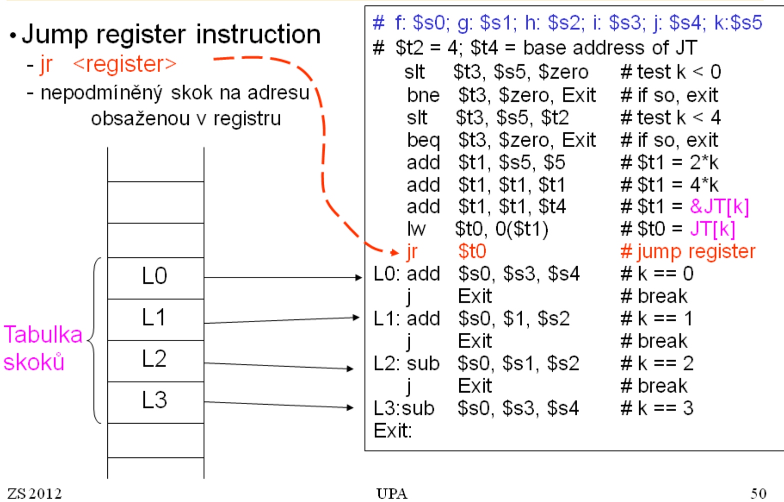
- **slti**: vhodné pro smyčky if (g >= 1) goto Loop



- Verze bez znaménka: **sltu** a **sltiu**

Tabulka skoků

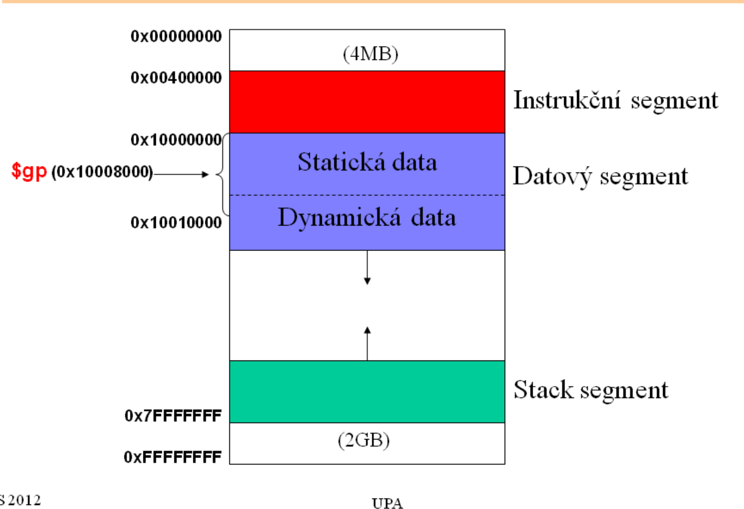
Tabulky skoků



ZS 2012 UPA 50

Mapa paměti - rozdělení paměti

Mapa paměti



ZS 2012 UPA 6

Volání procedur – dynamická povaha procedur

Volání procedur

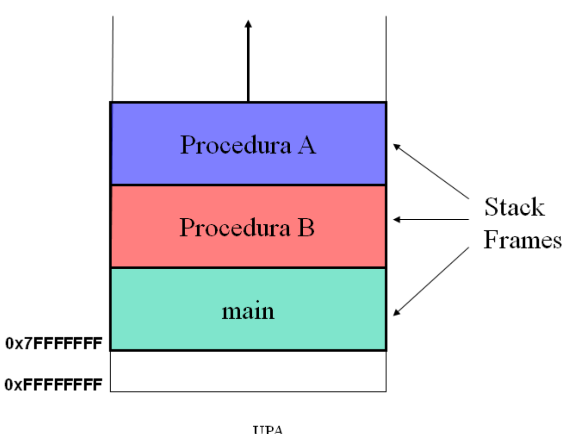
- Problémy
  - Adresa procedury
  - Návratová adresa
  - Argumenty
  - Lokální proměnné
  - Návratová hodnota
- Registry - konvence
  - Labels
  - $ra
  - $a0, $a1, $a2, $a3
  - $s0, $s1, …, $s7
  - $v0, $v1

- Dynamická povaha procedur
  - Rámce volání procedur (frames)
    - Argumenty, ukládání registrů, lokální proměnné

ZS 2012 UPA 9

Zásobník – stack

Stack



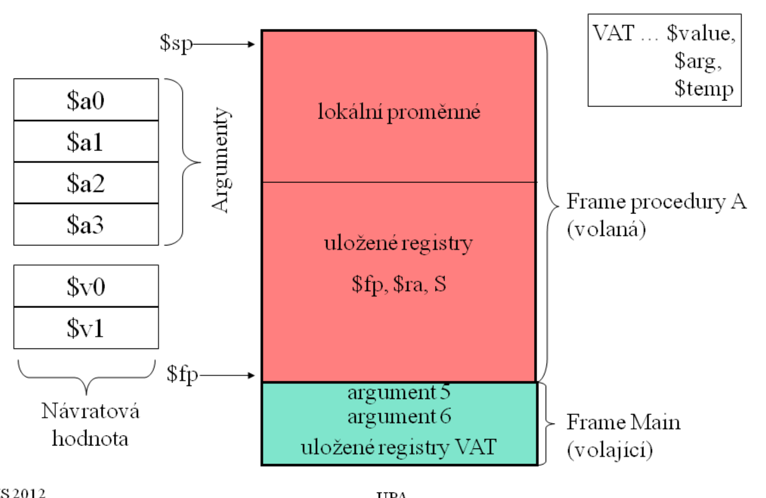
Registr $fp – frame pointer

Registr $fp

- Konvence MIPS
  - je-li funkci předáváno více parametrů než 4, zapíší se tyto parametry do stacku nad $fp
  - na tyto extra parametry se dostupuje pomocí pointeru $fp a příslušného offsetu
- Použití frame pointeru je ale nepovinné, některé softwarové produkty jej nevyužívají, na parametry lze dostupovat pomocí $sp (jako pointer s příslušným offsetem)
- $fp se během zpracování funkce nemění (představuje pevnou bázi v rámci jednoho provedení funkce)
- $sp se během zpracování funkce měnit může, na jednotlivé parametry se pak dostupuje s aktuálním offsetem, což je méně přehledné (překladač určí offsety správně !!)

Rámce stacku – frames

Rámce stacku (frames)



Volající/Volaný konvence – volání funkce a ukládání registrů

Volající/volaný - konvence

- Těsně před vyvoláním funkce volající
  - Předá argumenty ($a0 - $a3). Další arg.: uloží do stacku
  - Uloží ukládané registry volajícího ($a0 - $a3; $t0 - $t9)
  - Provede instrukci **jal** (skok na volanou proceduru a uložení návratové adresy)
- Těsně před zahájením výpočtu volané funkce se
  - Alokuje paměť pro frame ($sp = $sp – fsize)
  - Uloží ukládané registry volaného ($s0-$s7; $fp; $ra)
  - $fp = $sp + (fsize – 4)
- Těsně před návratem do volajícího:
  - Uložení funkční hodnoty do registru $v0
  - Obnovení všech registrů volané funkce
  - Pop stack frame ($sp = $sp + fsize); obnova $fp
  - Návrat provedením skoku na adresu uloženou v $ra

Instrukce skoku a link – funkce jal

Instrukce skoku a link

- Jednoduchá instrukce pro skok a uložení návratové adresy

jal: **jump & link** (*Společné části stavět rychlé*)
- **Formát J: jal label**
- **Měla by se nazývat** ***laj***
  1. **(link): uložení adresy příští instrukce do $ra**
  2. **(jump): skok na návěští**

```
1000 add  $a0,$s0,$zero  # $a0 = x
1004 add  $a1,$s1,$zero  # $a1 = y
1008 jal  sum  # $ra = 1012; jump to sum
1012 ...

2000 sum: add $v0,$a0,$a1
2004 jr   $ra                  # jump to 1012
```

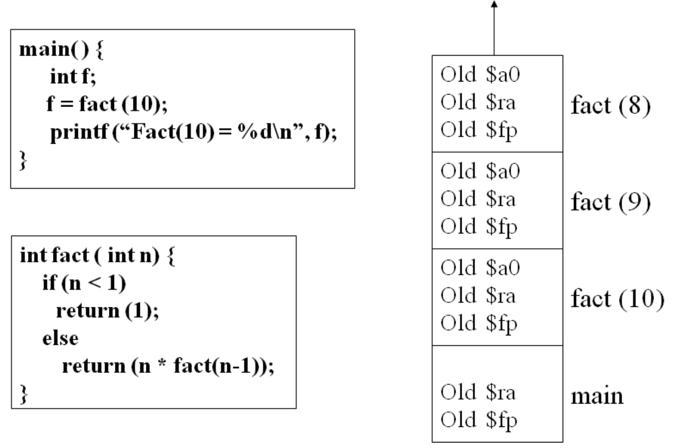
Podpora – instrukce **jal**

Vnořené procedury – příklad faktoriál

Příklad(1/2)

```
main() {
    int f;
    f = fact (10);
    printf ("Fact(10) = %d\n", f);
}
```

```
int fact ( int n) {
   if (n < 1)
     return (1);
   else
      return (n * fact(n-1));
}
```



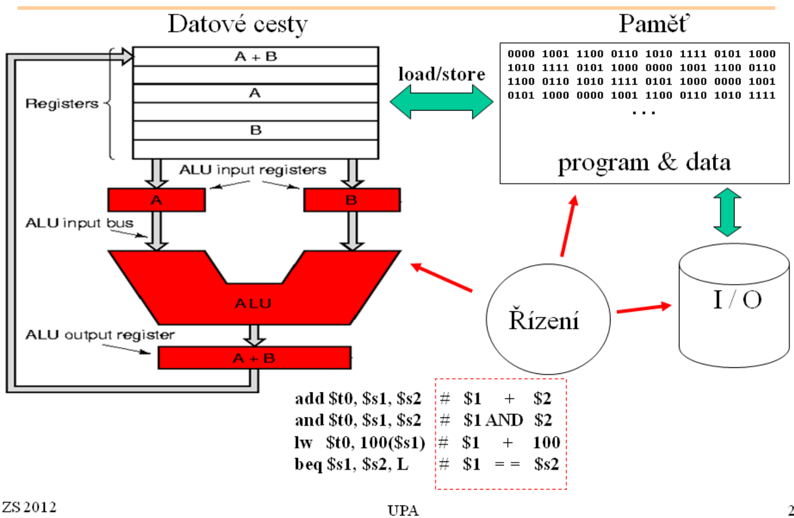
Příklad (2/2)

```
main:   subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sw      $fp, 16($sp)
        addiu   $fp, $sp, 28
        li      $v0, 4
        la      $a0, str
        syscall
        li      $a0, 10
        jal     fact
        addu    $a0, $v0, $zero
        li      $v0, 1
        syscall
        lw      $ra, 20($sp)
        lw      $fp, 16($sp)
        addiu   $sp, $sp, 32
        jr      $ra
```

```
fact:   subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sw      $fp, 16($sp)
        addiu   $fp, $sp, 28
        sw      $a0, 0($fp)
        lw      $v0, 0($fp)
        bgtz    $v0, L2
        li      $v0, 1
        j       L1
L2:     lw      $v1, 0($fp)
        subu    $v0, $v1, 1
        move    $a0, $v0
        jal     fact
        lw      $v1, 0($fp)
        mul     $v0, $v0, $v1
L1:     lw      $ra, 20($sp)
        lw      $fp, 16($sp)
        addiu   $sp, $sp, 32
        jr      $ra
```


Schéma ALU – aritmeticko logický jednotka

Arithmeticko-logická jednotka



Problémy se znaménkem a amplitudou – dvojí nula, komplikovanější aritmetické obvody

Problémy se znaménkem a amplitudou

1. Komplikovanější aritmetické obvody
   Jsou třeba speciální kroky v závislosti na tom, zda jsou znaménka shodná či nikoliv
   (např., $-x \cdot -y = xy = x * y$)
2. Dvojí reprezentace nuly
   – 0x00000000 = +0
   – 0x80000000 = - 0
   – Komplikace při porovnávání (+ 0 == - 0)

- Vzhledem k „zmatku“ kolem nuly se tato reprezentace (“přímý kód“) běžně nepoužívá (vyjma fp)

Dvojkový doplněk – převod na dekadický tvar, reprezentace v základním tvaru

Dvojkový doplněk

- Může reprezentovat kladná i záporná čísla –znaménkový bit (MSB). Zápis:
  $d_{31} x (-2^{31}) + d_{30} x 2^{30} + \ldots + d_2 x 2^2 + d_1 x 2^1 + d_0 x 2^0$
- Příklad
  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
  $= 1x(-2^{31})+1x2^{30}+1x2^{29}+\ldots+1x2^2+0x2^1+0x2^0$
  $= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^2 + 0 + 0$
  $= -2{,}147{,}483{,}648_{10} + 2{,}147{,}483{,}644_{10}$
  $= -4_{10}$
- Pozn.! Musí být známa délka zobrazení =>poloha MSB => MIPS používá 32 bitů, takže MSB je d_{31}

Definice kódů pro zobrazení záporných čísel – přímý, inverzní, doplňkový kód

Definice kódů pro zobrazení záporných čísel

- Přímý kód:

$$N_p = (1-2.x_{n-1}) . \sum_{i=0}^{n-2} x_i.2^i$$

- Inverzní kód (jedničkový doplněk)

$$N_i = -(2^{n-1}-1).x_{n-1} + \sum_{i=0}^{n-2} x_i.2^i$$

- Doplňkový kód (dvojkový doplněk)

$$N_d = -2^{n-1}.x_{n-1} + \sum_{i=0}^{n-2} x_i.2^i$$

Datové typy, datové typy MIPS

Datové typy

- Aplikace / HLL
  - Integer
  - Floating point
  - Character
  - String
  - Date
  - Currency
  - Text
  - Objects (ADT- Abstract Data Types)
  - Blob (Binary large object)
  - Double precision
  - Signed, unsigned
- Podpora hardware
  - Numerické datové typy
    - Integer
      - 8 / 16 / 32 / 64 bitů
      - Se znaménkem, bez znaménka
      - BCD čísla (COBOL, Y2K!)
    - Floating point
      - 32 / 64 /128 bitů
  - Nenumerické datové typy
    - Znaky
    - Řetězce
    - Boolean (bitové mapy)
    - Pointery

Datové typy MIPS (1/2)

- Základní „strojní“ datové typy: 32-bit slovo
  - 0100 0011 0100 1001 0101 0011 0100 0101
  - Integer čísla (se znaménkem a bez znaménka)
    - 1,128,878,917
  - Floating point čísla
    - 201.32421875
  - 4 ASCII znaky
    - C I S E
  - Adresy do paměti (pointery)
    - 0x43495345
  - Instrukce

Datové typy MIPS (2/2)

- 16-bitové konstanty (immediates)
  - addi $s0, $s1, 0x8020
  - lw $t0, 20($s0)
- Half word (16 bitů)
  - **lh** (**lhu**): load half word lh $t0, 20($s0)
  - **sh**: save half word sh $t0, 20($s0)
- Byte (8 bitů)
  - **lb** (**lbu**): load byte sh $t0, 20($s0)
  - **sb**: save byte sh $t0, 20($s0)

Konstanty

Konstanty

- Časté používání malých konstant (50% operandů)
  - např.: A = A + 5;
- Řešení
  - Uložení 'typických konstant' do paměti a jejich používání.
  - Vytvoření HW registrů (jako $zero) i pro některé další konstanty, např.: 1.
- Instrukce MIPS:

```
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 0x4a
addi $29, $29, 4
```

8	29	29	4
101011	10011	01000	0000 0000 0011 0100

Velké konstanty

Velké konstanty

- Naplnění 32 bitového registru konstantou:
  1. Naplnění (16) vyšších bitů
     `lui $t0, 1010101010101010`

1010 1010 1010 1010	0000 0000 0000 0000

  2. Pak se musí nižší bity přesunout doprava, t. zn.
     `ori $t0, $t0, 1010101010101010`

$t0:	1010 1010 1010 1010	0000 0000 0000 0000
ori	0000 0000 0000 0000	1010 1010 1010 1010
	1010 1010 1010 1010	1010 1010 1010 1010

Adresní režimy, typy adresních režimů dat, ortogonalita, bázové registrové adresování, immediate operandy

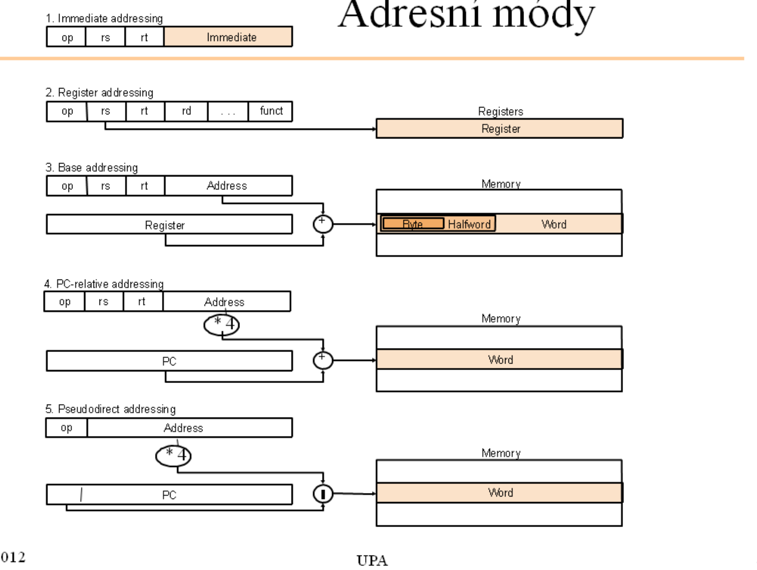
Adresní režimy

- Adresy pro *data* a *instrukce*
- Data (operandy a výsledky)
  - Registry
  - Místa v paměti
  - Konstanty
- Úsporné kódování adres (prostor: 32 bitů)
  - Registry (32) => použito 5 bitů pro zakódování adresy
  - *Destruktivní* instrukce: reg2 = reg2 + reg1
  - Akumulátor
  - Stack
- ***Ortogonalita*** operačního kódu, adresních režimů a datových typů

Adresní režimy dat

- Registrové adresování
  - Nejobvyklejší způsob (nejrychlejší a nejkratší)
  - add $3, $2, $1
- Bázované adresování
  - Operand je v paměti na místě udaném offsetem
  - lw $t0, 20 ($t1)
- „Immediate“ operandy
  - Operand je malá konstanta uvnitř instrukce
  - addi $t0, $t1, 4 (16-bit integer se znaménkem)

Adresní módy



Adresní módy instrukcí

Adresní módy instrukcí

- Adresy jsou dlouhé 32 bitů
- Speciální registr PC (**P**rogram **C**ounter) obsahuje adresu právě prováděné instrukce
- PC-relativní adresování (větvení, skoky)
  - Adresa: PC + (konstanta v instrukci) * 4
  - beq $t0, $t1, 20 (0x15090005)
- „Pseudopřímé“ adresování (skoky)
  - Adresa: PC[31:28] : (konstanta v instrukci) * 4

Pointery- definice

Pointery

- Pointer: proměnná, která obsahuje adresu jiné proměnné
  - Výraz pocházející z HLL pro adresu v paměti
- Proč používat pointery?
  - Někdy je to jediná cesta pro rychlý výpočet
  - Často jediná cesta pro získání úsporného kódu
- Proč ne?
  - Častý zdroj chyb v softwaru

  1) „Nestálé“ reference (předčasně uvolněné)

  2) „Díry“ v paměti (pozdě uvolněné): dlouho trvající úlohy nelze provozovat bez periodického restartu (???)

Předávání argumentů – volání hodnotou, volání odkazem

Předávání argumentů

- 2 možnosti
  - “Volání hodnotou”: funkci/proceduře se předá kopie položky (argumentu)
  - “Volání odkazem”: funkci/proceduře se předá pointer na položku (argument)
- Proměnné s délkou 1 slovo se předávají hodnotou
- Předání pole ? např., `a[100]`
  - Pascal (volání hodnotou) kopíruje 100 slov z pole `a[]` do stacku
  - C (volání odkazem) předává se pouze pointer (1 slovo) na pole `a[]` v registru

Přehled instrukcí MIPS

Přehled instrukcí

MIPS assembly language

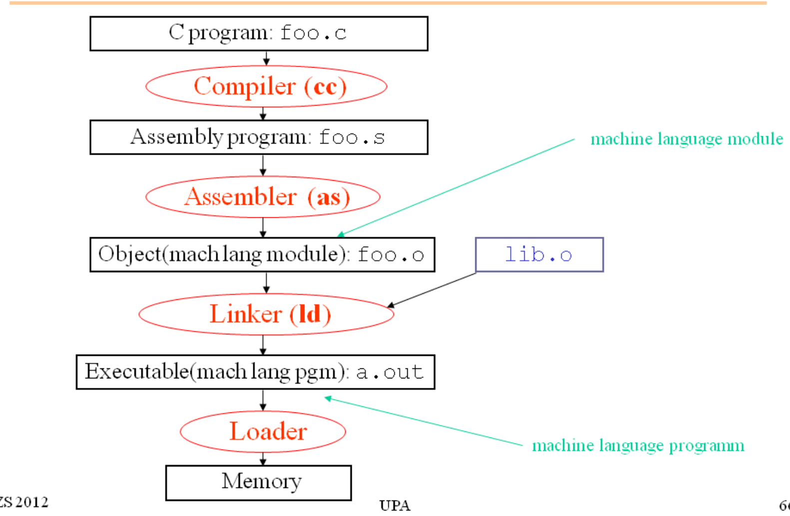
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	lb $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1, 100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

Datové typy MIPS – adresní módy

Nové - programy MIPS

- Datové typy a adresování zahrnuté v ISA
  - Kompromis mezi požadavky aplikací a hardwarovou implementací
- Datové typy MIPS
  - 32-bitová slova
  - 16-bitová poloviční slova
  - 8-bitové byty
- Adresní módy
  - Data
    - Registry
    - 16-bitové konstanty se znaménkem
    - Bázové adresování
  - Instrukce
    - PC-relativní
    - (Pseudo) přímé

Postup při vývoji programu



Direktivy assembleru

Direktivy assembleru

- Direktivy assembleru, které neprodukují strojní instrukce

`.align n`	Zarovnat další položku na 2^n bytové hranici
`.text`	Uložit další položky do uživatelského textového segmentu
`.data`	Uložit další položky do uživatelského datového segmentu
`.globl sym`	Na `sym` se lze odvolávat z jiných souborů
`.asciiz str`	Uložit řetězec `str` do paměti
`.word w1…wn`	Uložit *n* 32-bitových položek do následujících slov v paměti
`.byte b1..bn`	Uložit *n* 8-bitových položek do následujících bytů v paměti
`.float f1..fn`:	Uložit *n* floating-point čísel do následujících slov v paměti

Absolutní adresy – které instrukce vyžadují editaci realokace

Absolutní adresy

- Které instrukce vyžadují editaci relokace?
  - Load/store do proměnných ve statické oblasti

lw/sw	$gp	$x	address

  - Podmíněné skoky

beq/bne	$rs	$rt	address

    – PC-relativní adresování to nevyžaduje

  - Nepodmíněné skokové instrukce

j/jal	xxxxx

- Přímé (absolutní) reference na data (např. instrukce `la`)

Generování strojního kódu – jednoduché případy, podmíněné skoky, přímá adresa

Generování strojního kódu

- Jednoduché případy
  - Aritmetické a logické operace, posuvy, atd.
  - Všechny informace v instrukci jsou k dispozici.
- Podmíněné skoky (beq, bne)
  - Jakmile jsou makroinstrukce nahrazeny reálnými, lze určit cílové adresy skoků
  - PC-relativní, jednoduchá manipulace
- Přímá (absolutní) adresa.
  - Skoky (j a jal)
  - Přímé (absolutní) reference na data
  - Nelze určit nyní, proto jsou vytvářeny **dvě tabulky**

Tabulky assembleru – tabulky symbolů, relokační tabulka

Tabulky assembleru

- **Tabulka symbolů**
  - Seznam „položek“ tohoto souboru, který bude použit jinými soubory.
    - Návěští: volání funkcí
    - Data: cokoliv v sekci **.data**; proměnné, které mají být dostupné z více souborů
  - První průchod: záznam dvojic návěští-adresa
  - Druhý průchod: generování strojního kódu
  - Lze skákat na návěští deklarovaná na vyšších adresách (dále v textu)
- **Relokační tabulka**
  - Seznam „položek“, pro které je třeba adresa.
  - Libovolné návěští, na které se skáče: j nebo jal
    - internal
    - external (včetně knihovních souborů)
  - Jakákoliv data (např. instrukce **la**)

Formát objektového souboru

Formát objektového souboru

- Hlavička objektového souboru: velikost a poloha ostatních částí objektového soubK segment: strojní kód
- Kódový segment: binární reprezentace dat ve zdrojovém souboru
- Relokační informace: identifikuje řádky kódu, který musí být “ošetřen”
- Tabulka symbolů : seznam návěští v souboru a data, na která bude dostupováno (budou reference)
- Informace pro debugger

Linker – funkce

Linker (Link Editor)

- Sestavuje objektové soubory (.o) a vytváří spustitelný soubor - program.
- Umožňuje oddělenou (nezávislou) kompilaci souborů.
  - Rekompilují se pouze pozměněné soubory (moduly)
    - **Windows NT zdrojový kód má >30 M řádek!**
    - **Windows XP patrně není známo ani Microsoftu** ☺
- Edituje "odkazy" ve skokových instrukcích, vyhodnocuje reference do paměti.
- Proces (vstup: objektové soubory vygenerované assemblerem).
  - Krok 1: slučuje kódové segmenty všech **.o** souborů
  - Krok 2: slučuje datové segmenty všech **.o** souborů a připojuje je na konec kódových segmentů
  - Krok 3: vyhodnocuje reference. Prochází relokatační tabulku a ošetří každou položku (doplní všude **absolutní adresy**)

Vyhodnocení referencí linkeru- čtyři typy referencí

Vyhodnocení referencí

- Čtyři typy referencí (adres)
  - PC-relativní (např. **beq, bne**): nikdy se nerelokují
  - Absolutní adresy (**j, jal**): vždy se relokují
  - Externí reference (**jal**): vždy se relokují
  - Datové reference (**lui** and **ori**): vždy se relokují
- Linker *předpokládá,* že prvé slovo prvého programového segmentu leží na adrese 0x00000000.
- Údaje, které linker zná:
  - Délka každého programového a datového segmentu
  - Uspořádání programových a datových segmentů
- Linker vypočítává:
  - Absolutní adresy všech návěští, na které se skáče (interní nebo externí) a každá data, na které se program odkazuje

Loader - funkce

Loader

- Spustitelný program je uložen na disku.
- Činnost loaderu: natažení programu do paměti a spuštění
- Ve skutečnosti, loader je částí operačního systému (OS)
  1. Čte hlavičku, aby určil velikost programu a datových segmentů
  2. Vytvoří nový adresní prostor pro program tak velký, aby mohl obsahovat kódové a datové segmenty i stackový segment
  3. Kopíruje instrukce a data ze souboru programu do paměti
  4. Kopíruje argumenty předané programu do stacku
  5. Inicializuje registry, **$sp** = první volné místo ve stacku
  6. Skáče na startovací rutinu, která kopíruje argumenty programu ze stacku do registrů a nastavuje registr PC
  7. Když se rutina main vrací, startovací rutina ukončuje program systémovým voláním exit

Souhrn překladu programu – funkce jednotlivých částí

Souhrn - programy MIPS

- Kompilátor konvertuje HLL soubor do jednoho souboru – jazyk assembler.
- Assembler odstraní makroinstrukce, konvertuje vše co lze do strojového jazyka a vytváří relokační tabulku pro linker. Ten pro každý .s soubor vytváří .o soubor.
- Linker spojuje všechny .o soubory a vyhodnocuje absolutní adresy.
- Loader načítá spustitelný soubor do paměti a startuje provádění programu.

Architektura vs Mikroarchitektura

Architektura & Mikroarchitektura (opakování)

- **Architektura:**
  Souhrn vlastností procesoru (nebo systému) jak se jeví „uživateli“
  - Uživatel: „binární program“ běžící na procesoru nebo programátor na úrovni assembleru
- **Mikroarchitektura:**
  Souhrn vlastností implementace (které uživatel na instrukční úrovni nevidí)
  Vlastnosti, které se mění (časování, výkon, technologie), patří do mikroarchitektury
- Snahou každé firmy je navrhnout architekturu, která dokáže „přežít“ delší časové údobí. Mikroarchitektura se v tomto údobí může měnit.

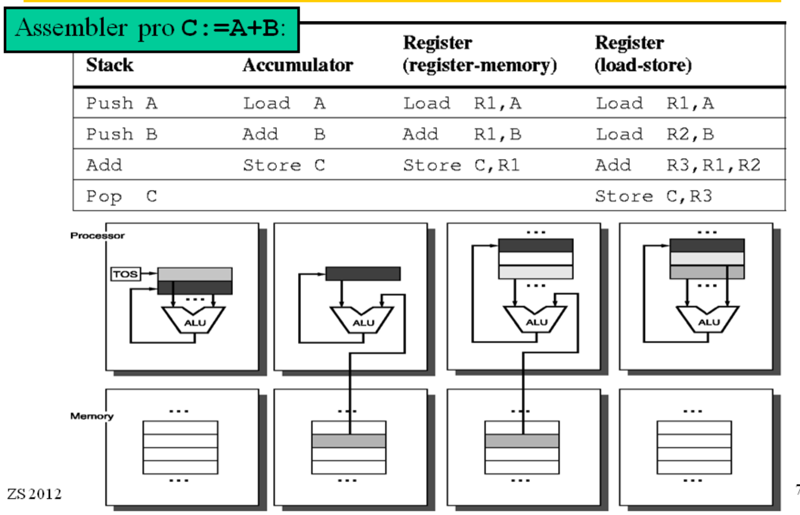
Architektury počítačů – orientace na stack, akumulátor, registr-paměť, registr-registr, architektura pro HLL

Architektury počítačů

Orientace architektury na:

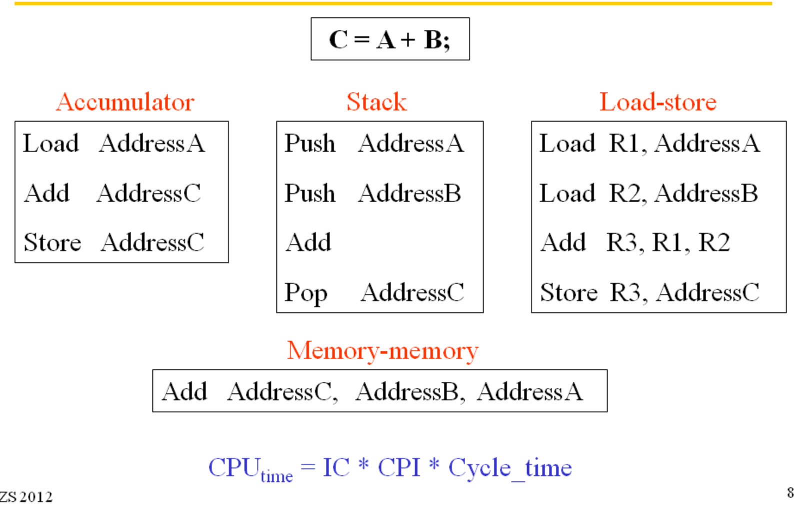
- Stack
  - Žádné registry (jednoduché kompilátory, kompaktní kódování)
- Akumulátor
  - Drahý hardware => pouze jeden registr
  - Akumulátor: jeden z operandů a výsledek
  - Adresní režimy vztažené na operand v hlavní paměti
- Registry se speciálním použitím (např. I8086)
  - Registr-paměť
  - Registr-registr (load-store)
- Architektura počítačů pro HLL

Typy architektur ISA– příklady



Stack	Accumulator	Register (register-memory)	Register (load-store)
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

Typy architektur ISA – příklad C=A+B, Memory-Memory, Load-Store, Stack, Akumulator (Accumulator)



Architektura RISC – filozofie návrhu

Architektura RISC

- RISC - počítače s redukovaným souborem instrukcí
- Filozofie návrhu
  - Load/store instrukce pro práci s pamětí
  - Instrukce pevné délky
  - Tříadresová architektura
  - Mnoho registrů
  - Jednoduché adresní módy
  - Instrukční pipelining
- Mnohé myšlenky, použité v moderních počítačích pocházejí ze CDC 6600 (1963)

PowerPC

PowerPC

- Podobné MIPS: 32 registrů, 32-bitové instrukce, RISC
- Rozdíly (porovnání: jednoduchost vs. common case)
  - Indexové adresování
    - Příklad: `lw $t1,$a0+$s3  #$t1=Memory[$a0+$s3]`
    - MIPS: `add $t0, $a0, $s3; lw $t1,0($t0)`
  - Adresní módy s aktualizací
    - Aktualizace registru jako část „load" (průchod polem)
    - Příklad: `lwu $t0,4($s3) #lw $t0,4($s3); addi $s3,$s3,4`
  - Speciální instrukce
    - Load multiple/store multiple: až 32 slov v jedné instrukci
    - Speciální registr - čítač
      - `bc Loop, $ctr!=0` *#decrement counter, if not 0 goto loop*
      - MIPS: addi, $t0, $t0, -1; bne $t0, $zero, Loop

ZS 2012 11

Architektura X86, Dvouadresní architektura, architektura Registr-Memory, Instrukce proměnné délky

Architektura X86

- Dvouadresní architektura
  - Jeden z operandů je zároveň cílem
    `add $s1,$s0 # s0=s0+s1 (C: a += b;)`
  - Výhoda: malé instrukce ⇒ malý kód ⇒ **rychlejší**
- Architektura typu Register-Memory
  - Jeden operand může být v paměti; druhý je v registru
    `add 12(%gp),%s0 # s0=s0+Mem[12+gp]`
  - Výhoda: méně instrukcí ⇒ menší rozsah kódu
- Instrukce proměnné délky (1 až 17 bytů)
  - Malý rozsah kódu (o 30% menší)
  - Vyšší účinnost instrukční cache
  - Instrukce mohou zahrnovat 8- nebo 32-bitové immediate op.

ZS 2012 13

Vlastnosti X86

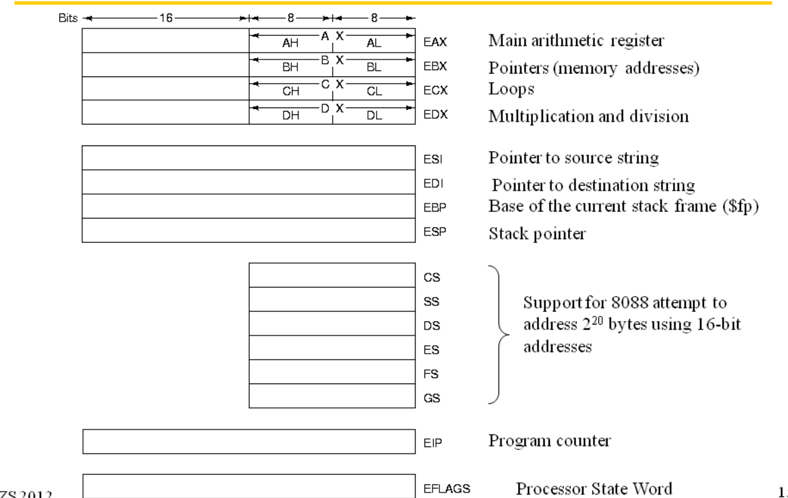
Vlastnosti X86

- Operační módy: reálný (8088), virtuální a chráněný
- Čtyři úrovně ochrany
- Paměť
  - Adresní prostor: 16,384 segmentů (4GB)
  - Uložení bytů ve slově - Little endian
- 8 32-bitových registrů (16-bit, 8086 jména, prefix **e**):
  - `eax, ecx, edx, ebx, esp, ebp, esi, edi`
- Datové typy
  - **S**igned/**U**nsigned integer (8, 16, a 32 bitů)
  - **B**inary **C**oded **D**ecimal integer čísla
  - **F**loating **P**oint (32 a 64 bitů)
- Floating point jednotka používá oddělený stack

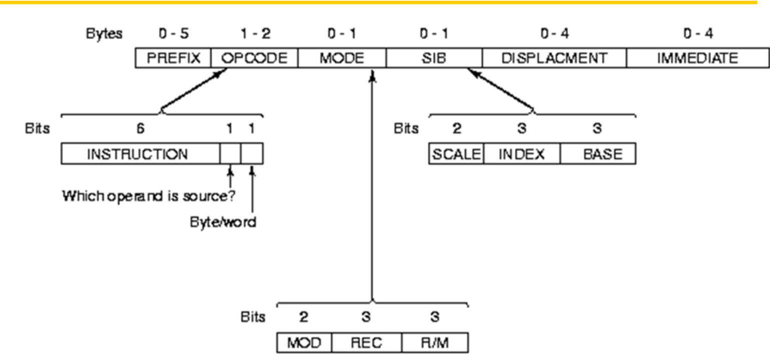
ZS 2012 14

Registry X86, Instrukční formáty X86, příklady instrukčních formátů X86

Registry X86

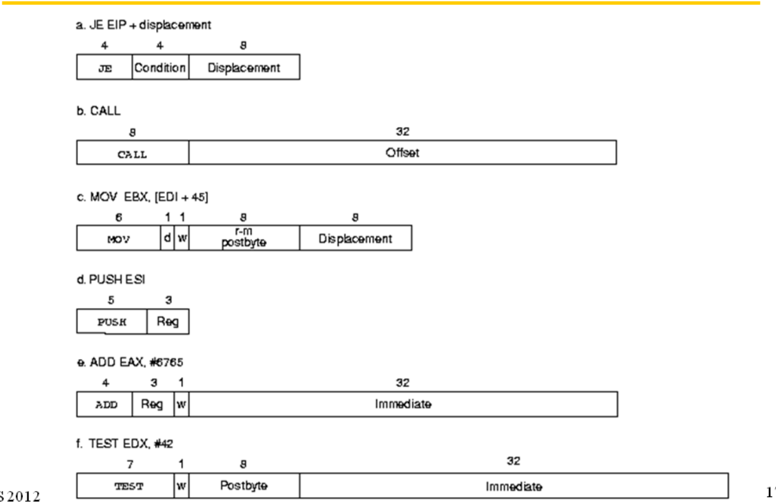


Instrukční formáty X86



- Velmi složité a neregulární
  - Šest polí s proměnnou délkou
  - Dalších pět polí se může vyskytnout

Příklady instrukčních formátů X86



Instrukce pro Integer (int) X86, příklady instrukcí X86

Instrukce pro čísla integer

- Řídící
  - JNZ, JZ
  - JMP
  - CALL
  - RET
  - LOOP
- Datové přenosy
  - MOV
  - PUSH, POP
  - LES
- Aritmetické
  - ADD, SUB
  - CMP
  - SHL, SHR, RCR
  - CBW
  - TEST
  - INC, DEC
  - OR, NOR
- Operace s řetězci
  - MOVS
  - LODS

Příklady instrukcí X86

- `leal` (load effective address)
  - Vypočítá adresu podobně jako load ale zapíše adresu do registru
  - Určí 32-bit adresu:

```
leal -4000000(%ebp),%esi    #esi = ebp - 4000000
```

- Paměťový stack je součástí instrukčního souboru
  - `call label` (esp-=4; M[esp]=eip+5; eip = label)
  - `push` uloží hodnotu do stacku, inkrementuje `esp`
  - `pop` vezme hodnotu ze stacku, dekrementuje `esp`
- `incl, decl` (increment, decrement)

```
incl %edx # edx = edx + 1
```


Dekódování adresních režimů

Dekódování adresních režimů

	MOD			
R/M	00	01	10	11
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX or AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX or CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX or DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX or BL
100	SIB	SIB with OFFSET8	SIB with OFFSET32	ESP or AH
101	Direct	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP or CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI or DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI or BH

- Vysoce neregulární, neortogonální adresní módy
  - Instrukce v 16-bitovém nebo 32-bitovém módu?
  - Zdaleka ne všechny módy lze aplikovat na všechny instrukce
  - Zdaleka ne všechny registry mohou být užity ve všech módech

Adresní režimy (módy), podpora větvení, smyčka while X86

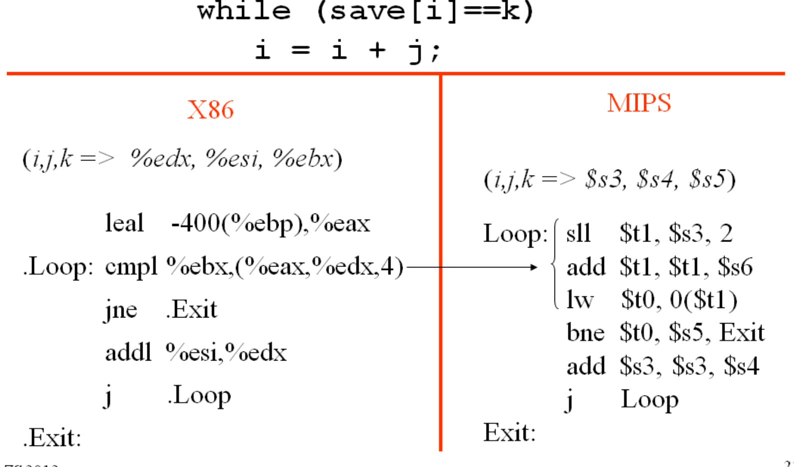
Adresní režimy (módy)

- Base reg + offset (jako MIPS)
  - `movl -8000044(%ebp), %eax`
- Base reg + index reg (2 registry formují adresu)
  - `movl (%eax,%ebx),%edi`
    `# edi = Mem[ebx + eax]`
- Scaled reg + index (posuv registru o 1,2)
  - `movl(%eax,%edx,4),%ebx`
    `# ebx = Mem[edx*4 + eax]`
- Scaled reg + index + offset
  - `movl 12(%eax,%edx,4),%ebx`
    `# ebx = Mem[edx*4 + eax + 12]`

Podpora větvení

- Namísto porovnání registrů používá x86 speciální 1-bitové registry nazývané "podmínkové kódy", které vytvářejí vedlejší efekty operací ALU
  - S - Sign Bit
  - Z - Zero (výsledek je celý roven 0)
  - C - Carry Out
  - P - Parity: nastaven na 1, je-li počet jedniček v osmi bitech výsledku operace vpravo sudý
- Instrukce podmíněných skoků používají tyto podmínkové kódy pro všechna porovnání: <, <=, >, >=, ==, !=

Smyčka While



Složitost instrukcí, principy návrhu X86

Souhrn

- Složitost instrukcí je pouze jeden faktor
  - menší počet instrukcí vs. vyšší CPI / nižší frekvence hodin
- Principy návrhu:
  - jednoduchost vyžaduje regularitu
  - menší je rychlejší
  - dobrý návrh vyžaduje dobré kompromisy
  - společné části stavět rychlé
- Instruction Set Architecture
  - velmi důležitá abstrakce!

ZS 2012 25

Aritmeticko logické operace – ALU, Problémy počítačové aritmetiky, Vlastnosti reprezentace čísel

Nové – Aritmeticko/logické operace

- Aritmeticko-logická jednotka (ALU)
  - Jádro počítače
  - Provádí aritmetické a logické operace nad daty
- Problémy počítačové aritmetiky
  - Reprezentace čísel
    - Integer a floating point
    - Omezená přesnost (overflow / underflow)
  - Algoritmy použité pro základní operace
- Vlastnosti reprezentace čísel
  - Jedna nula
  - Čísla rozložena symetricky kolem nuly
  - Efektivní hardwarová implementace algoritmů
- Dvojkový doplněk (algoritmus): negace čísla a přičtení jedničky

ZS 2012 26

Podmínky přetečení

Podmínky přetečení

Operace	Operand A	Operand B	Výsledek
A + B	>= 0	>= 0	< 0
A + B	< 0	< 0	>= 0
A – B	>= 0	< 0	< 0
A – B	< 0	>= 0	>= 0

ZS 2012 32

Podpora přetečení MIPS

Podpora MIPS

- U MIPS nastane výjimka, vznikne-li přetečení
  - Výjimky (interrupty) pracují jako volání procedury
  - Registr EPC uloží adresu *„závadné"* instrukce
  - mfc0 $t1, $epc # moves contents of EPC to $t1
  - Neexistuje podmíněný skok s testem přetečení
- Aritmetika dvojkového doplňku (add, addi a sub)
  - Výjimka při přetečení
- Aritmetika bez znaménka (addu a addiu)
  - Při přetečení nevznikne výjimka
  - Používáno při výpočtu adres
- Kompilátory
  - C ignoruje přetečení (vždy používá addu, addiu, subu)
  - Fortran používá vhodné instrukce

Podmíněné skoky při přetečení, podpora registru $k0 $k1

Podmíněné skoky při přetečení

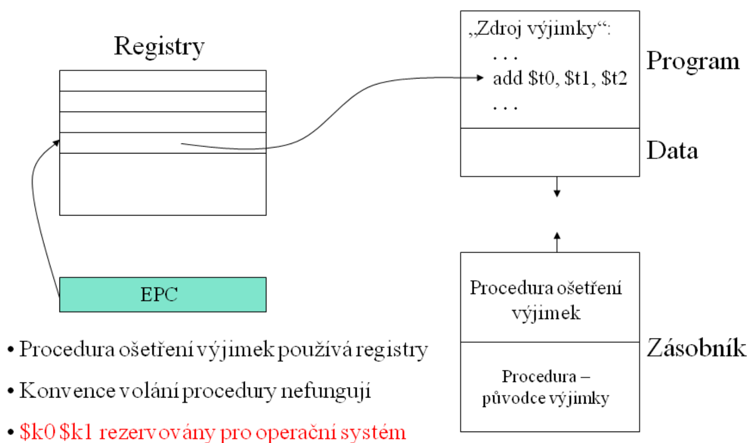
Sčítání se znaménkem – softwarová detekce přetečení

```
addu  $t0, $t1, $t2         # add but do not trap
xor   $t3, $t1, $t2         # check if sign differ
slt   $t3, $t3, $0          # $t3 =1 if signs differ
bne   $t3, $0, NO_OVFL      # signs of t1, t2 different
xor   $t3, $t0, $t1         # sign of sum (t0) different?
slt   $t3, $t3, $0          # $t3 = 1 if sum has different sign
bne   $t3, $0, OVFL         # go to overflow
```

Sčítání bez znaménka (range = $[0 : 2^{32} – 1]$ => **$t1 + $t2 <= 2^{32} – 1**)

```
addu  $t0, $t1, $t2         # $t0 contains the sum
nor   $t3, $t1, $0          # negate $t1 ($t3 = NOT $t1)
sltu  $t3, $t3, $t2         # 2^32 –1 – t1 < t2?
bne   $t3, $0, OVFL         # t1 + t2 > 2^32 –1 => overflow
```


Registry $k0 a $k1



- Procedura ošetření výjimek používá registry
- Konvence volání procedury nefungují
- $k0 $k1 rezervovány pro operační systém

Hardwarové komponenty, hradla (bloky)

Hardwarové komponenty (bloky)

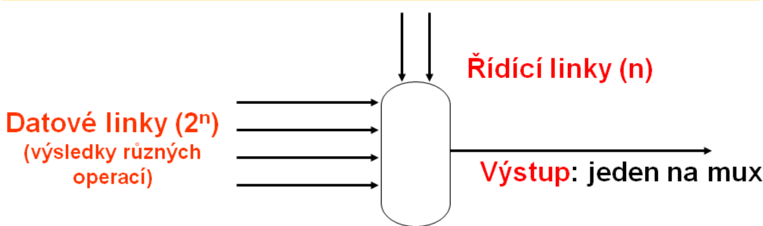
- ALU se staví z komponent nízké úrovně (implementace) – z logických hradel
- **Hradla** (přehled)
  - Hardwarový prvek, který zpracovává několik vstupů a generuje jeden výstup
  - Může být reprezentován pravdivostní tabulkou a nebo logickou rovnicí
  - Hradla se vytvářejí z tranzistorů na křemíku
- Hardwarové komponenty (bloky) - přehled
  - Hradlo And
  - Hradlo Or
  - Invertor (not)
  - Multiplexor (mux)

Modulární návrh ALU, implementace ALU

Modulární návrh ALU

- Fakta
  - Stavební bloky pracují s **individuálními** (I/O) bity
  - ALU pracuje se 32-bitovými registry
  - ALU provádí množinu operací (+, -, *, /, posuvy, atd.)
- Principy
  - Sestavit 32 samostatných 1-bitových ALU
  - Sestavit samostatné hardwarové bloky pro každou úlohu
  - Všechny operace se provádí paralelně
  - Pro výběr aktuální operace se použije multiplexor
- Výhody
  - Snadné připojení další operace (instrukce)
    - Připojení nových datových linek k multiplexoru; informovat „řízení" o změně

Implementace ALU

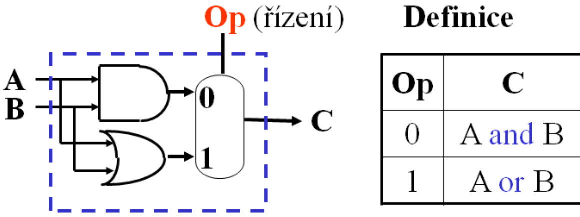


1. 32-bitová ALU použije 32 multiplexerů (pro každý výstupní signál jeden)
2. Projít instrukční soubor a pro implementaci odpovídajících operací přidat datové (a řídící) linky.

Mapování komponent na hardware

Jednobitové logické instrukce

- Přímé mapování na hardwarové komponenty
  - instrukce AND
    - Jedna datová linka pochází z pouhého hradla AND
  - instrukce OR
    - Další datová linka pochází z pouhého hradla OR

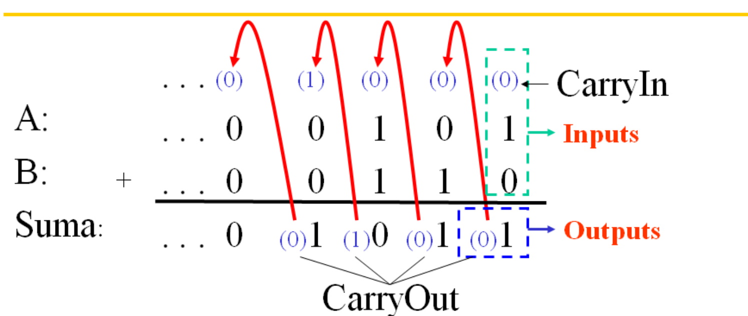


Definice

Op	C
0	A and B
1	A or B

Jednobitová úplná sčítačka, pravdivostní tabulka, zápis rovnic, obvody zapojení

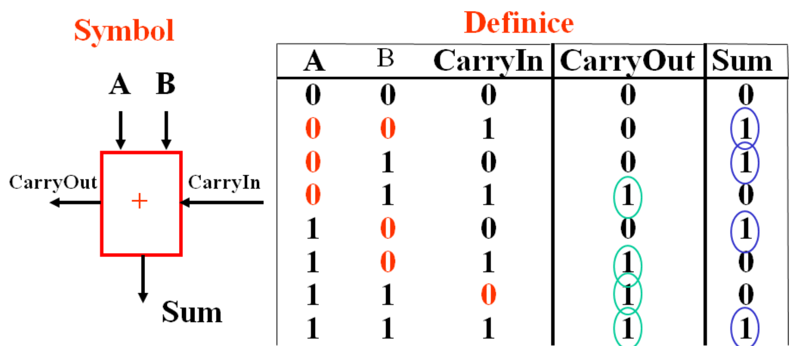
Jednobitová úplná sčítačka



- Každý „bit" sčítačky má:
  - Tři vstupní signály: A_i, B_i, $CarryIn_i$
  - Dva výstupní signály: Sum_i, $CarryOut_i$
    ($CarryIn_{i+1}$ = $CarryOut_i$)

Pravdivostní tabulka úplné sčítačky

Symbol **Definice**

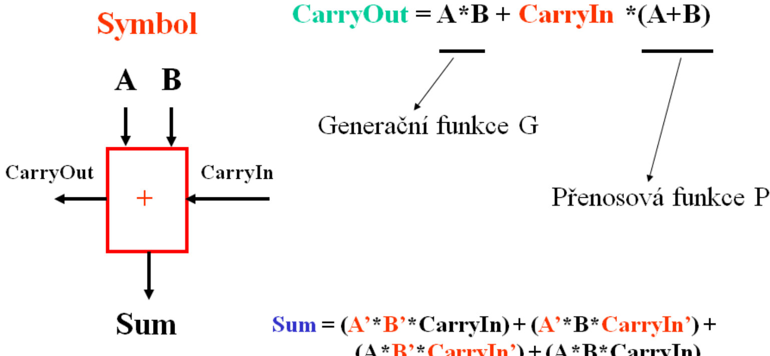


A	B	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

CarryOut = (A'*B*CarryIn) + (A*B'*CarryIn) + (A*B*CarryIn') +
(A*B*CarryIn) = (B*CarryIn) + (A*CarryIn) + (A*B)

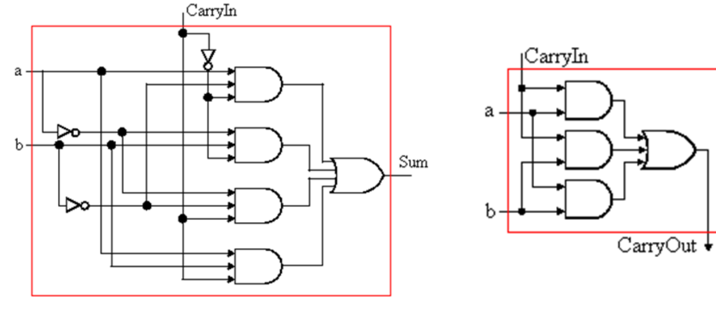
Sum = (A'*B'*CarryIn) + (A'*B*CarryIn') + (A*B'*CarryIn') + (A*B*CarryIn)

Ekvivalentní zápis rovnic úplné sčítačky

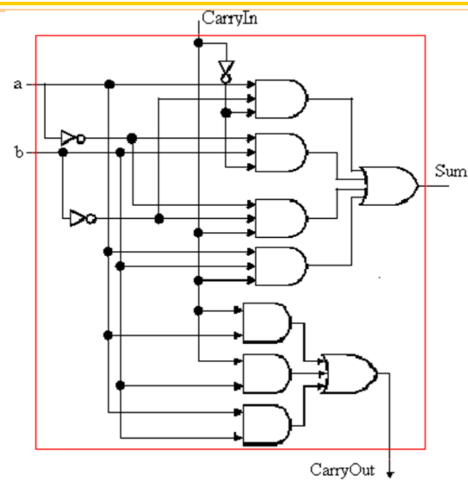


Obvody úplné sčítačky (1/2)

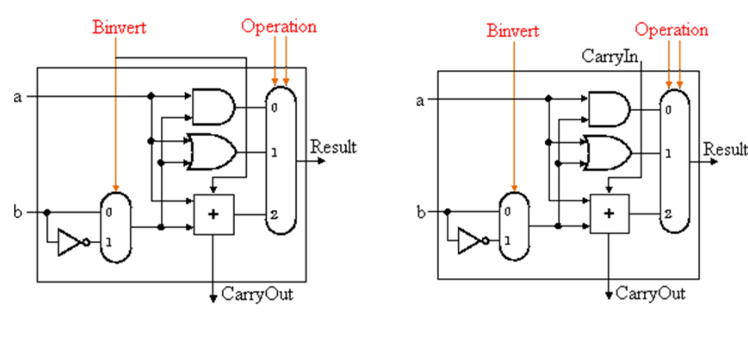
1. Sestavení funkce Sum
2. Sestavení funkce CarryOut
3. Propojení signálů se stejným jménem



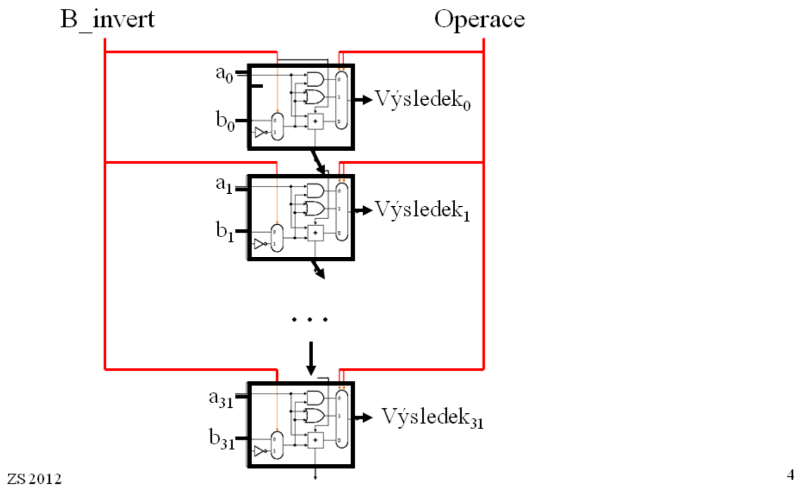
Obvody úplné sčítačky (2/2)



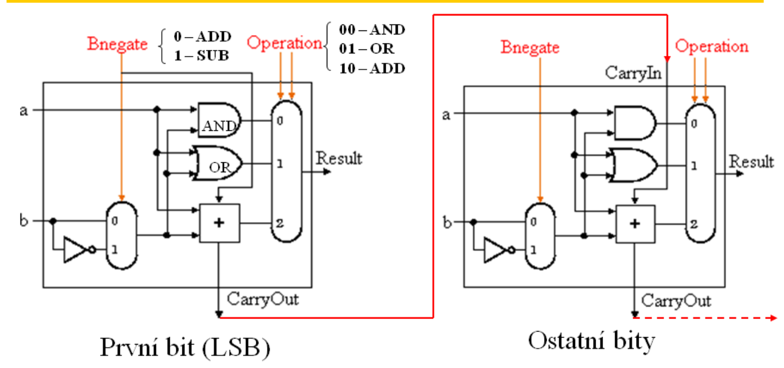
Jednobitová ALU



32-bitový ALU



Generická jednobitová ALU



Instrukce slt – porovnání znamének, detekce přetečení

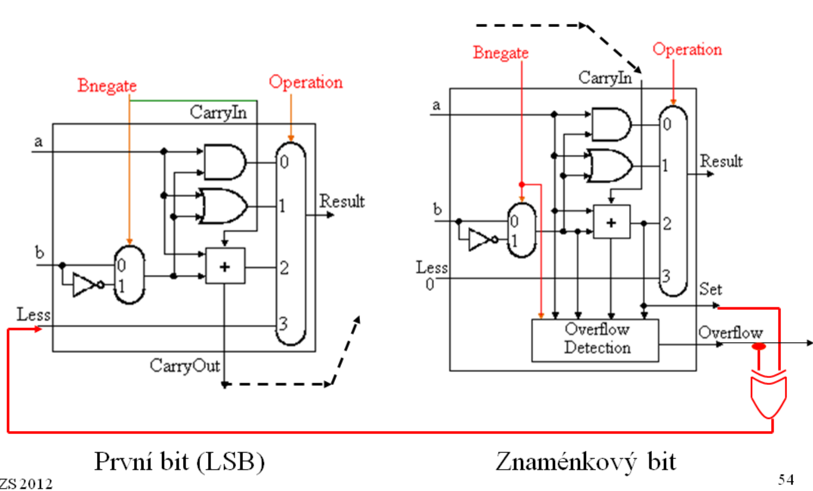
Instrukce Slt

- **Slt rd, rs, rt**

  rd: 0000 0000 0000 0000 0000 0000 0000 000**r** → { 1 if (rs < rt) ; 0 else }

- A < B => A – B < 0
  1. Vytvoření rozdílu použitím úplné sčítačky
  2. Test bitu s nejvyšší vahou (znaménkový bit)
  3. Znaménkový bit říká, zda A < B
- Nový vstupní signál (*Less*) jde přímo na mux
- Nová řídící linka (111) pro slt
- Výsledek pro slt není výstupem z ALU
  - Je třeba další 1-bitová ALU pro bit s nejvyšší vahou
    - Má novou výstupní linku (*Set*) použitou pouze pro slt
    - (S tímto bitem je také spojena logika detekce přetečení)

HW podpora pro Slt



První bit (LSB)   Znaménkový bit

Instrukce větvení programu – použití rozdílu, schéma zapojení

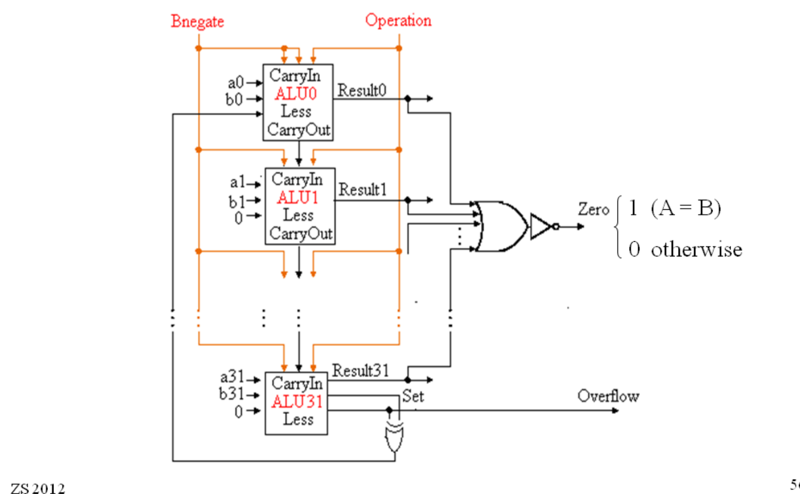
Instrukce větvení programu

- beq $t5, $t6, L
  - Použití rozdílu: (a-b) = 0 => a = b
  - Pro test výsledku na rovnost 0 - přidat HW
  - operace OR s 32 bity výsledku a následná inverze výstupu OR

$$Zero = \overline{(Result_1 + Result_2 + .. + Result_{31})}$$

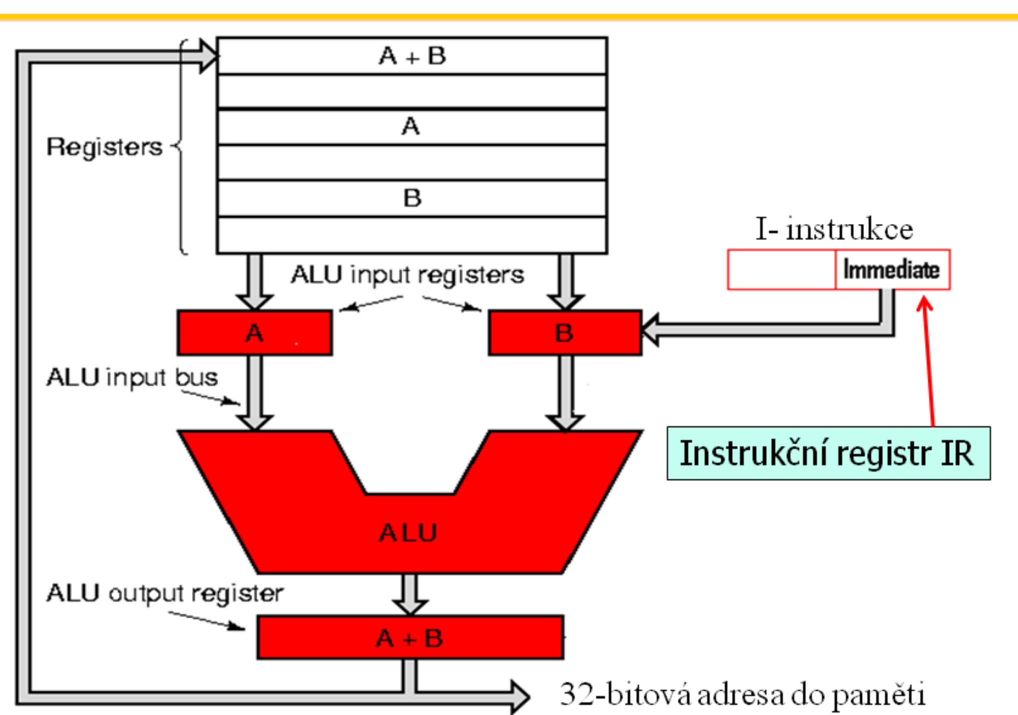
- Uvažujme operace A + B a A - B
  - Přetečení nastane, je-li
    - A = 0 ?
    - B = 0 ?

HW podpora větvení



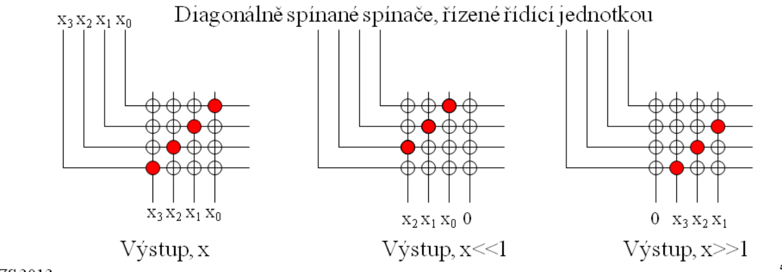
Instrukce posuvu – barrel shifter

Přehled



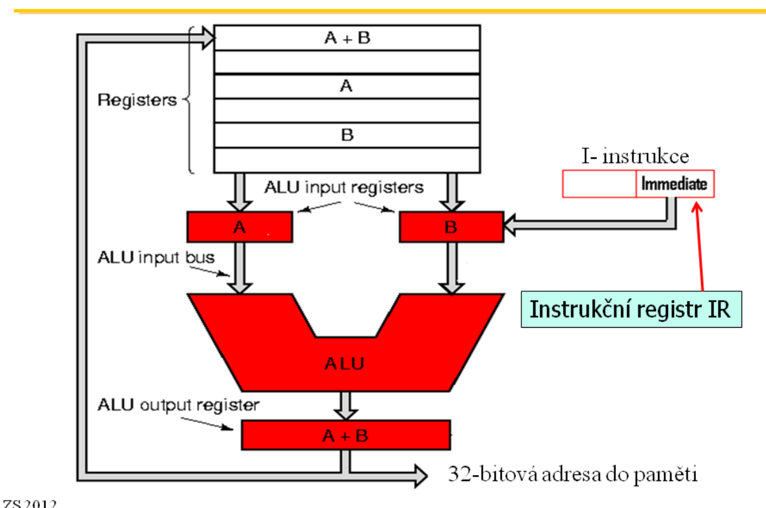
Instrukce posuvu

- SLL, SRL a SRA
- Potřebujeme datovou linku pro posuvy ($\overleftarrow{L}$ a $\overrightarrow{R}$)
- Nicméně posuvové jednotky se snáze implementují na úrovni tranzistorů (mimo ALU)
- Posuvové jednotky typu „Barrel shifter“



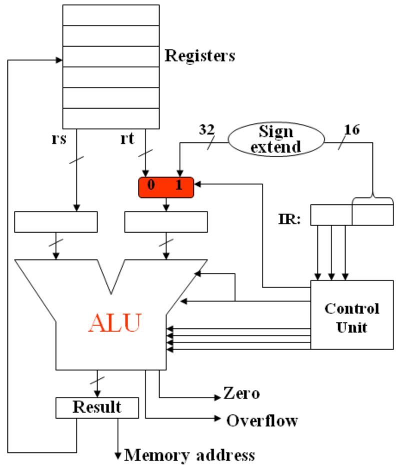
Instrukce s operandy typu immediate

Přehled



Instrukce s operandy typu immediate

- Prvý vstup do ALU tvoří prvý registr (rs)
- Druhý vstup
  - Data z registru (rt)
  - Nula- nebo immediate s rozšířením znaménka
- Přidáme multiplexer na druhý vstup ALU



Sčítačka typu Carry-lookhead, schéma, funkce P a G druhé úrovně

Sčítačka typu Carry-Lookahead (1/2)

- Řešení - kompromis mezi dvěma extrémy
- Motivace:
  - Co můžeme dělat, neznáme-li hodnotu carry-in?
  - Kdy budeme vždy generovat přenos? $g_i = a_i\ b_i$
  - Kdy jej budeme předávat dál? $p_i = a_i + b_i$
- Zbavili jsme se šíření vlny v cestě přenosu?

$c_1 = g_0 + p_0 c_0$

$c_2 = g_1 + p_1 c_1$ $\quad c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$

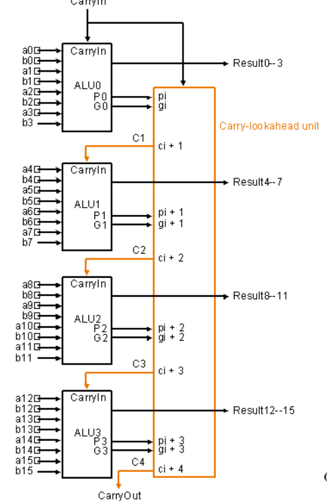
$c_3 = g_2 + p_2 c_2$ $\quad c_3 = g_2 + p_2 g_1 + \ldots$

$c_4 = g_3 + p_3 c_3$ $\quad c_4 =$

ZS 2012 61

Carry-Lookahead Adder (2/2)

- Uvedeným způsobem nelze postavit 16-ti bitovou sčítačku (příliš velká a složitá)
- Lze postavit „ripple carry“ sčítačku s použitím 4-bitových CLA sčítaček
- Lépe: použít princip CLA opakovaně!



ZS 2012 62

Funkce P a G druhé úrovně

$P_0 = p_3 + p_2 + p_1 + p_0$

$P_1 = p_7 + p_6 + p_5 + p_4$

$P_2 = p_{11} + p_{10} + p_9 + p_8$

$P_3 = p_{15} + p_{14} + p_{13} + p_{12}$

$G_0 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$

$G_1 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4$

$G_2 = g_{11} + p_{11} g_{10} + p_{11} p_{10} g_9 + p_{11} p_{10} p_9 g_8$

$G_3 = g_{15} + p_{15} g_{14} + p_{15} p_{14} g_{13} + p_{15} p_{14} p_{13} g_{12}$

ZS 2012 63

Ripple Carry vs Carry lookahead

Ripple Carry vs. Carry Lookahead

- Budeme uvažovat stejné zpoždění pro průchod signálu hradlem (AND nebo OR)
- Celková doba = dána počtem hradel v nejdelší cestě
- Předpokládejme 16-bitovou sčítačku
- Signály CarryOut c_{16} a c_4 určují nejdelší cestu
  - Ripple carry: 2 * 16 = 32
  - Carry lookahead: 2 + 2 + 1 = 5
    - 2 úrovně logiky pro vytvoření P_i a G_i
    - P_i je určen v jedné úrovni (AND) z jednotlivých p_i
    - G_i se vytváří ve dvoustupňové logice ze signálů p_i a g_i
    - p_i a g_i lze vytvořit v jedné úrovni ze signálů a_i a b_i
- „Carry lookahead adder“ je zde **šestkrát** rychlejší

Alternativy implementace

Alternativy implementace

- Logická rovnice pro součet může být zapsána mnohem jednodušeji použitím hradel XOR

  Sum = a XOR b XOR CarryIn
- Pro některé technologie vychází XOR efektivněji než dvě úrovně AND a OR
- Procesory se nyní implementují pomocí CMOS tranzistorů (spínače)
- CMOS ALU a posuvové jednotky typu „barrel shifter“ mají méně multiplexorů, než obsahoval náš návrh.
- Principy návrhu jsou stejné

Rychlost hradel, rychlost obvodu, klíčová myšlenka

Závěr

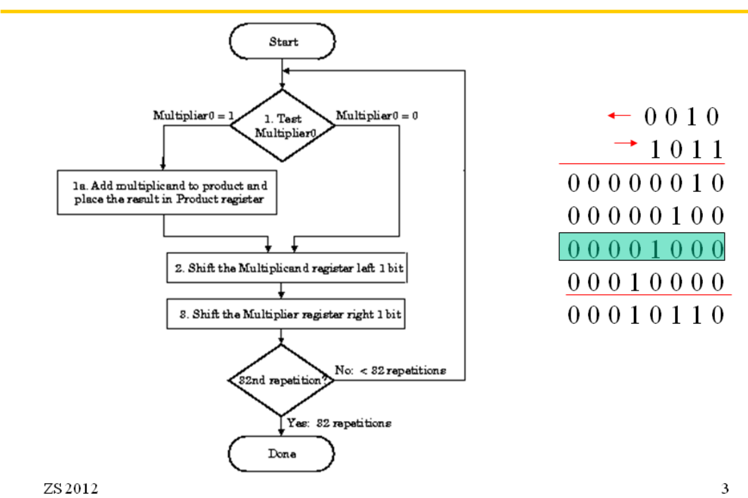
- ISA podporuje rozvoj architektury
- Hardware/Software, důraz na RISC/CISC
- Technologie je hnacím motorem rozvoje
- ALU = jádro procesoru
- ALU problém = přetečení
- Ošetření výjimek (přerušení) ☺

Závěr

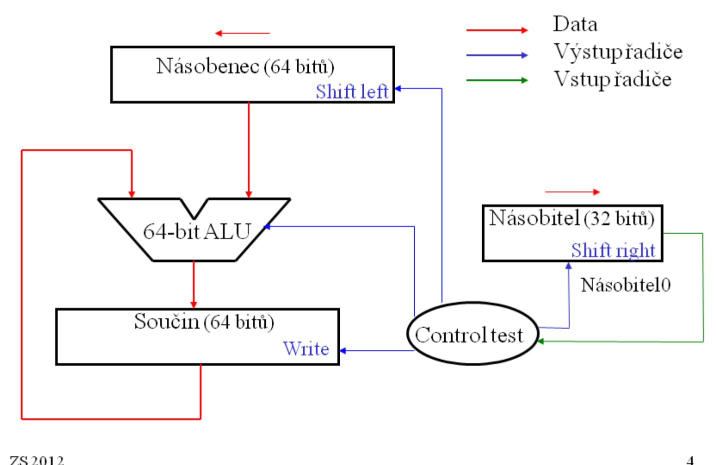
- Můžeme postavit ALU tak, aby vyhověla MIPS ISA
  - **Klíčová myšlenka**: Použít multiplexer pro výběr výstupu ALU
  - Pro odčítání se používá sčítání dvojkového doplňku
  - Pro sestavení 32-bitové ALU opakovaně použít 1-bitovou ALU
- Důležité poznámky k hardware
  - Všechna *hradla* v ALU *pracují paralelně*
  - Rychlost hradel závisí na počtu vstupů
  - Rychlost obvodu závisí na počtu sériově řazených hradel (v *kritické cestě,* nebo-li na *počtu úrovní logiky*)
- Primární cíl: (koncepční)
  - Promyšlené změny organizace mohou zlepšit výkon (podobně jako použití lepšího algoritmu při programování)

Algoritmus násobení verze 1 – hardware, schéma zapojení, vývojový diagram

První verze (V.1)

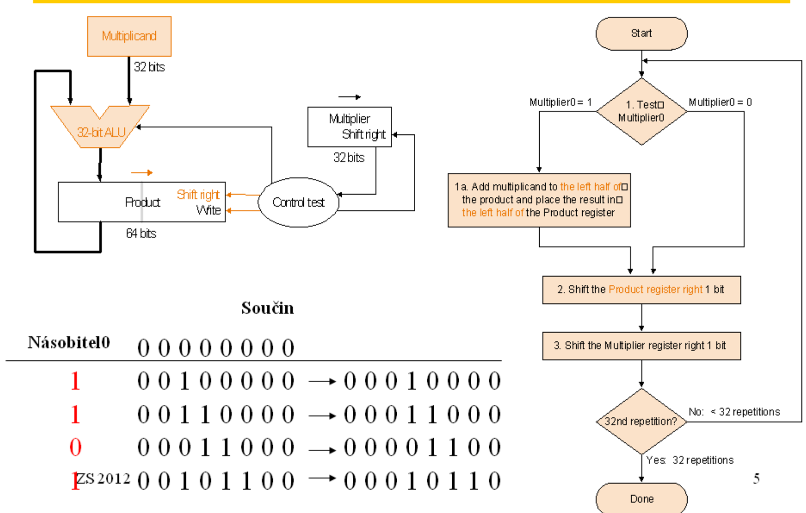


Hardware (V.1)



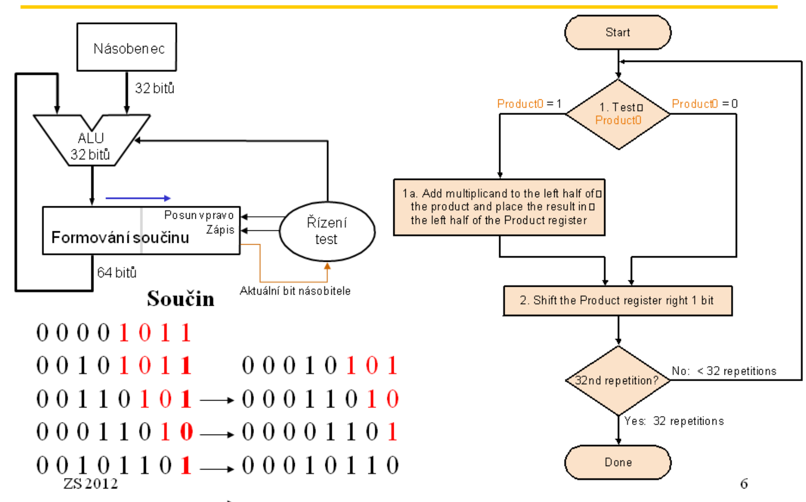
Algoritmus násobení verze 2

Druhá verze (V.2)



Algoritmus násobení verze 3 – konečná verze, souhrn

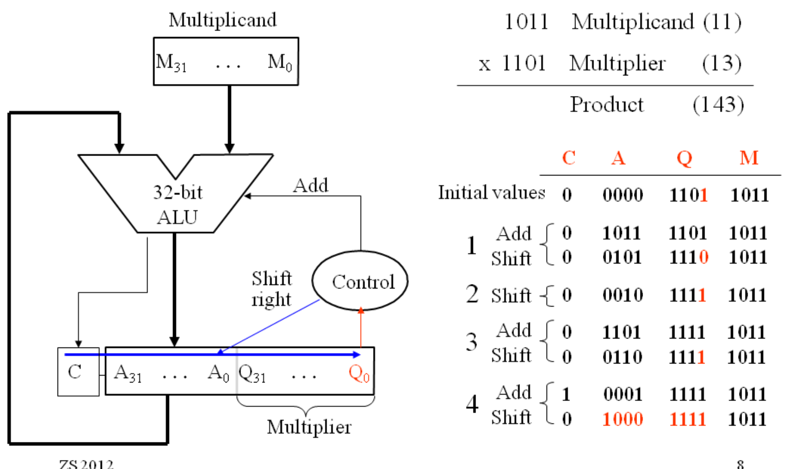
Konečná verze (V.3)



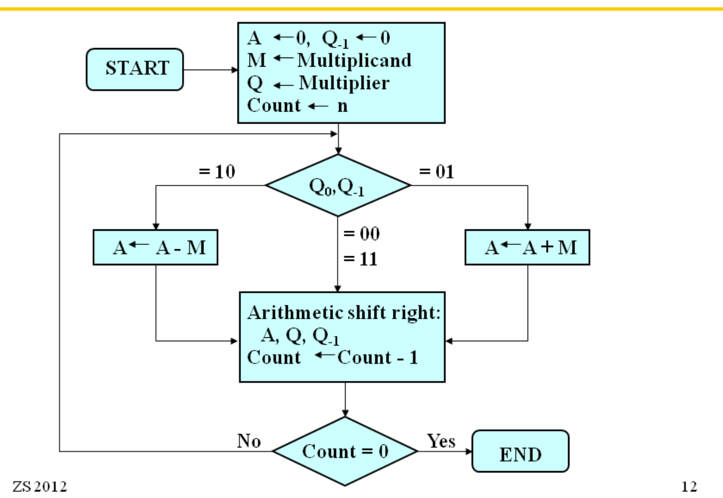
Souhrn

- Násobení bez znaménka
  - Generace parciálního součinu pro každou cifru násobitele
  - Parciální součin $= \begin{cases} \textbf{0} & \textbf{If cifra násobitele = 0} \\ \textbf{Násobenec} & \textbf{If cifra násobitele = 1} \end{cases}$
  - Celkový součin = součet parciálních součinů (správně posunutých vlevo)
  - Násobení dvou n-bitových binárních čísel dává výsledek o šířce 2n bitů

Násobení celkový pohled – konečná (finální) verze schéma, příklad



Boothův algoritmus – vývojový diagram



Boothův algoritmus

Boothův algoritmus pro dvojkový komplement

- Zpracovává kladné i záporné operandy
- Mnohem efektivnější než konvenční algoritmus

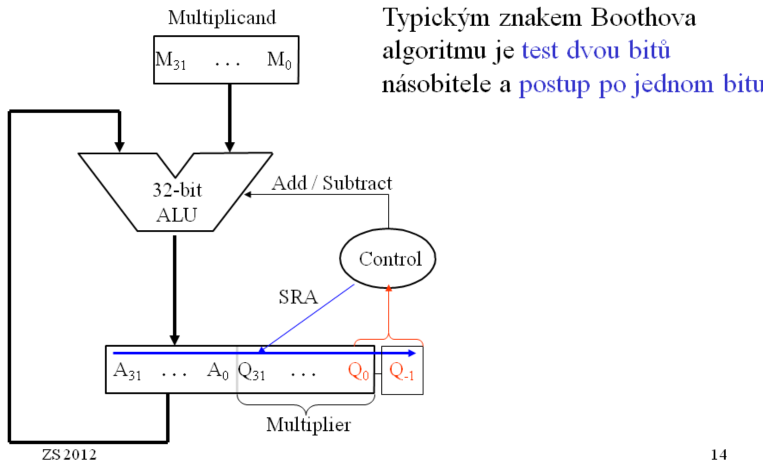
  $2^n + 2^{n-1} + 2^{n-2} + \dots + 2^k = 2^{n+1} - 2^k$

- Je možná další optimalizace

- Operaci násobení lze implementovat pomocí hardware pro posuvy, sčítání (! odčítání)
- Instrukce násobení MIPS ignorují přetečení
  - Multu: Hi = 0
  - Mult: Hi = rozšířené znaménko z Lo

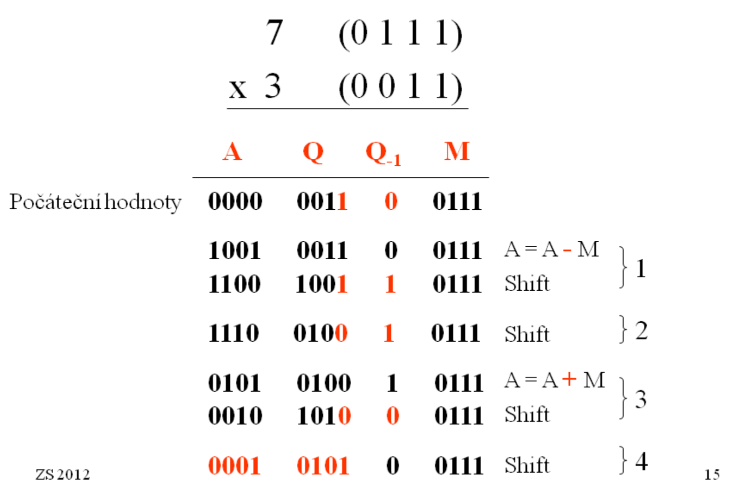
Hardware pro Boothův algoritmus - schéma

Hardware pro Boothův algoritmus



Boothův algoritmus - příklad

Příklad



7 (0 1 1 1)

x 3 (0 0 1 1)

	A	Q	Q_{-1}	M		
Počáteční hodnoty	0000	0011	0	0111		
	1001	0011	0	0111	A = A - M	} 1
	1100	1001	1	0111	Shift	
	1110	0100	1	0111	Shift	} 2
	0101	0100	1	0111	A = A + M	} 3
	0010	1010	0	0111	Shift	
	0001	0101	0	0111	Shift	} 4

Boothův algoritmus důkazy – důkaz kladný násobitel, důkaz záporný násobitel

Důkaz: kladný násobitel

- Předpokládejme *jednoduchý* kladný násobitel

  0 0 0 1 1 1 1 0 (jeden blok jedniček obklopený nulami)

  M x (0 0 0 1 1 1 1 0) $= M \times (2^4 + 2^3 + 2^2 + 2^1)$

  (pozice bitů: 5 4 3 2 1 0)

  $= M \times (16 + 8 + 4 + 2)$

  $= M \times 30$

- Poznámka: $2^n + 2^{n-1} + \ldots + 2^{n-k} = 2^{n+1} - 2^{n-k}$

  => M x (0 0 0 1 1 1 1 0) = M x $(2^5 - 2^1)$

- Boothův algoritmus odpovídá schématu:
  - Odečti, jestliže je nalezena kombinace (1-0)
  - Přičti, jestliže je nalezena kombinace (0-1)
- Toto pravidlo se použije na každý blok jedniček

Důkaz: záporný násobitel

- Reprezentace záporného čísla (X):

  $\{ 1\ x_{n-2}\ x_{n-3} \ldots x_1\ x_0 \}$

- $X = -2^{n-1} + x_{n-2}*2^{n-2} + x_{n-3}*2^{n-3} \ldots x_0*2^0$
- Předpokládejme 0 nejvíc vlevo v *k*-té pozici

  Reprezentace $X = \{ 1\ 1\ 1 \ldots 10\ X_{k-1} \ldots X_0 \}$

  $X = -2^{n-1} + 2^{n-2} \ldots 2^{k+1} + x_{k-1}*2^{k-1} \ldots x_0*2^0$

  $-2^{n-1} + 2^{n-2} + \ldots + 2^{k+1} = -2^{k+1}$

  $X = -2^{k+1} + x_{k-1}*2^{k-1} \ldots x_0*2^0$

  (1-0) tato změna znamená operaci odečtení

Násobení u MIPS – softwarová detekce

Násobení u MIPS

- Pro výsledek jsou vyhrazeny registry (Hi, Lo)
- Dvě instrukce pro násobení
  - Mult: se znaménkem
  - Multu: bez znaménka
- mflo, mfhi – přesune obsah Hi, Lo do obecných registrů (GPR)
- *Neprovádí se žádná hardwarová detekce přetečení*

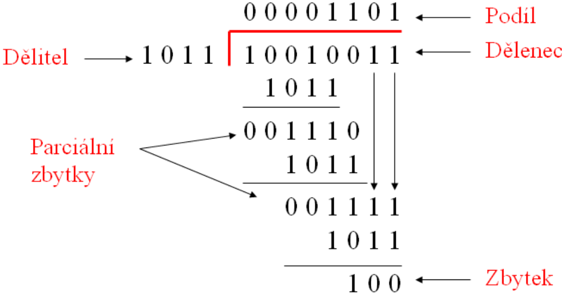
  => Softwarová detekce přetečení

  - Hi musí být 0 pro *multu* nebo rozšířené znaménko operandu Lo pro *mult*

Dělení

Dělení

- Dlouhé dělení binárních čísel integer bez znaménka



Dělenec = Podíl * Dělitel + Zbytek

Dělení

- Operace dělení je z principu sekvenční.
- Převážná většina sekvenčních metod pracuje podle rekurentního vztahu:

$$R_{j+1} = z.R_j - q_j.D$$

kde R_{j+1} ... zbytek do dalšího kroku

R_0 ... dělenec

R_j ... aktuální zbytek

q_j ... cifra podílu generovaná v j-tém kroku

z ... základ číselné soustavy

D ... dělitel

$Q = \{q_0, q_1, q_2, q_3, q_4, \dots, q_{n-2}, q_{n-1}\}$ ... podíl

Dělení – některé metody dělení

Některé metody dělení

Sekvenční metody binárního dělení se hlavně odlišují množinou generovaných cifer:

$z = 2$

$q_j \in \{0, 1\}$ … metoda binárního dělení s obnovou zbytku

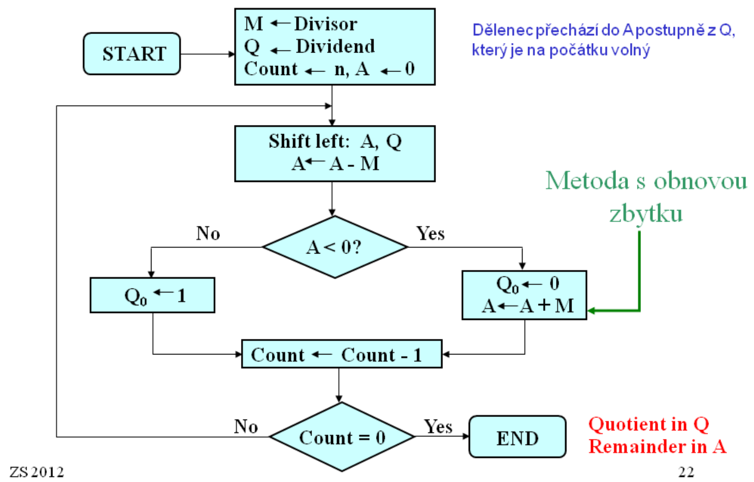
$q_j \in \{-1, 1\}$ … metoda binárního dělení bez obnovy zbytku

$q_j \in \{-1, 0, 1\}$ … binární metoda SRT (Sweeny-Robertson-Toucher)

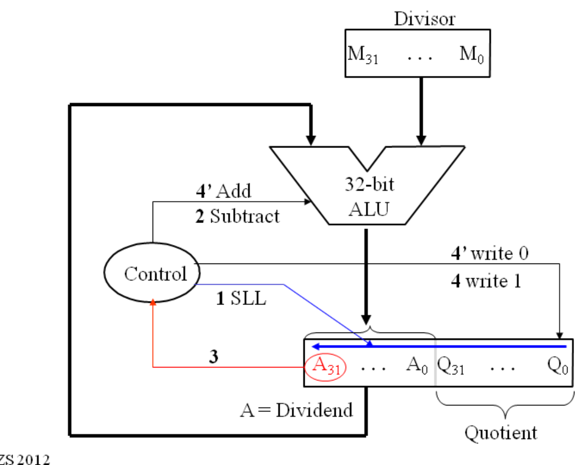
Poznámka: Metody, které generují záporné cifry výsledku, vyžadují korekční kroky. Lze je provádět většinou již v průběhu dělení (nezpůsobují další zpoždění).

Dělení bez znaménka – vývojový diagram, schéma

Dělení bez znaménka



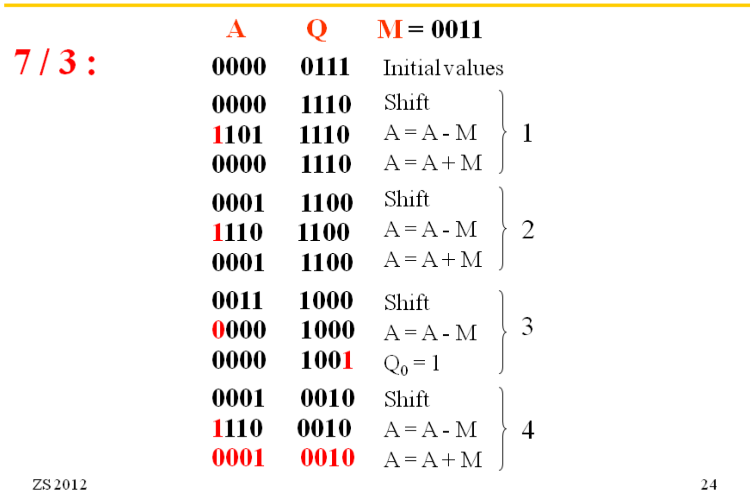
Hardware pro operaci dělení



Dělení bez znaménka příklad

Příklady

7 / 3 :

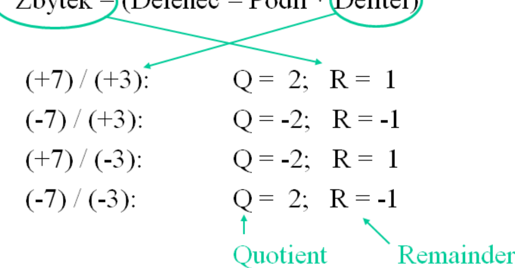


A	Q	M = 0011	
0000	0111	Initial values	
0000	1110	Shift	1
1101	1110	A = A - M	
0000	1110	A = A + M	
0001	1100	Shift	2
1110	1100	A = A - M	
0001	1100	A = A + M	
0011	1000	Shift	3
0000	1000	A = A - M	
0000	1001	$Q_0 = 1$	
0001	0010	Shift	4
1110	0010	A = A - M	
0001	0010	A = A + M	

Dělení se znaménkem - podmínka zbytku a dělitele ***(pozn. je tam chyba, správně – zbytek a dělitel musí mít stejná znaménka)***

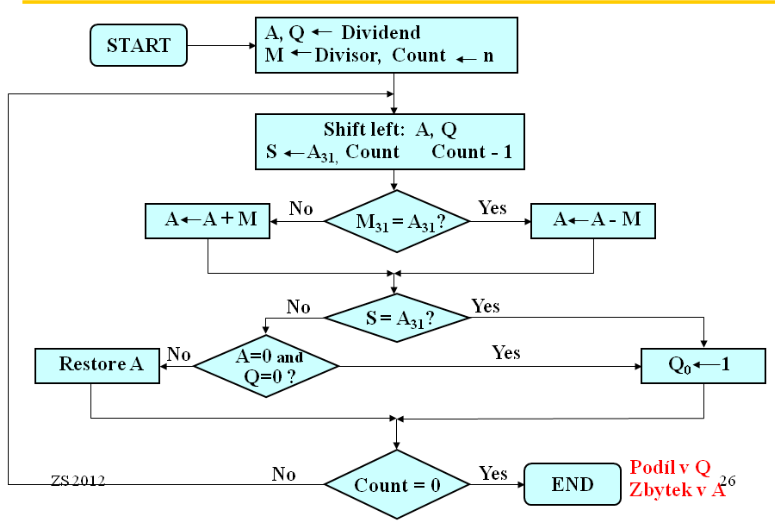
Operace dělení se znaménkem

- Jednoduché řešení
  - Negovat podil, jestliže znaménka dělitele a dělence nejsou shodná
  - Zbytek a dělenec musí mít shodná znaménka



Algoritmus dělení se znaménkem – schéma, příklady

Algoritmus dělení se znaménkem



Příklady (1/2)

A	Q	M = 0011	
0000	0111	Initial values	
0000	1110	Shift	1
1101		Subtract	
0000	1110	Restore	
0001	1100	Shift	2
1110		Subtract	
0001	1100	Restore	
0011	1000	Shift	3
0000		Subtract	
0000	1001	$Q_0 = 1$	
0001	0010	Shift	4
1110		Subtract	
0001	0010	Restore	

(7) / (3)

A	Q	M = 1101	
0000	0111	Initial values	
0000	1110	Shift	1
1101		Add	
0000	1110	Restore	
0001	1100	Shift	2
1110		Add	
0001	1100	Restore	
0011	1000	Shift	3
0000		Add	
0000	1001	$Q_0 = 1$	
0001	0010	Shift	4
1110		Add	
0001	0010	Restore	

(7) / (-3)

Příklady (2/2)

A	Q	M = 0011	
1111	1001	Initial values	
1111	0010	Shift	1
0010		Add	
1111	0010	Restore	
1110	0100	Shift	2
0001		Add	
1110	0100	Restore	
1100	1000	Shift	3
1111		Add	
1111	1001	$Q_0 = 1$	
1111	0010	Shift	4
0010		Add	
1111	0010	Restore	

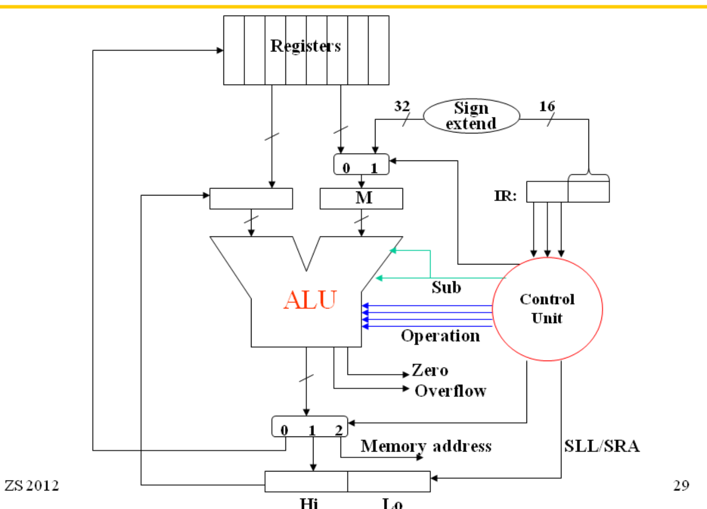
(-7) / (3)

A	Q	M = 1101	
1111	1001	Initial values	
1111	0010	Shift	1
0010		Subtract	
1111	0010	Restore	
1110	0100	Shift	2
0001		Subtract	
1110	0100	Restore	
1100	1000	Shift	3
1111		Subtract	
1111	1001	$Q_0 = 1$	
1111	0010	Shift	4
0010		Subtract	
1111	0010	Restore	

(-7) / (-3)

Schéma dělení procesoru MIPS

Procesor MIPS



MIPS speciální registry – Lo, Hi, Div, Divu, mfhi, mflo

Závěr

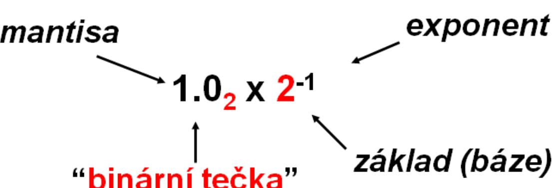
- Násobení => Posuv-&-součet
- Násobení bez znaménka = násobení se znaménkem
- Pro násobení se znaménkem se používá Boothův algoritmus
  - Test dvou bitů
  - Postup po jednom bitu
- MIPS má speciální (vyhrazené) registry (Hi, Lo) a dvě instrukce (`mult`, `multu`)

MIPS

- Operace násobení a dělení využívá existující hardware
  - ALU a posuvovou jednotku
- Extra hardware: 64-bitový registr pro operace SLL/SRA
  - Hi obsahuje zbytek (mfhi)
  - Lo obsahuje podíl (mflo)
- Instrukce
  - Div: dělení se znaménkem
  - Divu: dělení bez znaménka
- MIPS ignoruje přetečení ?
- Dělení nulou se musí testovat softwarově !

Binární notace floating point, proč se používá tato forma

Binární notace



- Aritmetika čísel FP
  - Binární tečka nemá pevnou polohu (jako tomu bylo u čísel typu integer)
  - V jazyce C se taková proměnná deklaruje jako `float`

Počítače a normalizovaná čísla

- Protože počítače pracují „pouze“ s binárními čísly, budeme je vyjadřovat pomocí normalizovaného vyjádření (scientific notation) s použitím binární tečky.
- Proč se používá právě tato forma?
  - Zjednodušuje výměnu dat, protože FP-čísla pak mají stejný tvar.
  - Zjednodušuje FP-algoritmy, protože čísla jsou vždy v této formě.
  - Zvyšuje se přesnost zobrazení, protože se nezobrazují nevýznamné úvodní nuly, naopak, vytváří se prostor pro cifry napravo od binární tečky.

Standard IEEE 754 floating point FP

Standard IEEE 754 FP

- Používán téměř ve všech počítačích (od r. 1980)
  - Přenositelnost FP programů
  - Kvalita počítačové aritmetiky FP čísel
- Znaménkový bit: { 1 znamená záporné číslo; 0 znamená kladné číslo
- Mantisa:
  - Prvá 1 je u normalizovaných čísel implicitní
  - (1 + 23) bitů jednoduchá, (1 + 52) bitů dvojitá
  - vždy platí: $0 < Mantisa < 1$
- 0.0 do pravidla nezapadá, má zvláštní vyjádření

$$(-1)^S * (1 + Mantisa) * 2^{Exp}$$

Exponent v IEEE 754, kód s posunutou nulou

Exponent v normě IEEE 754

- Operovat s FP čísly lze i bez FP hardware
  - Setřídění FP čísel použitím komparace pro čísla typu integer!
- Rozdělit FP číslo na 3 složky: porovnat znaménka, pak exponenty a nakonec mantisy
- Rychlejší (v ideálním případě, jednoduchá komparace při vhodném rozložení ve slově)
  - Nejvyšší bit je znaménko (záporné < kladné)
  - Následuje exponent, větší exponent => větší #
  - Nakonec mantisa: stejný exponent => větší # má větší mantisu

Exponent – kód s posunutou nulou

- Dvojkový komplement je pro exponent nefunkční
- Nejmenší exponent: 00000001_2
- Největší exponent: 11111110_2
- Posun: číslo, přičtené k reálnému exponentu
  - 127 pro jednoduchou přesnost
  - 1023 pro dvojitou přesnost
- $1.0 * 2^{-1}$

0	0111 1110	0000 0000 0000 0000 0000 000

$$(-1)^S * (1 + \text{Mantisa}) * 2^{(\text{Exponent - Posun})}$$

Mantisa – floating point, metody

Mantisa

- Metoda 1 (zlomky):
  - Desítkově: $0.340_{10} => 340_{10}/1000_{10} => 34_{10}/100_{10}$
  - Binárně: $0.110_2 => 110_2/1000_2 (6_{10}/8_{10}) => 11_2/100_2 (3_{10}/4_{10})$
  - Pomůže porozumět významu mantisy
- Metoda 2 („výčet hodnot“):
  - Konverze tzv. „scientific notation“ čísel
  - Desítkově: $1.6732 = (1x10^0) + (6x10^{-1}) + (7x10^{-2}) + (3x10^{-3}) + (2x10^{-4})$
  - Binárně: $1.1001 = (1x2^0) + (1x2^{-1}) + (0x2^{-2}) + (0x2^{-3}) + (1x2^{-4})$
  - Vhodné pro rychlý výpočet hodnoty mantisy

Převod z binárního a do binárního tvaru floating point, převod periodických čísel

Převod z binárního do desítkového tvaru FP

0	0110 1000	101 0101 0100 0011 0100 0010

- Znaménko: 0 => kladné
- Exponent:
  - $0110\ 1000_2 = 104_{10}$
  - Výpočet posunu: 104 - 127 = -23
- Mantisa:
  - $1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$
    $= 1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$
    $= 1.0 + 0.666115$
- Vyjadřuje: $1.666115*2^{-23} \sim 1.986*10^{-7}$

Implicitní část mantisy

Převod z desítkového do bin. tvaru FP (1/2)

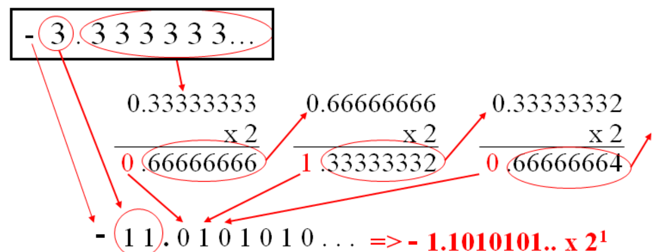
- Jednoduchý případ: Je-li jmenovatel mocninou 2 (2, 4, 8, 16, atd.), je to snadné.
- Binární FP reprezentace čísla -0.75
  - -0.75 = -3/4
  - $-11_2/100_2 = -0.11_2$
  - Normalizováno na $-1.1_2 \times 2^{-1}$
  - $(-1)^S \times (1 + \text{Mantisa}) \times 2^{(\text{Exponent}-127)}$
  - $(-1)^1 \times (1 + .100\ 0000 \ldots 0000) \times 2^{(126-127)}$

1	0111 1110	100 0000 0000 0000 0000 0000

Převod z desítkového do bin. tvaru FP (2/2)

- Jmenovatel není mocninou 2
  - Číslo nelze reprezentovat přesně
  - Mantisa má obvykle dost bitů na dosažení požadované přesnosti
  - Obtížnější krok: výpočet mantisy
- Racionální čísla mají periodu
- Převod
  - Zapište binární číslo s opakující se periodou.
  - Bity přesahující mantisu vpravo ořízněte (různý počet pro jednoduchou vs. dvojitou přesnost).
  - Odvoďte znaménko a pole exponentu a mantisy.

Převod z desítkového do binárního tvaru



- Mantisa: 101 0101 0101 0101 0101 0101
- Znaménko: záporné => 1
- Exponent: 1+ 127 = 128_{10} = $1000\ 0000_2$

1	1000 0000	101 0101 0101 0101 0101 0101

Standard IEEE 754 floating point – základní přesnost počet bitů

Standardy IEEE 754 (Floating Point)

- IEEE respektoval volby návrhu a doporučil velikost exponentu 8 bitů a 23 bitů pro mantisu (za předpokladu, že délka slova je 32 bitů).
- Tento formát je použit u MIPS a u většiny počítačů po roce 1980 – jedná se dobré kompromisní řešení.

s	exponent	mantisa
1 bit	8 bitů	23 bitů

Reprezentované číslo = $(-1)^S \times F \times 2^E$
kde S, F a E jsou pole znaménka, exponentu a mantisy
(1 v poli s znamená zápornou hodnotu čísla)

floating point výjimky, přetečení, podtečení

FP výjimky

- Standard IEEE 754 pokrývá velmi velký rozsah reálných čísel, která mohou být vyjádřena, od nejmenších 2.0×10^{-38} k největším 2.0×10^{38}.
- ! Je třeba podotknout, že rozsah je velký, ale ne nekonečný…
  - Přetečení v pohyblivé řádové čárce nastává, jestliže vypočtený exponent výsledku je příliš velký a nelze ho vyjádřit v poli exponentu (příliš velké číslo). ($> 2.0 \times 10^{38}$)
  - Podtečení v pohyblivé řádové čárce nastává, jestliže vypočtený exponent výsledku je příliš malý a nelze ho vyjádřit v poli exponentu (příliš malé číslo – co do abs. hodnoty ($>0, < 2.0 \times 10^{-38}$)

Floating point dvojitá přesnost, počet bitů, double v C

Dvojitá přesnost

- Aby se bylo možno s těmito případy lépe vyrovnat IEEE 754 standard zahrnuje specifikaci formátu *double precision*, ve které jsou použita dvě slova k zobrazení čísla.
- Exponent je rozšířen na 11 bitů a mantisa na 52 bitů…

s	exponent	mantisa
1 bit	11 bitů	20 bitů

mantisa (pokračování)
32 bitů

Čísla s dvojitou přesností (Double Precision FP)

- Pro zobrazení čísla jsou použita dvě slova (64 bitů)

31	30 … 20	19 … 0
S	Exponent	Mantisa
1 bit	11 bitů	20 bitů

Mantisa (pokračování)
32 bitů

- V jazyce C proměnná deklarována jako `double`
- Reprezentuje čísla v rozsahu od nejmenšího 2.0×10^{-308} až po největší 2.0×10^{308}
- Primární výhodou je větší přesnost (52 bitů) (přesnost určuje mantisa !!!)

Floating point výhody dvojité přesnosti

Výhody dvojité přesnosti

- Tento formát dovoluje vyjádřit čísla ve větším rozsahu a to od $2.0\text{x}10^{-308}$ do $2.0\text{x}10^{308}$.
- Přestože primárním důvodem rozšíření je podstatné zvýšení přesnosti zobrazení, bylo zvětšeno i pole pro zobrazení exponentu.

Floating point optimalizace

Optimalizace

- Protože bit nalevo od binární tečky je trvale „1“ a nenese proto žádnou informaci, rozhodli se návrháři normy IEEE 754 tento bit nezahrnout do standardního formátu.
- Čísla IEEE 754 mají mantisu o délce 24 bitů (1 implicitní a 23 ukládaných) pro jednoduchou přesnost a 53 bitů mantisy (1 implicitní a 52 ukládaných) ve dvojité přesnosti.
- Nula je zobrazena speciálním způsobem a to s nulou v exponentu, v mantise i ve znaménku.

Další optimalizace…

- Mantisa využívá „skrytou“ jedničku, hodnota čísla je pak rovna:

  $(-1)^S \times (1 + \text{mantisa}) \times 2^E$

  kde bity mantisy představují zlomek mezi nulou a jedničkou.
- Pro zjednodušení a zrychlení komparace čísel bylo vhodně zvoleno i pořadí jednotlivých polí v zobrazení čísla.
  - To je hlavní důvod proč znaménkový bit leží v MSB.

Kódování exponentu – kód s posunutou nulou v IEEE 754

Kódování exponentu

- Jako vhodná forma pro exponent se jeví takové zobrazení, u kterého je nejmenší (záporný) exponent zobrazen jako 00..00 a největší kladný exponet jako 11..11.
- Tato konvence se nazývá *kód s posunutou nulou (biased encoding)* – tento posun je přičten bez znaménka k exponentu. Tak je získán obsah pole exponentu.
- Standard IEEE 754 používá posun *(bias)* 127.
- Proto skutečný exponent –1 je kódován jako (-1)+127=126 (0111 1110) a exponent 1 je kódován jako 1+127=128 (1000 0000).

Kód s posunutou nulou v IEEE 754

- Z toho vyplývá, že hodnotu čísla kódovaného podle normy IEEE 754 určíme podle výrazu…

  $(-1)^{S} \times (1 + \text{mantisa}) \times 2^{(\text{exponent} - \text{bias})}$

- Stejný výraz platí i pro dvojnásobnou přesnost, pouze s tím rozdílem, že posun je pak roven1023. (00..00) je opět nejmenší exponent, a (11..11) představuje největší možný exponent.

Dekódování IEEE 754, jednoduchá, dvojitá přesnost

Příklad: dekódování IEEE 754

- Zakódujeme $(-0.75)_{10}$ podle IEEE 754.
- Zápis ve dvojkové soustavě…$(-0.11)_2$.
- Normalizovaná forma…$(-1.1)_2 \times 2^{-1}$.
- Požadovaný tvar…

  $(-1)^{S} \times (1 + \text{mantisa}) \times 2^{(\text{exponent} - \text{bias})}$

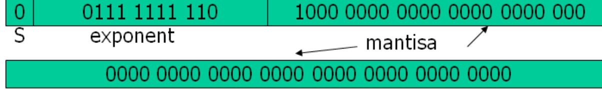
- Převod do požadovaného tvaru…

  $(-1)^{1} \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000) \times 2^{(126 - 127)}$

- Pro jednoduchou přesnost podle IEEE 754…

S	exponent	mantisa
1	0111 1110	1000 0000 0000 0000 0000 000

- Pro dvojnásobnou přesnost podle IEEE 754…



Příklad: dekódování IEEE 754

- Budeme dekódovat číslo podle IEEE 754…

S	exponent	mantisa
1	1000 0001	0100 0000 0000 0000 0000 000

- Pro reprezentaci čísla platí výraz…

  $(-1)^{S} \times (1 + \text{mantisa}) \times 2^{(\text{exponent} - \text{bias})}$

- Dosazením hodnot polí…

  $(-1)^{1} \times (1 + 0.25) \times 2^{(129 - 127)}$

- Vyčíslením výrazu…

  $-1 \times 1.25 \times 2^{2} = -1.25 \times 4 = -5.0$

- Uvedený obsah polí tedy vyjadřuje $-(5.0)_{10}$.

Sčítání floating point čísel – princip, příklad (v příkladu je zjednodušení)

Sčítání FP čísel

- Nyní když víme, jak se čísla v pohyblivé řádové čárce zobrazují, můžeme s nimi provádět operace – např. sčítání.
- Nejlépe lze porozumět této operaci tak, že ji sami krok po kroku vyzkoušíme.
- Pak se můžeme pokusit přidat hardware k ALU, který tyto kroky bude provádět podobně, jako jsme je dělali v předchozím případě „ručně".
- Sečteme krok po kroku čísla $9.999 \times 10^1 + 1.610 \times 10^{-1}$ (pro přehlednost použijeme desítkovou soustavu, ve dvojkové by operace probíhaly stejně).

Příklad - použité zjednodušení

- Zobrazení FP čísel v počítačích má pevnou délku.
- Pro jednoduchost budeme uvažovat formát, který používá 4 dekadické cifry pro mantisu a 2 dekadické cifry pro exponent.
- Stejný princip lze použít i na čísla podle standardu IEEE 754, uvedené zjednodušení je použito kvůli ilustraci procesu sčítání a ilustraci kompromisů s ohledem na omezenou délku zobrazení.

Sčítání floating point čísel – vyrovnání exponentů

Sčítání FP čísel
Krok 1: Vyrovnání exponentů

- Abychom správně sečetli čísla, je nutné upravit polohu desetinné tečky jednoho z operandů (abychom sčítali cifry se stejnou vahou).
- V našem případě upravujeme exponent čísla 1.610×10^{-1} tak, aby odpovídal exponentu čísla 9.999×10^1.
- $1.610 \times 10^{-1} = 0.1610 \times 10^0 = 0.01610 \times 10^1$
- Nezapomeňte, že můžeme ukládat pouze 4 cifry mantisy, takže dostaneme hodnotu 0.016×10^1 (ztratili jsme na přesnosti vlivem omezení HW prostředků – délka zobrazení).

Sčítání floating point čísel – sečtení mantis

Sčítání FP čísel
Krok 2: Sečtení mantis

- Potom, co byly srovnány exponenty, můžeme provést operaci součtu mantis…

```
   9.999
+  0.016
 -------
  10.015
```

- Součtem dostáváme výsledek 10.015×10^1.

Sčítání floating point čísel – normalizace součtu

Sčítání FP čísel
Krok 3: Normalizace součtu

- Nakonec provedeme normalizaci součtu – převedení do standardního tvaru, který byl operací součtu porušen.
- $10.015 \times 10^1 = 1.0015 \times 10^2$
- Nezapomeňte, že i zde je nutné provést kontrolu, zda nenastalo přetečení nebo podtečení. V tomto případě k chybám nedošlo, exponent výsledku je roven 2 a lze ho zobrazit.

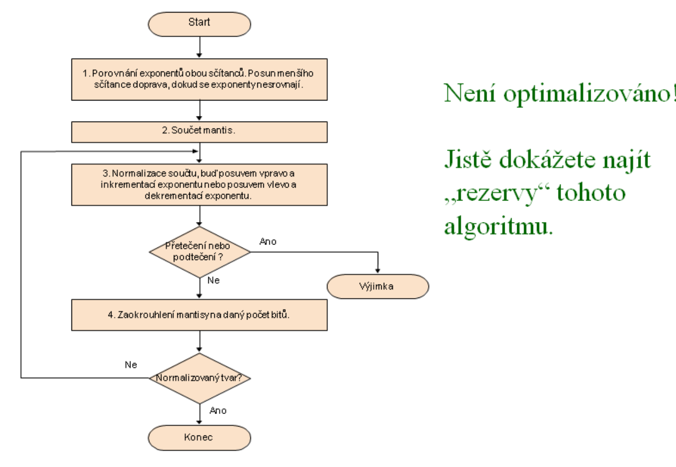
Sčítání floating point čísel – zaokrouhlení

Sčítání FP čísel
Krok 4: Zaokrouhlení

- Při sčítání vznikl výsledek, který překračuje nároky na délku zobrazení => musíme výsledek zaokrouhlit.
- Použijeme staré zaokrouhlovací pravidlo ze základní školy, 1.0015×10^2 zaokrouhlíme na 1.002×10^2.
- Nutno poznamenat, že i zaokrouhlením lze opět dostat nenormalizované číslo a je nutno se pak vrátit ke kroku 3.

Algoritmus sčítání floating point čísel

Algoritmus součtu FP-čísel



Není optimalizováno!

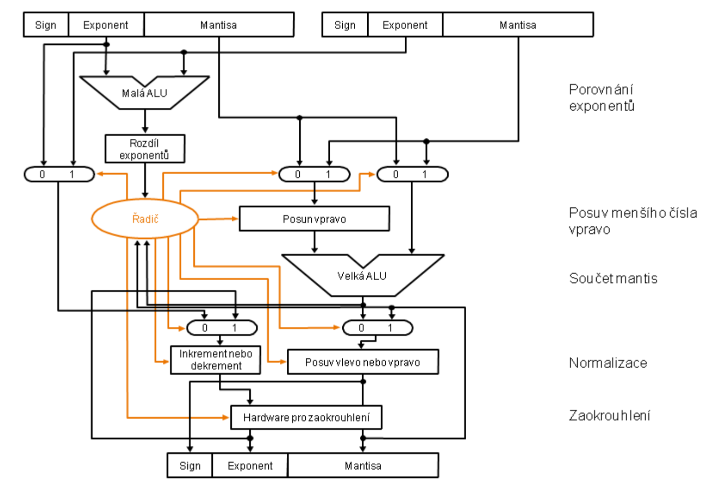
Jistě dokážete najít „rezervy" tohoto algoritmu.

Hardware sčítání floating point čísel – schéma zapojení

Hardware pro součet FP-čísel

- Moderní procesory mají často implementováno technické vybavení (hardware) pro rychlé provádění FP-operací, jako např. sčítání.
- Generický návrh takové implementace obsahuje dvě ALU, řídící jednotku, posuvný registr, mini-ALU (pro inkrement/dekrement) a obvody pro provedení zaokrouhlovacího procesu.

Hardware pro součet FP-čísel



Násobení floating point – 5 kroků

Násobení FP čísel

- Když jsme zvládli jednoduchou operaci sčítání FP čísel, můžeme přikročit ke složitějšímu problému – násobení FP čísel.
- Podobně jako v předchozím případě budeme postupovat krok po kroku.
- Opět použijeme k zobrazení desítkovou soustavu. Mantisa bude zobrazena na 4 dekadických cifrách, exponent na 2 dekadických cifrách.
- Uvědomte si, že stejnou proceduru můžete aplikovat na binárně zobrazená čísla podle normy IEEE 754, jedná se jen o zjednodušený příklad.
- Budeme násobit čísla 1.110 x 10^{10} a 9.200 x 10^{-5}.

Chyba v závorkách se přičítá 127

FP násobení Krok 1: Sečtení exponentů

- Výpočet exponentu součinu je jednoduchý. Sečteme exponenty násobence a násobitele.
- Sečteme 10 a (-5), dostaneme 5 – exponent součinu je roven 5.
- Nyní totéž provedeme s posunutými exponenty (protože v této formě se exponenty ukládají), posun je 127.
  - (10+137) + (-5+137) = 137+122 = 259
  - To není správný výsledek: 259 - 127 = 132 a nikoliv 5.
  - Posun jsme započetli dvakrát! Proto je nutno posun odečíst: 132 – 127 = 5 (správný výsledek!).

FP násobení Krok 2: Násobení mantis

- Nyní vynásobíme mantisy…

```
      1.110
x     9.200
---------
      0000
     0000
    2220
   9990
---------
10212000
```

- Desetinná tečka je umístěna po šesté cifře zprava, protože násobitel i násobenec mají tři desetinná místa – součin je roven 10.212000.
- Předpokládejme, že můžeme uložit pouze tři cifry vpravo od desetinné tečky, bude součin roven 10.212 x 10^5.

FP násobení
Krok 3: Normalizace součinu

- Součin je třeba normalizovat, protože zatím nemá požadovaný normalizovaný tvar, ve kterém ho lze uložit do paměti.
- $10.212 \times 10^5 = 1.0212 \times 10^6$
- Připomeňte si, že je třeba zkontrolovat, zda nedošlo k přetečení nebo k podtečení. V tomto případě žádná z uvedených chyb nenastala.

FP násobení
Krok 4: Zaokrouhlení

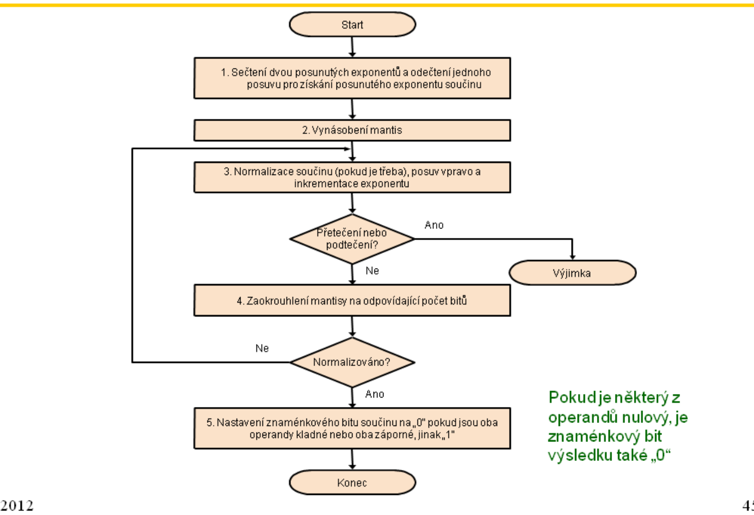
- Protože provedením operace se počet cifer zvýšil, je třeba provést zaokrouhlení výsledku.
- Použitím zaokrouhlovacích pravidel (ze základní školy) dostaneme: 1.0212×10^6 zaokrouhleno dává 1.021×10^6.
- Nakonec je opět třeba ověřit, zda zůstal zachován normalizovaný tvar stejně, jako tomu bylo v případě operace sčítání.

FP násobení
Krok 5: Určení znaménka

- Nakonec určíme znaménko součinu.
- Jsou-li znaménka obou operandů shodná, výsledek je kladný, v opačném případě záporný (násobení nulou neuvažujeme).
- V našem případě byly oba operandy kladné a proto i výsledek je kladný.
- Konečný výsledek: $+1.021 \times 10^6$.

Algoritmus násobení floating point

Algoritmus násobení FP čísel



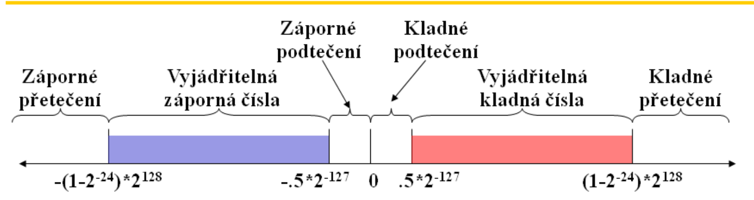
Dělení floating point

Dělení FP čísel

- Dělení FP čísel je složitá operace.
- K našemu postupnému budování hardware (metodou pokus-úspěch) zmiňme ještě některé urychlené komerční metody.
  - Newtonova iterační metoda se používá k nalezení převrácené hodnoty jednoho z operandů. Vynásobení optimalizovaným hardwarem s druhým operandem dostáváme podíl.
  - Sekvenční algoritmy (bit po bitu)
    - Binární dělení s obnovou zbytku
    - Binární dělení bez obnovy zbytku
    - SRT dělení – odhad většího počtu bitů podílu pomocí tabulek (Intel Pentium používá podobnou metodu).

Speciální hodnoty floating point – NaN (Not a number), denormalizovaný čísla

Speciální hodnoty



Speciální hodnoty	Exponent	Mantisa
+/- 0	0000 0000	0
denormalizované číslo	0000 0000	nenulová
NaN	1111 1111	nenulová
+/- nekonečno	1111 1111	0

Not a Number

- Co je výsledkem operace: `sqrt(-4.0)` or `0/0`?
  - Jestliže nekonečno není chyba, tohle by také nemělo.
  - Nazývá se Not a Number (NaN)
  - Exponent = 255, mantisa je nenulová
- Aplikace
  - někdy lze „NaNy“ využít při ladění programu
  - šíření v návazných operacích: op(NaN, X) = NaN

Denormalizovaná čísla

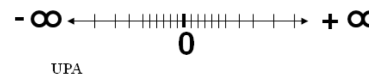
- Problém: Kolem nuly se mezi reprezentovatelnými čísly vytváří mezera
  - Nejmenší kladné číslo: $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$
  - Druhé nejmenší kladné číslo: $b = 1.001_2 * 2^{-126} = 2^{-126} + 2^{-150}$
  - $a - 0 = 2^{-126}$
  - $b - a = 2^{-150}$

Mezera! b

- ∞ + ∞

0 a

- Řešení:
  - Denormalizovaná čísla: nemají úvodní 1
  - Nejmenší kladné číslo: $a = 2^{-150}$
  - Druhé nejmenší kladné číslo: $b = 2^{-149}$



Omyl při práci s FP čísly – operace nejsou asociativní

Častý omyl při práci s FP čísly

- FP operace Add, Sub jsou asociativní: CHYBA!
  - $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, a $z = 1.0$
  - $x + (y + z) = -1.5\text{x}10^{38} + (1.5\text{x}10^{38} + 1.0)$
    $= -1.5\text{x}10^{38} + (1.5\text{x}10^{38}) = \underline{0.0}$
  - $(x + y) + z = (-1.5\text{x}10^{38} + 1.5\text{x}10^{38}) + 1.0$
    $= (0.0) + 1.0 = \underline{1.0}$
- Floating Point operace Add, Sub nejsou asociativní!
  - Proč? Výsledky operací s čísly FP **aproximují** výsledky operací s reálnými čísly
  - 1.5×10^{38} je mnohem větší než 1.0, takže $1.5 \times 10^{38} + 1.0$ v reprezentaci floating point dává stále 1.5×10^{38}

Zaokrouhlování a přesnost floating point – typy zaokrouhlování

Zaokrouhlení a přesnost

- Při práci s čísly v pohyblivé řádové čárce přichází v potaz další fenomén.
- Číslo, uložené v FP formátu (běžně tedy v IEEE 754), představuje pouze aproximaci reálného čísla (protože počet míst za desetinnou/binární tečkou je omezený).
- I nejlepší počítače mohou pro zobrazení reálných čísel pouze vybírat aproximaci FP číslem, nejbližším zobrazovanému reálnému číslu.
- Pro dosažení co nejlepších výsledků používá norma IEEE 754 několik režimů pro zaokrouhlování FP čísel.

Zaokrouhlování

- Matematika reálných čísel => zaokrouhlování
- Zaokrouhlování také při konverzi typů
  - Double ⇔ single precision ⇔ integer
- Zaokrouhlení směrem k + nekonečno
  - VŽDY zaokrouhluje "nahoru": 2.001 => 3; -2.001 => -2
- Zaokrouhlení směrem k - nekonečno
  - VŽDY zaokrouhluje "dolů": 1.999 => 1; -1.999 => -2
- Oříznutí
  - Vypuštění nejméně významných bitů (zaokrouhlení k 0)
- Zaokrouhlení k (nejbližšímu) sudému (default)
  - 2.5 => 2; 3.5 => 4

Podpora zaokrouhlování – guard bit, round bit

Standardní podpora zaokrouhlení

- V dosud uvedených příkladech jsme poněkud „neopatrně“ zacházeli s délkou zobrazení mezivýsledků.
- Kdybychom „ořízli“ vše co reprezentujeme na délku zobrazení, nemohli bychom zaokrouhlovat, protože tak bychom ztratili potřebnou informaci.
- Z toho důvodu používá norma IEEE 754 vždy dva extra bity, které prodlužují mezivýsledky zprava během průběžných operací. Nazývají se *guard bit* a *round bit*.
- Tyto bity jsou vypočítávány během výpočtu podobně jako všechny ostatní. Na konci algoritmu jsou pak použity v procesu zaokrouhlení výsledku (po zaokrouhlení jsou uvolněny).

Sticky bit

„Sticky“ bit

- Norma IEEE 754 dále zavádí třetí speciální bit, který se nazývá *sticky bit*.
- Tento bit leží úplně napravo a nastavuje se, jestliže jsou za *round bitem* nenulová data.
- Díky tomuto bitu dosahuje počítač stejných výsledků, jako kdyby byly mezivýsledky počítány s neomezenou přesností a pak zaokrouhleny.

Podpora floating point u MIPS – funkce MIPS pro floating point

Podpora pohyblivé řádové čárky u MIPS

- Architektura MIPS podporuje formáty IEEE 754 pro jednoduchou i dvojnásobnou přesnost…
  - **FP addition** – jednoduchá (*add.s*) a dvojitá (*add.d*)
  - **FP subtraction** – jednoduchá (*sub.s*) a dvojitá (*sub.d*)
  - **FP multiply** – jednoduchá (*mul.s*) a dvojitá (*mul.d*)
  - **FP divide** – jednoduchá (*div.s*) a dvojitá (*div.d*)
  - **FP comparison** – jednoduchá (*c.x.s*) a dvojitá (*c.x.d*), kde *x* je jedna z: equal (*eq*), not equal (*neq*), less than (*lt*), greater than (*gt*), less than or equal to (*le*) nebo greater than or equal (*ge*)
  - **FP branch** – pozitivní (*bc1t*) a negativní (*bc1f*)
  - (FP komparace nastavuje speciální bit na *true* nebo *false* a FP branch rozhoduje na základě této podmínky).

!? Řešení podmínek FP instrukcí klasickým způsobem ?!

MIPS floating point – problémy, instrukce, koprocesor

Architektura MIPS - FP (1/2)

- Rozdílné FP instrukce pro:
  - jednoduchou přesnost: `add.s, sub.s, mul.s, div.s`
  - dvojitou přesnost: `add.d, sub.d, mul.d, div.d`
- Tyto instrukce jsou mnohem složitější, než odpovídající operace s typem integer
- Problémy:
  - Pro celý procesor je nepříznivé, jestliže se doba zpracování intrukcí zásadně liší.
  - Obecně platí, že během zpracování určitého programu většinou data nemění svůj charakter (FP < = > Int).
  - Některé programy vůbec neprovádí FP operace
  - Hardware pro rychlé provádění FP operací je značně rozsáhlý v porovnání s hardwarem pro operace integer

Architektura MIPS - FP (2/2)

- 1990 Řešení: vyhrazený čip, který provádí pouze FP.
- Koprocesor 1: FP čip
  - Obsahuje 32 32-bitových registrů: **$f0, $f1, …**
  - Většina registrů je specifikována v `.s` a `.d` instrukcích ($f)
  - Separátní load a store: **lwc1** a **swc1** (“load word coprocessor 1”, “store …”)
  - Dvojitá přesnost: **konvence**, sudý/lichý pár obsahuje jedno DP FP číslo: `$f0/$f1`, … , `$f30/$f31`
- 1990 Počítače často obsahují více vyhrazených obvodů:
  - Procesor: provádí běžné operace
  - Koprocesor 1: pouze FP operace;
- Přenos dat mezi hlavním procesorem a koprocesorem:
  - **mfc0, mtc0, mfc1, mtc1,** atd.

Registry MIPS pro floating point

FP sada registrů MIPS

- Návrháři MIPS se rozhodli zařadit oddělenou sadu FP registrů $f0, $f1, atd.
- Pro plnění a ukládání FP registrů jsou také použity vyhrazené instrukce – *lwc1* a *swc1*. Jako bázové registry jsou použity normální registry z integer sady.
- Následující příklad ukazuje načtení dvou čísel v jednoduché přesnosti, jejich sečtení a uložení výsledku…

```
lwc1   $f2, x($sp)   # load 32-bit FP num into $f4
lwc1   $f6, y($sp)   # load 32-bit FP num into $f6
add.s  $f2,$f4,$f6   # $f2=$f4+$f6, single precision
swc1   $f2, z($sp)   # store 32-bit FP num from $f2
```

- Registr pro dvojitou přesnost je tvořen párem registrů jednoduché přesnosti (sudý/lichý), používající jméno sudého.

Floating point C vs MIPS

C => MIPS

```
Float f2c (float fahr) {
  return ((5.0 / 9.0) * (fahr – 32.0));
}
```

```
F2c:
  lwc1  $f16, const5($gp)    # $f16 = 5.0
  lwc1  $f18, const9($gp)    # $f18 = 9.0
  div.s $f16, $f16, $f18     # $f16 = 5.0/9.0
  lwc1  $f20, const32($gp)   # $f20 = 32.0
  sub.s $f20, $f12, $f20     # $f20 = fahr – 32.0
  mul.s $f0, $f16, $f20      # $f0 = (5/9)*(fahr-32)
  jr    $ra                  # return
```

Podpora FP u PowerPC – multiply-add

Podpora FP u architektury PowerPC

- Architektura PowerPC je z hlediska zobrazení čísel v pohyblivé řádové čárce podobná MIPS. Existuje několik rozdílů, které pramení hlavně z toho, že PowerPC je novější a pokročilejší architektura.
  - Neobsahuje žádné registry Hi a Lo – PowerPC instrukce operují přímo nad registry.
  - PowerPC má 32 registrů pro jednoduchou přesnost a 32 registrů pro dvojitou přesnost, tedy dvakrát tolik, než architektura MIPS.
  - Power PC zavádí také instrukci multiply-add (více na následujícím snímku).

Instrukce multiply-add

- Instrukce PowerPC multiply-add pro FP operandy čte tři operandy, dva z nich vynásobí, třetí připočítá k součinu a výsledek uloží do registru, kde ležel třetí operand.
  - dvě instrukce MIPS = jedna PowerPC instrukce
  - tato instrukce provádí na závěr jediné zaokrouhlení; dvě oddělené instrukce = dvě zaokrouhlení a tím i nižší přesnost výsledku
- Tato instrukce je také použita v PowerPC při provádění FP dělení (použitím Newtonovy iterační metody, jak bylo zmíněno) – přesnost dělení byla primárním důvodem pro redukci počtu zaokrouhlení (zaokrouhlení až na závěr této sdružené operace).

Podpora u IA-32 – zásobníková architektura

Podpora FP u architektury IA-32

- Podpora operací v pohyblivé řádové čárce u IA-32/x86 započala s koprocesorem 8087 v roce 1980 a velmi se lišila od architektur MIPS a PowerPC.
- Intel používá zásobníkově orientovanou architekturu s FP instrukcemi, jedná se o odlišný samostatný celek.
  - Operace Load ukládají FP čísla na vrcholek FP zásobníku a inkrementují FP stack pointer.
  - Operace Store odebírají FP čísla z vrcholku FP zásobníku, dekrementují FP stack pointer a ukládají FP čísla do paměti.

Push a pop jsou obráceně

Zásobníková FP architektura

- FP operace se provádějí se dvěma operandy na vrcholku zásobníku, operandy jsou nahrazeny jedním výsledkem (dvakrát pop, jeden push).
- Existuje také možnost provádět FP operaci s jedním operandem v paměti a druhým ležícím na vrcholku zásobníku nebo v jednom ze sedmi speciálních FP registrů.
- FP instrukce v IA-32 patří do jedné ze čtyřech skupin …
  - Přesun dat – load, load immediate, store, atd.
  - Aritmetika – add, sub, mul, div, sqrt, abs, atd.
  - Komparace – může zasílat výsledek do integer procesoru, který pak případně vykoná instrukci větvení
  - Transcendentní – sinus, kosinus, log, exp, atd.

Zásobníkově orientované stroje

Zásobníkově orientované stroje

- Tato zásobníkově orientovaná architektura se velmi liší od registrově orientované, kterou jsme se doposud zabývali.
- Data se pro provedení operací přenáší do/ze zásobníku namísto registrů procesoru.
- Tento typ architektury není neobvyklý. Některé počítače (i velmi moderní) používají podobnou architekturu …
  - Java Virtual Machine (JVM)
  - Microsoft Common Language Runtime (CLR)
  - Většina graf. HP kalkulátorů (interface, nikoliv použitý procesor)

Dvojitá rozšířená přesnost

Dvojitá rozšířená přesnost

- Zásobník v systému pohyblivé řádové čárky Intel IA-32 je široký 80 bitů => označení *double extended precision*.
- Všechna FP čísla jednoduché i dvojité přesnosti jsou konvertována do tohoto formátu, když se přesouvají z paměti do zásobníku a naopak.
- FP registry mají šířku 80 bitů.
- Leží-li jeden operand FP operace v paměti, je během operace (on-the-fly) konvertován do 80-bitového formátu.
- Tato forma není využívána kompilátory moderních programovacích jazyků, ale je k dispozici pro přímé programování v assembleru (časově náročné algoritmy).

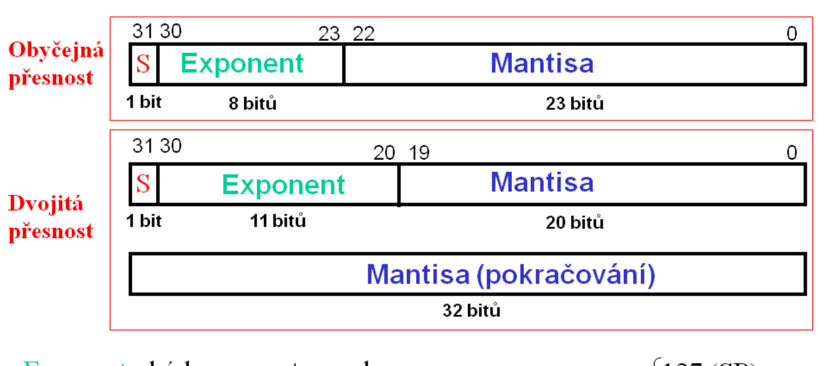
FP aproximace , reprezentace čísel FP. Jednoduchá a dvojitá přesnost, exponent posunutí

Závěr

- Čísla s pohyblivou řádovou čárkou (FP) pouze ***aproximují*** reálná čísla, která bychom chtěli používat, představují dokonce jen podmnožinu racionálních čísel.
- IEEE 754 Floating Point Standard je dnes široce akceptovaným standardem pro práci s FP aritmetikou.
- Nové prvky architektury MIPS
  - Registry (`$f0-$f31`)
  - Jednoduchá přesnost (32 bitů, $2x10^{-38}$ … $2x10^{38}$)
    - `add.s, sub.s, mul.s, div.s`
  - Dvojitá přesnost (64 bitů, $2x10^{-308}$ … $2x10^{308}$)
    - `add.d, sub.d, mul.d, div.d`
- Typ není asociován s daty, význam bitů závisí na kontextu (například *int* vs. *float*)

Reprezentace čísel FP

Normalizovaná notace: $+1.xxxx_2 * 2^{yyyy_2}$



Exponent: kód s posunutou nulou - - - - -> Posunutí {127 (SP), 1023 (DP)}

Mantisa: notace znaménko-amplituda

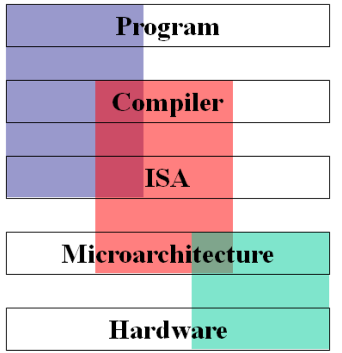
Návrh datových cest – implementace fází fetch-decode-execute, metodologie návrhu

Nové – Návrh datových cest

- **Datové cesty -** implementace fází fetch-decode-execute
- **Metodologie návrhu**
  - Určení tříd instrukcí a instrukčních formátů
  - Vytvoření sekcí datových cest pro každý instrukční fmt.
  - Sloučení sekcí tak, aby byly pokryty potřebné přesuny u MIPS
- *Otázka #1*: Co jsou to instrukční třídy?
- *Otázka #2*: Které komponenty jsou vhodné?

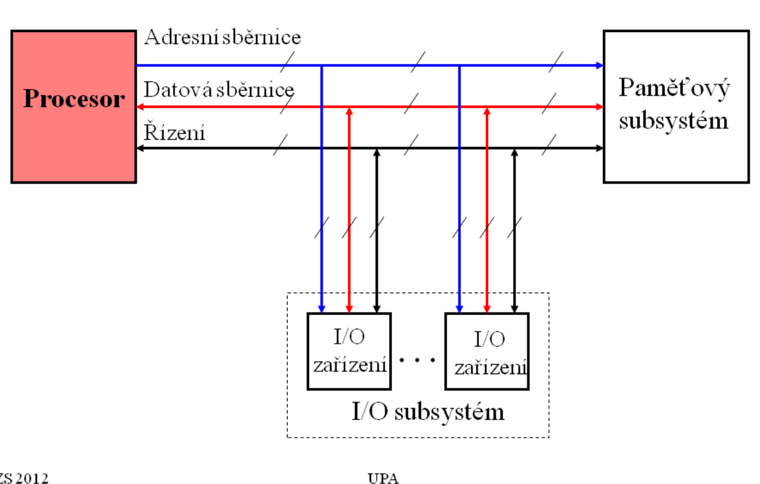
Výkony procesoru

Výkon procesoru



Organizace počítače - schéma

Organizace počítače



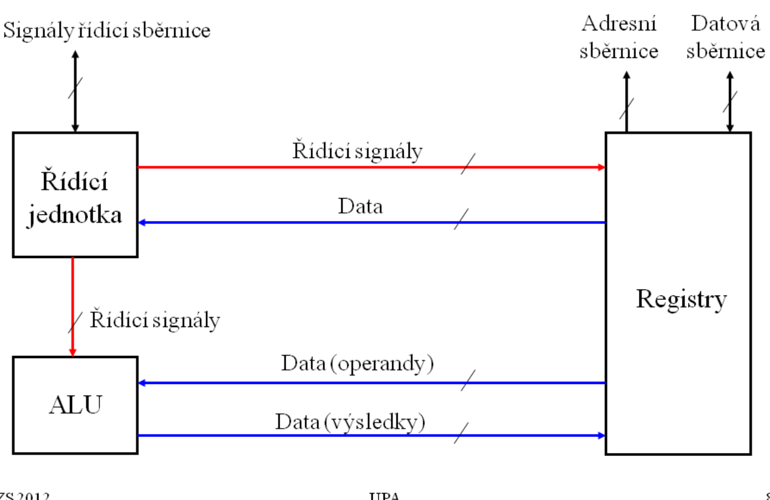
Procesor – složení, popis částí, datové cesty, řízení

Procesor

- Procesor (CPU)
  - Aktivní část počítače
  - Vykonává všechnu „práci“
    - Manipulace s daty
    - Rozhodování
- Datové cesty
  - Hardware, který vykonává všechny potřebné operace
  - ALU + registry + interní sběrnice
  - ***Výkonné prvky***
- Řízení
  - Hardware, který řídí, co se má provádět a kam se co má přesouvat
  - ***“Mozek”***

Organizace procesoru

Organizace procesoru



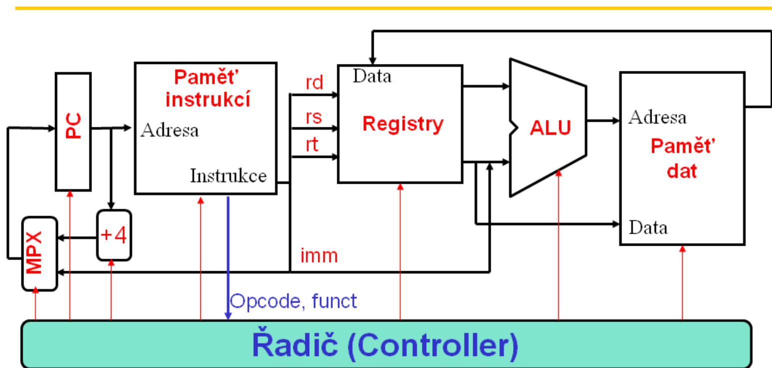
MIPS implementace

MIPS - implementace

- ISA určuje mnoho aspektů implementace
- Strategie implementace ovlivňuje hodinovou frekvenci a CPI
- Výběr atributů u MIPS pro ilustraci implementace
  - Instrukce pro práci s pamětí
    - Load word (lw)
    - Save word (sw)
  - Aritmetika čísel integer a logické instrukce
    - add, sub, and, or, a slt
  - Instrukce větvení
    - Branch if equal (beq)
    - Jump (j)

Přehled implementace - schéma

Přehled implementace



- Datové cesty umožňují přesuny obsahu registrů při provádění instrukcí
- Řízení zajišťuje provedení správných přesunů

Logika a taktování – kombinační a sekvenční prvky

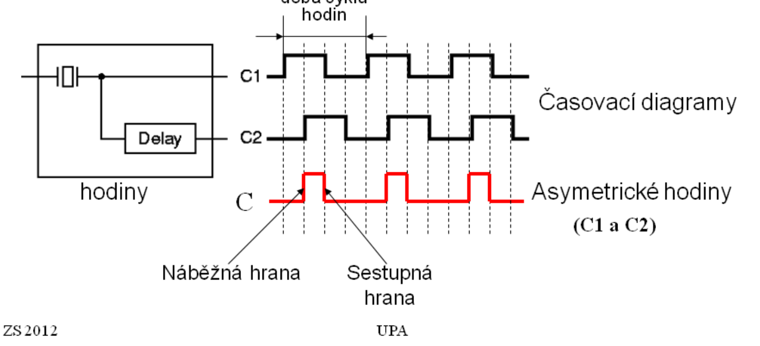
Logika a taktování

- Kombinační prvky
  - Výstupy závisí pouze na aktuálních vstupech
  - Příklad: ALU (sčítačky, multiplexery, posuvové jednotky)
- Sekvenční prvky
  - Obsahují stav (state)
  - Výstupy závisí na vstupech a na stavu
  - Vstupy: data a hodiny (clock)
  - Paměť, registry
- Signály v aserci: logická jednička (vysoká úroveň)

Metodologie taktování – taktování hranou hodin

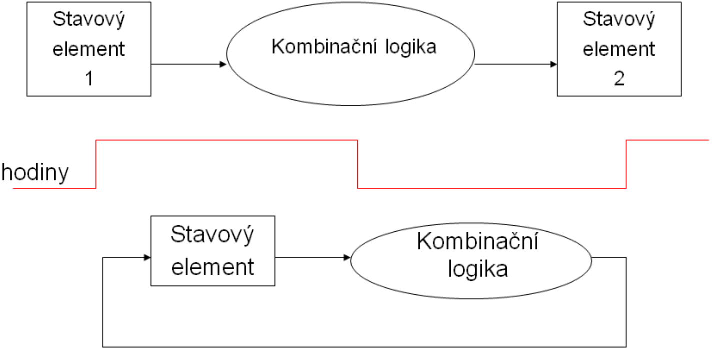
Metodologie taktování

- Určuje pořadí událostí
  - Určuje, kdy lze signály číst nebo zapisovat
- Hodiny: obvod, který generuje posloupnost pulsů



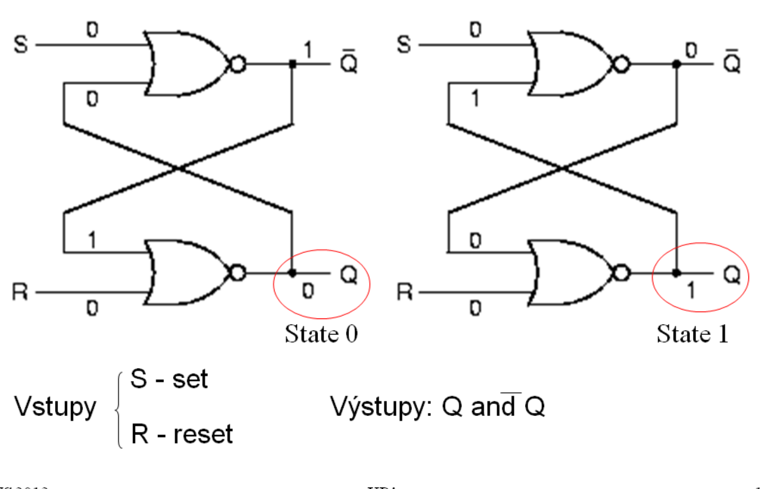
Taktování hranou hodin

- Aktivní je buď náběžná nebo sestupná hrana hodin
- Ke změnám stavu dochází pouze na aktivní hraně hodin



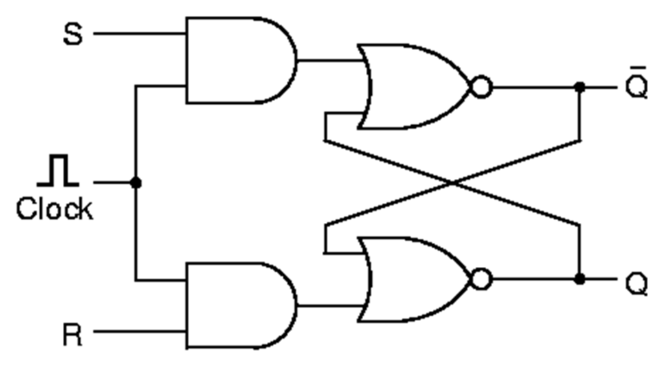
SR Latch s hradly NOR – RS klopný obvod

SR Latch s hradly NOR



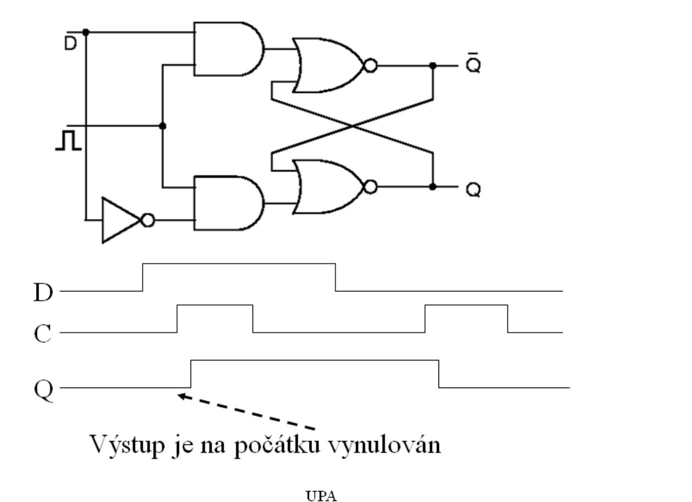
Taktovaný SR latch – taktovaný RS klopný obvod

Taktovaný SR Latch

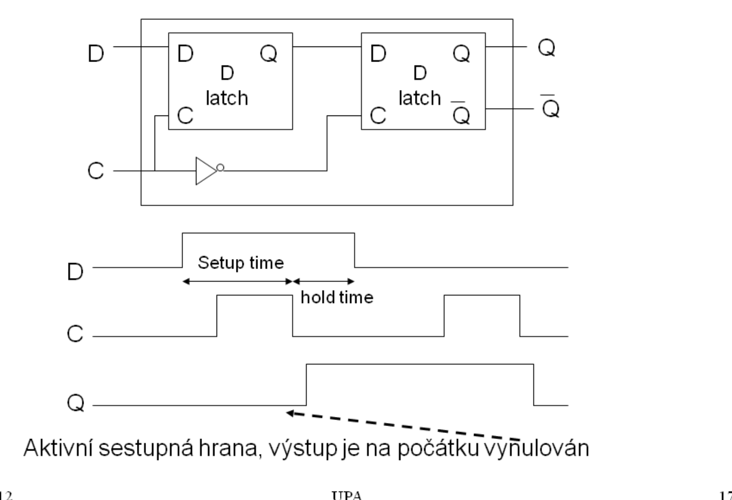


Taktovaný D klopný obvod – taktovaný D latch

Taktovaný D Latch

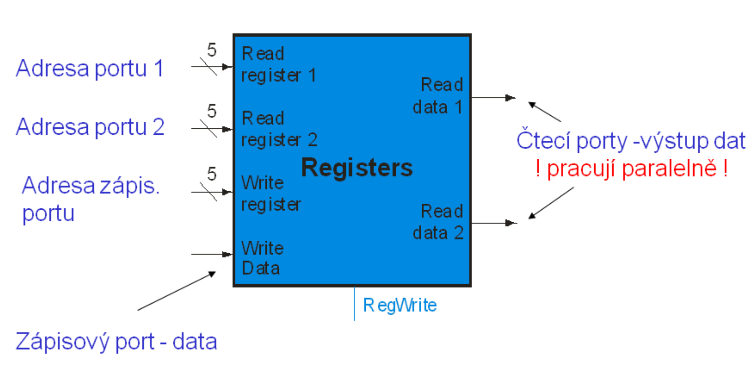


Klopný obvod typu D



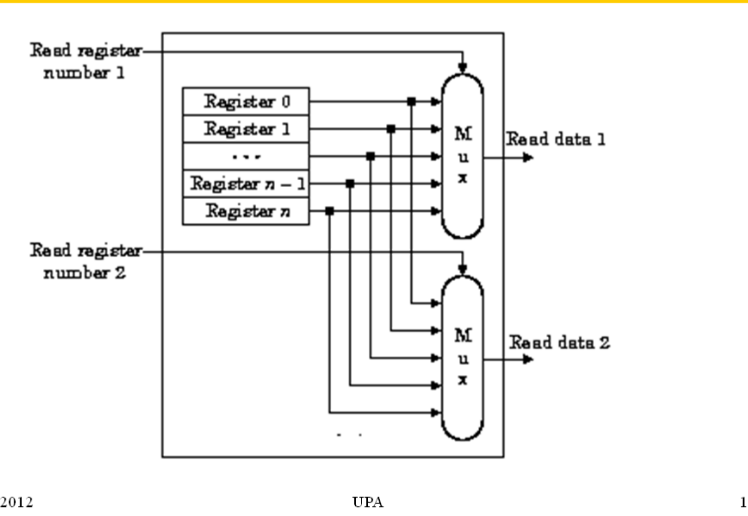
Registrový soubor – schéma, čtecí porty a výstup dat pracují paralelně

Registrový soubor

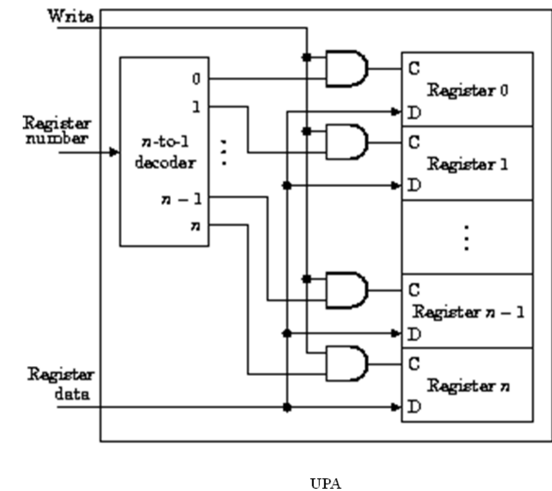


Čtecí / zápisové porty registrového souboru - schéma

Čtecí porty registrového souboru



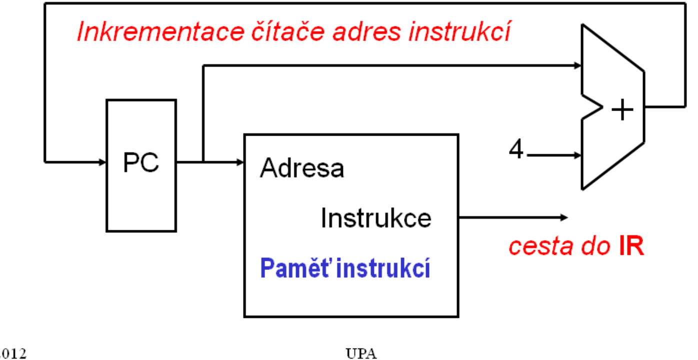
Zápisové porty registrového souboru



Základní datové cesty – paměť, ALU, PC

Základní datové cesty

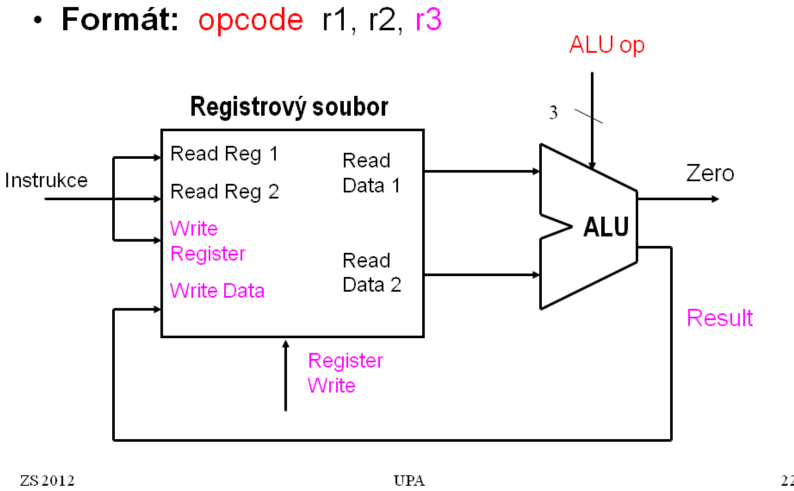
- **Paměť** obsahuje instrukce
- **PC** adresa aktuální instrukce
- **ALU** provádí aktuální instrukci



Datové cesty formátu R – schéma

Datové cesty pro R-formát

- **Formát:** opcode r1, r2, r3



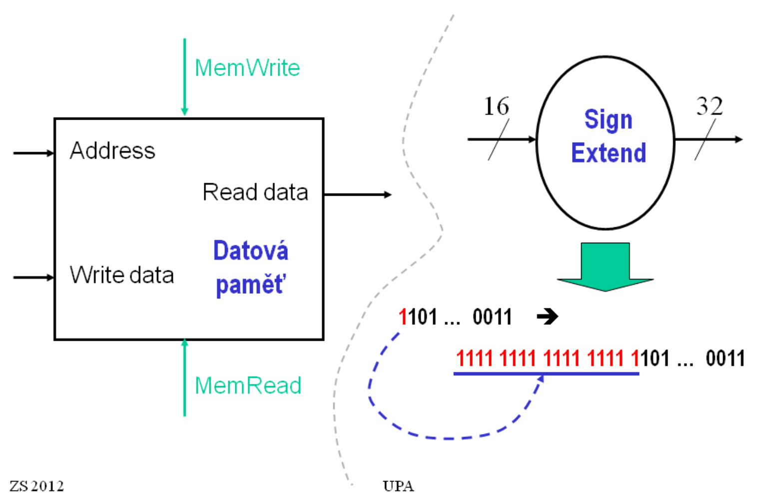
Elementární kroky při Load/Store

Elementární kroky při Load/Store

- **lw $t1, offset($t2)**
  - reference do paměti, báze v $t2 s offsetem
  - lw: čtení paměti, zápis do registru $t1
  - sw: čtení z registru $t, zápis do paměti
- Výpočet adresy – podle ISA:
  - 16-bitový offset se znaménkem se převede na 32-bitovou hodnotu se znaménkem
- *Hardware*: Datová paměť pro read/write

Prvky datové cesty pro Load/Store, formát I

Prvky datové cesty pro Load/Store

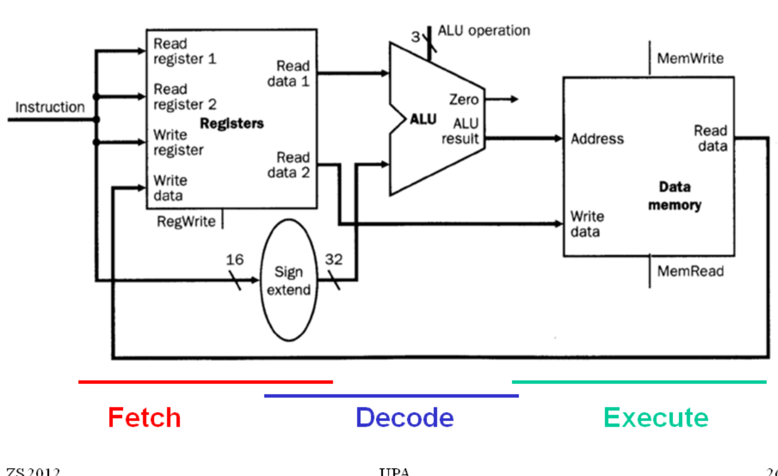


Provedení operace Load/Store, schéma datové cesty load/store

Provedení operace Load/Store

1. **Přístup do registru** — Registrový soubor
   -- ***Čtení*** instrukce/dat/adresy
2. **Výpočet adresy do paměti** — ALU
   -- ***Dekódování*** adresy
3. **Čtení/zápis z/do paměti** — Datová paměť
4. **Zápis do registrového souboru** — Registrový soubor
   -- ***Provedení*** instrukce Load/Store

Datové cesty pro Load/Store



Fetch Decode Execute

Datové cesty instrukce větvení Branch, formát I, pokud nepodmíněný skok pak formát J

Datové cesty - instrukce větvení (Branch)

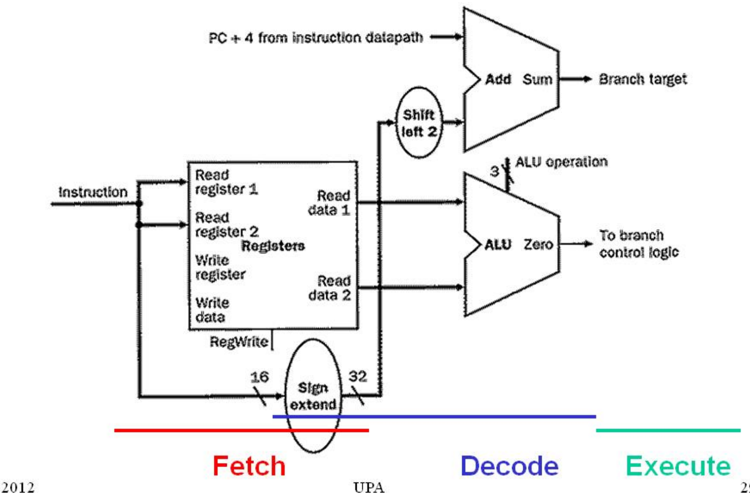
- **beq $t1, $t2, offset**
  - Dva registry ($t1, $t2) – test na rovnost
  - 16-bitový offset výpočet cílové adresy skoku
- Adresa cíle skoku – podle ISA:
  - Přičti znaménkem rozšířený offset k PC
  - Bázová adresa je instrukce za skokem (PC+4)
  - Posuň offset vlevo o 2 bity => word offset
- Skok na cílovou adresu
  - Nahraď dolních 26 bitů PC dolními 26 bity instrukce posunuté o 2 bity

Provedení větvení, schéma větvení

Provedení větvení (Branch)

1. **Přístup do registru** — Registrový soubor
   -- ***Čtení*** instrukce/dat
2. **Vyhodnocení podmínky skoku** — ALU #1
3. **Výpočet cílové adresy** — ALU #2
   -- Výpočet skoku – podobné ***dekódování***
4. **Skok na cílovou adresu** — Řadič
   -- ***Provedení*** instrukce větvení

Datové cesty pro větvení (Branch)



Opožděné podmíněné skoky MIPS

Opožděné podmíněné skoky (MIPS)

- **MIPS ISA**: Podmíněné skoky jsou vždy *opožděné*
  - Instrukce I_b ležící za skokem se vždy provede
  - *condition = false* => normální skok
  - *condition = true* => I_b se provede

Je to špatně?

1. Zlepší se efektivita
2. Skok se neprovádí (nesplněná podmínka) může být *častější případ*

Datové cesty, složení, MIPS ISA, stavební prvky datových cest

Závěr

- Datové cesty zajišťují přesuny dat v CPU
- Datové cesty propojují:
  - *ALU*: Provádí elementární operace
  - *Registrový soubor*: Čtení a zápis dat
- Registrový soubor je implementován pomocí D klopných obvodů, multiplexerů a dekodérů
  - Taktované (synchronní) logické obvody
  - Řídící signály určují zápis do registrů
  - Datové přenosy z registrů nejsou chráněné

Závěr

- **MIPS ISA**: Tři instrukční formáty (R, I a J)
- Datové cesty navrženy pro každý instrukční formát
- **Stavební prvky datových cest:**

  -- *R-formát:* ALU, registrový soubor

  -- *I-formát:* Sign Extender, datová paměť

  -- *J-formát:* ALU #2 pro výpočet cílové adresy

Trik: Opožděné skoky zlepšují efektivitu

MIPS procesor s jednoduchým cyklem, interface, ISA

MIPS procesor s jednoduchým cyklem

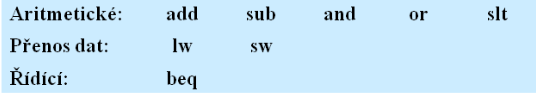
- Instruction Set Architecture je *interface* definující hardwarové operace, které má software k dispozici.

- Libovolný instrukční soubor může být implementován různými způsoby. V příštích hodinách porovnáme dvě důležité implementace.
  - V základní implementaci s jednoduchým cyklem trvají všechny operace stejnou dobu – jeden cykl.
  - V implementaci s režimem pipeline se v procesoru při provádění instrukce překrývají, což přináší potenciálně vyšší výkon.

Implementace s jednoduchým cyklem

Implementace s jednoduchým cyklem

- Popíšeme implementaci instrukčního souboru procesoru na bázi MIPS s následujícími operacemi.



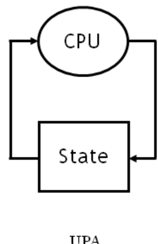
Aritmetické:	add	sub	and	or	slt
Přenos dat:	lw	sw			
Řídící:	beq				

  - Použijeme architekturu MIPS, protože ji lze podstatně snáze implementovat než architekturu x86.
- Začneme s implementací instrukčního souboru s jednoduchým cyklem.
  - Doba provádění všech instrukcí bude stejná. Tím je určena doba cyklu pro rovnici výkonu.
  - Bude vysvětlena funkce datových cest a řídící jednotky.

Počítače jako automaty – velký automat (state machine)

Počítače jsou automaty

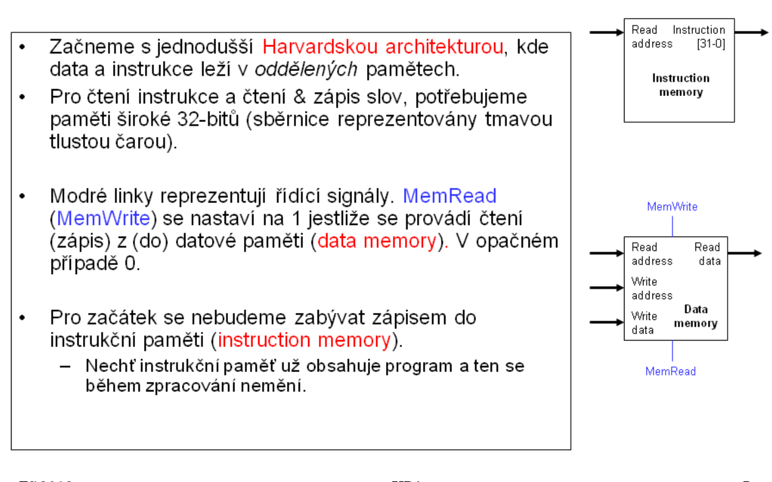
- Počítač je vlastně „velký automat“ (state machine).
  - Registry, paměť, pevné disky a jiné paměťové prvky formují stav.
  - Procesor provádí „aktualizaci“ stavu podle instrukcí programu.
- Modelování chování počítačů se opírá o teorii automatů (konečných automatů).



ZS 2012 UPA 6

Paměti- harvardská architektura

Paměti

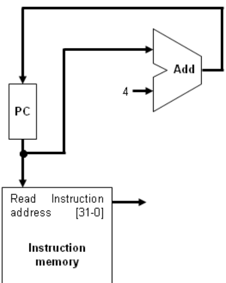


ZS 2012 UPA 7

Čtení instrukce, program counter, PC

Čtení instrukce

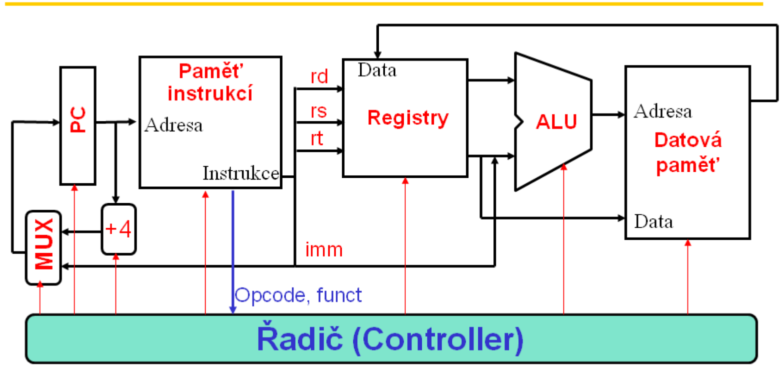
- CPU pracuje v nekonečné smyčce – čte instrukce z paměti a provádí je.
- Program counter (register PC) obsahuje adresu aktuální instrukce.
- Instrukce MIPS jsou dlouhé 4 byte. Proto je PC inkrementován o 4, aby ukazoval na následující instrukci.



ZS 2012 UPA 8

Přehled implementace, řadič, datové cesty

Přehled implementace



- Datové cesty umožňují přesuny obsahu registrů při provádění instrukcí
- Řízení zajišťuje provedení správných přesunů

Dekódování instrukcí typu R, příklad instrukce a dekódování

Dekódování instrukcí (typu-R)

- Aritmetické instrukce typu Register-to-Register používají formát typu-R.
  - op je operační kód a func specifikuje blíže prováděnou aritmetickou operaci
  - rs, rt a rd jsou zdrojové a cílový registr.

op	rs	rt	rd	shamt	func
6 bitů	5 bitů	5 bitů	5 bitů	5 bitů	6 bitů

- Příklad instrukce a jejího dekódování:

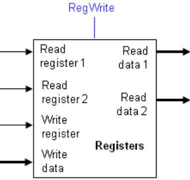
add $s4, $t1, $t2

000000	01001	01010	10100	00000	1000000

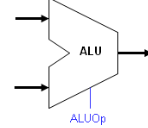
Registry a ALU – instrukce typu R, RegWrite, ALUOp tabulka funkcí

Registry a ALU

- Instrukce typu R pracují s registry a s ALU.
- Registrový soubor obsahuje 32 32-bitových hodnot.
  - Každý specifikátor registru je dlouhý 5 bitů.
  - V jednom okamžiku lze číst dva registry.
  - RegWrite je 1, má-li být zapisováno do registru.
- Níže je jednoduchá ALU s pěti operacemi, výměr se provádí 3-bitovým polem (řídicí signály) ALUOp.



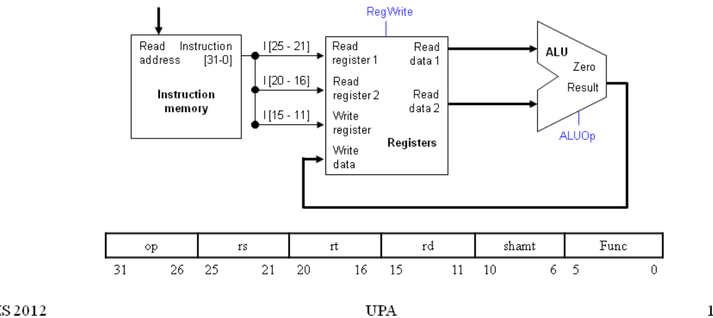
ALUOp	Funkce
000	and
001	or
010	add
110	subtract
111	slt



Provedení instrukce typu R, R formát - provedení

Provedení instrukce (typu-R)

1. Čtení instrukce z instrukční paměti.
2. Zdrojové registry, určené v instrukci poli rs a rt, se čtou z registrového souboru.
3. ALU provede požadovanou operaci.
4. Výsledek se uloží do cílového registru, určeného polem rd v instrukci.



ZS 2012 UPA 12

R-formát - provedení

Instrukce: `add $t0, $t1, $t2`

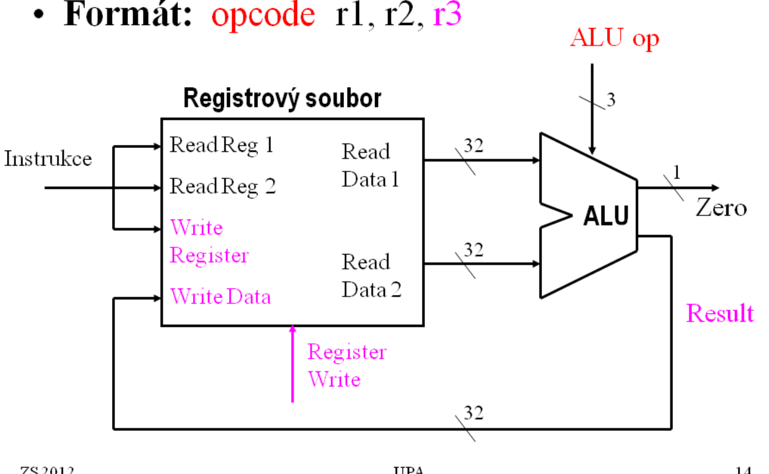
1. Načtení instrukce a inkrement PC
2. Vstup $t0 a $t1 z registrové sady
3. ALU zpracovává $t0 a $t1 podle pole *funct* instrukce MIPS (bity 5-0)
4. Výsledek z ALU je zapsán do registrové sady, bity 15-11 instrukce vybírají cílový registr (např, $t0).

ZS 2012 UPA 13

R formát komponenty . schéma

Komponenty: R-formát

- **Formát:** opcode r1, r2, r3



ZS 2012 UPA 14

Dekódování instrukce typu I

Dekódování instrukcí typu - I

- Instrukce lw, sw a beq jsou kódovány ve formátu typu-I.
  - rt je *cílový registr* pro lw, ale *zdrojový registr* pro beq a sw.
  - adresa je 16-bitová konstanta se znaménkem.

op	rs	rt	adresa
6 bitů	5 bitů	5 bitů	16 bitů

- Dva příklady instrukcí:

Instrukce	op	rs	rt	adresa
lw $t0, -4($sp)	100011	11101	01000	1111 1111 1111 1100
sw $a0, 16($sp)	101011	11101	00100	0000 0000 0001 0000

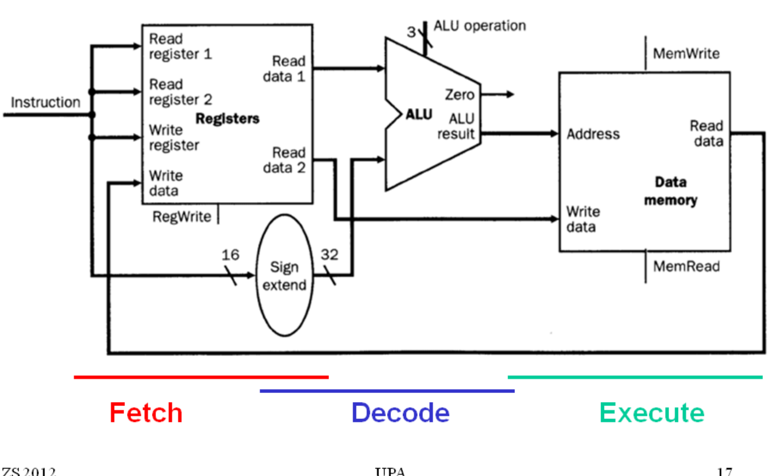
Load/store provedení, Komponenty Load/store

Load/Store - provedení

Instrukce: `lw $t1, offset($t2)`

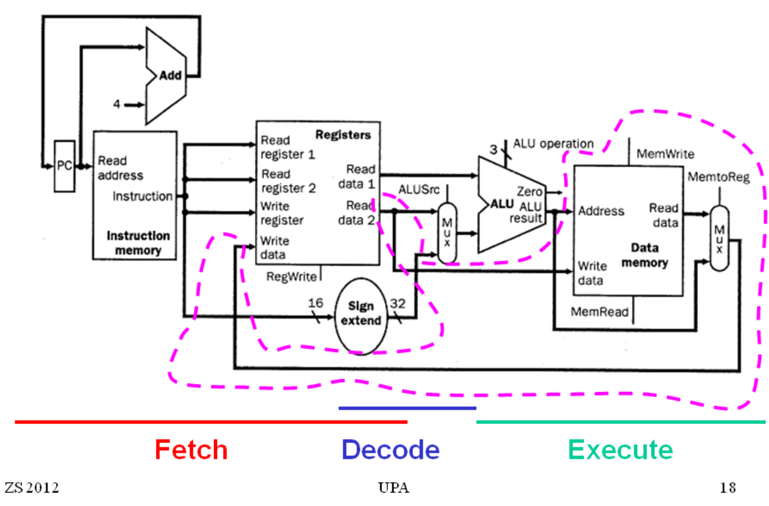
1. Načtení instrukce a inkrement PC
2. Čtení obsahu registru (např., bázová adresa v $t2) z registrového souboru
3. ALU přičte hodnotu z $t2 ke (znaménkem rozšířeným) dolním 16 bitům instrukce (`offset`)
4. Výsledek z ALU = adresa do datové paměti
5. Čtení dat z paměti, zápis do registrové sady, podle čísla registru, uvedeného v $t1 (bity 20-16)

Komponenty: Load/Store



R formát + Load/store HW - schéma

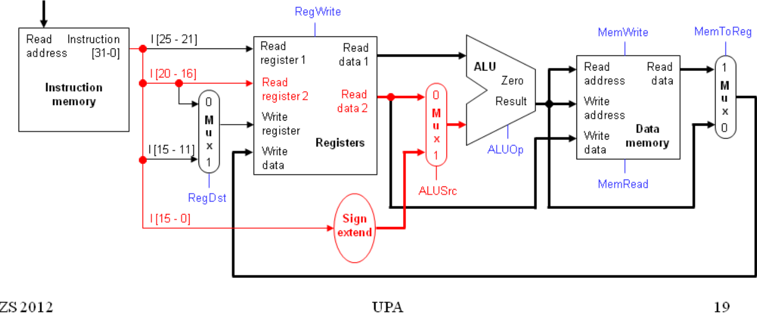
R-formát + Load/Store HW



Přístup do datové paměti – schéma + popis, ALUSrc

Přístup do datové paměti

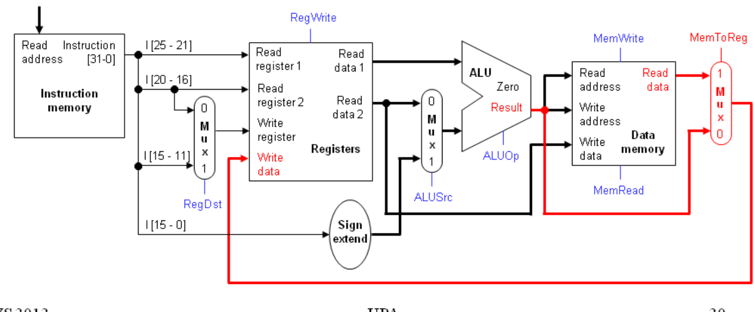
- Pro instrukci typu jako např. lw $t0, -4($sp) je bázový registr $sp přičten ke konstantě, rozšířené na 32 bitů (se znaménkem). Tak vznikne adresa do paměti.
- To znamená, že ALU musí mít na vstupu buď registrový operand pro aritmetickou instrukci a nebo *přímý operand* (*immediate*) pro lw a sw.
- Doplníme multiplexer, řízený polem ALUSrc, který vybere registrový operand (0) nebo konstantu (1).



Přesun z paměti do registrů, MemToReg

Přesun z paměti do registru

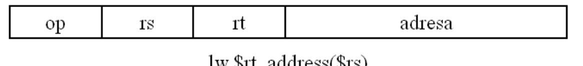
- Vstup do registrového souboru "Write data" také musí musí nabýt hodnotu výstupu ALU pro instrukce typu-R *nebo* hodnotu dat na výstupu paměti pro lw.
- Doplníme multiplexer řízený signálem MemToReg, který vybere výsledek ALU (0) nebo výstup datové paměti (1).



Cílový registr – RegDst, schéma

Cílový registr

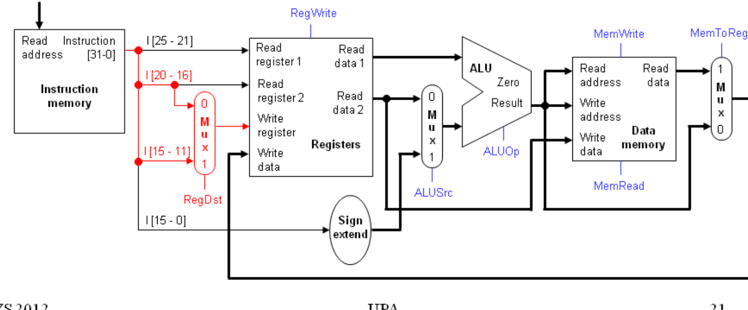
- Zbývá vyřešit výběr cílového registru, pro instrukci lw je to pole *rt* namísto rd.



op	rs	rt	adresa

lw $rt, address($rs)

- Doplníme další multiplexer řízený signálem RegDst pro výběr cílového registru podle instrukčního pole rt (0) nebo pole rd (1).



Větvení – podmíněné skoky, komponenty větvení, komponenty instrukce větvení

Větvení

- Pro instrukce větvení nepředstavuje konstanta adresu, ale *offset* registru PC od cílové adresy.

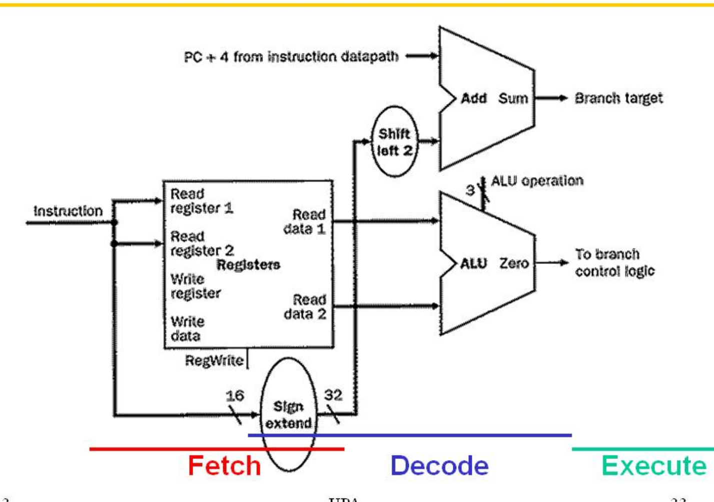
```
        beq   $at, $0, L
        add   $v1, $v0, $0
        add   $v1, $v1, $v1
        j     Somewhere
L:      add   $v1, $v0, $v0
```

- Cílová adresa L leží tři *instrukce* za beq, z toho vyplývá obsah adresního pole (offsetu) 0000 0000 0000 0011.

000100	00001	00000	0000 0000 0000 0011
op	rs	rt	adresa

- Délka instrukcí je čtyři byte. Proto je aktuální offset do paměti 12 bytů.

Komponenty: instrukce větvení



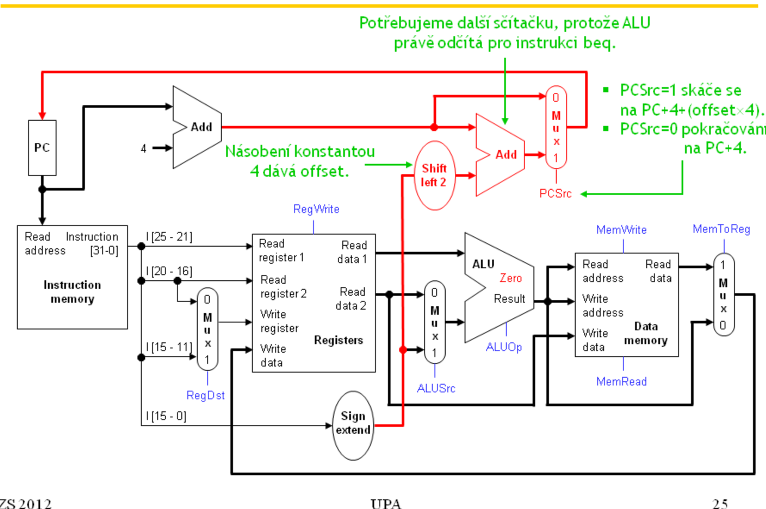
Provádění instrukce beq

Provádění instrukce beq

1. Čtení instrukce, např. beq $at, $0, offset z paměti.
2. Čtení zdrojových registrů, $at a $0, z registrového souboru.
3. Odečtením v ALU se obsahy porovnají.
4. Je-li výsledek rozdílu 0 (signál ALU **zero**), zdrojové operandy byly stejné a PC musí být naplněn cílovou adresou PC + 4 + (offset x 4).
5. V opačném případě se skok nekoná a PC je pouze inkrementován na PC + 4 a čte se následující instrukce.

Hardware pro provádění skoků, schéma včetně skoků

Hardware pro provádění skoků



Řídící kódy ALU – ALUOp input, ALU Control Input

Řídící kódy ALU

ALU má dva řídící kódy (celkem = 5 bitů):

1) *ALUop* – vybírá specifickou operaci ALU
2) *Control Input* – vybírá funkci ALU

ALUop Input	Operation
00	load/store
01	beq
10	determined by opcode

ALU Control Input	Function
000	and
001	or
010	add
110	sub
111	slt

Řídící bity ALU

Řídící bity ALU

ALU má následující řídící bity:

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

Instrukční formáty MIPS

Opakování: Instrukční formáty MIPS

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
I:	op	rs	rt	address / immediate		
J:	op	target address				

op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions ($+/-2^{15}$)
immediate: constants for immediate instructions

MIPS instrukční pole – pravidla

MIPS instrukční pole - pravidla

Všimněte si, že vždy platí:

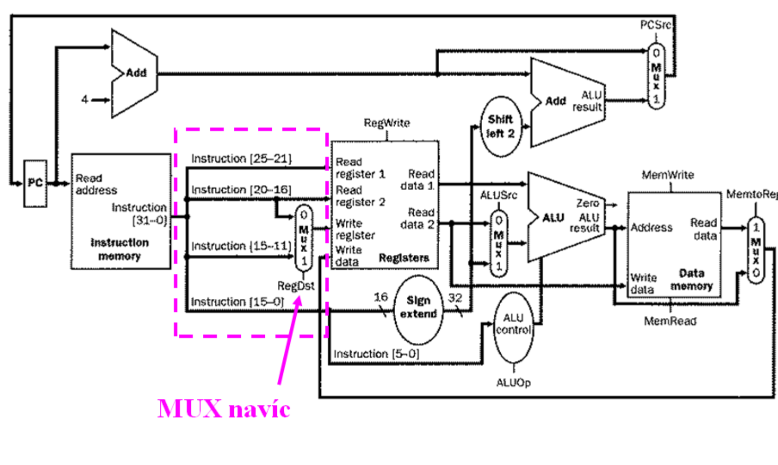
- **Bity 31-26**: *opcode* – vždy v tomto místě
- **Bity 25-21** a **20-16**: *specifikace vstupů* – vždy v tomto místě

Dále podle typu instrukce platí:

- **Bity 25-21**: *bázový registr* pro operace Load/Store – vždy v tomto místě
- **Bity 15-0**: *16-bitový offset* pro podmíněné skoky – vždy v tomto místě
- **Bity 15-11**: *registr určení* pro R-formát instrukcí – vždy v tomto místě
- **Bity 20-16**: *registr určení* pro operace Load/Store – vždy v tomto místě

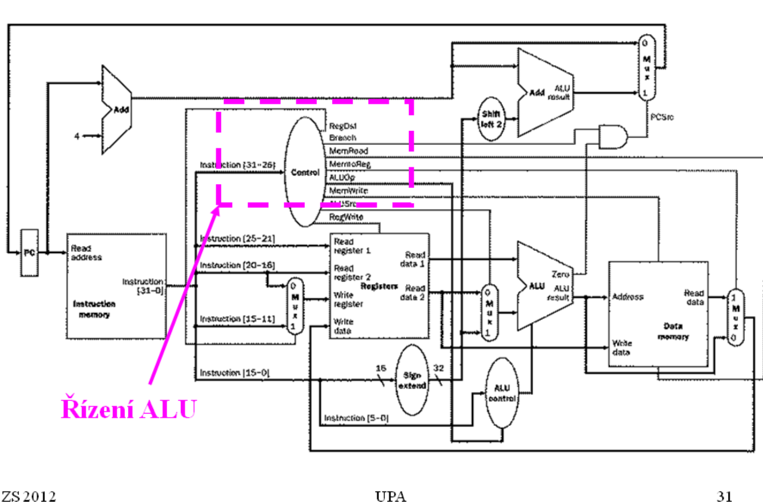
Datové cesty řízení zápisu do registrů

Datové cesty - řízení zápisu do registrů



Datové cesty řídící signály

Datové cesty – řídící signály



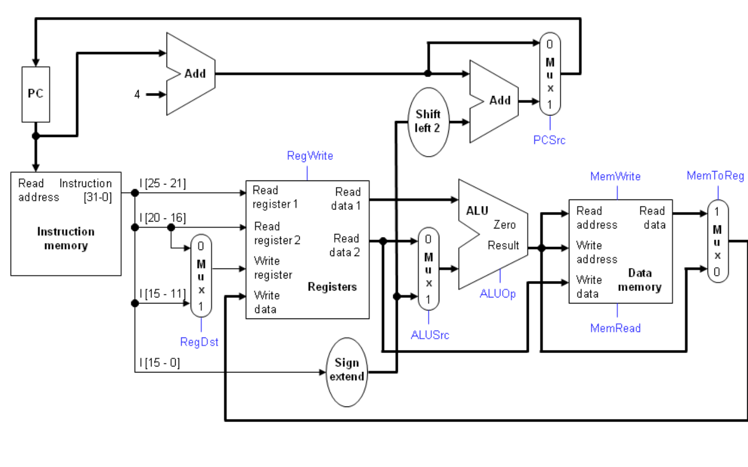
Řízení datových cest souhrn tabulka

Řízení datových cest (souhrn)

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Výsledné datové cesty

Výsledné datové cesty



Instrukce JUMP

Rozšíření: instrukce Jump

Instrukce: `j address`

1. Načtení instrukce a inkrement PC
2. Čtení *adresy* z pole instrukce immediate
3. Cílová adresa skoku (JTA) má tyto bity:
   - **Bity 31-28**: horní čtyři bity PC+4
   - **Bity 27-02**: pole *immediate* instrukce skoku
   - **Bity 01-00**: Zero (00_2)
4. Mux řízený signálem Jump vybere JTA nebo cílovou adresu skoku jako nový obsah PC

Instrukce JUMP, formát instrukce J

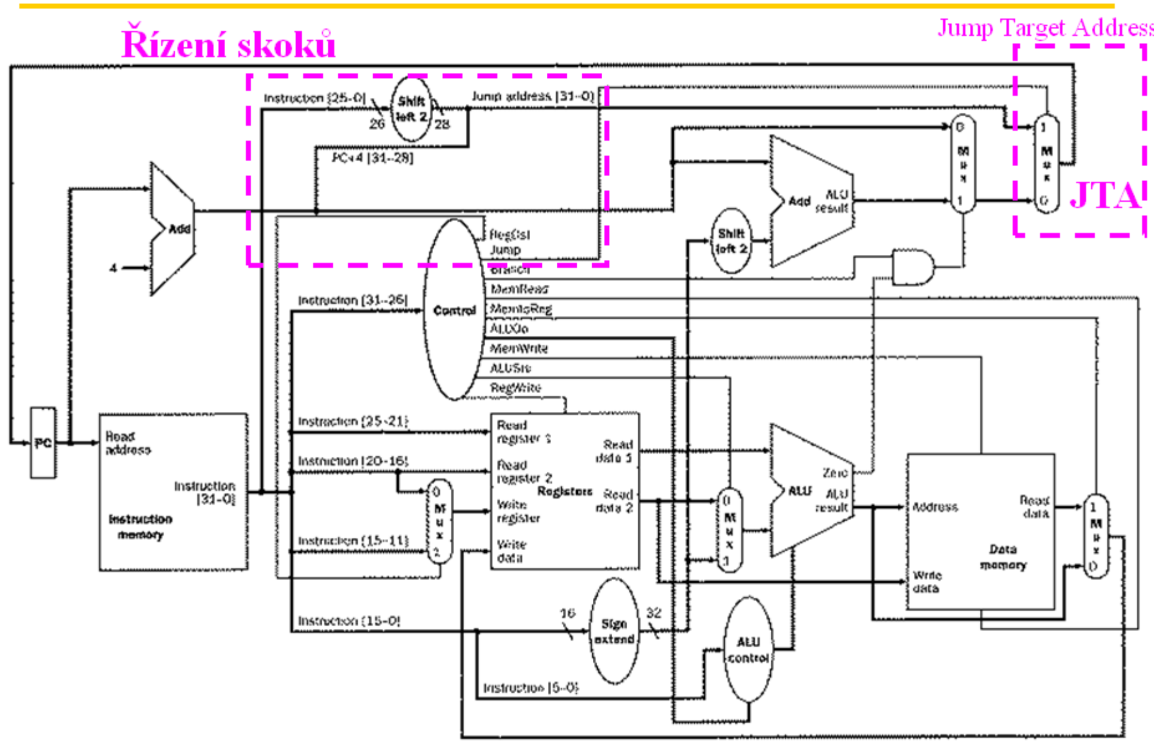
Rozšíření: instrukce Jump

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
I:	op	rs	rt	address / immediate		
J:	op	target address				

- **Bity 31-28**: Horní čtyři bity (PC + 4)
- **Bity 27-02**: pole immediate instrukce skoku
- **Bity 01-00**: Zero (002) - zarovnání slov

Instrukce JUMP schéma

Rozšíření: instrukce Jump



Řízení – řídící jednotka

Řízení

- Řídící jednotka zajišťuje generování řídících signálů tak, že všechny instrukce se správně provádějí.
  - Vstupem řídící jednotky je 32-bitové instrukční slovo.
  - Výstupem jsou řídící signály v datových cestách (označené modře).
- Většina signálů je odvozována pouze z operačního kódu instrukce, nikoliv z celého 32-bitového slova.

Shrnutí implementace s jednoduchým cyklem – datové cesty, řídící jednotka

Shrnutí implementace s jednoduchým cyklem

- Datové cesty obsahují všechny funkční jednotky a spoje, potřebné pro implementaci dané ISA (Instruction Set Architecture).
  - Pro implementaci s jednoduchým cyklem jsme použili dvě oddělené paměti, ALU, několik sčítaček a řadu multiplexerů.
  - MIPS je 32-bitový procesor, a proto má většina sběrnic šířku 32-bitů.
- Řídící jednotka určuje činnost podle toho, jaká instrukce se právě vykonává.
  - Náš procesor má 10 řídicích signálů, které ovládají datové cesty.
  - Řídicí signály mohou být generovány kombinačními obvody, odvozenými od 32-bitového instrukčního slova.
- Dále se budeme zabývat omezením výkonu, které uvedená implementace s jednoduchým cyklem přináší.

Problémy s jednocyklovou jednotkou

Problémy

- Lze postavit jednotku operující v jednom cyklu?
  - Ano – "Návrh lze provést"
  - Všechny instrukce prováděny s CPI = 1 ☺
  - Doba cyklu je diktována dobou ustálení všech obvodů ☹
  - Všechny operace trvají jako nejdelší operace, obvykle (load) ☹
  - Vyšší efektivita *software u architektury MIPS*

☹ **Problémy s jednocyklovou jednotkou**
  - Zpoždění signálu odpovídá průchodu 1-5 prvky
  - Není rozloženo do fází: Dokončení během 1 hodin taktu
  - Maximální zpoždění = instrukce Load (5 komponent)
  - ***Zvětšuje dobu cyklu hodin***
  - ***Pokles výkonu*** $t_{cpu} = IC * CPI * t_{cyc}$

Multicyklové zpracování, multicyklové jednotky

Přehled – multicyklové zpracování

- **Každá instrukce je zpracována ve více stupních**
- **Každý stupeň vykoná operaci během jednoho cyklu**

1. Načtení instrukce
2. Dekódování instrukce / Načtení dat

Používají všechny instrukce

3. Operace ALU / provedení instrukcí R-formátu
4. Dokončení provedení R-formátu
5. Provedení paměťového cyklu

☺ **Každý stupeň může opět použít hardware předchozího stupně**

☺ *Efektivnější využití hardware a času*

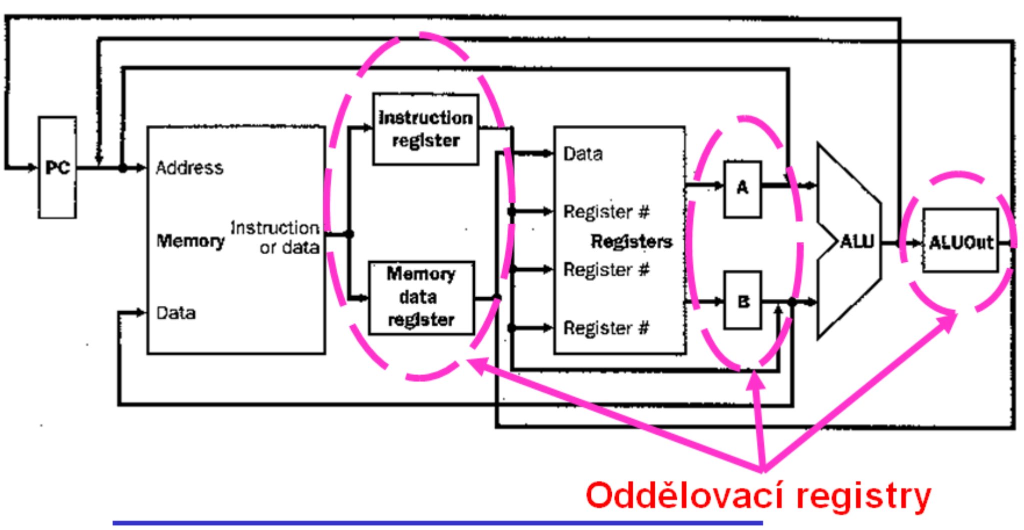
>> **Nový hardware** vyžaduje registrový výstup

>> **Nové multiplexery (MUX)** pro opětné využití hardware

>> **Rozšíření řídící jednotky (řadiče)** pro nový hardware

Princip multicyklové datové cesty

Princip: Multicyklové datové cesty



Načtení instrukce Dekódování/Načtení dat Provedení

Princip multicyklové datové cesty – jak realizovat multicyklové jednotky- 4 rozdíly

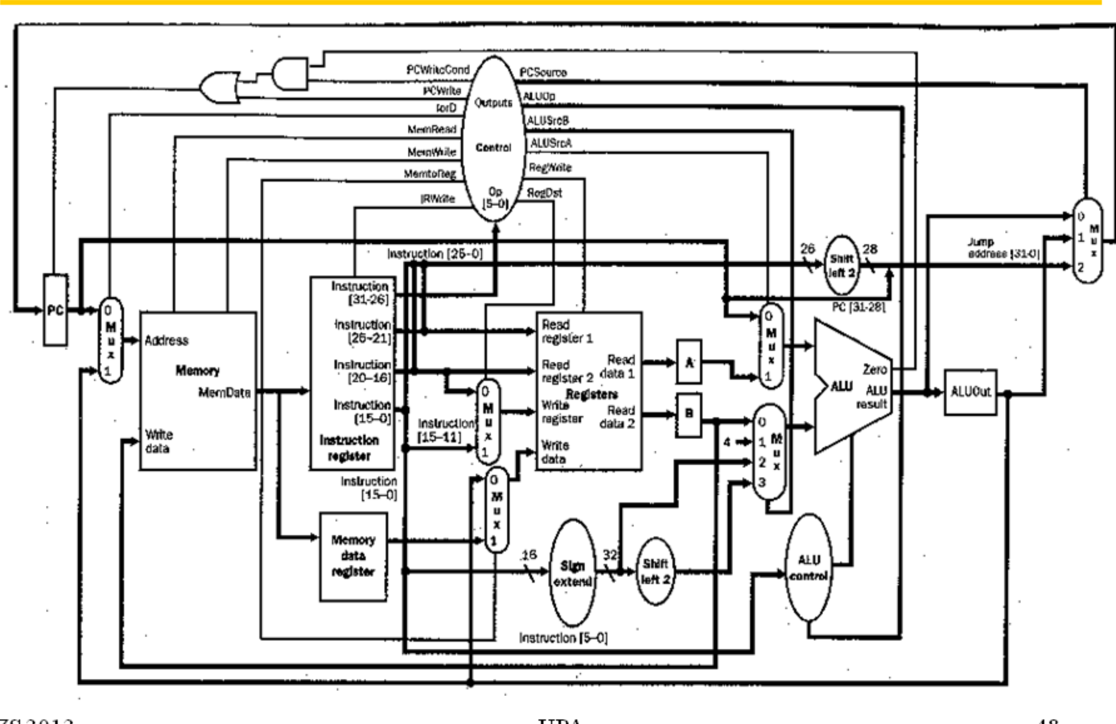
Princip: Multicyklové datové cesty

Jak realizovat multicyklové jednotky (DP)???

1. **Nahradíme** 3 ALU ve struktuře jedinou ALU
2. **Přidáme** jeden multiplexer pro výběr vstupu ALU
3. **Přidáme** jednu řídící linku pro vstupní multiplexer ALU
   - Nové vstupy: Konstanta = 4 [PC + 4]
   - Sign-ext., posunutý offset [výpočet BTA]
4. **Přidáme** *temporary (buffer) registry*: (oddělovací registry)
   - MDR: Datový registr paměti
   - IR: Instrukční registr
   - A, B: Registr operandů ALU
   - ALUout: Výstupní registr ALU

Multicyklová jednotka kompletní

Multicyklová jednotka (komplet)



ZS 2012 UPA 48

Multicyklová jednotka 1-bitové řídící signály, 2-bitové řídící signály

Multicyklová jednotka: 1-bitové řídící signály

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None	The output of the memory is written into the IR.
PCWrite	None	The PC is written; the source is controlled by PCSource.
PCWriteCond	None	The PC is written if the Zero output from the ALU is also active.

ZS 2012 UPA 49

Multicyklová jednotka: 2-bitové řídící signály

Signal name	Value	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25–0] shifted left 2 bits and concatenated with PC + 4[31–28]) is sent to the PC for writing.

Plánování činnosti multicyklové jednotky

Plánování činnosti multicyklové jednotky

Krok 1: Dekompozice provedení MC/DP na cykly

Krok 2: Ověřit, na které instrukce lze které cykly aplikovat

Jednocyklové kroky (akce)	R-fmt	lw	sw	beq	j
1. Načtení instrukce	■	■	■	■	■
2. Dekódování instrukce / čtení dat	■	■	■	■	■
3. Operace ALU/ provedení R-formátu	■	■	■	■	■
4. Dokončení R-formátu	■	■	■		
5. Dokončení přístupu do paměti		■			

Multicyklová jednotka R formát, ALUsrcA, ALUsrcB, ALUop, RegWrite, RegDst, CPI pro R formát

Multicyklová jednotka: R-formát

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole opcode, rd, rs, rt, funct

Načtení dat: Výběr registrů podle rs, rt

Data načtena do pomocných registrů A, B (vstup ALU)

Krok 3: Operace ALU (ALUsrcA, ALUsrcB, ALUop)

Výstup z ALU je zapsán do registru ALUout

Krok 4: Obsah registru ALUout jde na zápisový port registrové sady
V rd je zapsáno číslo registru

Aktivovány signály: RegWrite, RegDst

CPI pro R-formát = 4 cykly

Multicyklová jednotka store word, ALUsrcB, ALUop, ALUout, MemWrite, CPI pro store

Multicyklová jednotka: Store Word (sw)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole opcode, rs, rt, offset
Načtení dat: rt adresuje registrovou sadu => Bázová adresa
Data načtena do buffer registru A (báze)
SignExt, Shift pole offsetu do buffer registru B

Krok 3: Operace ALU (ALUsrcB, ALUop) => Báze + Offset
výstup ALU jde do registru ALUout

Krok 4: Obsah registru ALUout je použit jako adresa do paměti
Aktivován signál: MemWrite [ALUout => reg. sadu]

CPI pro Store = 4 cykly

Multicyklová jednotka load word, ALUsrcB, ALUop, ALUout, MemRead, CPI pro load

Multicyklová jednotka: Load Word (lw)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4
Krok 2: Dekódování instrukce: pole opcode, rs, rt, offset
Načtení dat: rt – pointer do registrové sady => Bázová adresa
Data načtena do buffer registru A (báze)
SignExt, Shift pole offsetu do buffer registru B
Krok 3: Operace ALU (ALUsrcB, ALUop) => Báze + Offset
výstup ALU jde do registru ALUout
Krok 4: Obsah registru ALUout je použit jako adresa do paměti
Aktivován signál: MemRead
Krok 5: Data z paměti jdou na zápisový port registrové sady,
adresa určena obsahem rd - proveden zápis

CPI pro Load = 5 cyklů

Multicyklová jednotka pro podmíněný skok, ALUsrcA, ALUsrcB, ALUop, Zero, CPI pro podmíněný skok

Multicyklová jednotka: Podmíněný skok

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole opcode, rs, rt, offset

Načtení dat: rs a rt určují adresy do sady registrů

Výpočet BTA: SignExt, Shift pole offsetu do buffer registru B

ALU určí PC, offset => BTA

Krok 3: Operace ALU (ALUsrcA, ALUsrcB, ALUop) = komparace
podle výstupu z ALU (registr Zero) dojde k
výběru BTA nebo PC+4

CPI pro podmíněný skok = 3 cykly

Multicyklová jednotka pro skok, PCsource, PCwrite, CPI pro podmíněný skok

Multicyklová jednotka: Skok (Jump)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: Pole opcode, address

Výpočet JTA: Pole SignExt, Shift offset [Bity 27-0]

sestaveny z části PC [Bity 31-28] => JTA

Krok 3: Obsah PC nahrazen cílovou adresou skoku (JTA)

PCsource = 10, *Aktivován signál:* PCWrite

CPI pro Jump = 3 cykly

MIPS ISA- formáty, Cena multicyklové jednotky – Komplikovanější návrh řízení

Závěr

- **MIPS ISA**: Tři instrukční formáty (R, I, J)
- Jeden cykl/stupeň, různé stupně na formát
- **Jednocyklové kroky (akce)**

Jednocyklové kroky (akce)	R-fmt	lw	sw	beq	j
1. Načtení instrukce	■	■	■	■	■
2. Dekódování instrukce / čtení dat	■	■	■	■	■
3. Operace ALU/provedení R-formátu	■	■	■	■	■
4. Dokončení R-formátu	■	■	■		
5. Dokončení přístupu do paměti		■			

6. Cena: Komplikovanější návrh řízení

Multicyklová jednotka – 1 cykl/krok, konečný automat FSM, Final state machine, řadič

Opakování – Multicyklová jednotka

- Multicyklová jednotka – 1 cykl/krok :
  1. Načtení instrukce (Instruction Fetch)
  2. Dekódování instrukce / čtení dat
  3. Operace ALU / Provádění instrukcí formátu-R
  4. Dokončení instrukcí formátu-R
  5. Dokončení přístupu do paměti
- Konečný automat (Finite-State Machine)
  - současný stav, příští stav, přechodová funkce
- Řadič (konečný automat)
  - **Stav:** Generování řídících signálů
  - **Hrany:** Podmínky přechodu
  - Lze implementovat v hardware (ROM, PLA)

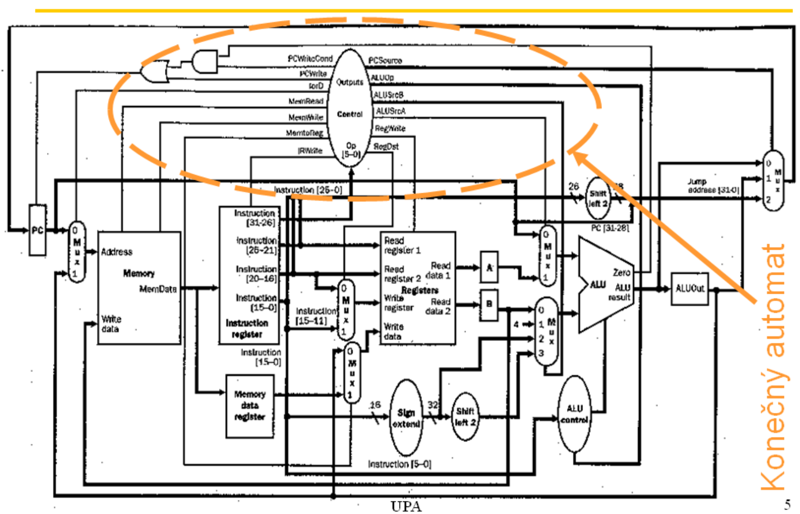
Problémy spojené s řízením, Mikroprogramování (Wilkesův automat), mikroinstrukce

Přehled

- Problémy spojené s řízením
  - Reálné procesory mají stovky stavů
  - Tisíce interakcí mezi stavy
  - Grafický návrh automatu je pro reálné architektury nemožný
- Řešení: Mikroprogramování (Wilkesův automat)
  - Návrh založený na „softwarových principech“ (firmware)
  - Řídící signály generované při vykonávání ***mikroinstrukce*** => určeny výstupním polem *mikroinstrukce*
  - Posloupnost mikroinstrukcí => ***mikroprogram***
- Mikroprogramování
  - Návrh mikroinstrukcí a jejich časování
  - Ošetření výjimek

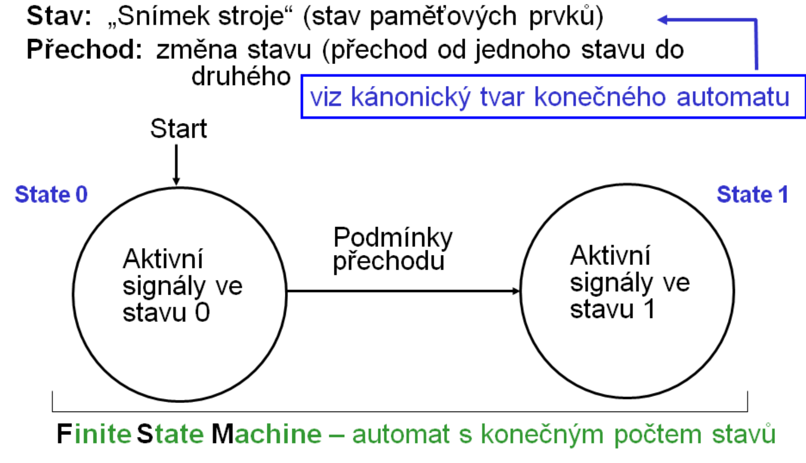
Řízení multicyklové jednotky – konečný automat

Řízení multicyklové jednotky



FSM – základy řízení pomocí FSM – stav, přechod, kánonický tvar

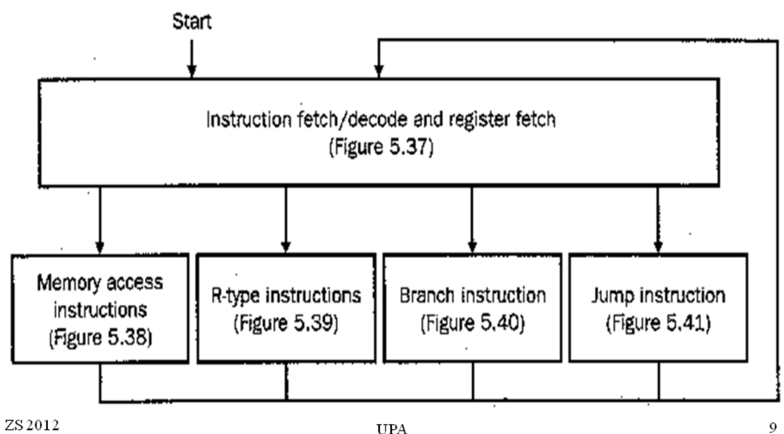
Základy řízení pomocí FSM



FSC pro multicyklové jednotky

FSC pro multicyklové jednotky

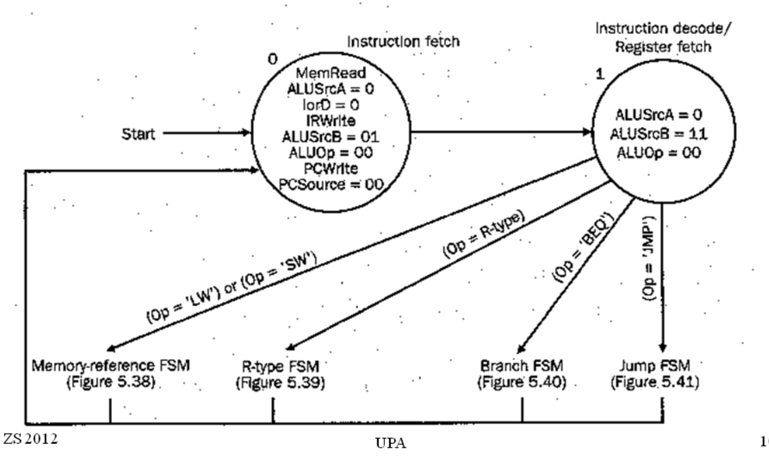
„Pohled shora“ – vhodné pro abstrakci



FSC pro natáčení a dekódování instrukce

FSC pro načtení & dekódování instrukce

Společné: Načtení instrukce/dat, dekódování



Multicyklový jednotka pro R formát

Multicyklová jednotka: R-formát

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole opcode, rd, rs, rt, funct

Načtení dat: Výběr registrů podle rs, rt

Data načtena do pomocných registrů A, B (vstup ALU)

Krok 3: Operace ALU (ALUsrcA, ALUsrcB, ALUop)

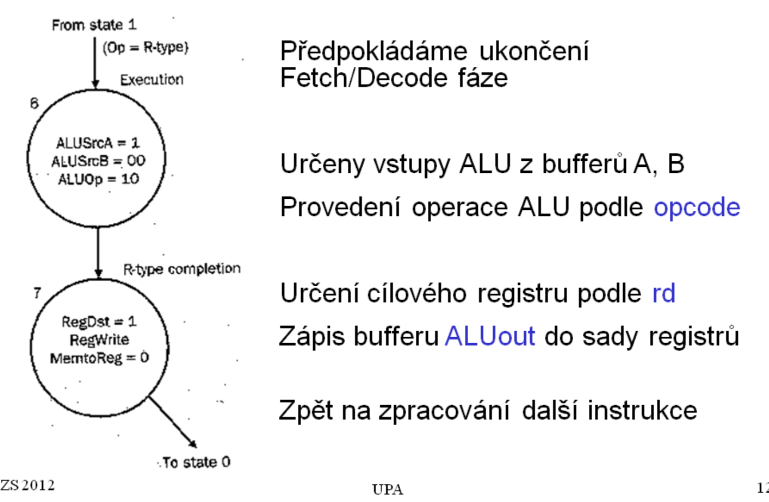
Výstup z ALU je zapsán do registru ALUout

Krok 4: Obsah registru ALUout jde na zápisový port registrové sady
V rd je zapsáno číslo registru

Aktivovány signály: RegWrite, RegDst

CPI pro R-formát = 4 cykly

FSC pro R formát instrukcí, ALUout



Multicyklový jednotka pro store word

Multicyklová jednotka: Store Word (sw)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole opcode, rs, rt, offset

Načtení dat: rt adresuje registrovou sadu => Bázová adresa

Data načtena do buffer registru A (báze)

SignExt, posun pole offset do buffer registru B

Krok 3: Operace ALU (ALUsrcB, ALUop) => Báze + Offset

výstup ALU jde do registru ALUout

Krok 4: Obsah registru ALUout je použit jako adresa do paměti

Aktivován signál: MemWrite [ALUout => reg. sadu]

CPI pro Store = 4 cykly

Multicyklový jednotka pro load word

Multicyklová jednotka: Load Word (lw)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole opcode, rd, rt, offset

Načtení dat: rt – pointer do registrové sady => Bázová adresa

Data načtena do buffer registru A (báze)

SignExt, posun pole offset do buffer registru B

Krok 3: Operace ALU (ALUsrcB, ALUop) => Báze + Offset

výstup ALU jde do registru ALUout

Krok 4: Obsah registru ALUout je použit jako adresa do paměti

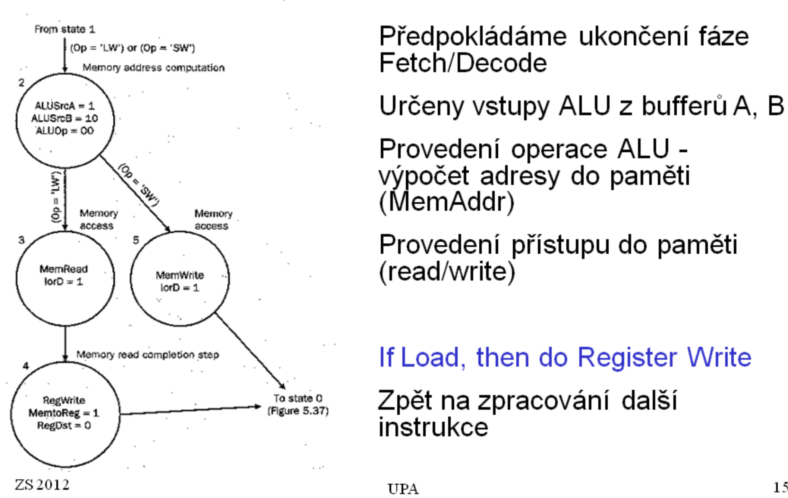
Aktivován signál: MemRead

Krok 5: Data z paměti jdou na zápisový port registrové sady, adresa určena obsahem rd - proveden zápis

CPI pro Load = 5 cyklů

FSC pro instrukce Load/store

FSC pro instrukce Load/Store



- Předpokládáme ukončení fáze Fetch/Decode
- Určeny vstupy ALU z bufferů A, B
- Provedení operace ALU - výpočet adresy do paměti (MemAddr)
- Provedení přístupu do paměti (read/write)
- If Load, then do Register Write
- Zpět na zpracování další instrukce

Multicyklová jednotka pro podmíněný skok

Multicyklová jednotka: Podmíněný skok

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole opcode, rs, rt, offset

Načtení dat: rs a rt určují adresy do sady registrů

Výpočet BTA: SignExt, posun pole offset do buffer registru B

ALU určí PC, offset => BTA

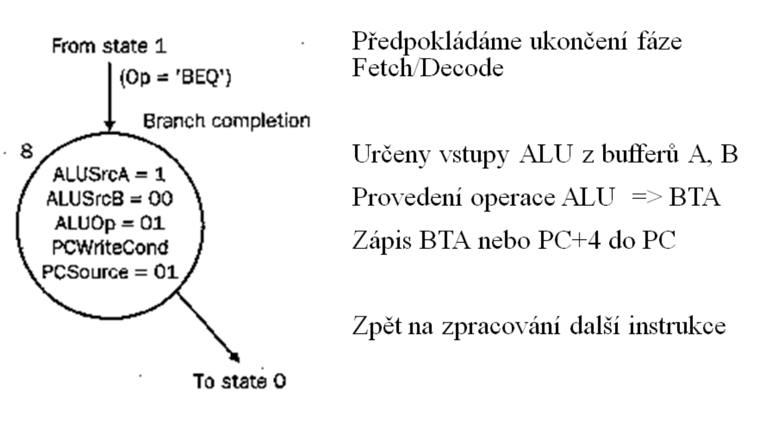
Krok 3: Operace ALU (ALUsrcA, ALUsrcB, ALUop) = komparace podle výstupu z ALU (výsledek Zero) dojde k výběru BTA nebo PC+4

CPI pro podmíněný skok = 3 cykly

Pozn.: BTA … adresa cíle instrukce větvení

FSC pro podmíněný skok

FSC pro podmíněný skok



- Předpokládáme ukončení fáze Fetch/Decode
- Určeny vstupy ALU z bufferů A, B
- Provedení operace ALU => BTA
- Zápis BTA nebo PC+4 do PC
- Zpět na zpracování další instrukce

Multicyklová jednotka pro skok

Multicyklová jednotka: skok (Jump)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: Pole opcode, address
Výpočet JTA: Pole SignExt, Shift offset [Bity 27-0]
sestaveny z části PC [Bity 31-28] => JTA

Krok 3: Obsah PC nahrazen cílovou adresou skoku (JTA)

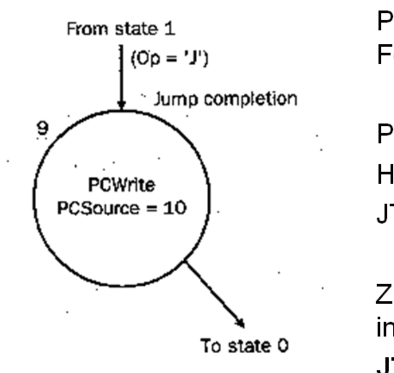
PCsource = 10, *Aktivován signál:* PCWrite

CPI pro Jump = 3 cykly

Pozn.: JTA … adresa cíle instrukce skoku

FSC pro instrukci skoku

FSC pro instrukci skoku



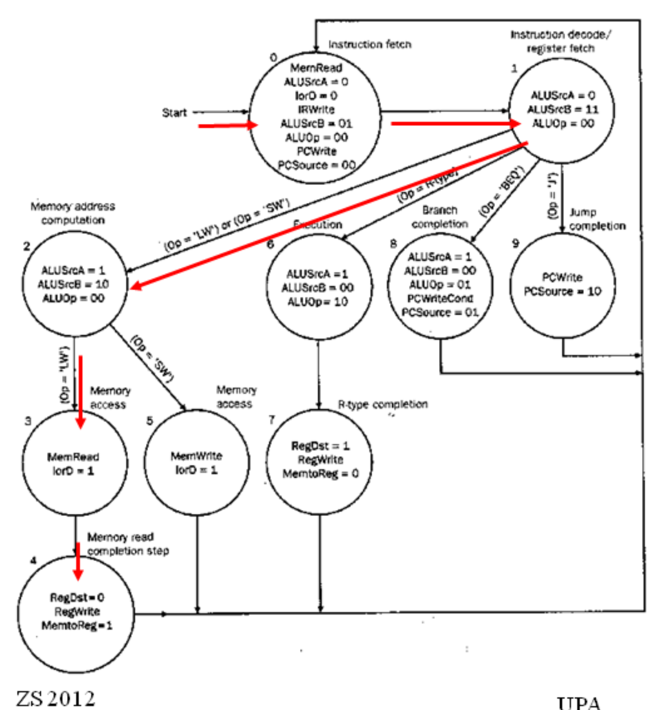
Předpokládáme ukončení fáze Fetch/Decode

PC bude přepsán
Hardware sestaví JTA
JTA se zapíše do PC

Zpět na zpracování další instrukce
JTA … Jump Target Address

FSC komplet

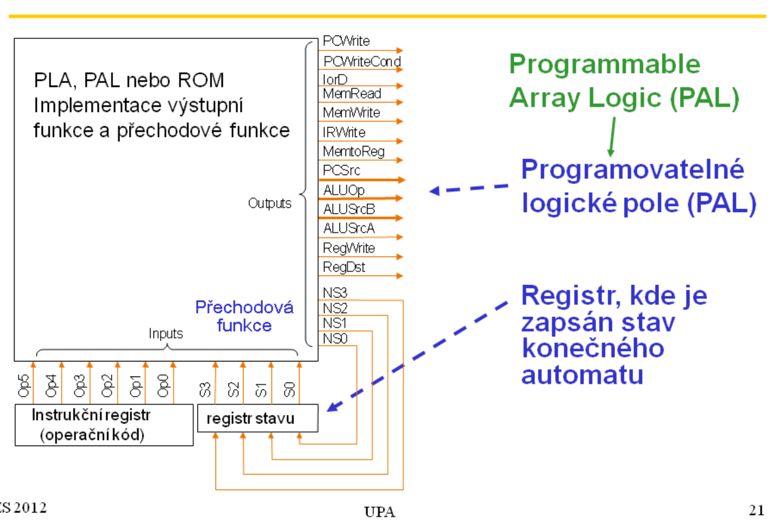
Kompletní FSC



- 10 stavů
- CPI = počet stavů nutných pro provedení dané instrukce
- R-formát = 4 stavy
- Store = 4 stavy
- **Load = 5 stavů [0,1,2,3,4]**
- Branch = 3 stavy
- Jump = 3 stavy

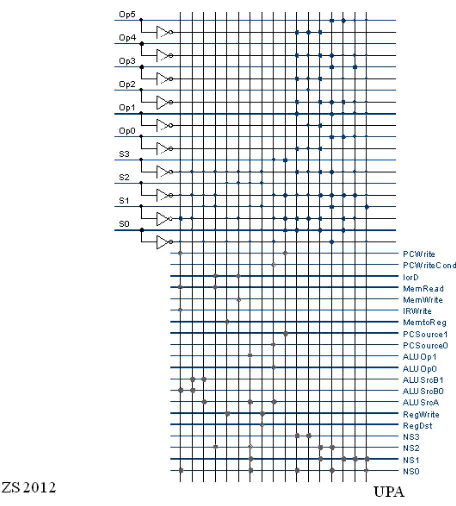
Hardware pro multicyklovou FSC, PAL

Hardware pro multicyklovou FSC



Implementace pomocí PLA

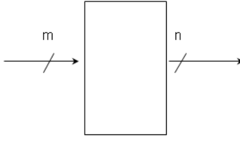
Implementace pomocí PLA



Implementace pomocí ROM – read only memory

Implementace pomocí ROM

- ROM = "Read Only Memory"
  - Hodnoty jsou zapsány a nelze je měnit
- Pamět ROM lze využít k implementaci pravdivostní tabulky
  - Má-li adresa m-bitů, můžeme adresovat 2^m položek v ROM.
  - Výstupem jsou data, na které ukazuje adresa.
    m je „výška" a n je „šířka" , odpovídající počtu výstupů.



Implementace pomocí ROM

- Kolik je v našem příkladu vstupů?
  6 bitů operační kód, 4 bity pro stav => 10 adresních linek
  (t.j. 2^{10} = 1024 různých adres)
- Kolik je výstupů?
  16 výstupních řídících signálů, 4 stavové bity => 20 výstupů
- Organizace ROM je 2^{10} x 20 = 20K bitů (trochu neobvyklá velikost a proto bereme nejbližší vyšší)
- Velmi nehospodárné vzhledem k velkému počtu situací, které nás nezajímají => výstupy závisejí pouze na stavech, nikoliv na op. kódu.

ROM vs PLA

ROM versus PLA

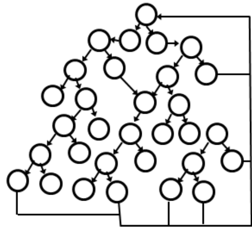
- Rozdělíme tabulku na dvě části
  - — 4 stavové bity určují celkem 16 výstupů, 2^4 x 16 bitů ROM
  - — 10 bitů určuje 4 bity příštího stavu, 2^{10} x 4 bitů ROM
  - — Celkem: 4.3K bitů ROM => poměrně značná úspora.
- PLA je mnohem menší
  - — může sdílet součinové termy
  - — obsahuje jen položky, které produkují aktivní výstup
  - — ošetřuje i případy (don't cares)
- Velikost je rovna (#vstupů x #produktových-termů) + (#výstupů x #produktových-termů)

  V tomto případě = (10x17)+(20x17) = 510 PLA buněk,
  buňka PLA je téměř tak velká jako buňka ROM (trochu větší).

Alternativa k FSM pro muticyklovou verzi

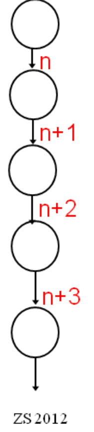
Alternativa k FSM pro multicyklovou verzi?

- MIPS-lite má (asi) 7 instrukcí, 10 FSM stavů
- Reálné stroje mají 100 a více instrukcí; reálné řadiče mají stovky až tisíce stavů!
- Problém: FSM bublinový diagram by byl příliš velký



Pozorování FSC na reálných strojích, mikroprogramové řízení, FSM řadič, strojový jazyk

Pozorování na reálných strojích



- Strojový jazyk: příští instrukce je určena implicitně.
  - PC registr určuje instrukci
  - Příští instrukce leží vždy na adrese PC+4 (vyjma skoku)
- FSM řadič: často produkuje jen jeden přechod od současného k příštímu stavu
- Vypůjčíme-li si tuto myšlenku z instrukční úrovně, bude každý řídící krok znamenat určitý druh “instrukce”?
- To vede na mikroprogramové řízení

Mikroprogramové řízení, mikroinstrukce mikroprogramu, čítače mikroinstrukcí mPC

Mikroprogramové řízení

- U mikroprogramového řízení vlastně stavy FSM přechází na mikroinstrukce mikroprogramu ("mikrokód")
  - Jeden stav FSM = jedna mikroinstrukce
  - Pomocí mikroinstrukcí jsou interpretovány instrukce
- Stavový registr FSM začne hrát roli čítače mikroinstrukcí (mPC)
  - Normální provádění: přičti 1 k mPC => adresa další mikroinstrukce
  - Větvení mikroprogramu: další logika určuje příští mikroinstrukci

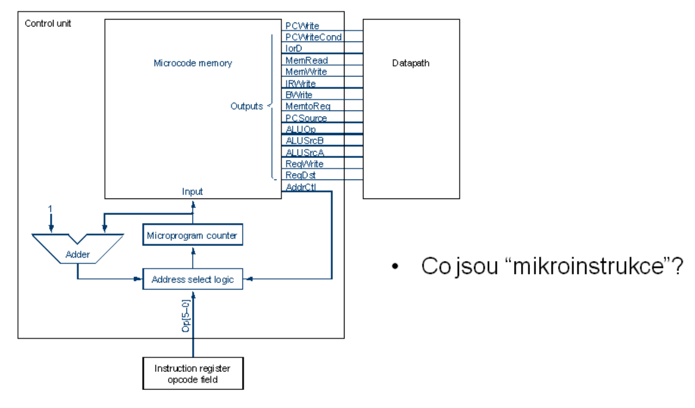
Mikroprogramování vs HW řízení

Mikroprogramování vs HW řízení

- Mikroprogramování přináší flexibilitu pro návrh a změny architektury. Řídící paměť (ROM) lze přeprogramovat nebo nahradit. Hardwarové řízení se obtížně navrhuje, je-li soubor instrukcí složitý. Po dokončení návrhu nejsou změny možné.
- Mikroprogramování je pomalejší, protože se do řídící paměti přistupuje v každém cyklu. Přístup do paměti je pomalý. Hardwarové řízení je rychlé, protože doba cyklu závisí pouze na zpoždění kombinačních obvodů řídící jednotky, které je mnohem kratší než doba přístupu do paměti.

Mikroprogramování - schéma

Mikroprogramování



- Co jsou "mikroinstrukce"?

Mikroprogramování - metodologie

Mikroprogramování

- Specifikační metodologie
  - Vhodné pro stovky op. kódů a režimů
  - Signaly se specifikují symbolicky pomocí mikroinstrukcí

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

Mají dvě implementace stejné architektury stejný mikrokód (firmware)?
Jakou úlohu plní mikroasembler?

Mikroinstrukční formát

Mikroinstrukční formát

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, lorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Maximální vs minimální kódování, žádné kódování, výrazné kódování

Maximální vs. minimální kódování

- Žádné kódování:
  - 1 bit pro každý signál mikrooperace
  - rychlé, vyžaduje více paměti (logika)
  - použito pro Vax 780 — celých 400K paměti!
- Výrazné kódování:
  - Signály mikrooperací se získávají dekódováním výstupního pole
  - Méně paměti, ale pomalejší, menší míra paralelizmu
- Historický kontext procesorů CISC:
  - Příliš mnoho logiky, která by se měla umístit na jeden chip spolu s ostatním
  - Použití ROM (nebo i RAM) pro uložení mikrokódu
  - Je jednoduché přidávat další instrukce

Mikrokód – kompromisy, výhody nevýhody

Mikrokód: Kompromisy

- Rozdíl mezi specifikací a implementací je někdy „neostrý“
- Výhody specifikace:
  - Snadný návrh a psaní
  - Současný návrh architektury a mikrokódu
- Výhody implementace (v externí ROM)
  - Snadná změna, protože se jedná o změnu obsahu paměti
  - Lze emulovat jiné architektury
  - Lze využívat interní registry
- Nevýhody implementace, dnes POMALÉ, protože:
  - Řadič bývá implementován na stejném čipu jako zbytek procesoru
  - ROM není dnes rychlejší než RAM
  - Obvykle není třeba se vracet a dělat změny

Mikroprogramování historie

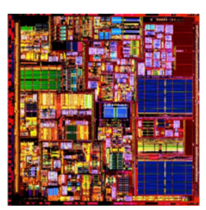
Historický přehled

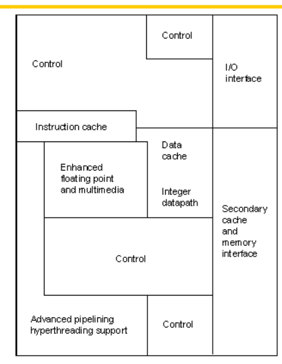
- V 60-tých a 70-tých letech bylo mikroprogramování velmi důležité při implementaci počítačů
- Vedlo na velmi propracované ISA, např. na VAX
- V 80-tých letech se staly populární procesory RISC, založené na hlubokém pipeliningu
- Pipelining je u mikroprogramování také možný!
- Implementace architektury procesoru IA-32 počínaje 486 používala:
  - “hardwarové řízení” jednoduchých instrukcí (malý počet cyklů, FSM implementován využitím PLA nebo „random logic“)
  - “mikroprogramové řízení” složitějších instrukcí (velký počet cyklů, centrální řídící paměť)
- Architektura IA-64 využívá styl RISC ISA a může být implementována bez velké centrální řídící paměti.

Random logic is a semiconductor circuit design technique that translates high-level logic descriptions directly into hardware features such as AND and OR gates. The name derives from the fact that few easily discernible patterns are evident in the arrangement of features on the chip and in the interconnects between them. In VLSI chips, random logic is often implemented with standard cells and gate arrays.

Pentium 4

Pentium 4





- Někde musí být ošetřeny i složité instrukce.
- Procesor provádí jednoduché mikroinstrukce, 70 bitů široké (hardwired).
- 120 řídicích linek pro integer jednotku (400 pro floating point).
- Jestliže některá instrukce vyžaduje pro implementaci více než 4 mikroinstrukce, řízení pak přichází z řídicí paměti ROM (8000 mikroinstrukcí). **Složité!**

MIPS ISA – konečný automat zjednodušení řízení

Přehled

- **MIPS ISA**: Tři instrukční formáty (R, I, J)
- Jeden cykl/stupeň, každý formát - různé stupně

• Jednocyklové kroky (akce)	R-fmt	lw	sw	beq	j
1. Načtení instrukce	■	■	■	■	■
2. Dekódování instrukce / čtení dat	■	■	■	■	■
3. Operace ALU/provedení R-formátu	■	■	■	■	■
4. Dokončení R-formátu	■	■	■		
5. Dokončení přístupu do paměti		■			

Konečný automat zjednodušuje návrh řízení

Formát mikroinstrukcí MIPS

Formát mikroinstrukcí MIPS

Mikroinstrukce: Abstrakce řízení datových cest

Pole	Specifikuje
• *ALU control*	operaci ALU v daném cyklu hodin
• *SRC1*	zdroj 1. operandu ALU
• *SRC2*	zdroj 2. operandu ALU
• *Register Control*	registr read/write, zapisovaná data
• *Memory*	read/write pro paměť, zapisovaná data cílový registr pro MemRead
• *PCWrite control*	zdroj pro PC (PC+4, BTA, JTA)
• *Sequencing*	výběr příští mikroinstrukce

Režimy výběru – sequencing, výběrové tabulky, mikroinstrukce inkrementace, větvení, řízený výběr, seq, fetch, dispatch i

Režimy výběru (sequencing)

Specifikuje výběr příští mikroinstrukce

- *Inkrementace* – Je-li aktuální adresa mikroinstrukce A, potom příští 32-bitová mikroinstrukce leží na A + 4 (hodnota = *Seq*)
- *Větvení* – Skok na mikroinstrukci, která načítá příští instrukci MIPS (hodnota = *Fetch*)
- *Řízený výběr* – Příští mikroinstrukce je vybrána podle vstupu řadiče (hodnota = *Dispatch i*)

Výběrové tabulky:

- Podobné *tabulce skoků* při programování MIPS
- Určuje mikroprogramovému řadiči, kde se startuje specifická rutina v mikroprogramu (např., přípravná fáze instrukce)

Načtení instrukce a dekódování . mikroinstrukce

Načtení instrukce a dekódování

Mikroinstrukce:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
Fetch	Add	PC	4	---	Read PC	ALU	Seq
---	Add	PC	Extshft	Read	---	---	Dispatch 1

Akce:

1. ALU provede PC+4, přesune ALUout do PC, přečte příští mikroinstrukci
2. ALU sečte PC a znaménkem rozšířený posunutý offset, čímž určí BTA, potom čte data z registrové sady do bufferů A a B

Přístup do paměti - mikroinstrukce

Přístup do paměti

Mikroinstrukce:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
Mem1	Add	A	Extend	---	---	---	Dispatch 2
LW2	---	---	---	---	Read ALU	---	Seq
---	---	---	---	Write MDR	---	---	Fetch

Akce:

1. ALU sečte bázi v A + rozšířený offset => adresa do paměti
2. Při čtení se paměť čte na adrese = ALUout
3. Výstup z paměti se zapíše do datového registru paměti (MDR)

   (Instrukce zápisu je symetrická)

Provedení instrukce formátu R - mikroinstrukce

Provedení instrukce R-formátu

Mikroinstrukce:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
Rformat1	Func code	A	B	---	---	---	Seq
---	---	---	---	Write ALU	---	---	Fetch

Akce:

1. Operace ALU je specifikována polem v mikroinstrukci *funct,* pracuje s A a B – výsledek je zapsán do registru ALUout
2. Výsledek v ALUout je zapsán do registrové sady a provede se skok zpět k místu volání tak, aby se načetla další instrukce MIPS

Větvení a skoky - mikroinstrukce

Větvení a skoky

Mikroinstrukce:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
Beq1	Subt	A	B	---	---	ALUout-cond	Fetch
Jump1	---	---	---	---	---	Jump address	Fetch

Akce:

- *Branch* provede A-B v ALU tak, aby se nastavil výstup Zero v případě, že A=B, potom se skáče na BTA, vypočítanou během kroku dekódování instrukce
- *Jump* – skok se provede zápisem JTA do PC
- Obě mikroinstrukce se navrací do bodu, odkud byly volány pomocí výběrové tabulky (např., volání Beq1 nebo Jump1)

Kompletní mikroprogram – výběrová tabulka

Kompletní mikroprogram

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
Fetch	Add	PC	4	---	Read PC	ALU	Seq
---	Add	PC	Extshft	Read	---	---	Dispatch 1
Mem1	Add	A	Extend	---	---	---	Dispatch 2
LW2	---	---	---	---	Read ALU	---	Seq
---	---	---	---	Write MDR	---	---	Fetch
SW2	---	---	---	---	Write ALU	---	Fetch
Rformat1	Func code	A	B	---	---	---	Seq
---	---	---	---	Write ALU	---	---	Fetch
Beq1	Subt	A	B	---	---	ALUout-cond	Fetch
Jump1	---	---	---	---	---	Jump address	Fetch

- **Výběrová tabulka 1:** Mem1, Rformat1, Beq1, Jump1
- **Výběrová tabulka 2:** LW2 (load), SW2 (store)

Problémy mikroprogramování, mprog sequencer, nesprávná představa, omyl

Problémy mikroprogramování

- Hardwarová implementace
  - Podobné řízení s pevným řadičem (FSC)
  - Stav uložen v registru, přechodová funkce v ROM nebo PLA
  - *Možnosti*: mprog sequencer používá *čítač (counter)*
- Nesprávná představa: Mikroprogram je rychlejší
  - Kdysi: Paměť mikroprogramu tvořila rychlá paměť
  - Dnes: Použití cache, výhoda se ztrácí
  - Základ: Je snazší měnit software než hardware
- Omyl: Nové mikroinstrukce jsou "zadarmo"
  - Paměť mikroprogramu nemusí být zcela zaplněna (pro začátek)
  - Později lze přidávat další instrukce

Výjimky a interrupty, přerušení – definice, typy, rozdíly

Nové: Výjimky a interrupty

Definice:
Událost, způsobující změnu toku instrukcí mimo normální běh programu.

Typy: (1) Výjimka [overflow]
(2) Interrupt [I/O]

Rozdíly: *Výjimka* generovaná uvnitř procesoru (synchronní)
Interrupt odvozen od *externí* události (asynchronní)

Úkoly:

- *Detekce výjimky* – Jak odhalit výjimku
- *Ošetření výjimky* – Co dělat

MIPS: 2 typy – Nedefinované instrukce, aritmetické přetečení

Detekce výjimek – nedefinované instrukce , aritmetické přetečení, overflow

Detekce výjimek

- **Nedefinované instrukce:**
  1. Doplnění stavu 10 [Exception] do FSC
  2. Každá instrukce vyjma lw, sw, beq, R-formát nebo jump způsobí přechod do stavu 10
- **Aritmetické přetečení (overflow):**
  1. Opakování: ALU obsahuje logiku detekce přetečení. Vytvoření dalšího stavu 11 v FSC pro obsluhu přetečení
  2. Vznikne-li *přetečení* (*Overflow) na výstupu* ALU, pak řadič přejde do stavu 11

Ošetření vyjimek, vektorové interrupty, EPC, Cause

Ošetření výjimek

- **Dvě metody:** *EPC/Cause* a *Vektorové interrupty*
- *Vektorové interrupty*:
  1. Každá výjimka má vyhrazenou určitou adresu A_E
  2. Výjimka detekována => A_E je kvůli ošetření zapsána do PC
- *EPC / Příčina:* (MIPS) (EPC …Exception Program Counter)
  1. Výjimka detekována => Adresa instrukce uložena do **$epc**
     Cause registr obsahuje kód výjimky
  2. Obsluha výjimky vyšetří registr **Cause** a zkouší restartovat výpočet na místě, kam ukazuje **$epc.**

Úprava MIPS pro výjimky, EpcWrite, CauseWrite

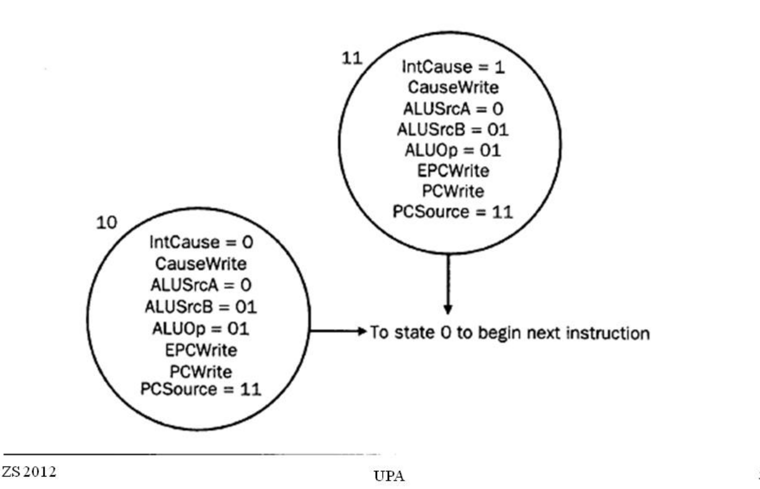
Úprava MIPS pro výjimky

1. Nové registry - **$epc** a **Cause** (32-bitů)
2. Nové řídící signály – EpcWrite, CauseWrite
3. Nové řídící linky – 0 pro nedefinovanou instrukci
   1 pro přetečení
4. Nový Mux signál pro zdroj PC - PCsource = 11_2
   - Staré vstupy PC: PC+4, BTA, JTA
   - Nový vstup: A_E = $C0000000_{16}$ u MIPS

Opakování: detekce přetečení ALU "již instalováno"

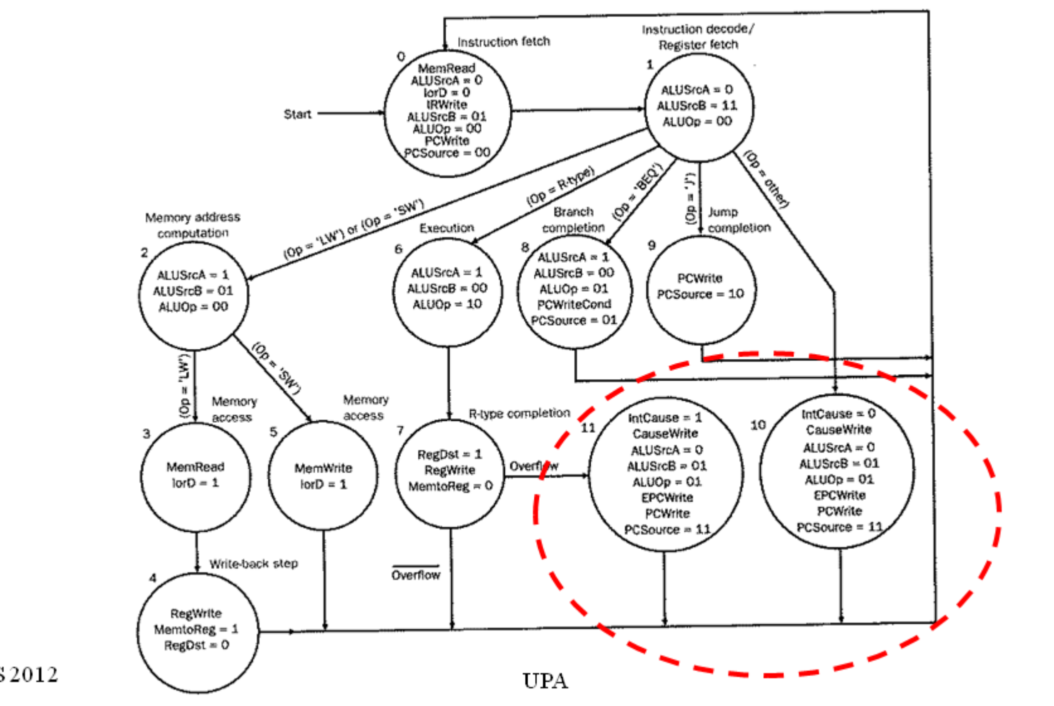
Nové stavy pro ošetření vyjimek

Nové stavy pro ošetření výjimek



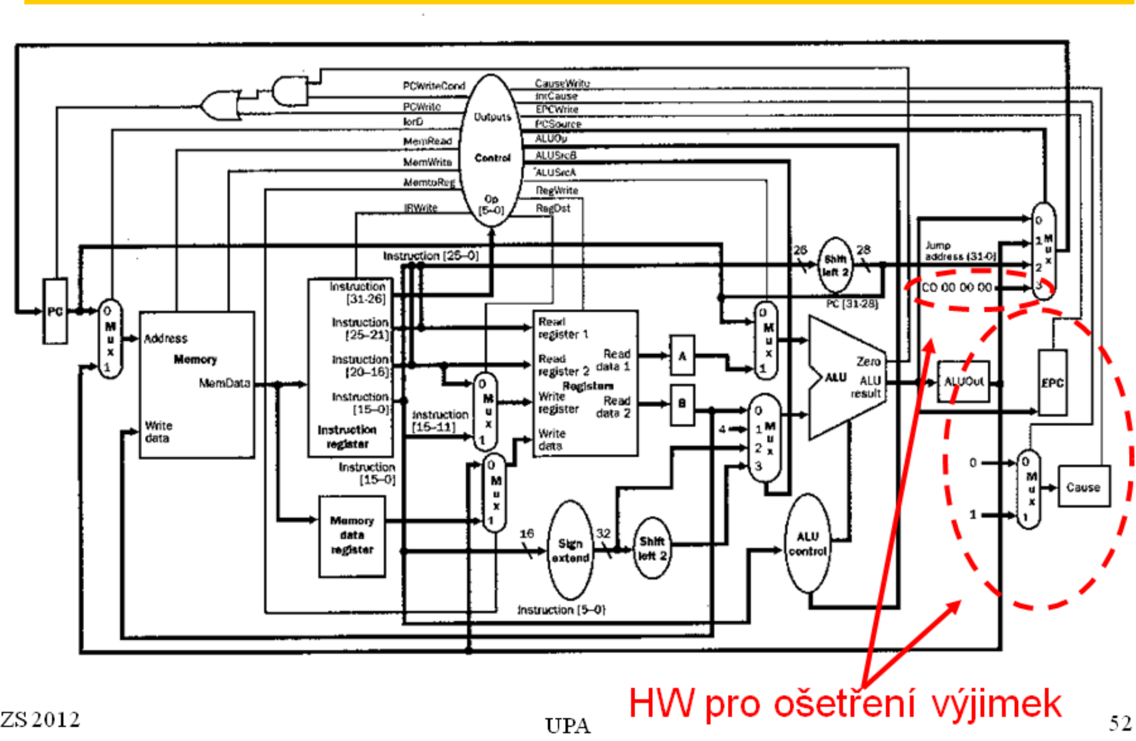
FSC včetně výjimek

Nový FSC se stavy výjimek



Uspořádání MIPS včetně vyjíme – hw pro výjimky, schéma

Nové uspořádání MIPS



Problémy spojené s výjimkami – Rollback, restart, Jak napravit výjimky

Problémy spojené s výjimkami

- Rollback a restart
  - *Rollback*: Reverzní proces
  - *Restart*: Opětné provedení procesu nebo operace
  - Detekce přetečení *po* zápisu výsledku ALUout
  - To znamená, že operace ALU způsobila neopravitelnou chybu ☹
  - Současný FSC nedovoluje podporu procesu *restart* nebo *rollback*

Jak to napravit?

- **Obtížné** – Návrh řídícího systému je komplikovaný
  – K provedení *rollback*u je třeba mnoho dalšího HW

Mikroprogramování, úkol , Omyl, Výjimky, Interrupty– detekce, ošetření, datové cesty

Závěr

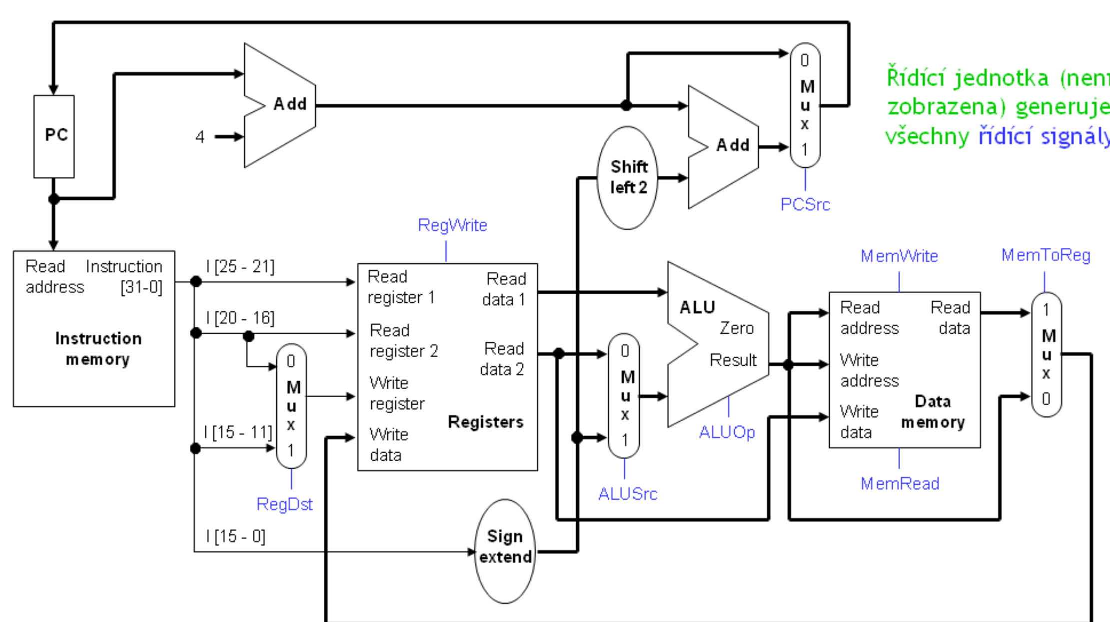
- **Mikroprogramování –** Flexibilnější pro rozlehlé ISA
- **Úkol:** Navrhnout pole a signály konzistentní
- **Omyl:** Mikroprogramy jsou nutně pomalejší
- **Omyl:** Přidávání instrukcí “je zdarma”
- **Výjimky** (interní události) vs. **Interrupty** (externí)
  - *Detekce výjimek* vyžaduje speciální hardware
  - *Ošetření výjimek* vyžaduje více hardware (cena!)

Závěr

- Jestliže rozumíme instrukcím…
  Můžeme postavit jednoduchý procesor!
- Trvají-li instrukce různou dobu, je výhodnější multicyklové zpracování
- Datové cesty se implementují s využitím:
  - Kombinační logiky pro aritmetiku
  - Paměťové prvky (klop. obvody) pro zapamatování informace
- Implementace řízení s využitím:
  - Kombinační logiky pro jednocyklovou implementaci
  - FSM pro vícecyklovou implementaci

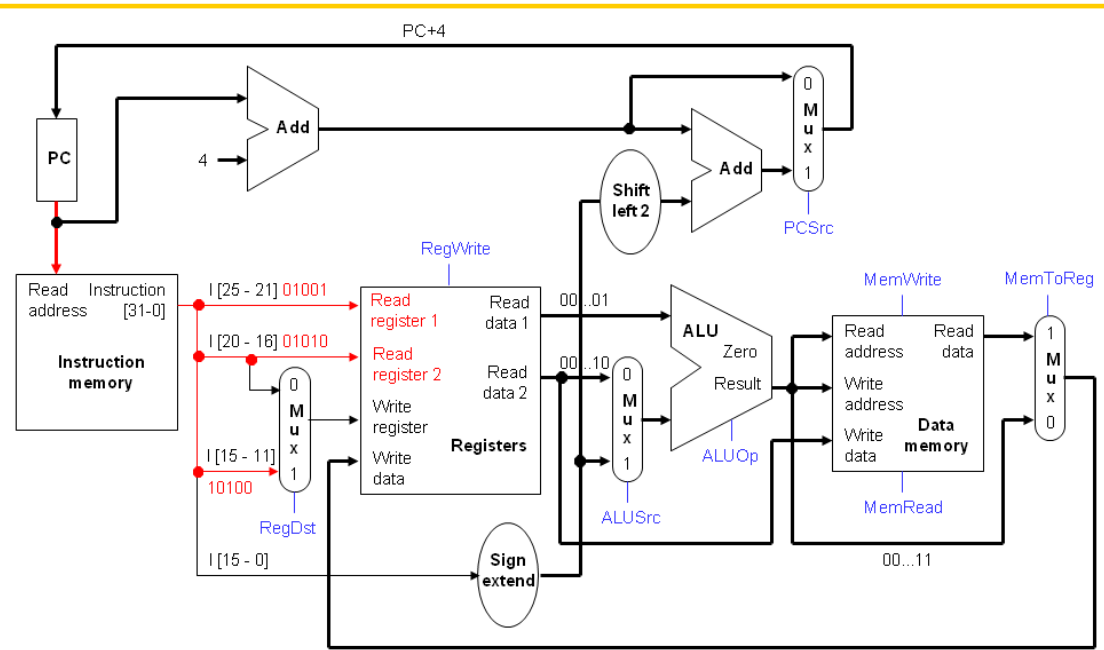
Jednocyklová jednotka

Jednocyklová jednotka



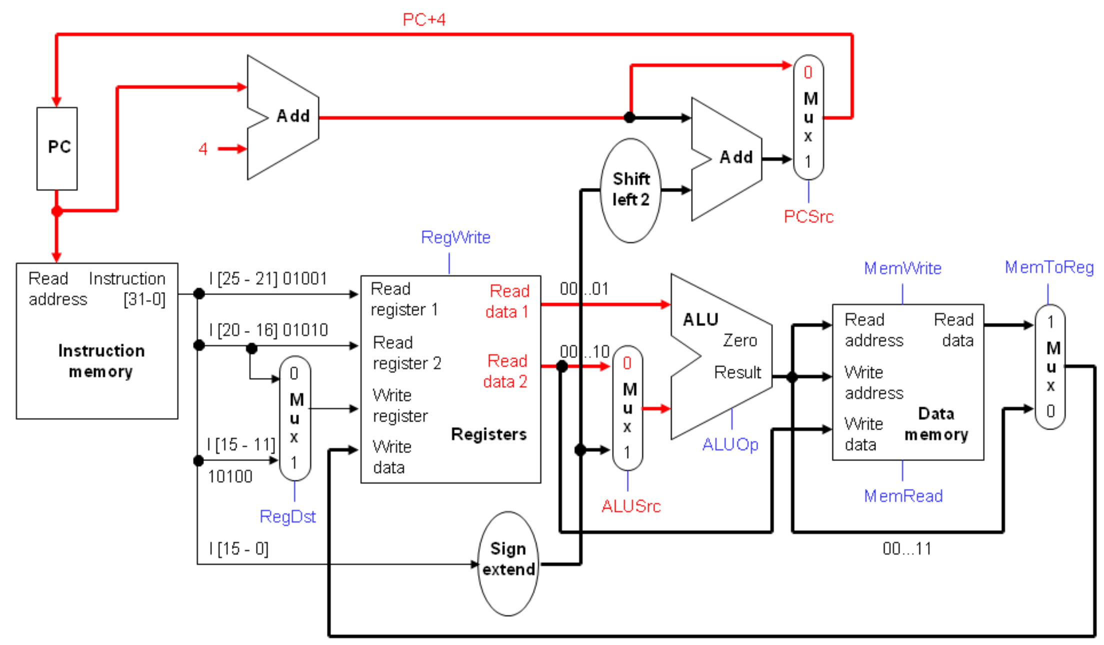
Provádění instrukce v jednocyklové jednotce 4 kroky

Provádění instrukce add $s4, $t1, $t2



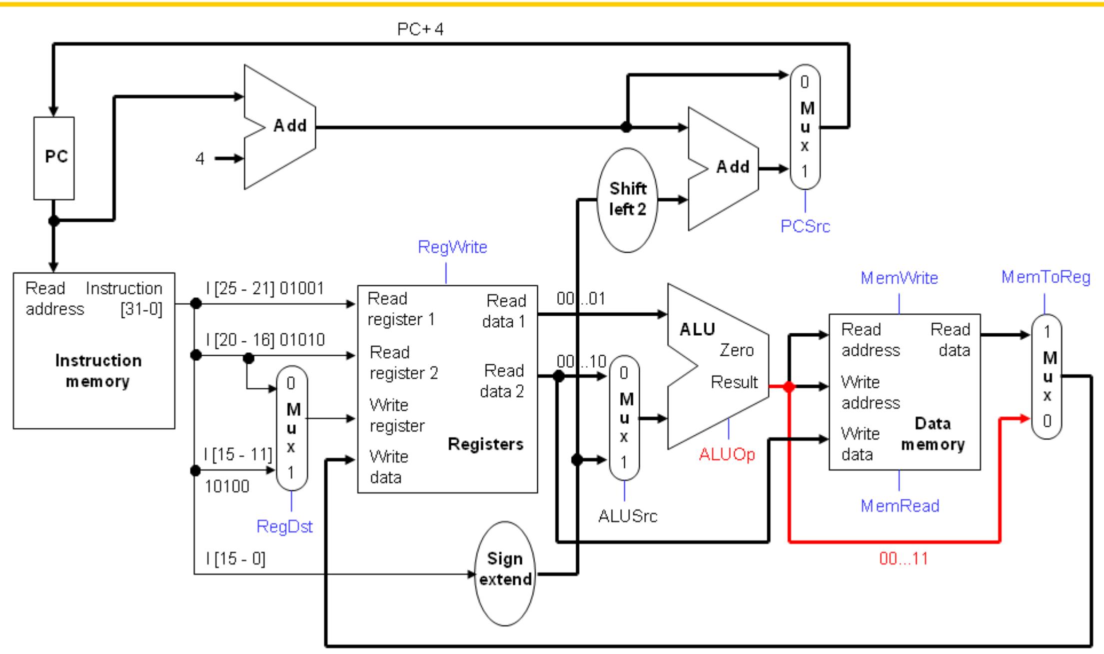
Čtení instrukce z paměti

Provádění instrukce add $s4, $t1, $t2



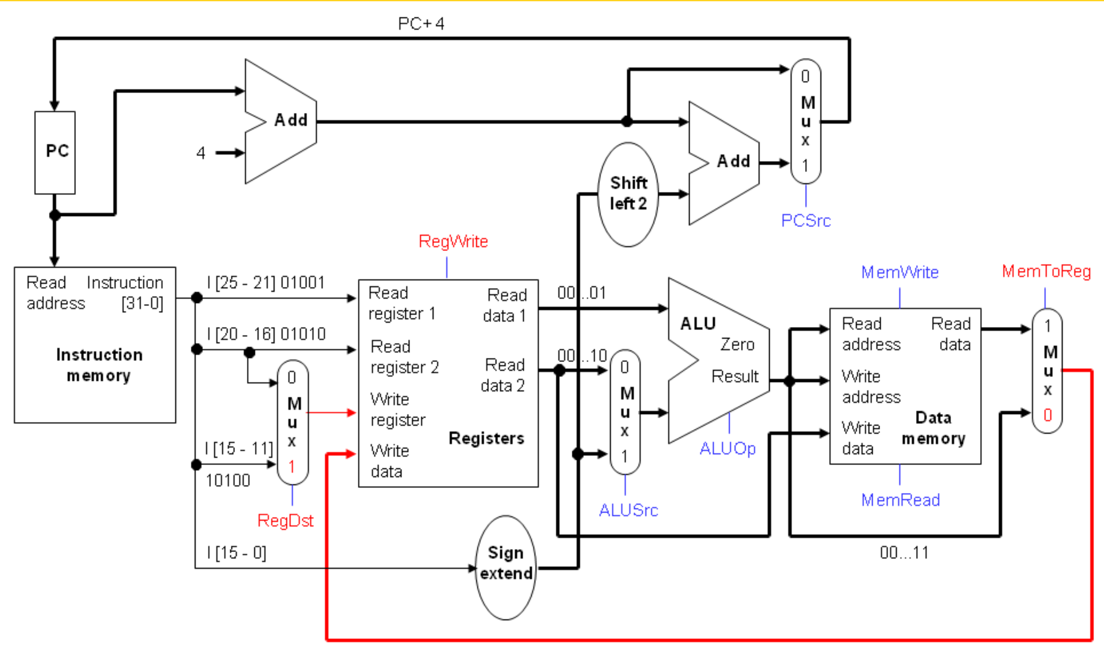
Čtení registrů, inkrement PC + 4, zápis nové hodnoty PC

Provádění instrukce add $s4, $t1, $t2



Provedení součtu v ALU

Provádění instrukce add $s4, $t1, $t2

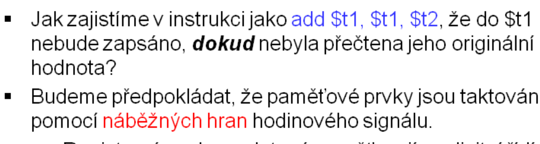


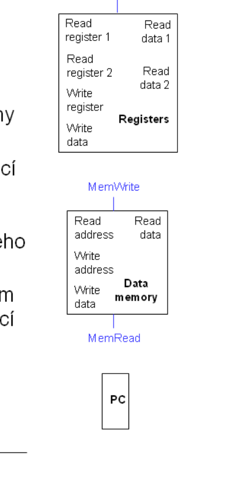
Zápis výsledku do registru

Časování hrany hodinového signálu, taktování pomocí náběžných hran

Časování: hrany hodinového signálu

- Jak zajistíme v instrukci jako add $t1, $t1, $t2, že do $t1 nebude zapsáno, ***dokud*** nebyla přečtena jeho originální hodnota?
- Budeme předpokládat, že paměťové prvky jsou taktovány pomocí náběžných hran hodinového signálu.
  - Registrový soubor a datová paměť mají explicitní řídící signály pro zápis, RegWrite a MemWrite. Do těchto jednotek se zapisuje v okamžiku, kdy je řídící signál aktivní a ***současně*** nastala náběžná hrana hodinového signálu.
  - V jednocyklovém stroji je do PC zapisováno v každém hodinovém cyklu, takže nepotřebujeme explicitní řídící signál.





Výkonnost jednocyklového procesoru, jednocyklová verze

Výkonnost jednocyklového procesoru

CPU time$_{X,P}$ = Instructions executed$_P$ * CPI$_{X,P}$ * Clock cycle time$_X$

CPI = 1 pro jednocyklový návrh

S náběžnou hranou hodin je do PC zapsána nová adresa.

1. Je přečtena nová instrukce z paměti. Řídící jednotka aktivuje řídící signály tak, aby se provedlo:
   — čtení registrů,
   — generování výstupu ALU,
   — čtení datové paměti pro instrukci lw
   — výpočet cílové adresy skoku.
2. S ***další*** náběžnou hranou hodin se provede:
   — Zápis do registrového souboru pro aritmetické instrukce nebo pro instrukci lw.
   — Zápis do datové paměti pro instrukci sw.
   — Stanoven nový obsah PC – ukazuje na další instrukci.

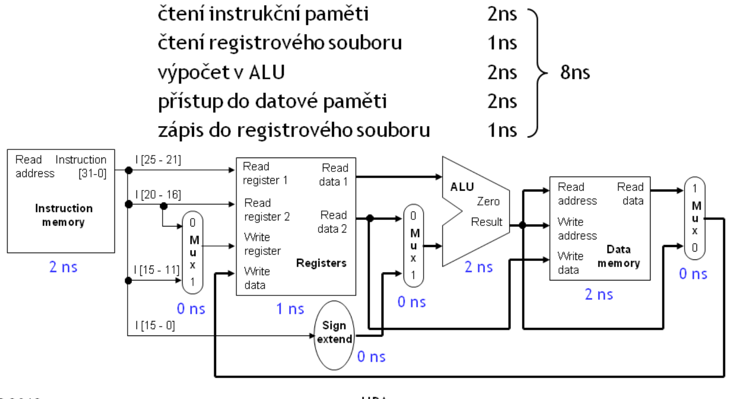
- U jednocyklové verze všechno v bodě 2 se musí odehrát během jednoho hodinového cyklu, před příchodem další náběžné hrany hodin.

Jak dlouho má trvat perioda hodin?

Prvky procesoru – všechny instrukce podle nejpomalejší

Prvky procesoru

- Jestliže se všechny instrukce musejí dokončit během jednoho cyklu, musí jeho délka vyhovět ***nejpomalejší*** instrukci.
- Každý prvek procesoru vykazuje určité zpoždění (latence)

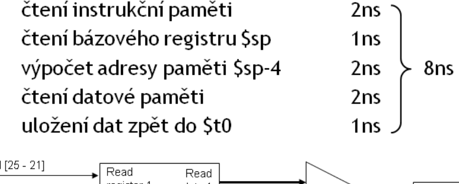


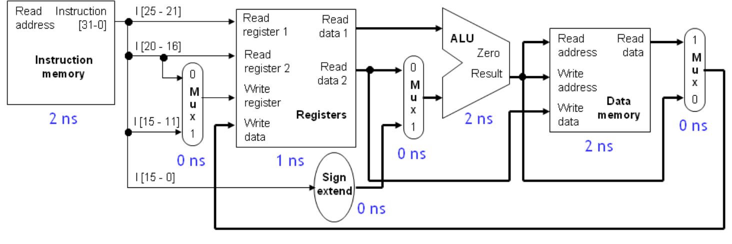
Nejpomalejší instrukce určuje dobu periody

Nejpomalejší instrukce...

Instrukce lw používá ***všechny*** prvky procesoru

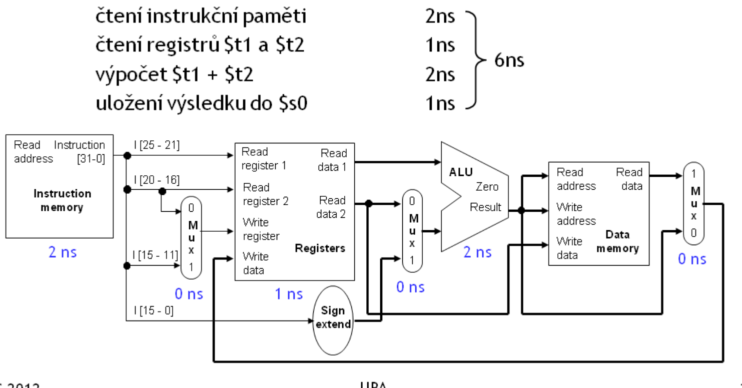
- Například, lw $t0, –4($sp) potřebuje 8 ns, použijeme-li následující odhad.





...určuje dobu periody hodin

- Zvolíme-li dobu cyklu 8 ns, potom *každá* instrukce bude trvat 8 ns, i když by tolik času nepotřebovala.
- Například instrukci add $s4, $t1, $t2 ve skutečnosti postačí jen 6 ns.



Lze to akceptovat – gcc, tabulka četnosti instrukcí, střední doba výpočtu instrukce pro gcc

Lze to akceptovat?

- Při stejných zpožděních by instrukce sw potřebovala 7 ns a beq jen 5 ns.
- Předpokládejme instrukční mix **gcc** z učebnice.

Instrukce	Četnost
Aritmetické	48%
Čtení (lw)	22%
Uložení (sw)	11%
Větvení	19%

- V případě jednocyklového návrhu by každá instrukce potřebovala 8 ns.
- Jestliže budeme provádět výpočet jak nejrychleji je možné, bude střední doba výpočtu instrukce pro gcc:

$$(48\% \times 6ns) + (22\% \times 8ns) + (11\% \times 7ns) + (19\% \times 5ns) = 6.36\ ns$$

- Jednocyklový návrh je přibližně 1.26 krát pomalejší!

Nejhorší případ cyklu load/store, přístup do hlavní paměti moderních strojů

Může být hůř...

- Použili jsme příliš optimistický předpoklad o době cyklu paměti:
  - Přístup do hlavní paměti moderních strojů je >50 ns.
  - Pro srovnání, operace ALU v Pentiu4 trvá ~0.3 ns.
- Náš nejhorší případ cyklu (load/store) zahrnuje 2 přístupy do paměti
  - U moderních jednocyklových implementací bychom skončili pod <10Mhz.
  - Cache paměti zlepšují střední přístupovou dobu, nikoliv nejhorší případ.

Zlepšení výkonu – 2 možnosti, vícecyklová implementace, pipeline

Zlepšení výkonu

- Dvě možnosti jak zlepšit výkon:
  1. Rozdělit každou instrukci do více kroků, každý bude trvat 1 cykl
     - kroky: IF (instruction fetch), ID (instruction decode), EX (execute ALU operation), MEM (memory access), WB (register write-back)
     - pomalé instrukce zaberou více cyklů, než ty rychlejší
     - = *vícecyklová* implementace
  2. Důležité „zjištění“: každá instrukce využívá pouze část hardware v každém kroku
     - instrukce lze ve zpracování *překrývat*; každá využívá jen část hw
     - = implementace s režimem *pipeline*
- Příklady pipeliningu: jakýkoliv proces montáže (auta, bagety), prádelna (pračka + sušička – lze provádět v režimu pipeline), atd.

Příklad pipeline – sestavení bagety

Příklad

- Sestavení bagety:
  - OBJ (8 sekund) Objednávka
  - TST (0 nebo 10 sekund) Nahřátí
  - EXT (0 nebo 10 sekund) Případná výplň
  - PLT (5 sekund) Placení

Jednoduchá bageta trvá 13 až 33 sekund

- Použijeme-li pipelining můžeme sestavit bagetu každých 10 sekund:

0–10	10–20	20–30	30–40	40–50	50–60
OBJ	TST	EXT	PLT		
	OBJ	TST	EXT	PLT	
		OBJ	TST	EXT	PLT

0 10 20 30 40 50 60

Pipelining – zvýšení propustnosti, optimální pipeline

Pipelining

- Pipelining může *zvýšit propustnost* (#baget za hodinu), ale ...

1. Každá bageta musí využívat *všechny* stupně
   — brání konfliktům v pipeline
2. každý stupeň musí zabírat stejné množství času
   — limitováno *nejpomalejším* stupněm (v tomto případě 10 sekund)

- Tyto dva faktory *snižují latenci* (doba/1bagetu)!
- V optimální k-stupňové pipeline struktuře:
  1. každý stupeň koná smysluplnou činnost
  2. délka stupňů je vyvážená
- Za těchto podmínek lze dosáhnout *téměř* optimální zrychlení: *k*
  — “téměř” protože stále tu existuje doba *plnění* a *vyprázdnění*

Pipelining není multiprocessing – obecný princip zvyšování efektivity

Pipelining není „multiprocessing“

- Pipelining je vlastně určitý druh paralelního zpracování.
- Jak multiprocessing, tak pipelining znamenají aktivaci více procesů ve více „funkčních jednotkách“
  - **Multiprocessing – každý proces běží celý v samostatné funkční jednotce**
    - např. pokladny v supermarketu
  - **Pipelining** – každý proces je rozčleněn na drobné části a každá je prováděna ve specializované funkční jednotce.
- Pipelining a multiprocessing se navzájem nevylučují
  - Moderní procesory uplatňují oba principy, násobné pipeline struktury (superskalární procesory)
- Pipelining je obecný princip zvyšování efektivity, používaný v počítačových systémech:
  - Sítě, I/O zařízení, softwarová architektura serverů

Pipelining u MIPSu – 5 kroků

Pipelining u MIPSu

- Provedení instrukce MIPS může zabrat 5 kroků

Krok	Jméno	Popis
Instruction Fetch	IF	Čtení instrukce z paměti
Instruction Decode	ID	Čtení zdrojových registrů a generace řídících signálů
Execute	EX	Výpočet výsledku R-typu popř. větvení
Memory	MEM	Čtení nebo zápis do datové paměti
Writeback	WB	Uložení výsledku do cílového registru

- Všechny instrukce nepotřebují všech 5 stupnů…

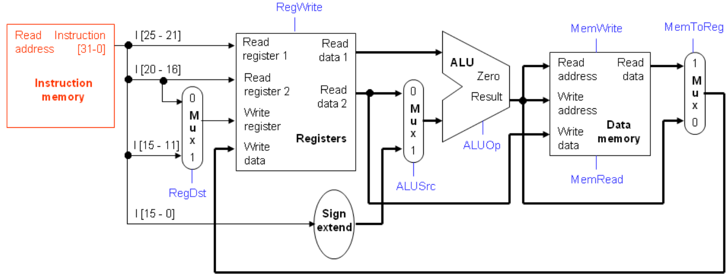
Instrukce	Vyžadované kroky				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

- …jednocyklová verze ale musí zvládnout všech 5 stupňů během jednoho taktu

Provádění jednocyklového čtení instrukce u MIPS – IF, ID, EX, MEM

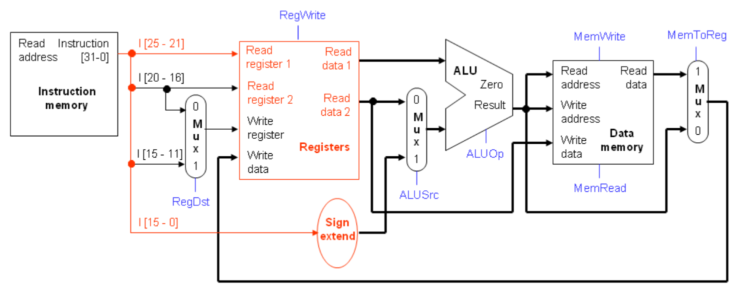
Čtení instrukce (IF)

- Zatímco se provádí IF, zbytek jednotky je nečinný …



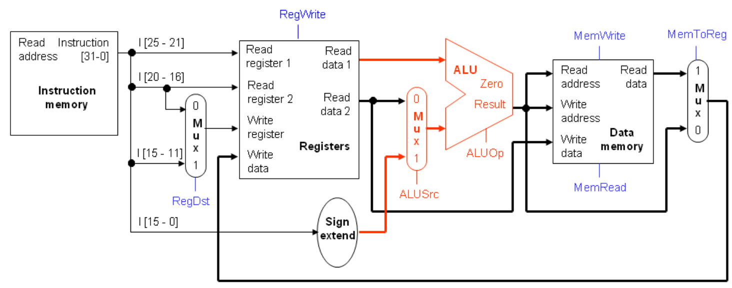
Dekódování instrukce (ID)

- Při provádění ID je část, odpovídající IF nečinná …



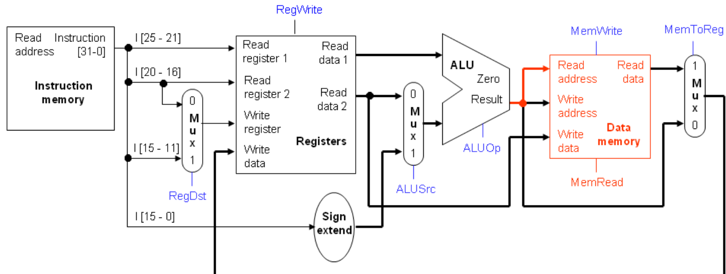
Prováděcí fáze (EX)

- ..totéž platí pro EX …



Paměť (MEM)

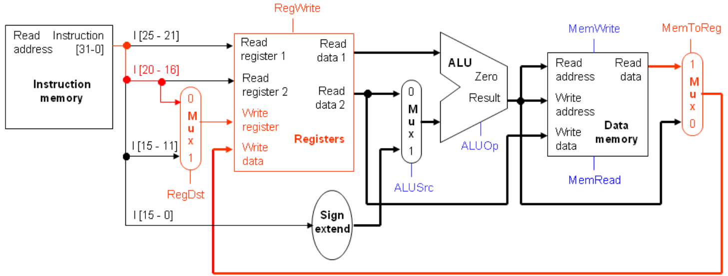
- … část MEM …



Zápis výsledků WB jednocyklové čtení, konflikt - nenastane

Zápis výsledku (WB)

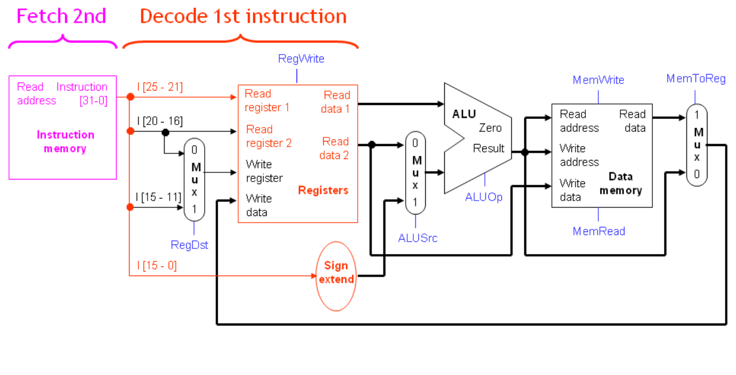
- … totéž pro část, odpovídající WB.
- „konflikt“ ve stupni IF v registrovém souboru?
- Odpověď: Do registrového souboru se zapisuje s náběžnou hranou, čtení probíhá v další části hodinového cyklu. Konflikt tedy nenastává.



Dekódování a čtení instrukce současně – pipeline, pipelining v krocích

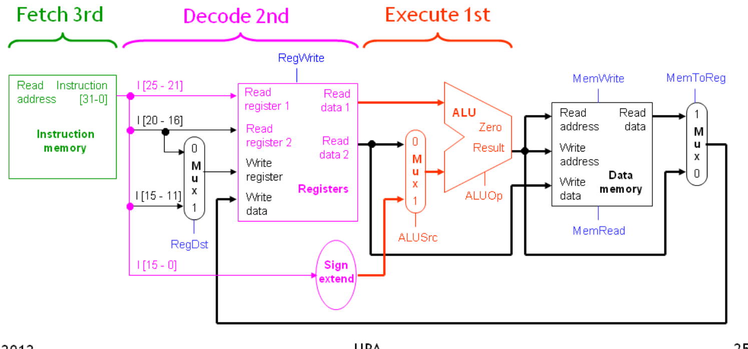
Dekódování a čtení instrukce současně

- Můžeme jít dále a číst další instrukci, zatímco se přečtená dekóduje?



Provádění, dekódování a čtení instrukce

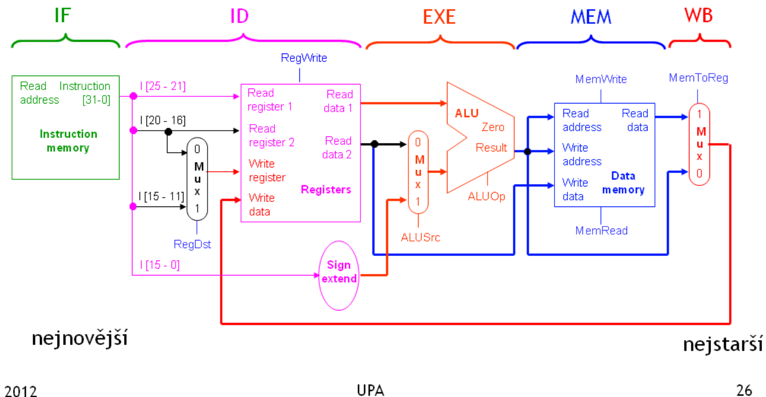
- Podobně, jakmile vstoupí instrukce do prováděcího stupně, lze zároveň dekódovat druhou instrukci.
- Instrukční paměť je ale teď opět volná a proto můžeme načíst třetí instrukci!



Rozdělení zpracování do 5 stupňů – pipeline rozdělení

Rozdělení jednotky do 5 stupňů

- Každý stupeň má svoji vlastní funkční jednotku
- Plný režim pipeline ⇒ procesor současně zpracovává 5 instrukcí!



Pipeline diagram

Pipeline diagram

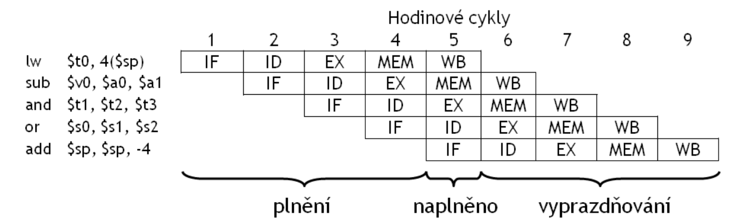
Hodinové cykly

	1	2	3	4	5	6	7	8	9
lw $t0, 4($sp)	IF	ID	EX	MEM	WB				
sub $v0, $a0, $a1		IF	ID	EX	MEM	WB			
and $t1, $t2, $t3			IF	ID	EX	MEM	WB		
or $s0, $s1, $s2				IF	ID	EX	MEM	WB	
addi $sp, $sp, -4					IF	ID	EX	MEM	WB

- Pipeline diagram ukazuje provádění série instrukcí
- Posloupnost instrukcí se zobrazuje vertikálně shora dolů
- Hodinové cykly ze zobrazují horizontálně zleva doprava
- Každá instrukce je rozdělena do stupnů
- Názorně ukazuje překrývání instrukcí. Například ve třetím cyklu jsou aktivní tři instrukce.
- Instrukce "lw" je ve stupni Execute.
- Současně "sub" je ve stupni Decode.
- Konečně instrukce "and" se právě načítá.

Pipeline terminologie, hloubka pipeline, plnění, vyprazdňování

Terminologie



Hodinové cykly

	1	2	3	4	5	6	7	8	9
lw $t0, 4($sp)	IF	ID	EX	MEM	WB				
sub $v0, $a0, $a1		IF	ID	EX	MEM	WB			
and $t1, $t2, $t3			IF	ID	EX	MEM	WB		
or $s0, $s1, $s2				IF	ID	EX	MEM	WB	
add $sp, $sp, -4					IF	ID	EX	MEM	WB

plnění naplněno vyprazdňování

- Hloubka pipeline je počet stupňů, zde pět
- V prvních čtyřech probíhá plnění, jsou zde nevyužité funkční jednotky
- V cyklu 5 je pipeline plná. Probíhá zpracování pěti instrukcí současně, jsou využity všechny hardwarové jednotky
- V cyklech 6-9 se pipeline vyprazdňuje

Výkon pipeline – vzorec výpočtu

Výkon pipeline struktury

		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
lw	$t1, 8($sp)		IF	ID	EX	MEM	WB			
lw	$t2, 12($sp)			IF	ID	EX	MEM	WB		
lw	$t3, 16($sp)				IF	ID	EX	MEM	WB	
lw	$t4, 20($sp)					IF	ID	EX	MEM	WB

Hodinové cykly; plnění

- Doba výpočtu v ideální pipeline:
  - —doba pro naplnění pipeline + jeden cykl na instrukci
  - —Kolik času pro N instrukcí? $k - 1 + N$, kde k = hloubka pipeline
- Jiný způsob odvození: k cyklů pro první instrukci, plus 1 pro každou ze zbývajících $N - 1$ instrukcí.
- Porovnejte tuto pipeline implementaci (2 ns perioda hodin) s jednocyklovou implementací (8 ns perioda hodin). Kolikrát bude rychlejší pro N=1000 ?

Datové cesty pro pipelining (pipeline) – požadavky (ADDER)

Datové cesty pro pipelinning: Požadavky

		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
lw	$t1, 8($sp)		IF	ID	EX	MEM	WB			
lw	$t2, 12($sp)			IF	ID	EX	MEM	WB		
lw	$t3, 16($sp)				IF	ID	EX	MEM	WB	
lw	$t4, 20($sp)					IF	ID	EX	MEM	WB

Hodinové cykly

- Musíme provádět více operací v každém cyklu.
  - – Inkrementace PC a sečtení registrů ve stejnou dobu.
  - – Čtení instrukce v době, kdy jiná instrukce čte nebo zapisuje data.
- Jaké změny to přináší pro hardware?
  - – Oddělená sčítačka (ADDER) a ALU
  - – Dvě paměti (instrukční paměť a datová paměť)

Pipelining u dalších instrukcí – IF, ID, EX, WB, typ R

Pipelining u dalších instrukcí

- Instrukce typu-R vyžadují pouze 4 stupně: IF, ID, EX a WB
  - —Nepotřebujeme stupeň MEM
- Co se stane, načteme-li do pipeline instrukce typu-R?

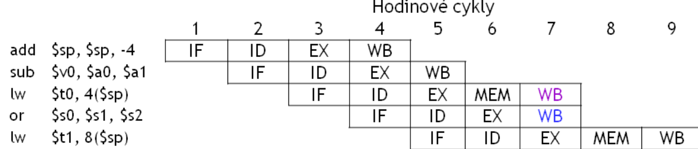
		1	2	3	4	5	6	7	8	9
add	$sp, $sp, -4	IF	ID	EX	WB					
sub	$v0, $a0, $a1		IF	ID	EX	WB				
lw	$t0, 4($sp)			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	WB		
lw	$t1, 8($sp)					IF	ID	EX	MEM	WB

Hodinové cykly

Zjištění – pipeline, každá funkční jednotka použita jen jednou, každá funkční jednotka musí být použita ve stejném stupni

Důležité zjištění

- Každá funkční jednotka smí být použita jen jednou na instrukci
- Každá funkční jednotka musí být použita ve stejném stupni pro všechny instrukce:
  - —Load používá zápisový port registrového souboru v 5. stupni
  - —R-typ používá zápisový port registrového souboru ve 4. stupni



Hodinové cykly

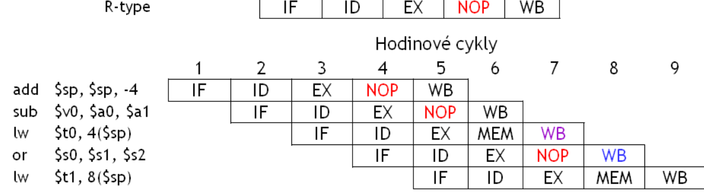
	1	2	3	4	5	6	7	8	9
add $sp, $sp, -4	IF	ID	EX	WB					
sub $v0, $a0, $a1		IF	ID	EX	WB				
lw $t0, 4($sp)			IF	ID	EX	MEM	WB		
or $s0, $s1, $s2				IF	ID	EX	WB		
lw $t1, 8($sp)					IF	ID	EX	MEM	WB

Řešení zjištění – vložení NOP, bloky neaktivní

Řešení: Vložení NOP

- Vynucení uniformity
  - —Navrhnout tak, aby všechny instrukce trvaly 5 cyklů.
  - —Navrhnout se stejným počtem stupňů, ve stejném pořadí
    - některé stupně budou neaktivní pro některé instrukce

R-type	IF	ID	EX	NOP	WB



Hodinové cykly

	1	2	3	4	5	6	7	8	9
add $sp, $sp, -4	IF	ID	EX	NOP	WB				
sub $v0, $a0, $a1		IF	ID	EX	NOP	WB			
lw $t0, 4($sp)			IF	ID	EX	MEM	WB		
or $s0, $s1, $s2				IF	ID	EX	NOP	WB	
lw $t1, 8($sp)					IF	ID	EX	MEM	WB

- Zápisy a větvení mají NOPy …

store	IF	ID	EX	MEM	NOP
branch	IF	ID	EX	NOP	NOP

Pipelining – maximalizace propustnosti, urychlení, datové cesty

Závěr

- Pipelining maximalizuje propustnost překrývaným zpracováním instrukcí.
- Pipelining poskytuje značné urychlení.
  - —V optimálním případě se v každém cyklu dokončuje jedna instrukce a zrychlení je rovno hloubce pipeline.
- Datové cesty se podobají cestám pro jednocyklové zpracování, navíc jsou doplněny pipeline registry
  - —Každý stupeň potřebuje svoji funkční jednotku

Pipelining - zrychlení

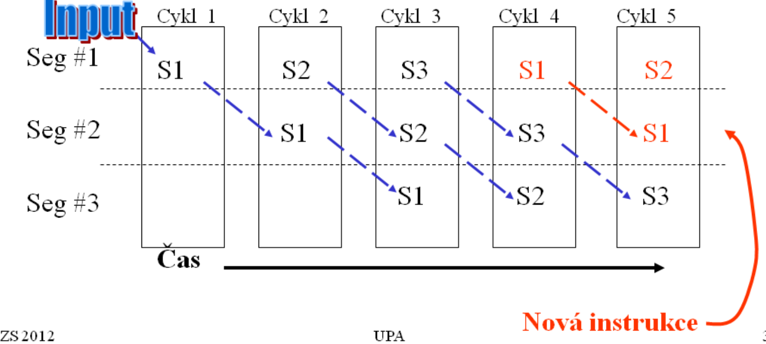
Pipelining

- Provedení instrukcí s překrýváním
- Paralelismus na úrovni instrukcí (souběžnost)
- Příklad na pipelining: linka ("T" u Forda)
- Doba odezvy pro každou instrukci je stejná
- Průchodnost instrukcí se zvyšuje ☺
- Zrychlení = *k* x počet kroků (stupňů)
  - Teorie: *k* je velká konstanta
  - Realita: Pipelining provází "další náklady"

Pipelining příklad - předpoklad

Příklad pipeliningu

- Předpoklad: Jeden instrukční formát (snadné)
- Předpoklad: Každá instrukce má 3 kroky S1..S3
- Předpoklad: Pipeline má 3 segmenty (jeden/krok)



Návrh ISA pro pipelining – dekódování a čtení obsahu registrů ve stejnou dobu, operandy zarovnány, instrukce mají stejnou délku

Návrh ISA pro pipelining

- Instrukce mají mít stejnou délku
  - Snadné provedení IF a ID
  - Podobně je tomu u *multicyklových jednotek*
- Malý počet konzistentních instrukčních formátů
  - ID registrů na stejném místě (rd, rs, rt)
  - Dekódování a čtení obsahu registrů ve stejnou dobu
- Paměťový operand *pouze* u instrukcí lw a sw
- Operandy jsou *zarovnány* v paměti

Co by se mělo změnit – změny oproti jednocyklové verzi

Co by se mělo změnit?

- Celkem nic! Struktura je téměř shodná s původní jednocyklovou variantou.
  - Paměti pro instrukce a data jsou oddělené.
  - Procesor obsahuje jednu ALU a dvě sčítačky pro výpočty adres.
  - Řídicí signály jsou stejné.
- Pouze byly provedeny některé kosmetické úpravy tak, aby se zmenšilo schéma.
  - Byla vynechána návěšti a zmenšeny multiplexery.
  - Datová paměť má pouze jeden vstup Address. Aktuální operaci paměti určují řídicí signály MemRead a MemWrite.
- Byl vytvořen prostor pro vložení *pipeline registrů*.

Pipeline registry – IF/ID, ID/EX, EX/MEM, MEM/WB

Pipeline registry

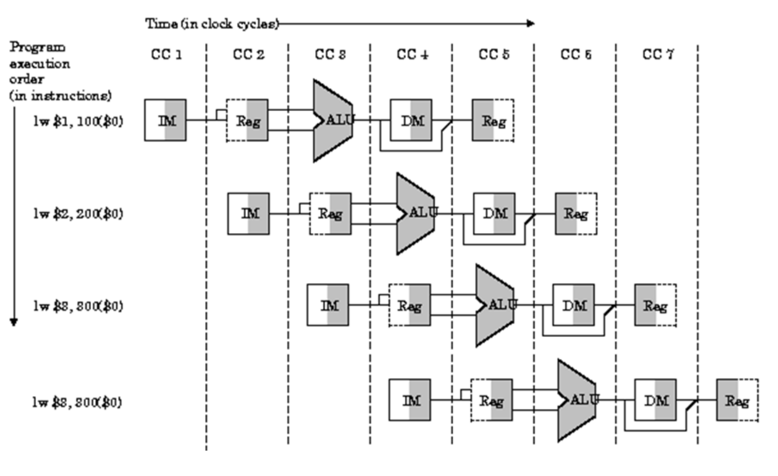
- V režimu pipeline dělíme vykonání instrukce do více cyklů.
- Informace vypočtená během jednoho cyklu se použije v dalších cyklech:
  - Instrukce načtená ve stupni IF určuje, které registry se plní ve stupni ID, které přímé operandy se použijí ve stupni EX a kam se zapisuje výsledek ve fázi WB.
  - Hodnoty registrů načtené v ID se použijí ve stupních EX a/nebo MEM
  - Výstup ALU generovaný v EX představuje efektivní adresu pro MEM nebo výsledek ve fázi WB
- Je třeba ukládat spoustu informace!
  - Ukládá se do registrů, které se nazývají pipeline registry
- Registry jsou pojmenovány podle stupňů, které propojují.

IF/ID ID/EX EX/MEM MEM/WB

- Na konci stupně WB není třeba žádný registr, protože po průchodu stupněm WB je instrukce dokončena.

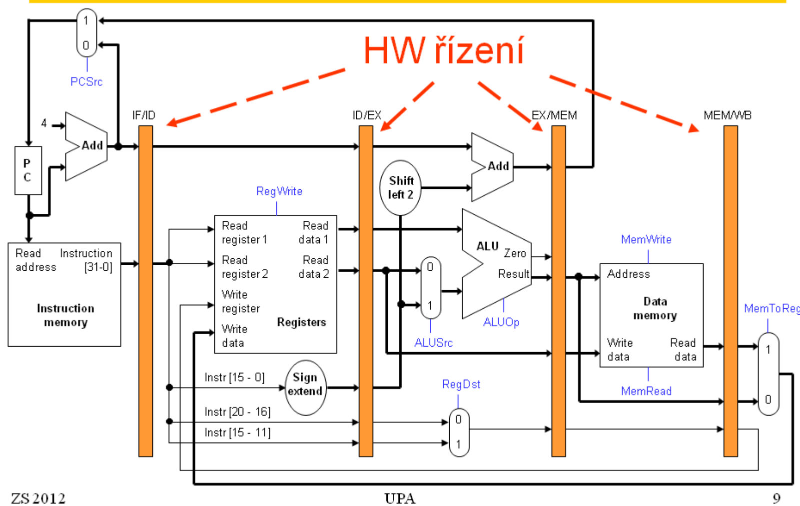
Výpočetní struktura uspořádání pipeline

Výpočetní struktura – uspořádání pipeline



Režim pipeline – jednotka, HW řízení, schéma pipeline registrů

Jednotka - režim pipeline



Přesun rozpracovaných dat – nejdelší cesta registru

Přesun rozpracovaných dat

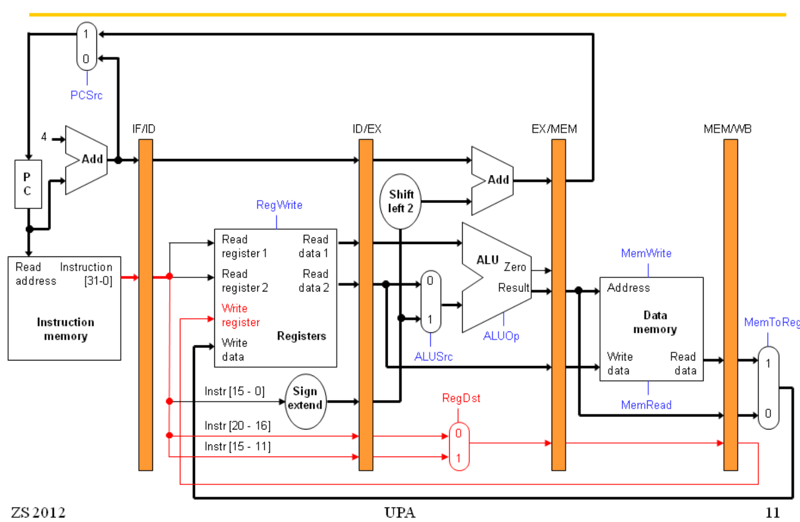
Data potřebná *později* procházejí postupně pipeline registry.

- Nejdelší cestu musí vykonat registr určení (rd nebo rt)
  - Hodnota je získána v IF, hodnota je použita ve WB
  - Musí tedy projít *všemi* stupni pipeline, jak je znázorněno na dalším obrázku

- Poznámka: Nelze uchovávat jednoduchý “instrukční registr”, protože procesory pracující v režimu pipeline musí v každém cyklu načítat novou instrukci.

Registr adresy výsledku - schéma

Registr adresy výsledku



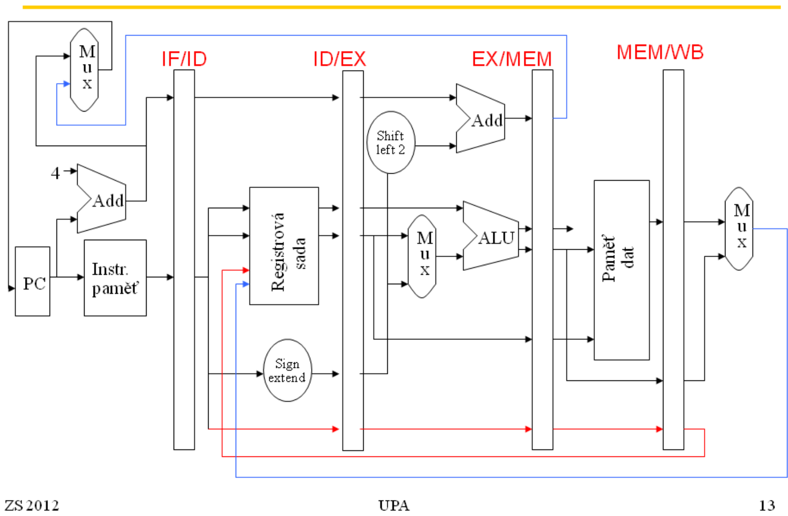
Pipelining – průchodnost, Teorie, realita

Pipelining

- Provedení instrukcí s překrýváním
- Paralelismus na úrovni instrukcí (souběžnost)
- Fyzický pipelining: automobilová linka
- Doba odezvy pro každou instrukci je stejná
- Průchodnost instrukcí se zvyšuje
- Zrychlení = *k* x počet kroků (stupňů)
  - Teorie: *k* je velká konstanta
  - Realita: Pipelining provází "další náklady"

Jednotka v režimu pipeline - schéma

Jednotka v režimu pipeline



Pipeline u MIPS – kroky, segmenty pipeline struktury

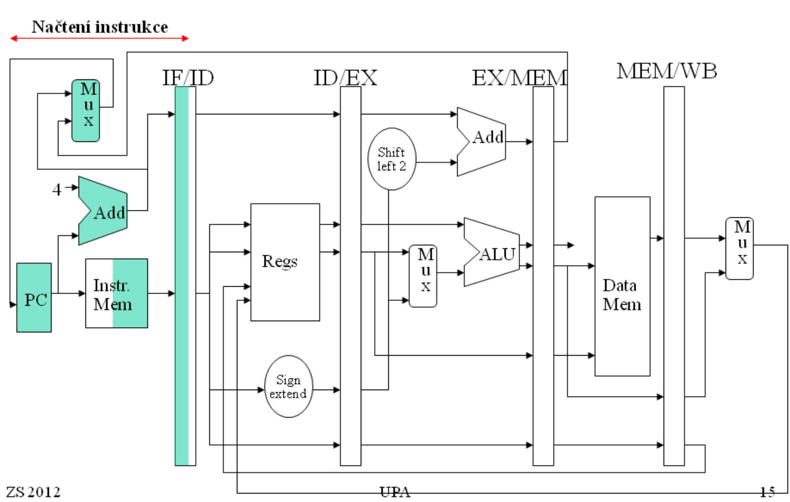
Pipeline u MIPS

- Podmnožina MIPS
  - *Přístup do paměti*: lw a sw
  - *Aritmetické a logické instrukce*: and, sub, and, or, slt
  - *Instrukce větvení*: beq

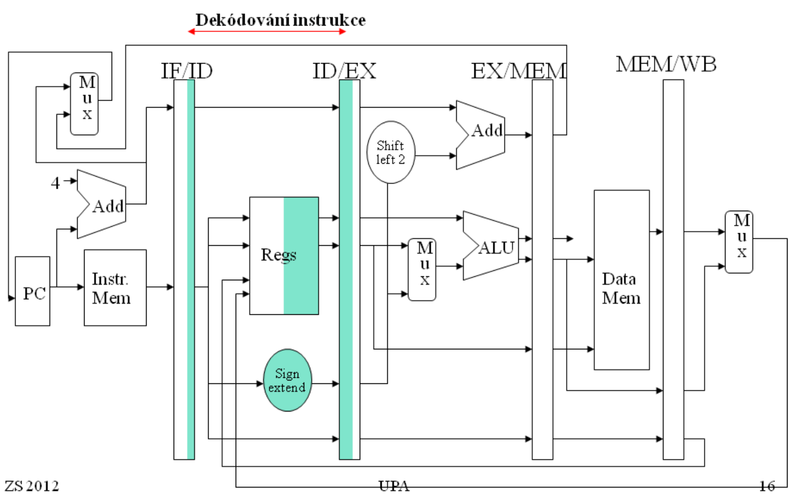
- Kroky (segmenty pipeline struktury)
  - **IF**: načtení instrukce z paměti
  - **ID:** dekódování instrukce a čtení registrů
  - **EX:** provedení operace nebo výpočet adresy
  - **MEM:** přístup k operandu do datové paměti
  - **WB:** zápis výsledku do registru

Provedení LW v jednocyklu – 5 kroků

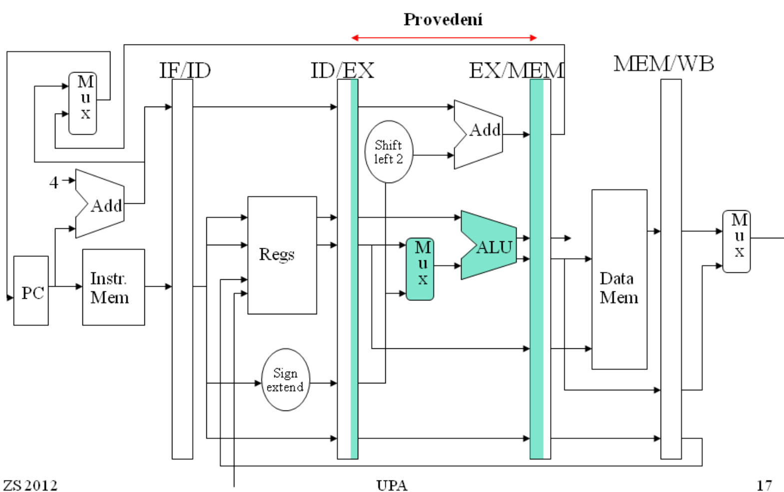
Provedení lw (1/5)



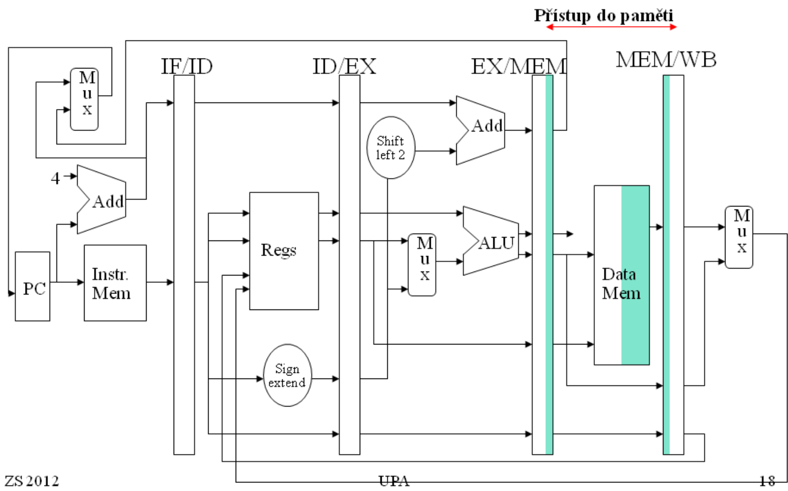
Provedení lw (2/5)



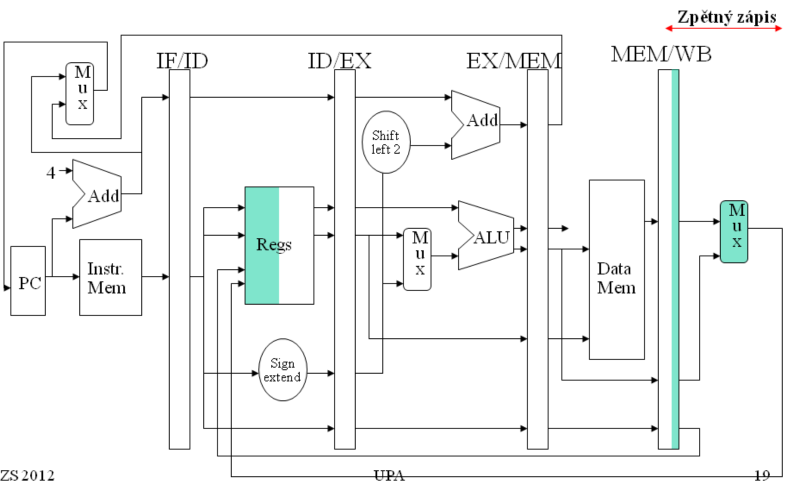
Provedení lw (3/5)



Provedení lw (4/5)

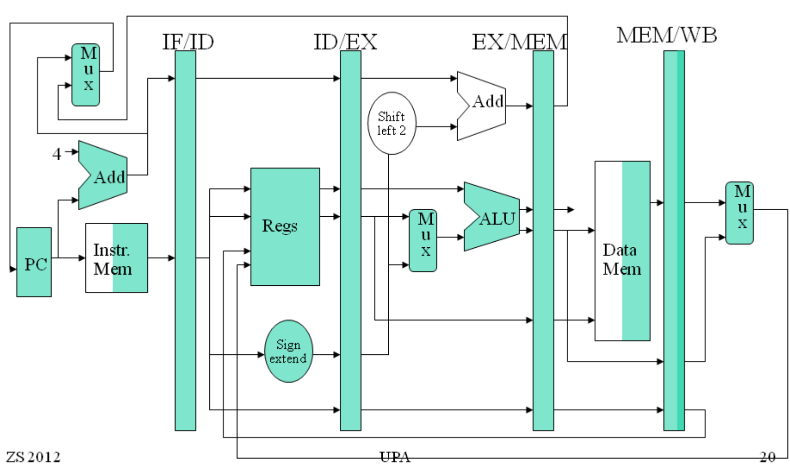


Provedení lw (5/5)



Zdroje použité při vykonání LW v jednocyklu

Zdroje použité pro vykonání lw



Časový rozvoj pipeline - diagram

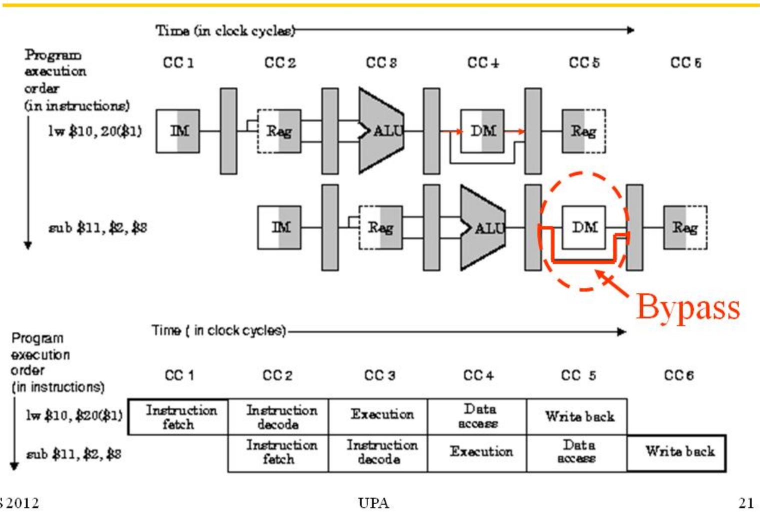
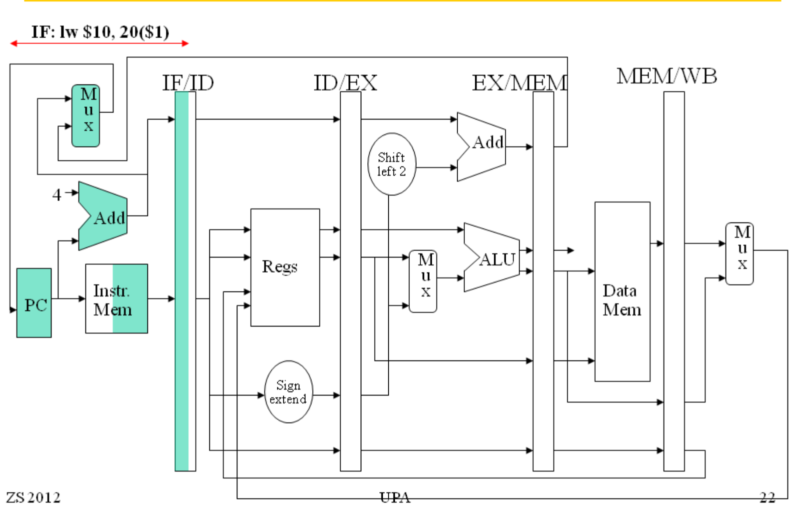


Diagram pipeline – 5 kroků



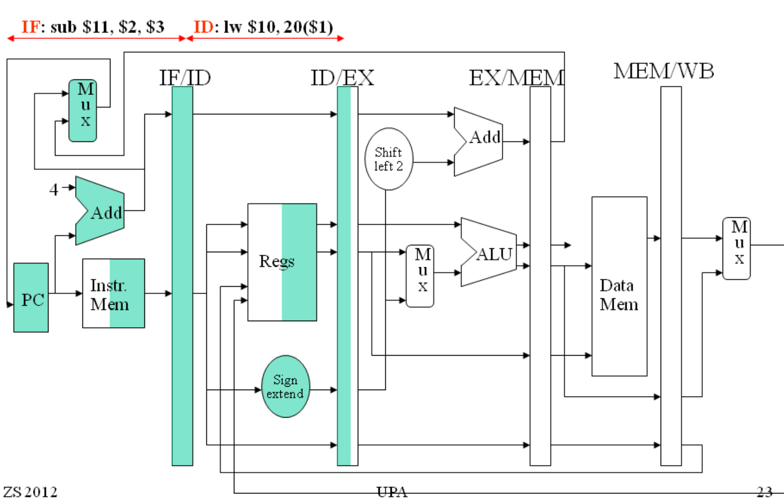


Diagram jednoho cyklu (cykl 3)

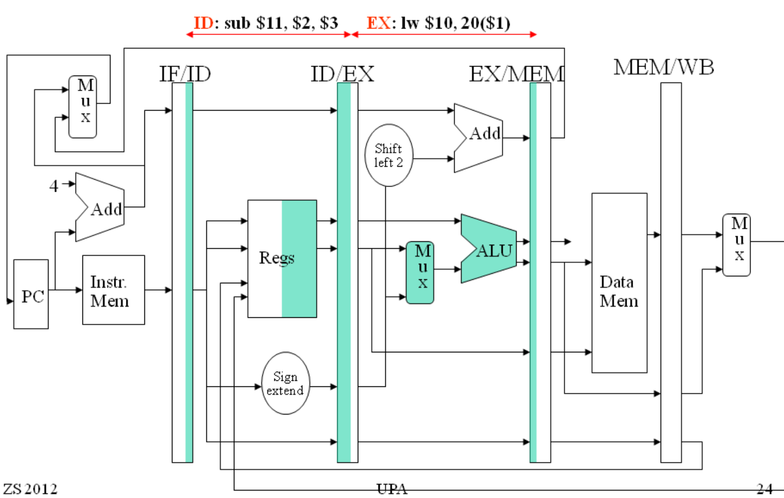


Diagram jednoho cyklu (cykl 4)

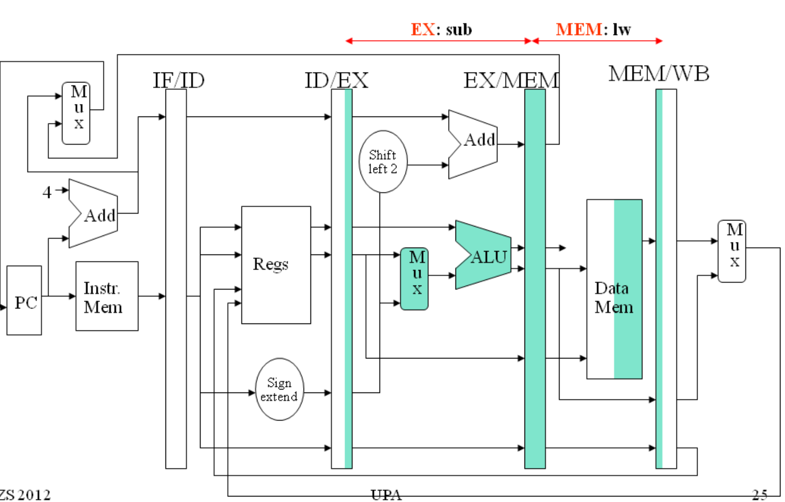


Diagram jednoho cyklu (cykl 5)

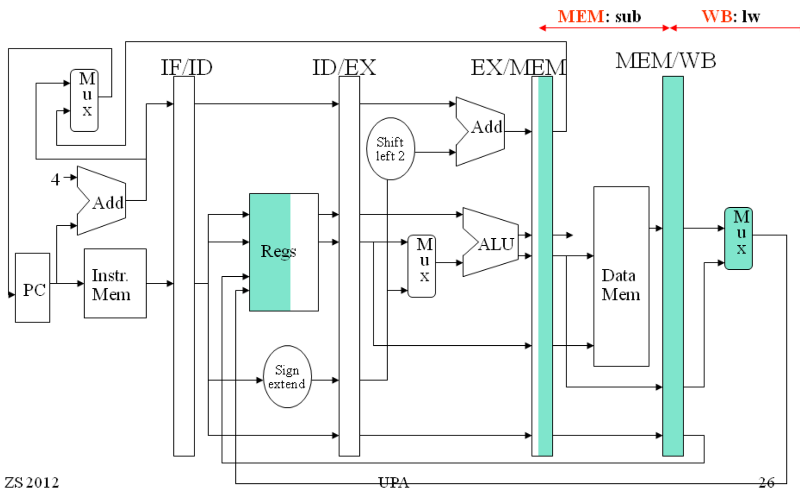
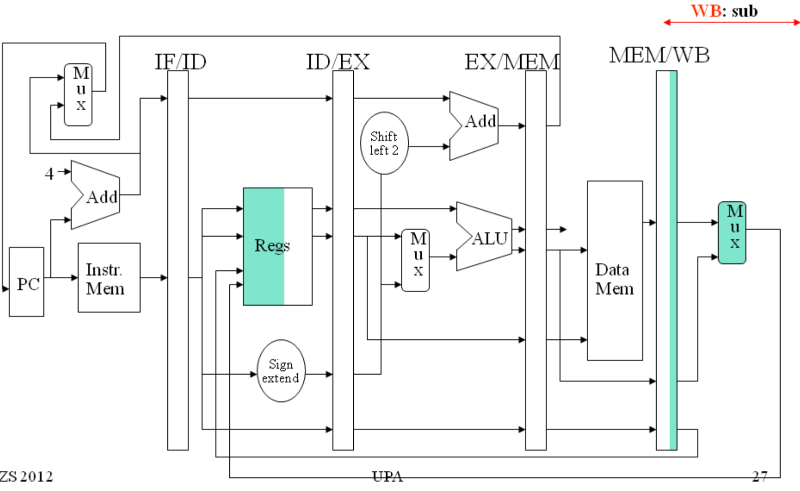
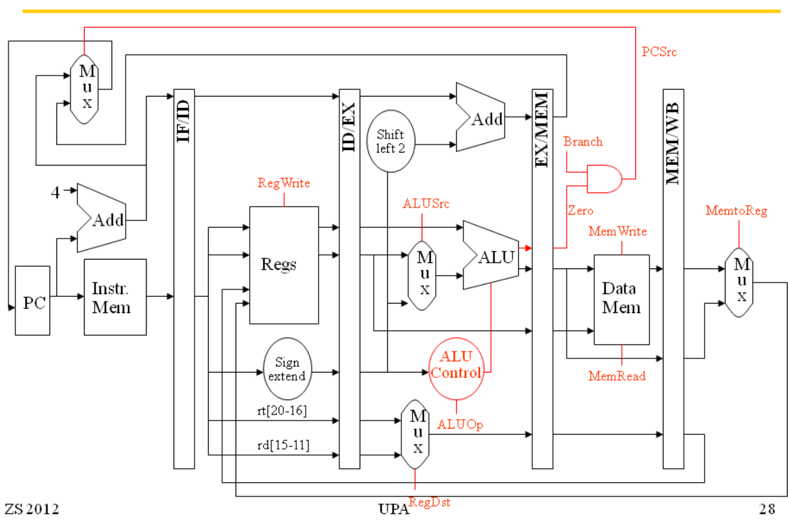


Diagram jednoho cyklu (cykl 6)



Řídící signály v pipeline

Řídící signály



Řídící vstupy ALU v pipeline

Řídící vstupy ALU

Instrukce	Operace ALU	Funkční kód	Akce ALU	Řízení ALU
Lw	00	xxxxxx	Add	010
Sw	00	xxxxxx	Add	010
Beq	01	xxxxxx	Subtract	110
Add	10	100000	Add	010
Sub	10	100010	Subtract	110
And	10	100100	And	000
Or	10	100101	Or	001
Slt	10	101010	Set on less than	111

Řídící linky v pipeline

Řídící linky

Instrukce	Řídící linky prováděcích stupňů				Řídící linky přístupu do paměti			Řídící linky zpětného zápisu	
	Reg Dst	ALU Op1	ALU Op2	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem 2Reg
R-formát	1	1	0	0	0	0	0	1	0
Lw	0	0	0	1	0	1	0	1	1
Sw	x	0	0	1	0	0	1	0	x
Beq	x	0	1	0	1	0	0	0	x

Implementace řízení

Implementace řízení

- Význam 9 řídících linek zůstává beze změny
- Nastavení řídících linek (na definované hodnoty) v každém stupni pro každou instrukci
- Rozšíření pipeline registrů, aby bylo možno zahrnout řídící informaci
- Během IF a ID není třeba nic řídit
- Vytváření řídící informace během ID

Generování/předávání řízení v pipeline

Generování/předávání řízení

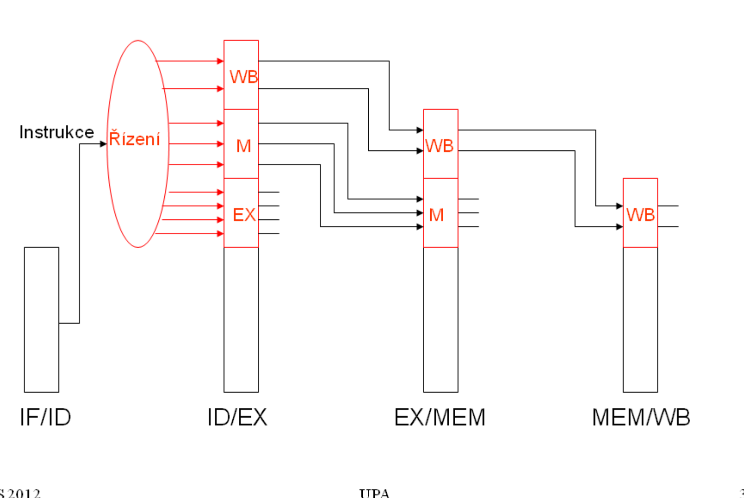
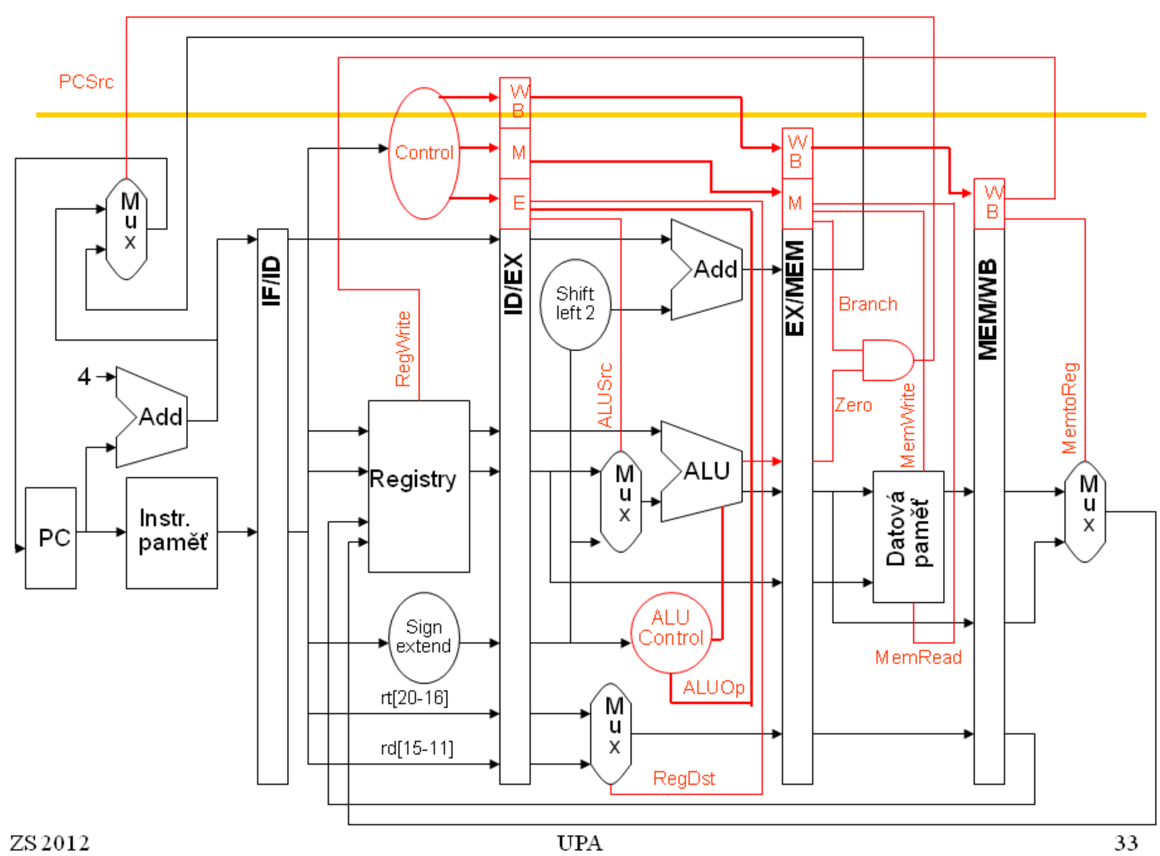


Schéma předávání řízení v pipeline



Příklad posloupnosti instrukcí v pipeline– ukázka funkce pipeline

Příklad posloupnosti instrukcí

Příklad zpracovávané posloupnosti instrukcí:

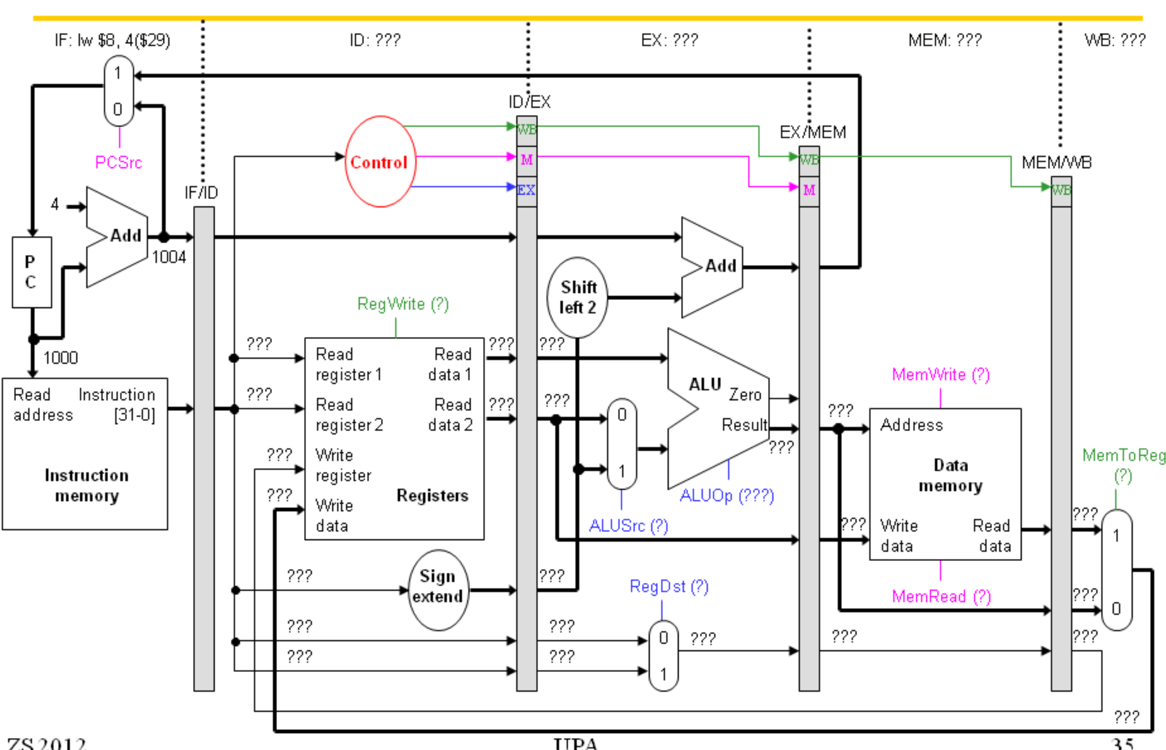
Adresy dekadicky →

```
1000:   lw    $8, 4($29)
1004:   sub   $2, $4, $5
1008:   and   $9, $10, $11
1012:   or    $16, $17, $18
1016:   add   $13, $14, $0
```

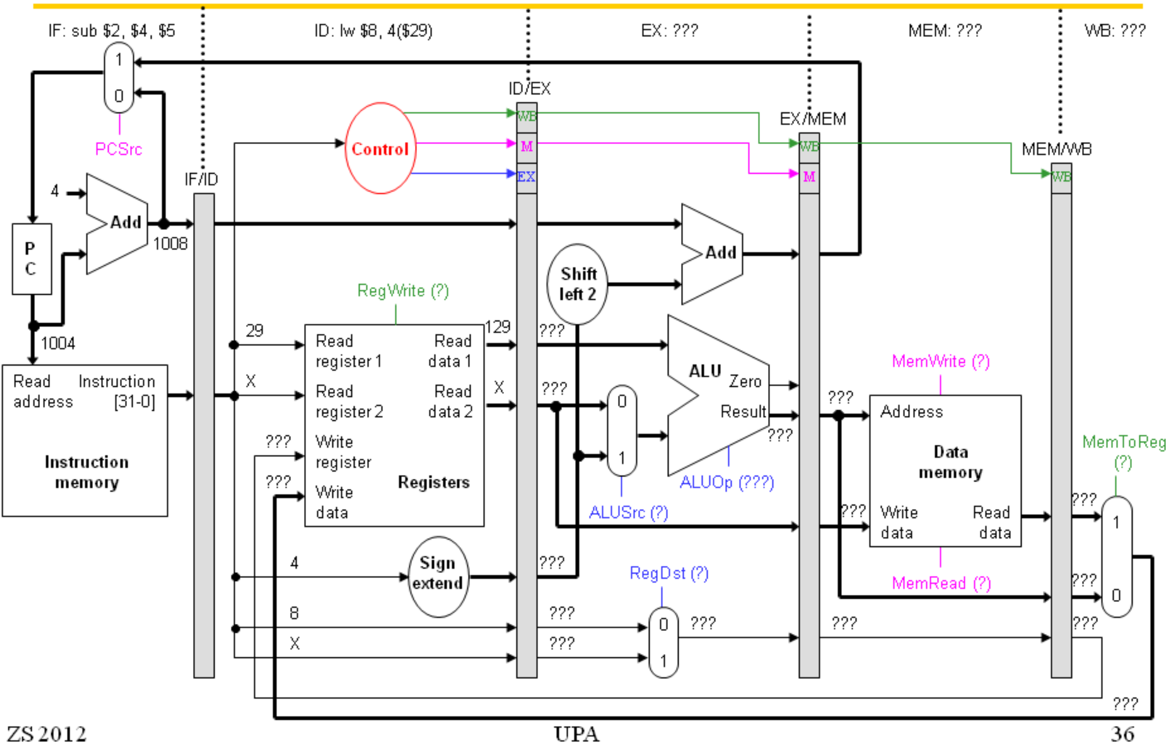
- Pro snazší vyhodnocení uděláme některé předpoklady.
  - Každý registr obsahuje hodnotu, odpovídající adrese plus 100. Např. registr $8 obsahuje 108, registr $29 obsahuje 129, atd.
  - Každá paměťová buňka obsahuje 99.
- Do pipeline diagramů zavedeme následující konvence.
  - X označuje hodnoty, které nejsou důležité,jako např. pole konstanty u instrukcí typu R.
  - Otazníky ??? Označují hodnoty, které neznáme, obvykle ty, které záviseji na předchozích instrukcích v našem příkladu.

9 cyklů ukázka pipeline

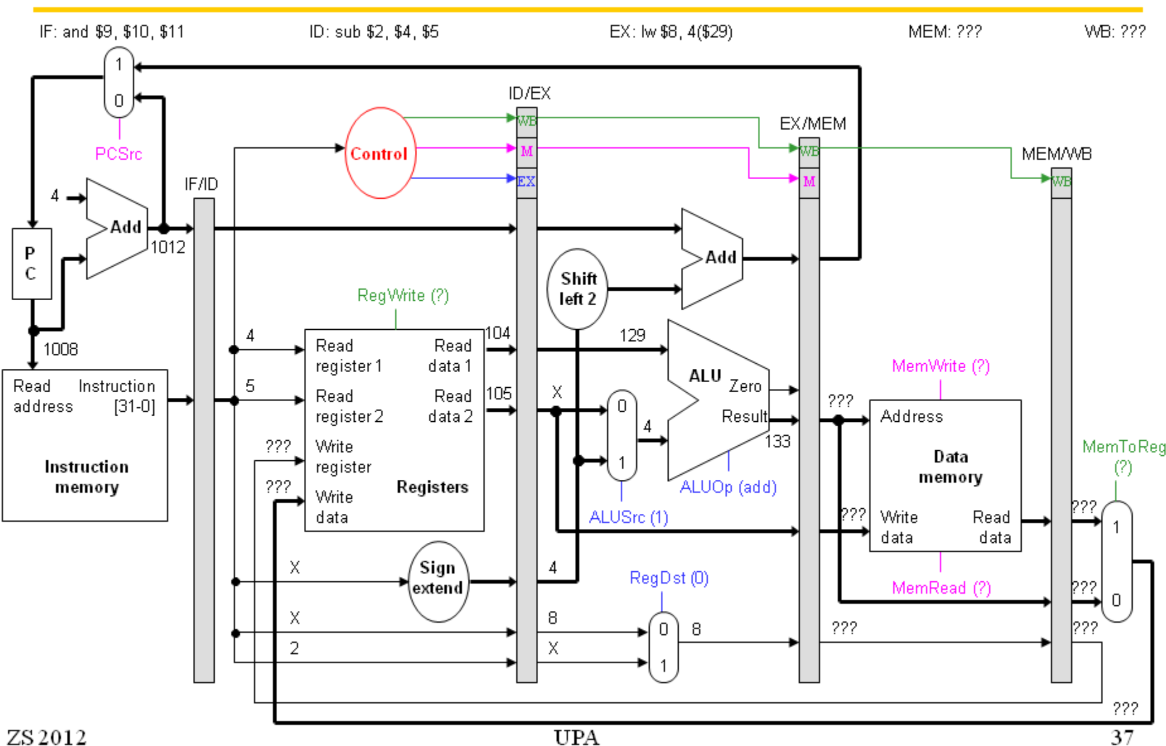
Cykl 1 (plnění)



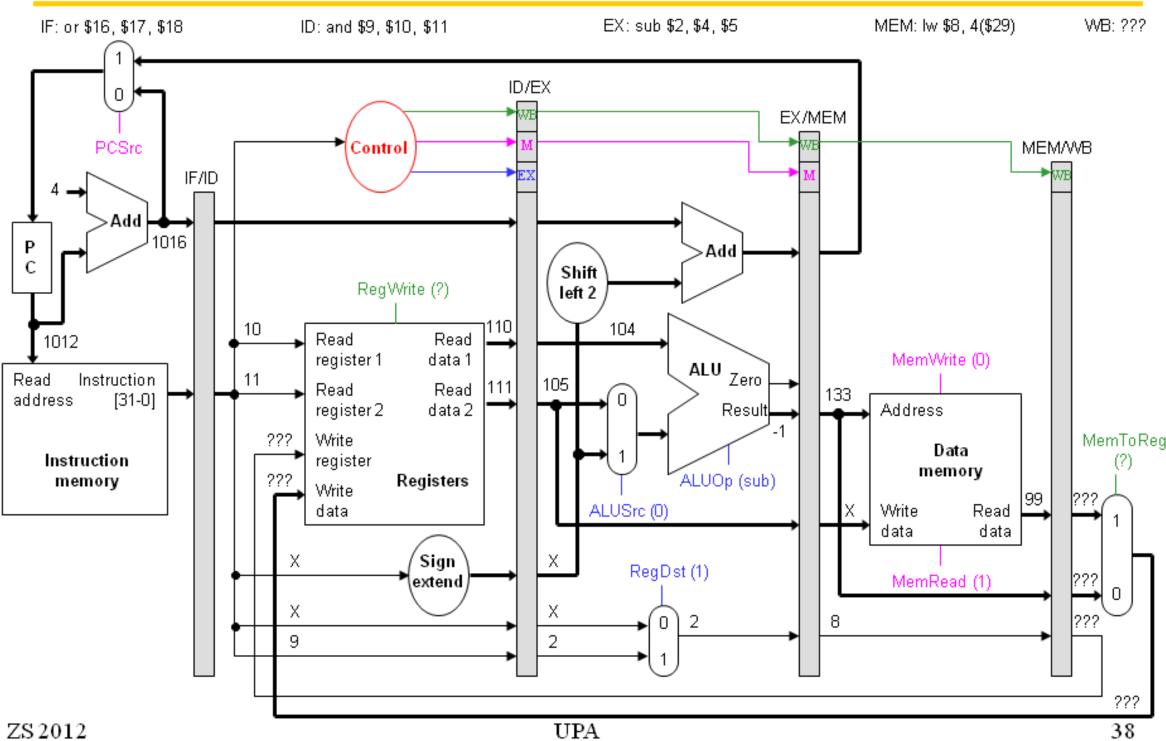
Cykl 2



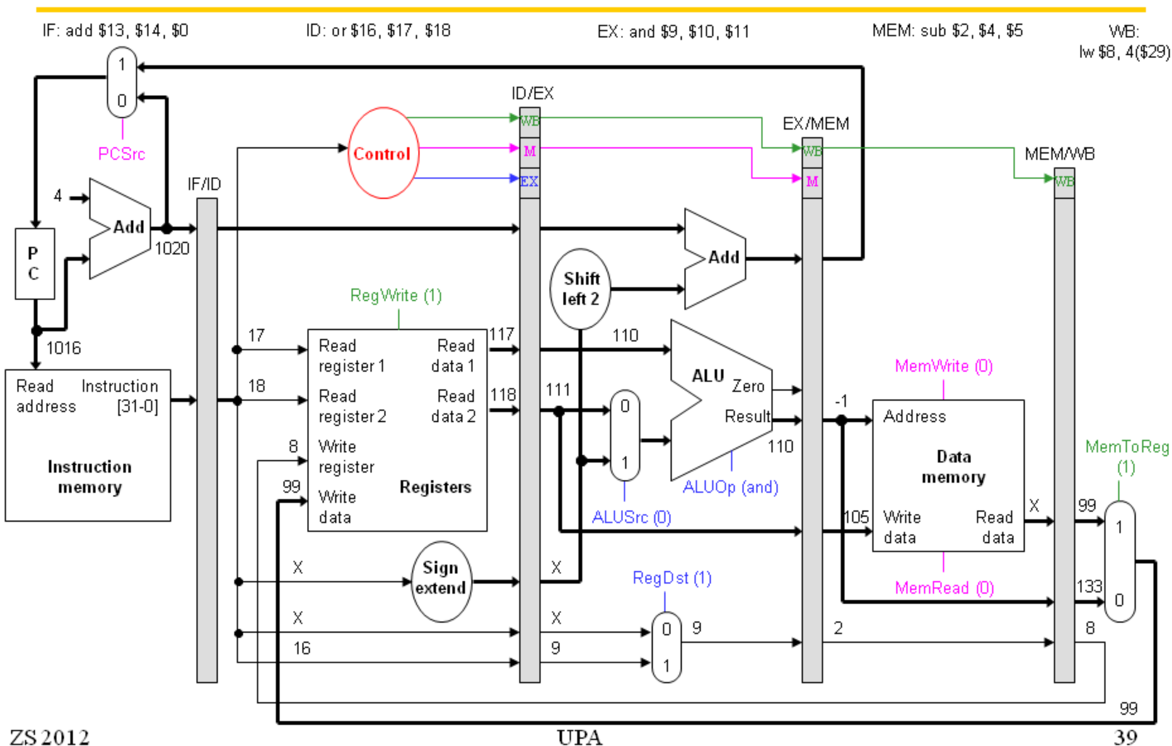
Cykl 3



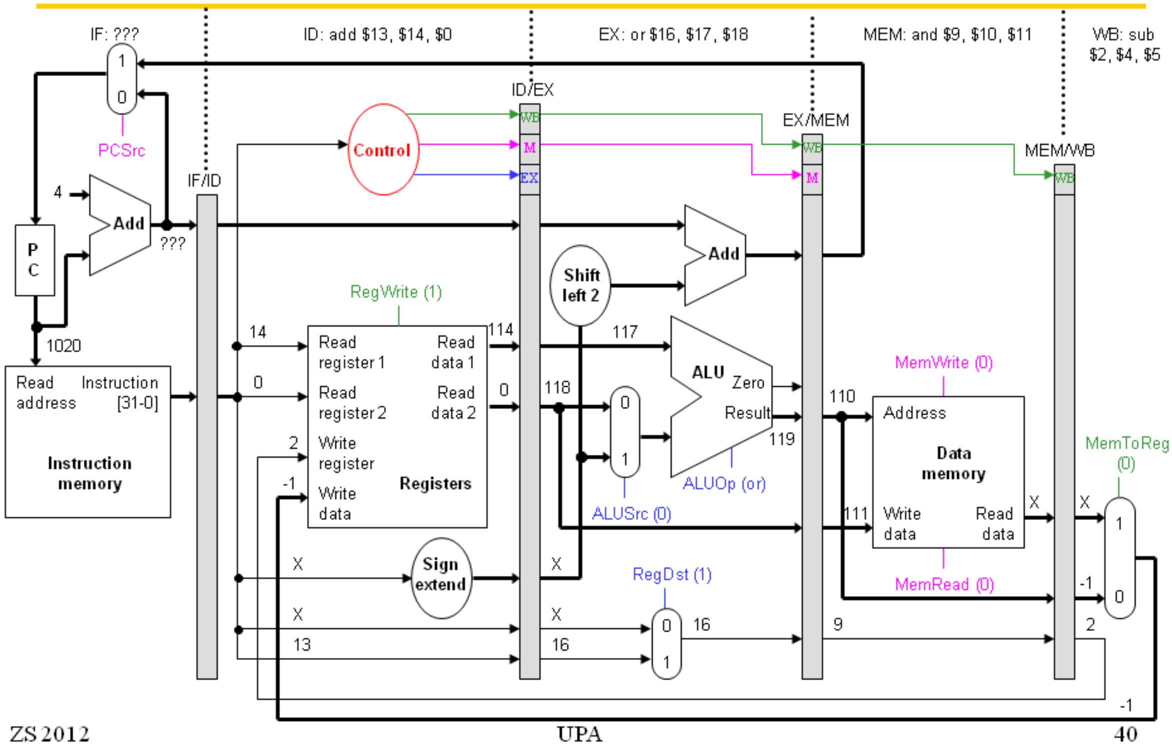
Cykl 4



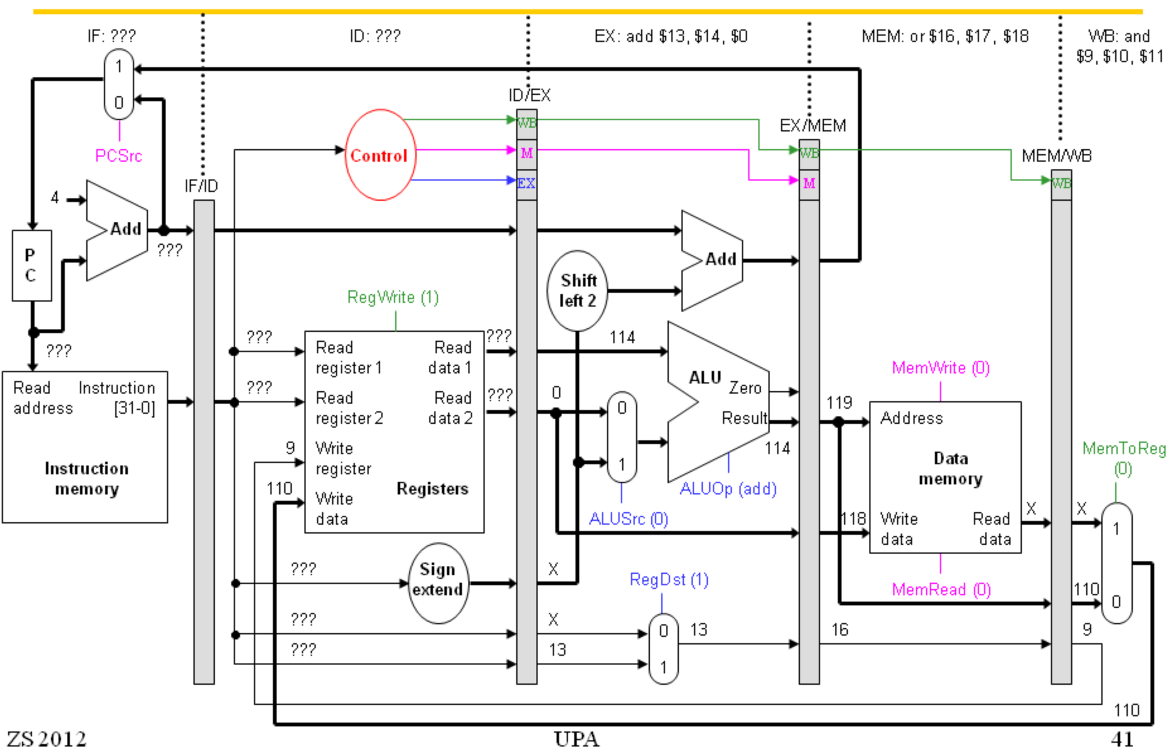
Cykl 5 (zaplněno)



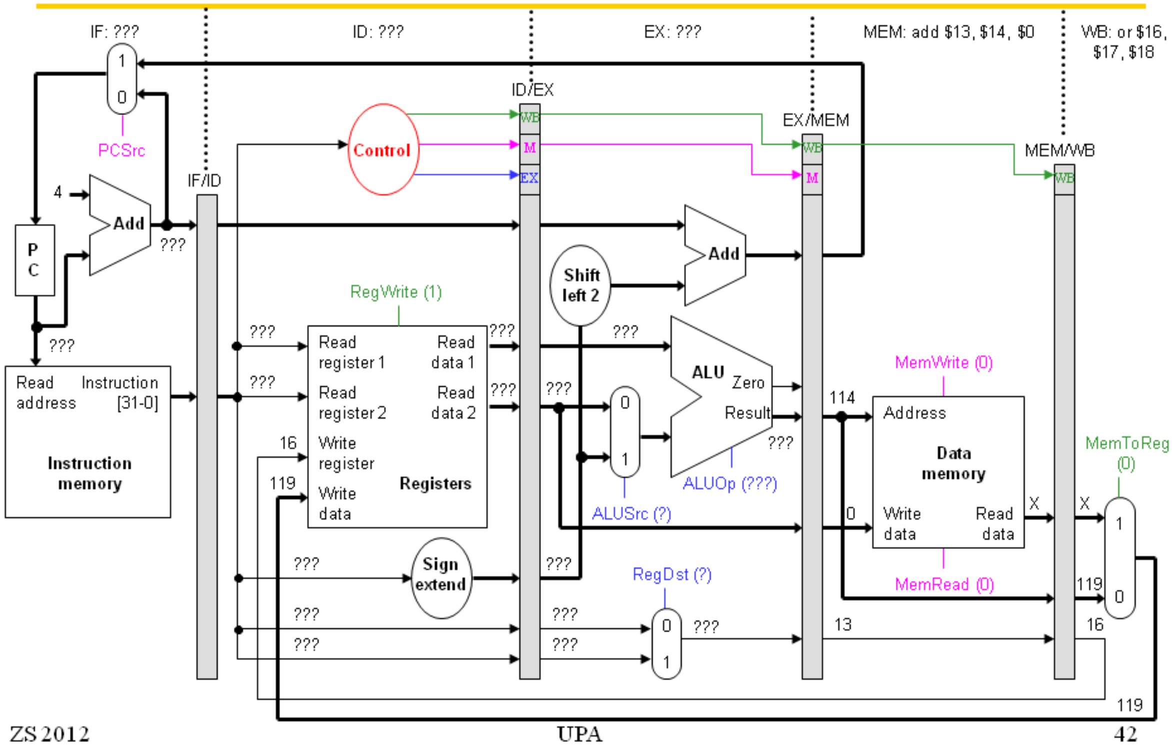
Cykl 6 (vyprazdňování)



Cykl 7



Cykl 8



Cykl 9

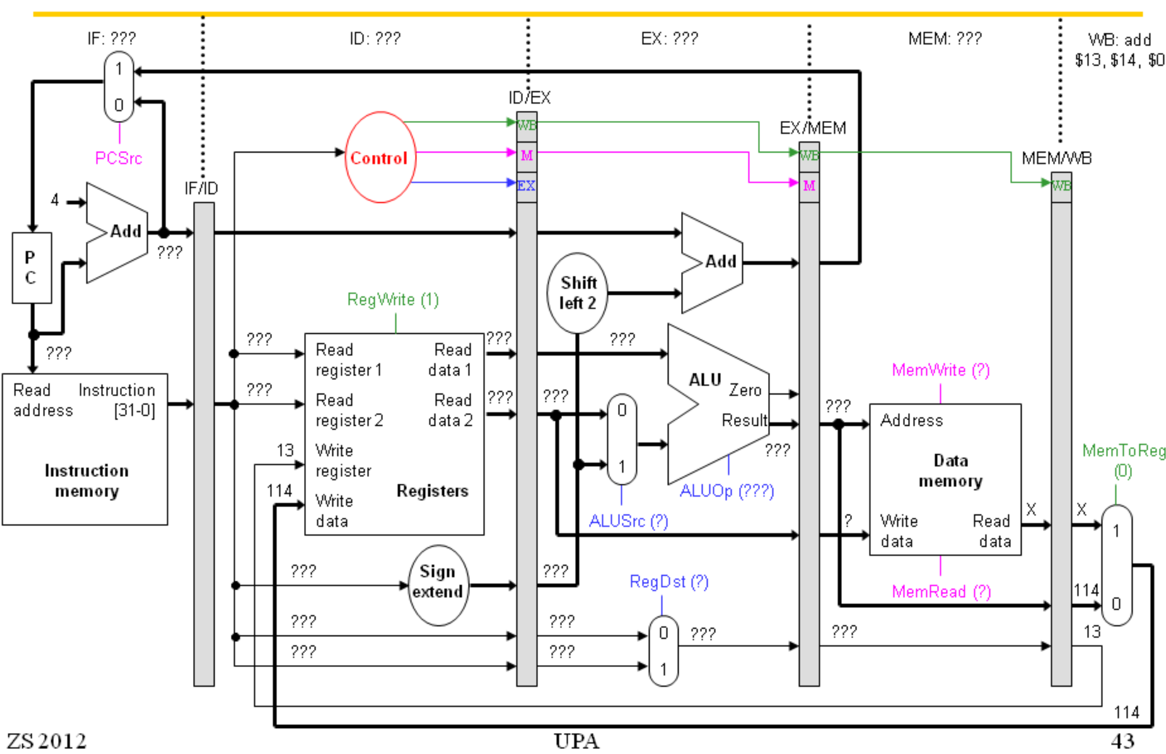


Diagram překrývání v pipeline

Diagram překrývání

Cykly hodin

		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
sub	$v0, $a0, $a1		IF	ID	EX	MEM	WB			
and	$t1, $t2, $t3			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	MEM	WB	
add	$t5, $t6, $0					IF	ID	EX	MEM	WB

Porovnejte devět posledních snímků s diagramem nahoře.

- Lze sledovat překrývání instrukcí.
- Každá funkční jednotka je v každém cyklu využita *jinou* instrukcí.
- Pipeline registry zachytí řízení a data generované v předchozím cyklu pro pozdější použití.
- Po zaplnění pipeline v cyklu 5 jsou všechny jednotky využity. To je ideální situace a také důvod, proč jsou procesory s pipeliningem tak rychlé.

ZS 2012 UPA 44

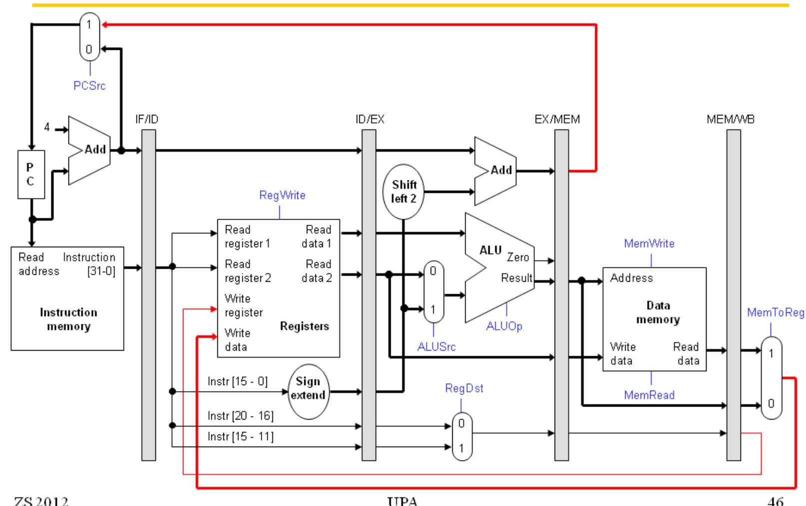
ISA a pipelining – instrukční soubor MIPS pro pipelining, Pipelining se těžko implementuje pro komplexní instrukční soubory

ISA a pipelining

- **Instrukční soubor MIPS byl navržen speciálně pro snadný pipelining.**
  - Všechny instrukce jsou 32-bitů dlouhé, načtení instrukce ve stupni IF znamená jen jeden čtecí cyklus paměti.
  - Pole jsou ve stejné pozici pro různé instrukční formáty—kód operace je prvních šest bitů, rs je dalších pět, atd. Vede to na jednoduchý stupeň ID.
  - MIPS - aritmetické operace se odehrávají v registrech, neobsahují tedy reference do paměti. To pomáhá udržet pipeline krátkou a jednoduchou.
- **Pipelining se těžko implementuje pro komplexní instrukční soubory.**
  - Jestliže různé instrukce mají různou délku nebo formát, budou cykly IF a ID potřebovat extra čas pro určení aktuální délky každé instrukce a polohy polí.
  - Pro instrukce paměť-paměť jsou třeba další stupně pipeline pro výpočet efektivních adres operandů a pro čtecí cykly ještě *před* vlasním provedením ve stupni EX.

ZS 2012 UPA 45

Vše jde zleva doprava s výjimkou …



ZS 2012 UPA 46

Instrukce nezávislé – úvod do hazardu

Naše příklady jsou příliš jednoduché

Příklad posloupnosti instrukcí použitý k ilustraci pipeliningu na předchozí stránce.

```
lw    $8, 4($29)
sub   $2, $4, $5
and   $9, $10, $11
or    $16, $17, $18
add   $13, $14, $0
```

- Instrukce v příkladu jsou nezávislé.
  - Každá instrukce čte a modifikuje úplně jiné registry.
  - Taková posloupnost se snadno ošetřuje.
- Většina posloupností instrukcí v reálných programech ale nejsou nezávislé!

Příklad závislosti instrukcí

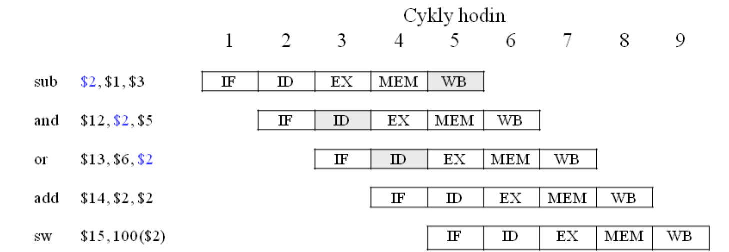
Příklad se závislostmi

```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

- Mezi instrukcemi je několik závislostí.
  - První instrukce SUB ukládá hodnotu do $2.
  - Tento registr je pak použit jako zdroj v dalších instrukcích.
- Pro jednocyklovou jednotku by to nepředstavovalo problém.
  - Každá instrukce je zakončena před započetím další instrukce.
  - To zajistí, že instrukce 2 až 5 použijí nově určenou hodnotu $2 (výsledek odečtení) tak, jak jsme předpokládali.
- Jak se tato posloupnost instrukcí provede v jednotce s pipeliningem ?

Datové hazardy v pipeline diagramu

Datové hazardy v pipeline diagramu



- Instrukce SUB zapisuje do registru $2 až v cyklu 5. To způsobí dva datové hazardy v naší současné jednotce, pracující v režimu pipeline.
  - AND čte registr $2 v cyklu 3. Protože SUB dosud nezapsala do $2, bude přečtena *stará* hodnota $2, nikoliv nová.
  - Podobně instrukce OR používá registr $2 v cyklu 4, přestože dosud nebyl aktualizován instrukcí SUB.

Ukázka kdy problémy nevznikají

Problémy nevznikají

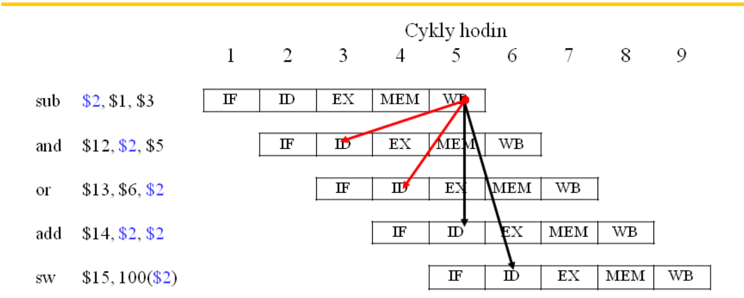
Cykly hodin

		1	2	3	4	5	6	7	8	9
sub	\$2, \$1, \$3	IF	ID	EX	MEM	WB				
and	\$12, \$2, \$5		IF	ID	EX	MEM	WB			
or	\$13, \$6, \$2			IF	ID	EX	MEM	WB		
add	\$14, \$2, \$2				IF	ID	EX	MEM	WB	
sw	\$15, 100(\$2)					IF	ID	EX	MEM	WB

- Instrukce ADD je v pořádku, vzhledem k návrhu registrové sady.
  - Registry jsou zapisovány na počátku hodinového cyklu.
  - Nová hodnota je k dispozici na konci cyklu.
- Instrukce SW nečini problém vůbec, protože čte \$2 až po ukončení SUB.

Závislosti – vektory, datový hazard

Závislosti



- Vektory znázorňuji tok dat mezi instrukcemi.
  - Počátky vektorů ukazuji, kdy je zapisováno do registru \$2.
  - Šipky ukazuji okamžiky, kdy je \$2 čten.
- Každá šipka, která ukazuje zpět v čase představuje datový hazard v pipeline. Hazardy existuji tedy mezi instrukcemi 1 & 2 a 1 & 3.

Hazardy souhrn – datové hazardy, forwarding, změna pořadí instrukcí

Hazardy - souhrn (1/2)

- **Datové hazardy**
  - *Závislosti*: Instrukce je závislá na výsledku předchozí instrukce, která je stále v pipeline
    Add \$s0, \$t0, \$t1
    Sub \$t2, \$s0, \$t3
  - *Pozastavení*: vložení třech bublin (no-ops) do pipeline
  - Řešení: forwarding (zaslání dat také do dalšího stupně)
    - MEM => EX
    - EX => EX
  - Změna pořadí instrukcí, aby se omezilo pozastavování

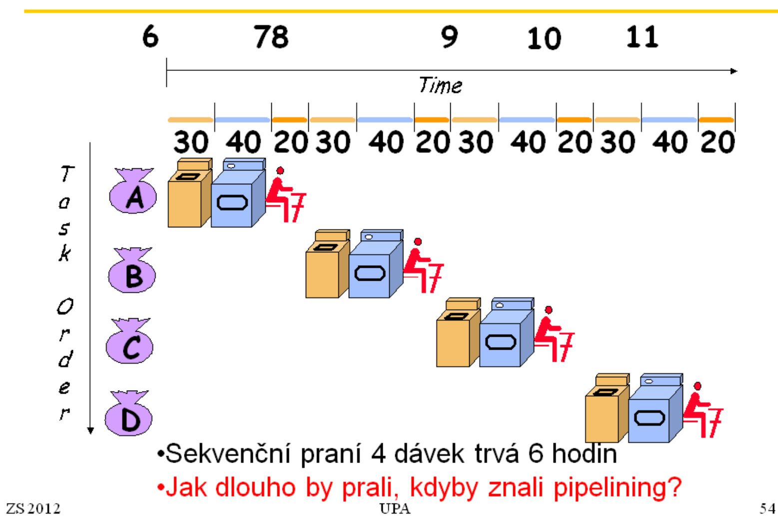
Hazardy souhrn – strukturní hazardy, řídící hazardy, duplikace hardware, opoždění instrukcí větvení

Hazardy - souhrn (2/2)

- **Strukturní**
  - Různé instrukce se snaží používat současně stejné funkční jednotky (např. paměť, registrovou sadu)
  - Řešení: duplikovat hardware
- **Řídící** (větvení)
  - Cílová adresa je známa až na konci třetího cyklu => POZASTAVENÍ
  - Řešení
    - Predikce (statická a dynamická): Smyčky
    - Opoždění instrukcí větvení (branches)

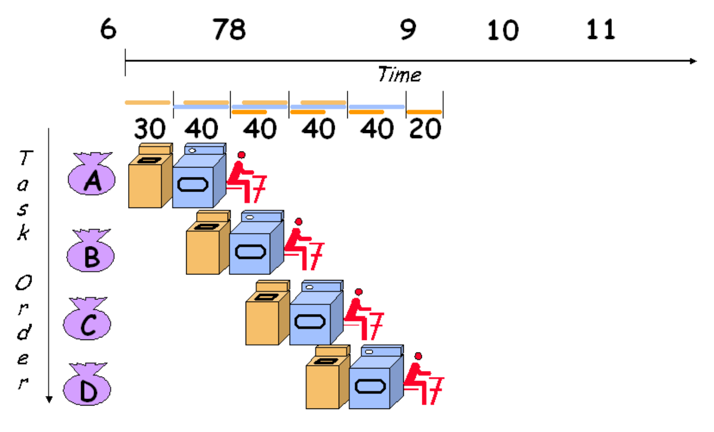
Sekvenční prádelna

Sekvenční prádelna



Prádelna v režimu pipeline

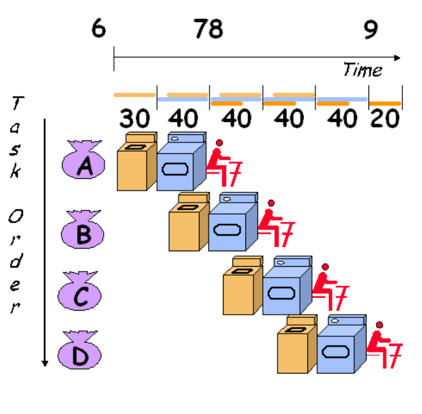
Prádelna v režimu pipeline



•Praní v režimu pipeline trvá 3.5 hodiny pro 4 dávky

Lekce pipeliningu – latence, propustnost, počet stupňů pipepilne, naplnění, doběh

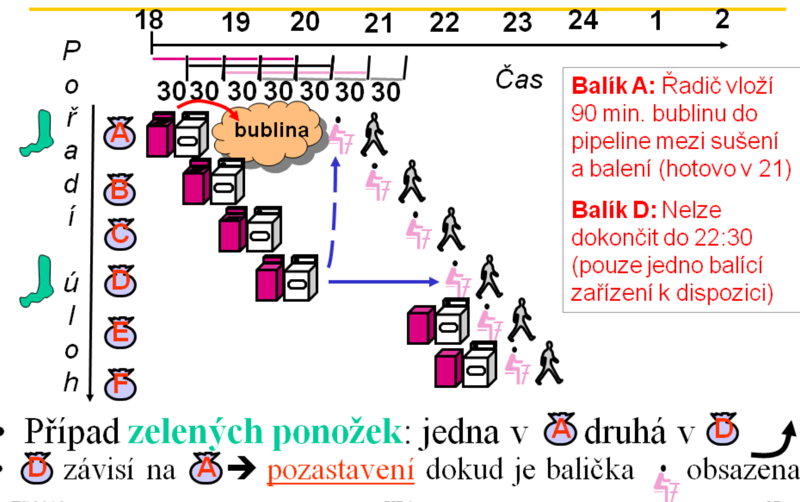
Lekce pipeliningu



- Pipelining nezlepšuje latenci jednotlivých úloh, ale propustnost celé dávky
- Rychlost je omezena nejpomalejším stupněm
- Více úloh se zpracovává paralelně
- Potenciální urychlení = počtu stupňů pipeline
- Nevyvážená délka stupnů redukuje rychlost
- Čas pro naplnění a doběh pipeline také redukuje rychlost

Hazardy v pipeline – příklad, prádelna, bublina, pozastavení

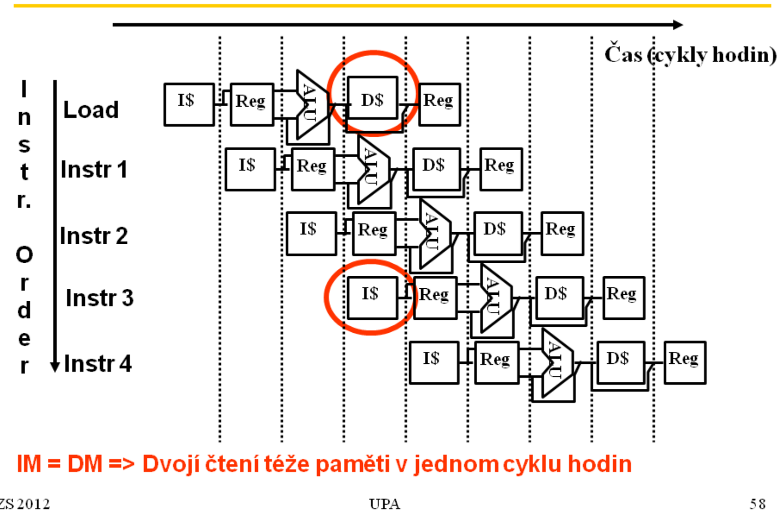
Hazardy v pipeline (příklad)



- Případ zelených ponožek: jedna v A druhá v D
- D závisí na A → pozastavení dokud je balička obsazena

Strukturní hazard 1 – jediná paměť, diagram, dvojí čtení téže paměti v jednom cyklu hodin

Strukturní hazard 1: Jediná paměť



IM = DM => Dvojí čtení téže paměti v jednom cyklu hodin

Strukturní hazard 2 registrová sada, současný zápis a čtení z/do sady registrů

Strukturní hazard 2: Registrová sada

Čas (cykly hodin)

Instr. Order

Load: I$ | Reg | ALU | D$ | Reg

Instr 1: I$ | Reg | ALU | D$ | Reg

Instr 2: I$ | Reg | ALU | D$ | Reg

Instr 3: I$ | Reg | ALU | D$ | Reg

Instr 4: I$ | Reg | ALU | D$ | Reg

Současný zápis a čtení do/ze sady registrů

Strukturní hazard řešení – dvě cache paměti úrovně jedna (L1), strukturní hazard registrový soubor

Strukturní hazardy: Řešení

- Strukturní hazard 1: Jediná paměť
  - Dvě paměti? neproveditelné a neefektivní
    => Dvě cache paměti úrovně 1 (instrukce a data)
- Strukturní hazard 2: Registrový soubor
  - Přístup do registru trvá méně než ½ času, který potřebuje stupeň ALU
    => Používají se následující konvence:
    - Write vždy během první poloviny každého cyklu
    - Read vždy během druhé poloviny každého cyklu
  - Obojí, Read i Write lze provést během téhož cyklu hodin (s malým zpožděním mezi)

Hazard řízení – větvení, nejvhodnější provádění skoků

Hazard řízení: Instrukce větvení (1/2)

- Hardware ve stupni ALU, podporující větvení
  - *Vždy* se načtou dvě další instrukce ležící za skokem, ať se skok koná nebo nikoliv
- Jaké by bylo nejvhodnější provádění skoků:
  - jestliže se skok nebude konat, neztrácet čas a pokračovat normálně ve výpočtu
  - jestliže se skok koná, neprovádět žádnou instrukci za skokem a jít rovnou na dané návěští (adresu)

Hazard řízení – instrukce větvení – výchozí řešení, lepší řešení, přesunutí komparátoru

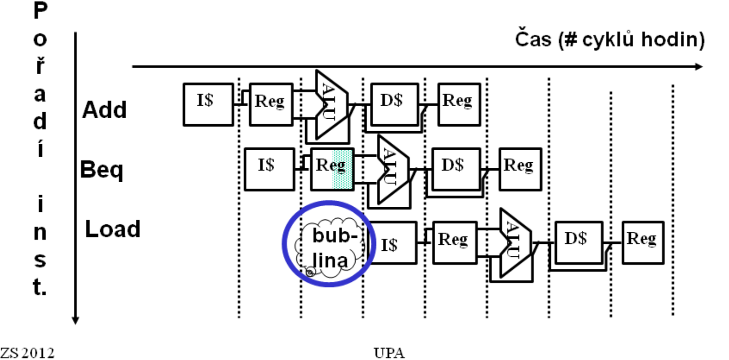
Hazard řízení : Instrukce větvení (2/2)

- **Výchozí řešení**: Zastavení dokud není rozhodnuto
  - Vložení "nop" instrukcí: takové, co nic nedělají, jen spotřebují čas
  - Nevýhoda: větvení spotřebují 3 cykly hodin (každé) (předpokládáme, že se komparátor nachází ve stupni ALU)
- **Lepší řešení**: Přesunout komparátor do stupně 2
  - Výhoda: jelikož větvení je dokončeno ve stupni 2, je načtena jenom jedna „zbytečná" instrukce
  - Je třeba jen jedna instrukce „nop"
  - To znamená, že větvení jsou neaktivní ve stupních 3, 4 a 5.

Hazard řízení – lepší řešení schéma

Hazard řízení: Lepší řešení

- Přesun komparátoru do stupně 2
- Výhoda: protože větvení je dokončeno ve stupni 2, je načtena jen jedna prázdná instrukce, je zapotřebí jeden „nop"
- To znamená, že větvení jsou neaktivní ve stupních 3, 4 a 5



Nejlepší řešení hazard řízení – zpožděná větvení, branch delay slot, worst-case

Nejlepší: Zpožděná větvení (1/2)

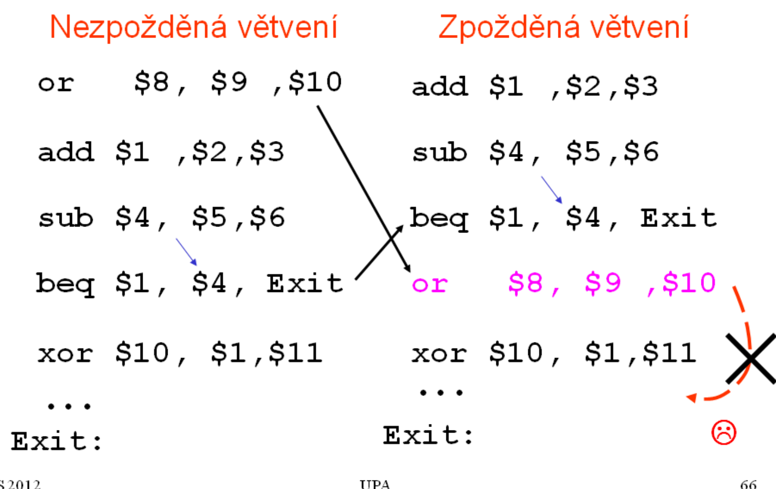
- Provedeme-li instrukci větvení, žádná z instrukcí, která leží za skokem není provedena náhodně.
- Nová definice: Ať se větvení koná nebo nikoliv, instrukce ležící bezprostředně za skokem se vždy provede (nazývá se branch-delay slot)

Nejlepší: Zpožděná větvení (2/2)

- Poznámky k technice Branch-Delay Slot
  - Scénář Worst-Case : vždy lze vložit „nop“
  - Lepší případ: Lze najít instrukci, která se umístí do branch-delay slotu, aniž by se ovlivnil chod programu
    - Přeskupení instrukcí je technika, vedoucí ke zvýšení výkonu – realizována v kompilátoru
    - Kompilátor musí být optimalizovaný tak, aby nalezl vhodnou instrukci, kterou lze přesunout
    - Obvykle lze takovou instrukci nalézt ve více než 50% případů

Nezpožděná vs. Zpožděná větvení

Nezpožděná vs. zpožděná



Pipelining myšlenka, reprezentace, není jednoduchý, výhody nevýhody viz dále

Závěr (1/3)

- Pipelining je významná myšlenka: Často používaná koncepce
- Pipelining není jednoduchá záležitost
  - Používání různých zdrojů v jednom cyklu *na instrukci*
  - Složitější než multicyklové jednotky
  - Násobná reprezentace
- Pipeline obsahuje reprezentaci
  - Jednocyklových jednotek: vhodné
  - Jednoduchých i multicyklových diagramů

Pipelining zvyšuje efektivitu, řízení jednotky složité, forwarding, detekce hazardů

Závěr (2/3)

- Pipelining zlepšuje efektivitu tím, že:
  - Zavádí regulární formáty => podporuje jednoduchost
  - Rozkládá každou instrukci na kroky
  - V každém kroku se provádí srovnatelné množství „práce“
  - Pipeline je téměř pořád zaplněná (obsazena) tak, aby se *maximalizovala propustnost procesoru*
- Řízení jednotky v režimu pipeline je složité
  - Forwarding
  - Detekce hazardů a jejich omezení
- Návrh řízení pipeline jednotky a operací

Optimální pipeline, podmínky pro úspěšnou činnost, co snižuje dokonalost

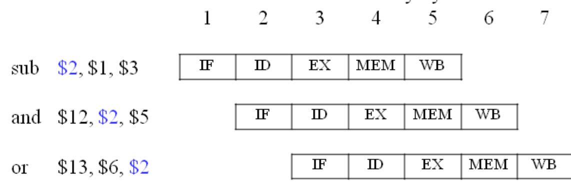
Závěr (3/3)

- Optimální pipeline
  - Každý stupeň provádí část instrukčního cyklu.
  - Během každého cyklu je dokončena jedna instrukce.
  - V průměru probíhá výpočet mnohem rychleji
- Jaké jsou podmínky pro úspěšnou činnost?
  - Podobnost mezi jednotlivými instrukcemi.
  - Každý stupeň vyžaduje přibližně stejný čas pro práci jako ostatní.
- Co snižuje její dokonalost?
  - Strukturní hazardy: Potřeba HW zdrojů
  - Hazardy řízení: Opožděná větvení
  - Datové hazardy: Instrukce závisí na předchozí instrukci

Detailní pohled na pipeline, datové hazardy, forwarding

Detailní pohled na pipeline

- Budeme se zabývat problémy, které způsobují datové hazardy v procesoru s pipeliningem a jak je eliminovat metodou, která se nazývá forwarding.

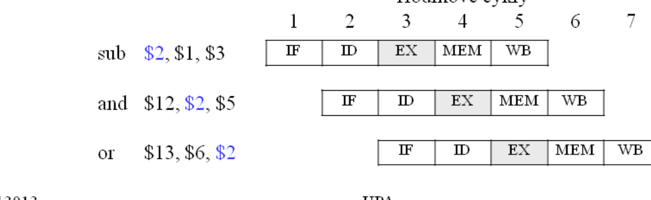


- Odstraníme hazardy tak, aby instrukce AND a OR v našem příkladu používaly korektní hodnoty pro registr $2.
- Kdy data aktuálně vznikají a kdy se „konzumují“?
- Jaká opatření můžeme uplatnit?

Bypass registrového souboru

„Bypass“ registrového souboru

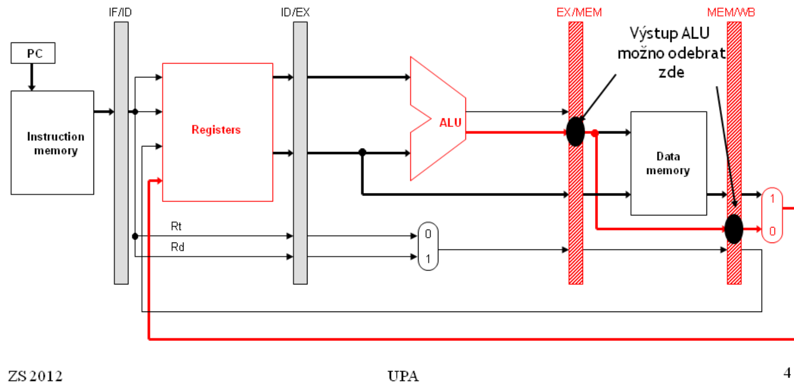
- Aktuální výsledek $1 - $3 je vypočítán v cyklu 3 *předtím*, než je použit v cyklech 4 a 5.
- Kdybychom mohli „vynechat“ stupeň zpětného zápisu a stupně čtení registrů když je třeba, hazardy by nevznikly.
  - Zaměříme se na hazardy provázející aritmetické instrukce.
  - Zároveň se budeme zabývat problémy s instrukcí lw.
- Podstatné, je třeba přivést výstup ALU instrukce SUB přímo k instrukcím AND a OR, aniž by výsledek procházel registrovou sadou.



Kde odebrat výsledek z ALU

Kde odebrat výsledek ALU

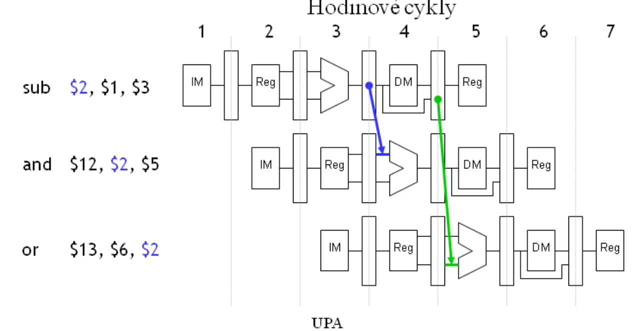
- Výsledek ALU, generovaný ve stupni EX, prochází obvykle pipeline registry do stupňů MEM a WB, nakonec je pak zapsán do registrové sady.
- Následující obrázek znázorňuje zjednodušený diagram jednotky s pipeliningem.



Forwarding – ukázka, schéma

Forwarding

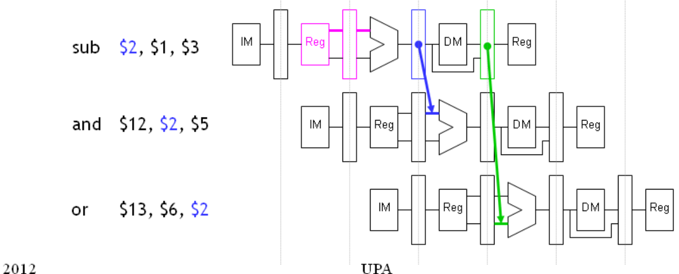
- Protože pipeline registry již obsahují výsledek ALU, přesuneme jej přímo (forward) do následující instrukce, abychom zabránili datovým hazardům.
  - V hodinovém cyklu 4 dostane instrukce AND hodnotu $1 - $3 z pipeline registru EX/MEM, použitém při odečtení (sub).
  - Potom v cyklu 5 instrukce OR dostane tentýž výsledek z pipeline registru MEM/WB, využitém při zpracování instrukce SUB.



Návrh hardware pro forwarding, ForwardA, ForwardB

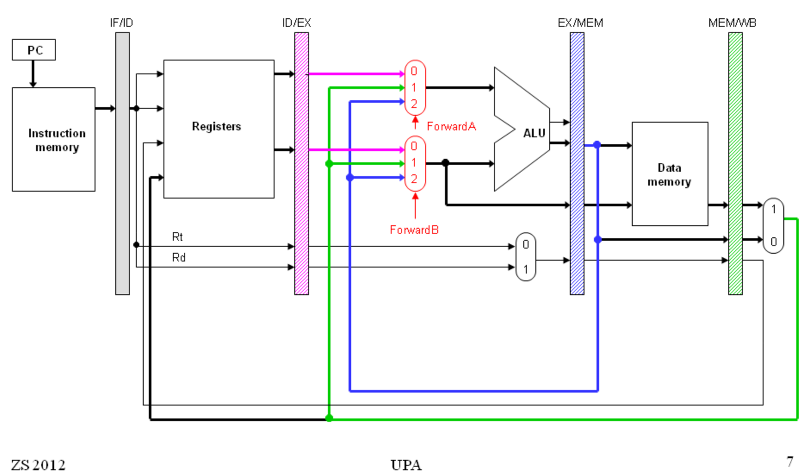
Návrh hardware pro forwarding

- Jednotka pro forwarding vybírá korektní vstupy ALU pro stupeň EX.
  - Nevznikají-li hazardy, jsou operandy ALU přisunuty z registrového souboru, podobně jako tomu bylo předtím.
  - Vzniká-li hazard, jsou operandy odebrány buď z pipeline registrů EX/MEM nebo MEM/WB.
- Zdrojové operandy ALU jsou vybrány pomocí dvou nových multiplexerů s řídícími signály ForwardA a ForwardB.



Zjednodušené datové cesty s multiplexery pro forwarding - schéma

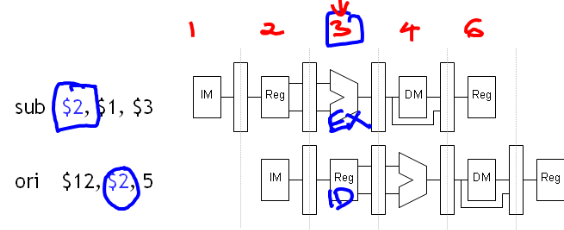
Zjednodušené datové cesty s multiplexery pro forwarding



Detekce hazardů EX/MEM

Detekce hazardů EX/MEM

- Sekce: detekuje hazard mezi instrukcemi typu R a I

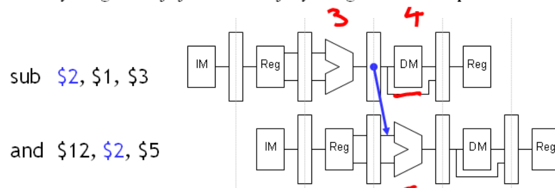


- Nemůže detekovat hazard do cyklu 3: `sub` je ve stupni EX, `ori` je v ID
- Vzniká hazard, protože: ID/EX.RegisterRd = IF/ID.RegisterRs
- V terminologii MP 5 (Java): `instr[EX].rd() == instr[ID].rs()`

Eliminace datových hazardů EX/MEM

Eliminace datových hazardů EX/MEM

- Kdy se potřebujeme dozvědět, že vznikne hazard?
- Jak lze hardwarově rozpoznat vznik hazardu?
- Hazard EX/MEM nastává ve stupni EX mezi po sobě jdoucími instrukcemi jestliže:
  1. Předchozí instrukce zapisuje do registrového souboru *a současně*
  2. Cílovým registrem je jeden ze zdrojových registrů ALU stupně EX.



- Data v pipeline registru lze označovat syntaxí podobnou té, která se používá v objektovém programování. Například ID/EX.RegisterRt znamená odkaz na pole rt, uložené v pipeline ID/EX.

Rovnice datových hazardů EX/MEM, RegisrerRd, RegisterRs, RegisterRt

Rovnice datových hazardů EX/MEM

- Prvý zdrojový operand ALU přichází z pipeline registru, když je to zapotřebí.

```
if (EX/MEM.RegWrite = 1
    and EX/MEM.RegisterRd = ID/EX.RegisterRs)
then ForwardA = 2
```

- Podobně je tomu i v případě druhého operandu.

```
if (EX/MEM.RegWrite = 1
    and EX/MEM.RegisterRd = ID/EX.RegisterRt)
then ForwardB = 2
```

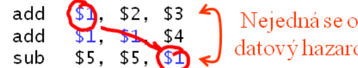
sub $2, $1, $3

and $12, $2, $5

Datové hazardy MEM/WB

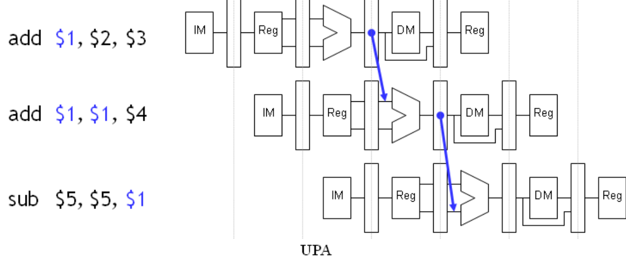
Datové hazardy MEM/WB

- Hazard MEM/WB se může objevit mezi instrukcí ve stupni EX a instrukcí z předchozích dvou cyklů.
- Nový problém vzniká, je-li registr aktualizován dvakrát v jedné řádce.



Nejedná se o datový hazard

- Registr $1 je zapisován *oběma* předchozími instrukcemi, ale jen poslední výsledek (druhá instrukce ADD) má být „forwardován“.



Rovnice datových hazardů MEM/WB, RegisrerRd, RegisterRs, RegisterRt

Rovnice datových hazardů MEM/WB

- Rovnice pro detekci a ošetření MEM/WB hazard prvého operandu ALU.

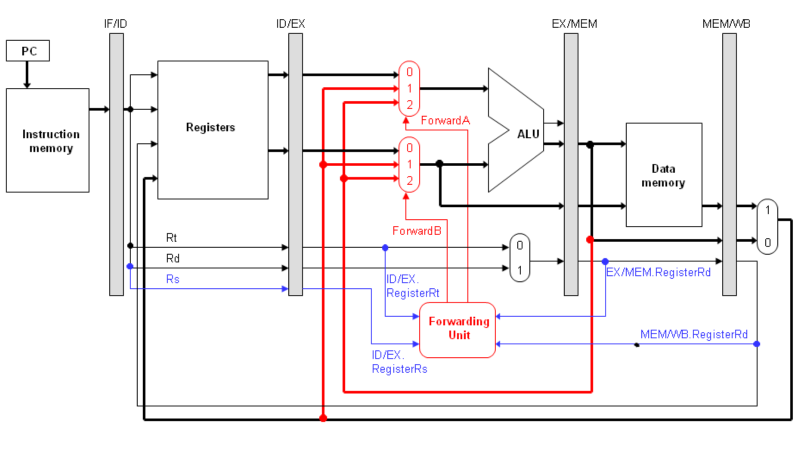
  if (MEM/WB.RegWrite = 1
  and MEM/WB.RegisterRd = ID/EX.RegisterRs
  and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs or EX/MEM.RegWrite = 0)
  then ForwardA = 1

- Druhý operand ALU je ošetřen podobně.

  if (MEM/WB.RegWrite = 1
  and MEM/WB.RegisterRd = ID/EX.RegisterRt
  and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt or EX/MEM.RegWrite = 0)
  then ForwardB = 1

Zjednodušená jednotka včetně forwardingu – schéma

Zjednodušená jednotka včetně forwardingu



Jednotka pro forwarding, RegisrerRd, RegisterRs

Jednotka pro forwarding

- Jednotka pro forwarding má řadu vstupních a výstupních-řídicích signálů.

  ID/EX.RegisterRs EX/MEM.RegisterRd
  MEM/WB.RegisterRd
  ID/EX.RegisterRt EX/MEM.RegWrite
  MEM/WB.RegWrite

  (Dva signály RegWrite nejsou v diagramu zachyceny, pocházejí z řídící jednotky.)
- Výstupy jednotky pro fowarding jsou výběrové signály multiplexerů ForwardA a ForwardB, připojených k ALU. Tyto výstupy jsou odvozeny podle rovnic na předchozích snímcích.
- K novým multiplexerům vedou také nové datové cesty.

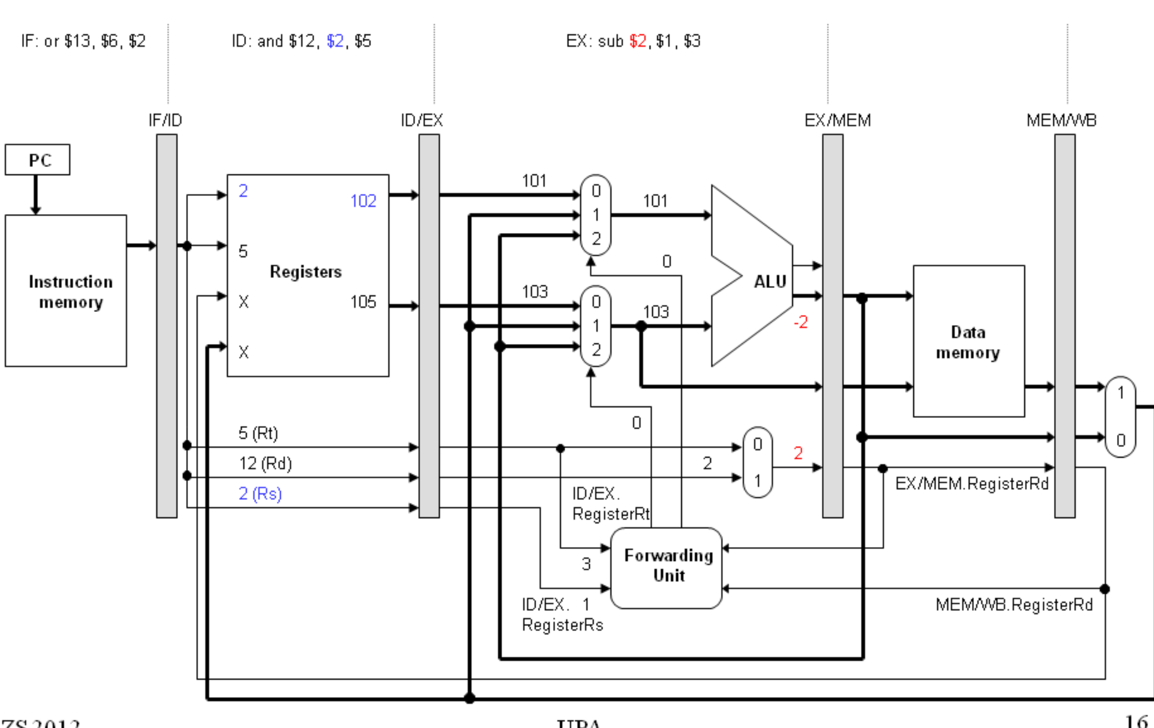
Příklad forwardingu v krocích

Příklad

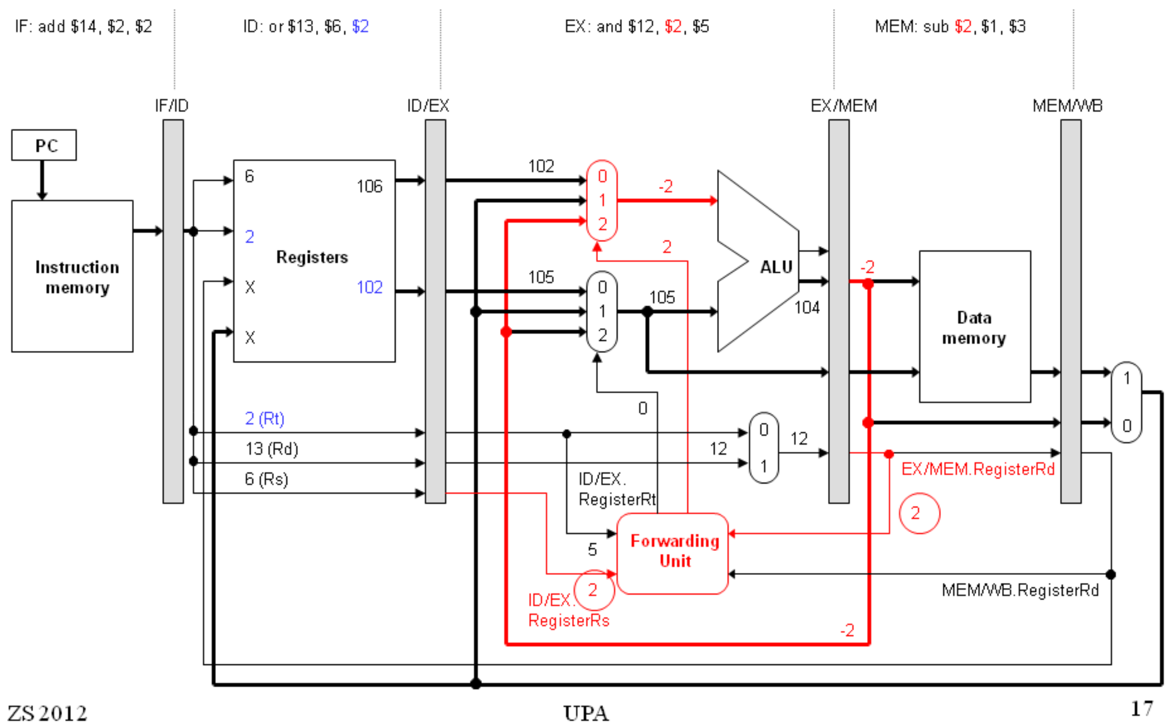
```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

- Předpokládejme, že každý registr obsahuje svoje číslo plus 100.
  - Po provedení prvé instrukce $2 by měl obsahovat −2 (101 − 103).
  - Ostatní instrukce použijí −2 jako jeden z operandů.
- Příklad bude popsán stručně.
  - Předpokládejme, že není třeba žádný forwarding, vyjma registr $2.
  - Přeskočíme první dva cykly, protože jsou shodné jako předtím.

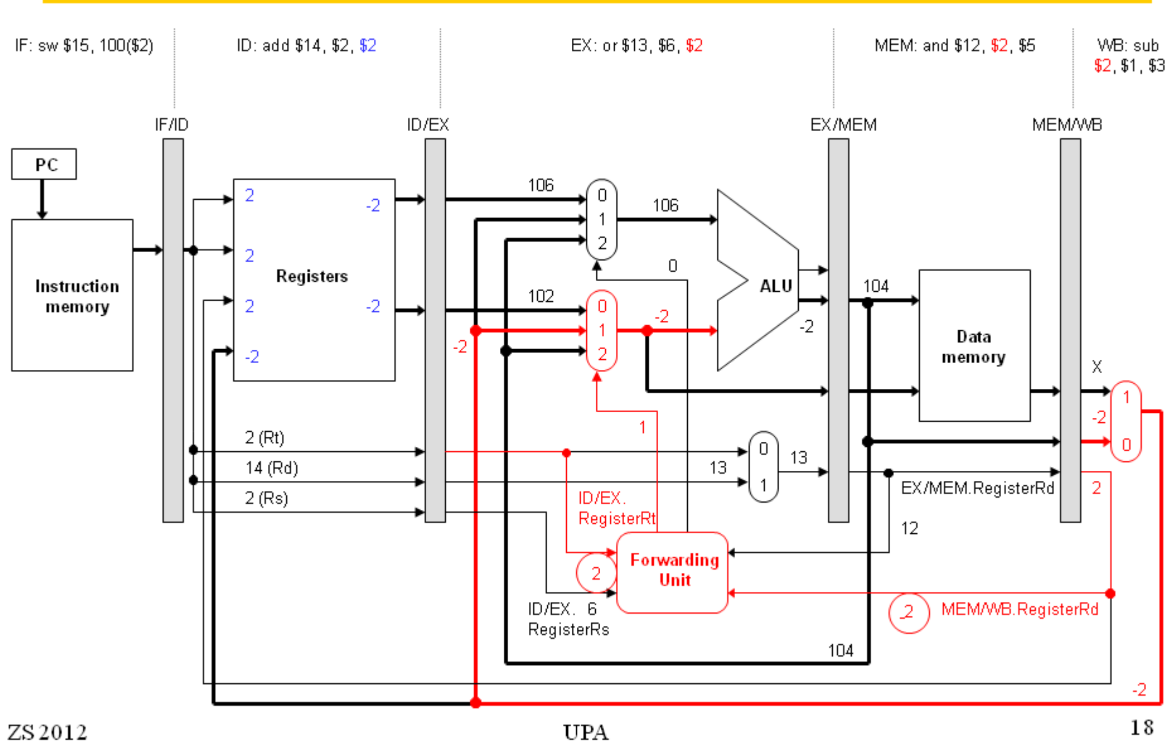
Hodinový cykl 3



Hodinový cykl 4: odběr $2 z EX/MEM



Hodinový cykl 5: odběr $2 z MEM/WB



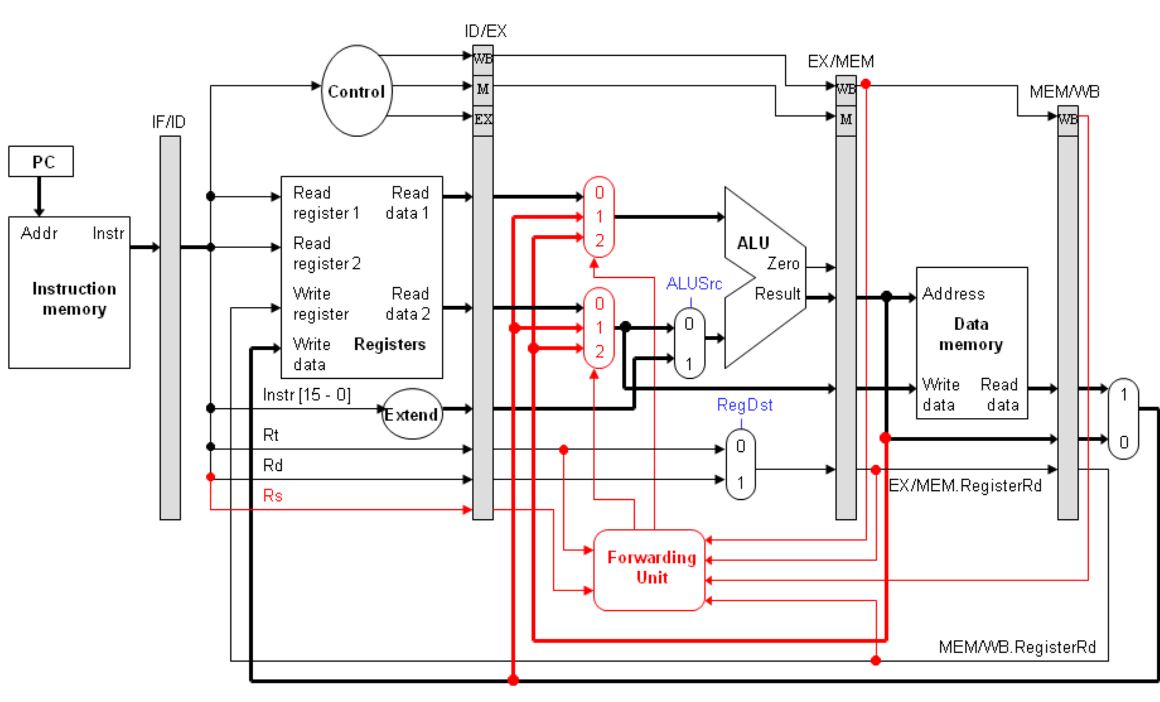
Vzniká řada hazardů

Vzniká řada hazardů

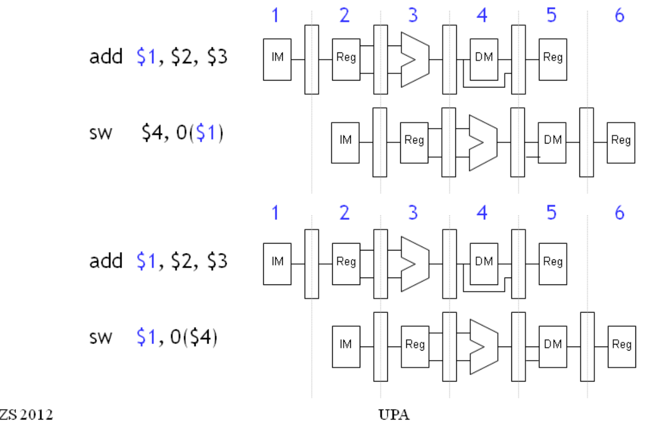
- První datový hazard nastává během cyklu 4.
  - Jednotka pro forwarding zjistí, že prvý zdrojový registr ALU pro AND je zároveň cílem instrukce SUB.
  - Správná hodnota se „forwarduje“ z registru EX/MEM. K použití nekorektní staré hodnoty v registrovém souboru nedojde.
- Druhý hazard nastává během hodinového cyklu 5.
  - Druhý zdroj ALU (pro OR) je zároveň cílem pro SUB.
  - Tady je třeba na rozdíl od předchozího případu „forwardovat“ obsah pipeline registru MEM/WB.
- Další hazardy ve spojitosti s instrukcí SUB nevznikají.
  - Během cyklu 5 zapisuje instrukce SUB výsledek zpět do registru $2.
  - Instrukce ADD může číst novou hodnotu z registrového souboru ve stejném cyklu.

Úplné datové cesty s pipeliningem – schéma, forwarding unit

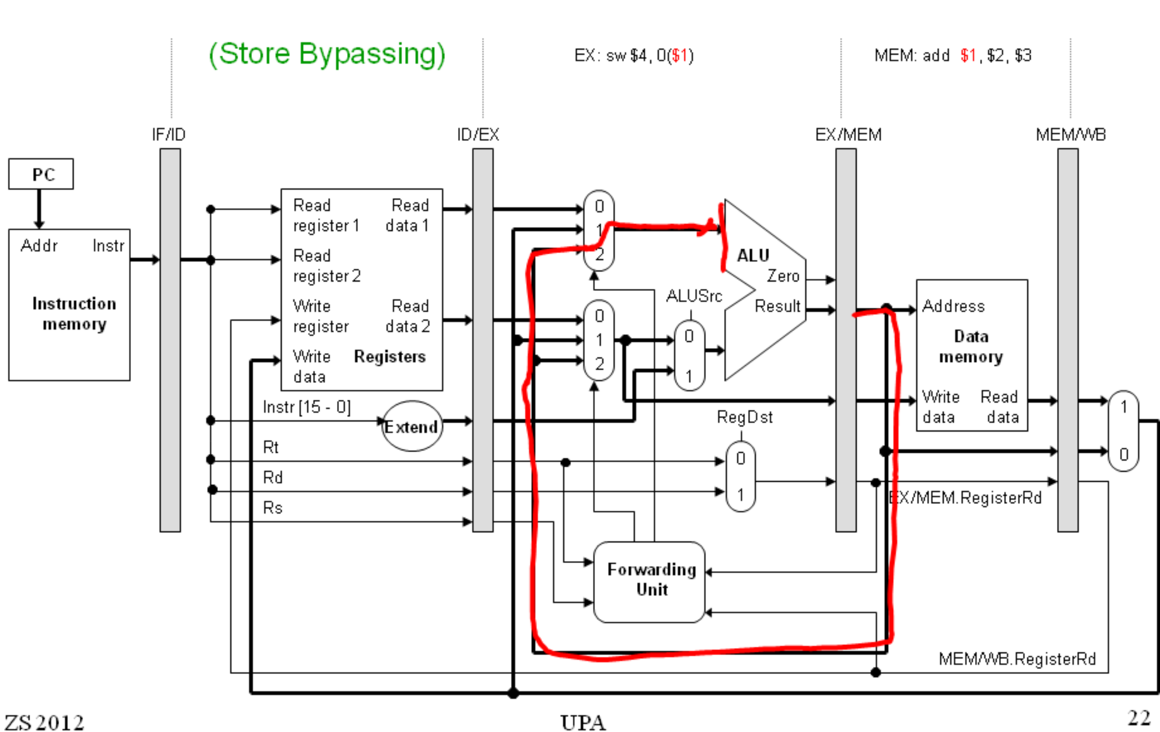
Úplné datové cesty s pipeliningem



Operace zápisu

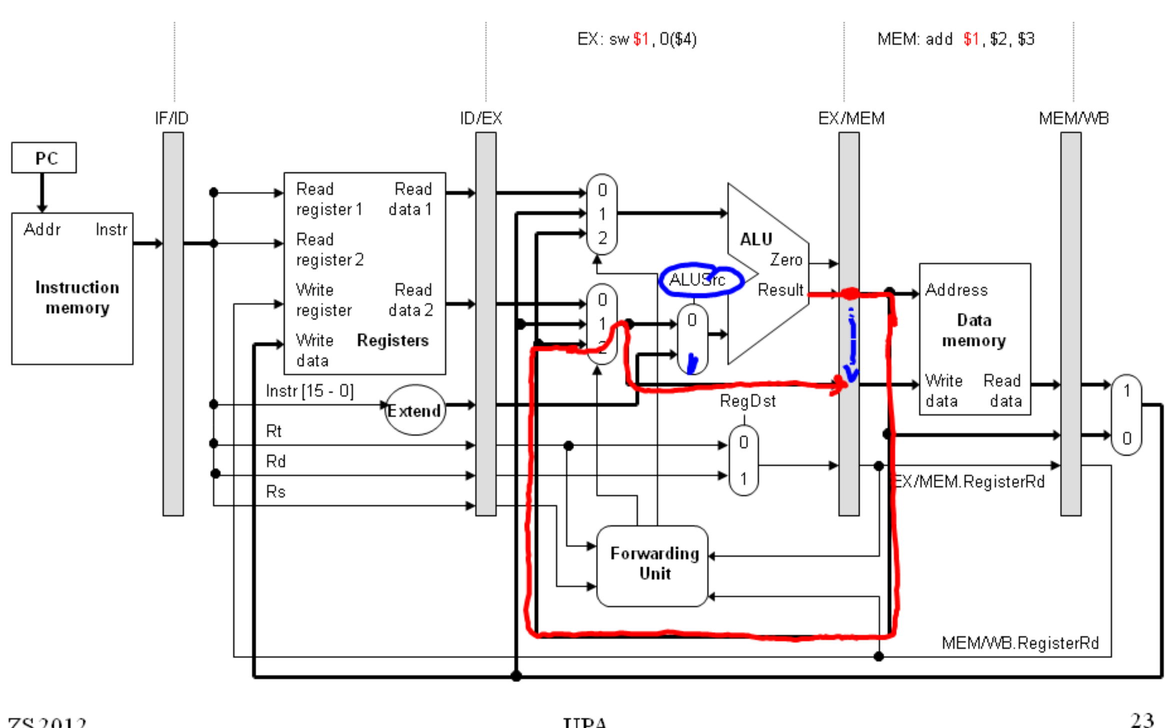


Bypass při zápisu verze 1 – store bypassing



Bypass při zápisu verze 2

Bypass při zápisu: Verze 2

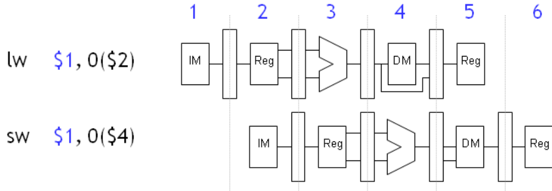


ZS 2012 UPA 23

Vlastní zápis – čím je třeba doplnit jednotku

Vlastní zápis?

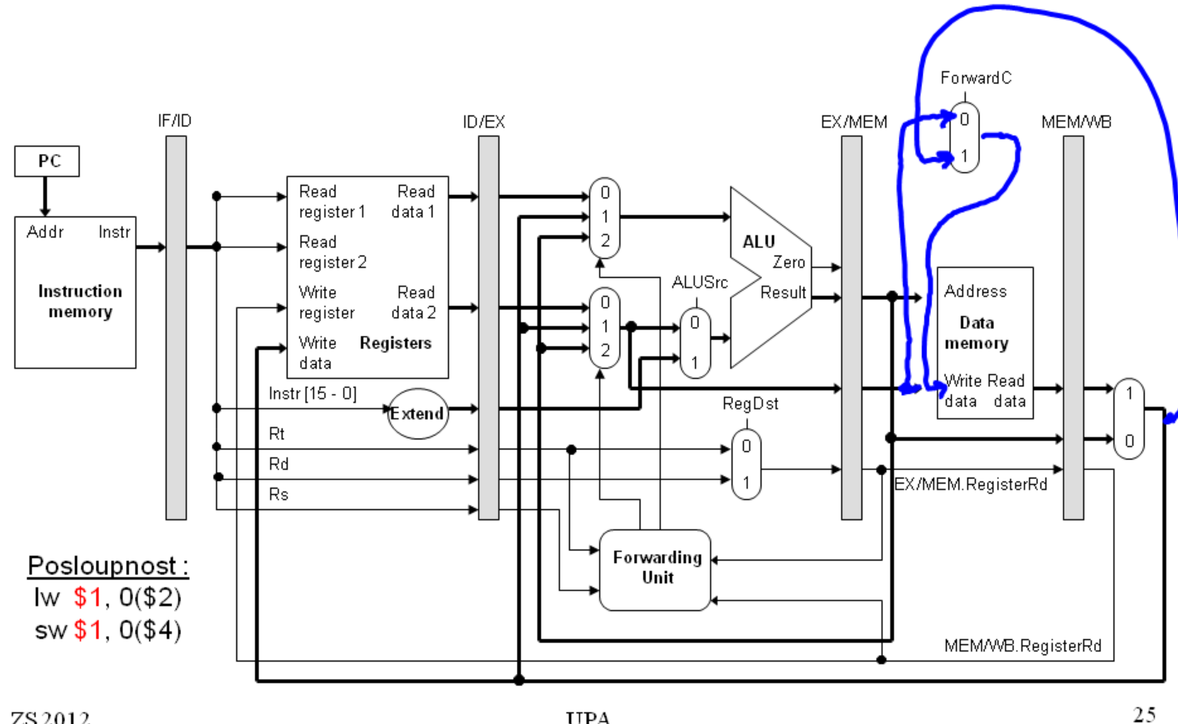
- Komplikovanější případ:



- Ve kterém cyklu:
  - Je k hodnota k dispozici? Konec cyklu 4
  - Je třeba ukládaná hodnota? Počátek cyklu 5
- Čím je třeba doplnit jednotku?

Rozšíření jednotky – Load bypass, Store bypass, Load/Store bypass – forward C

Rozšíření jednotky - Load/Store bypass



Poznámky, každá instrukce MIPS zapisuje maximálně do jediného registru, forwarding důležitý u hlubokých pipeline

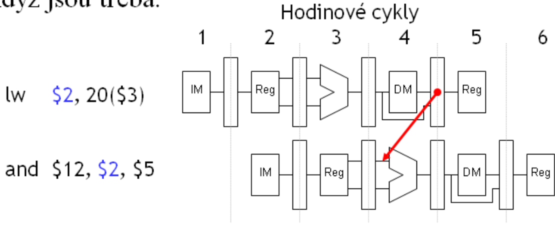
Poznámky

- Každá instrukce MIPS zapisuje maximálně do jediného registru.
  - Proto lze hardware pro forwarding snáze navrhnout. „Forwarduje" se nejvýše jediný cílový registr.
- Forwarding je zvláště důležitý u procesorů s hlubokým stupněm pipeliningu. To se týká hlavně současných procesorů, používaných v PC.
- Sekce 6.4 učebnice obsahuje některé doplňující materiály, které zde nebyly uvedeny.
  - Rovnice pro detekci hazardu testuje, zda zdrojový registr není $0, který nikdy nemůže být modifikován.
  - Je uveden složitější příklad forwardingu.

Problém se čtením – hazard, load, pravý datový hazard

Problém se čtením

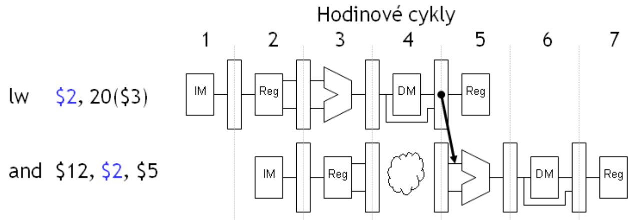
- Předpokládejme níže uvedenou posloupnost instrukcí.
  - Čtená data dorazí z paměti nejdříve na *konci* cyklu 4.
  - Operace AND vyžaduje tuto hodnotu na *počátku* stejného cyklu!
- Toto je "pravý" datový hazard – data nejsou k dispozici, když jsou třeba.



Pozastavení, hazard, bublina, stall

Pozastavení

- Nejjednodušším řešením je zastavit pipeline (stall).
- Zpozdíme instrukci AND zařazením zpoždění v délce 1 cyklu do pipeline. Toto zpoždění se nazývá bublina.

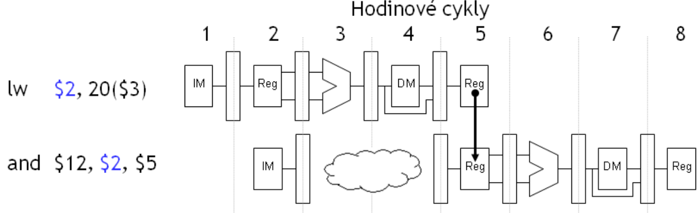


- Poznámka: Stále se používá forwarding v cyklu 5, abychom dopravili data z pipeline registru MEM/WB do ALU.

Pozastavení a forwarding, hazard, load

Pozastavení a „forwarding"

- Bez forwardingu bychom byli nuceni pozastavit na *dva* cykly a čekat na provedení zpětného zápisu instrukce LW.

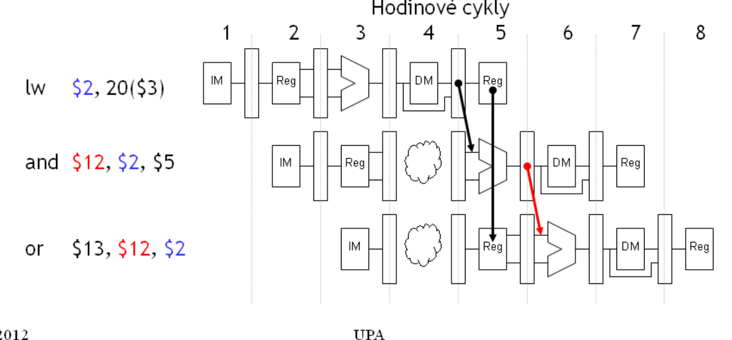


- Obecně lze hazardy vždy řešit vkládáním bublin. Prakticky to ale vzhledem k častým závislostem mezi instrukcemi vede k podstatné redukci výkonu.

Pozastavení zdrží celou pipeline

Pozastavení zdrží celou pipeline

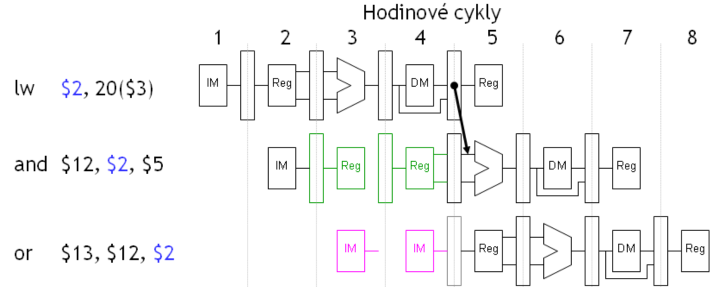
- Jestliže zpozdíme provedení druhé instrukce, musíme zpozdit také třetí a čtvrtou.
  - Je nutné provést forwarding mezi AND a OR.
  - Zabrání se tak problému, že se dvě instrukce snaží v jednom cyklu psát do stejného registru.



Co s registry EX, MEM, WB

Co s registry EX, MEM, WB ?

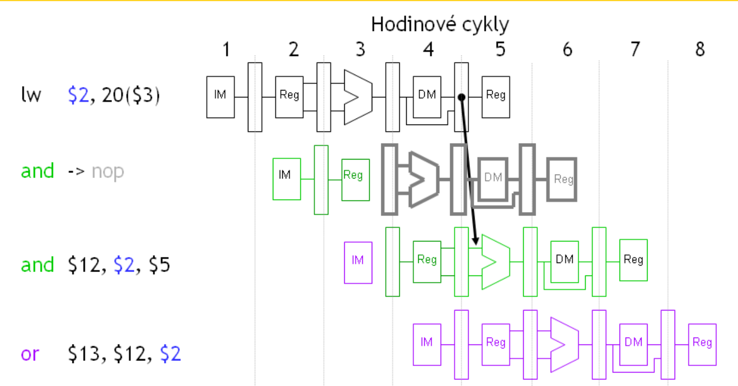
- Co bude provádět ALU během cyklu 4, datová paměť v cyklu 5 a co se bude zapisovat do registrového souboru v cyklu 6 ?



- Tyto funkční bloky nejsou v uvedených cyklech vzhledem k pozastavení využity a proto řídící signály EX, MEM a WB jsou neaktivní („0“).

Pozastavení = konverze na NOP, load stall

Pozastavení = konverze na Nop



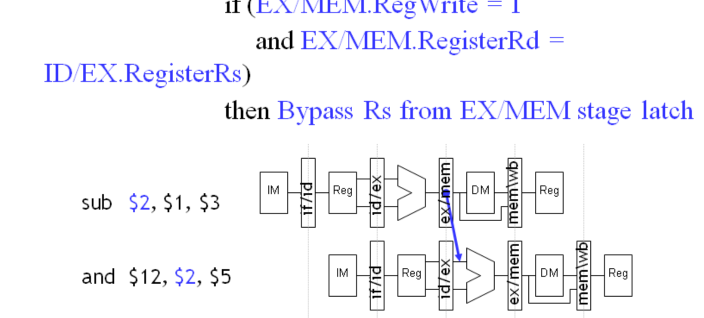
- Efekt pozastavení vlivem čtení (load stall) je vložení prázdné instrukce (nop) do pipeline

Detekce pozastavení, rovnice detekce hazardu, stage latch

Detekce pozastavení

- Detekce pozastavení se podobá detekci datových hazardů. Připomeňte si rovnici detekce hazardů:

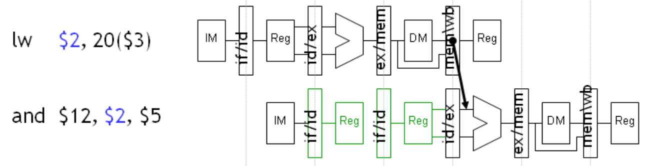
  if (EX/MEM.RegWrite = 1
  and EX/MEM.RegisterRd = ID/EX.RegisterRs)
  then Bypass Rs from EX/MEM stage latch



Detekce pozastavení, stalls, podmínka pro zařazení bubliny, kde se má detekovat pozastavení

Detekce pozastavení, pokračování

- Kdy se mají pozastavení (**stalls**) detekovat? Ve stupni EX

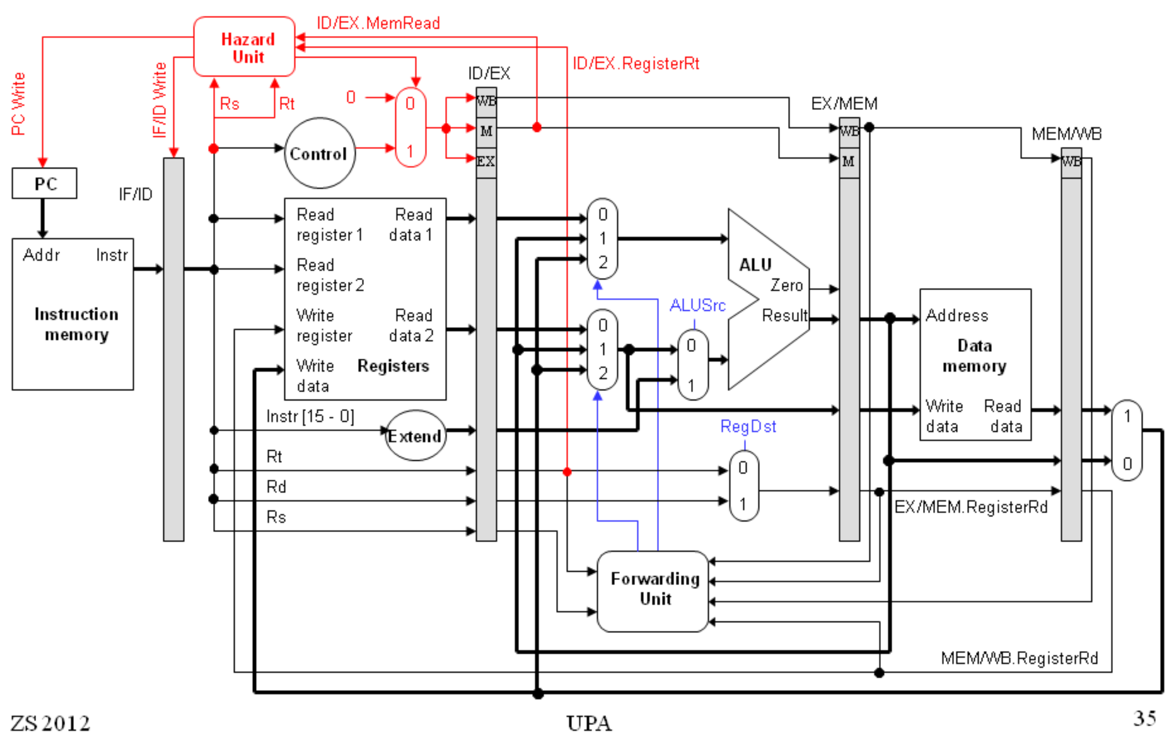


- Co je podmínkou pro zařazení bubliny (stall) ?

  if (ID/EX.MemRead = 1 and
  (ID/EX.RegisterRt = IF/ID.RegisterRs or ID/EX.RegisterRt = IF/ID.RegisterRt))
  then stall

Doplnění detekce hazardů do CPU, hazard unit - schéma

Doplnění detekce hazardů do CPU



Jednotka pro detekci hazardů, PCWrite, IF/ID Write, mux select

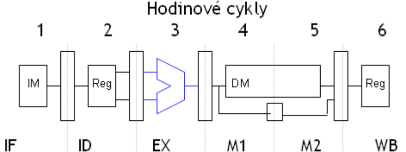
Jednotka pro detekci hazardů

- Jednotka pro detekci hazardů má následující vstupy.
  - IF/ID.RegisterRs a IF/ID.RegisterRt, zdrojové registry aktuální instrukce.
  - ID/EX.MemRead a ID/EX.RegisterRt, pro určení, zda předchozí instrukce byla LW. Jestliže ano, do kterého registru bude zapisovat.
- Na základě vyšetření těchto hodnot jednotka generuje tři výstupy.
  - Dva nové řídicí signály PCWrite a IF/ID Write, které určují, zda pipeline zastaví a nebo bude pokračovat.
  - Signál mux select pro nový multiplexer, který vynutí „0“ na řídicích signálech pro aktuální stupeň EX a příští stupeň MEM/WB v případě, že nastane „stall“.

Zobecnění forwardingu/pozastavení

Zobecnění forwardingu/pozastavení

- Co když je přístup do paměti pomalý tak, že jej potřebujeme rozprostřít přes dva cykly?



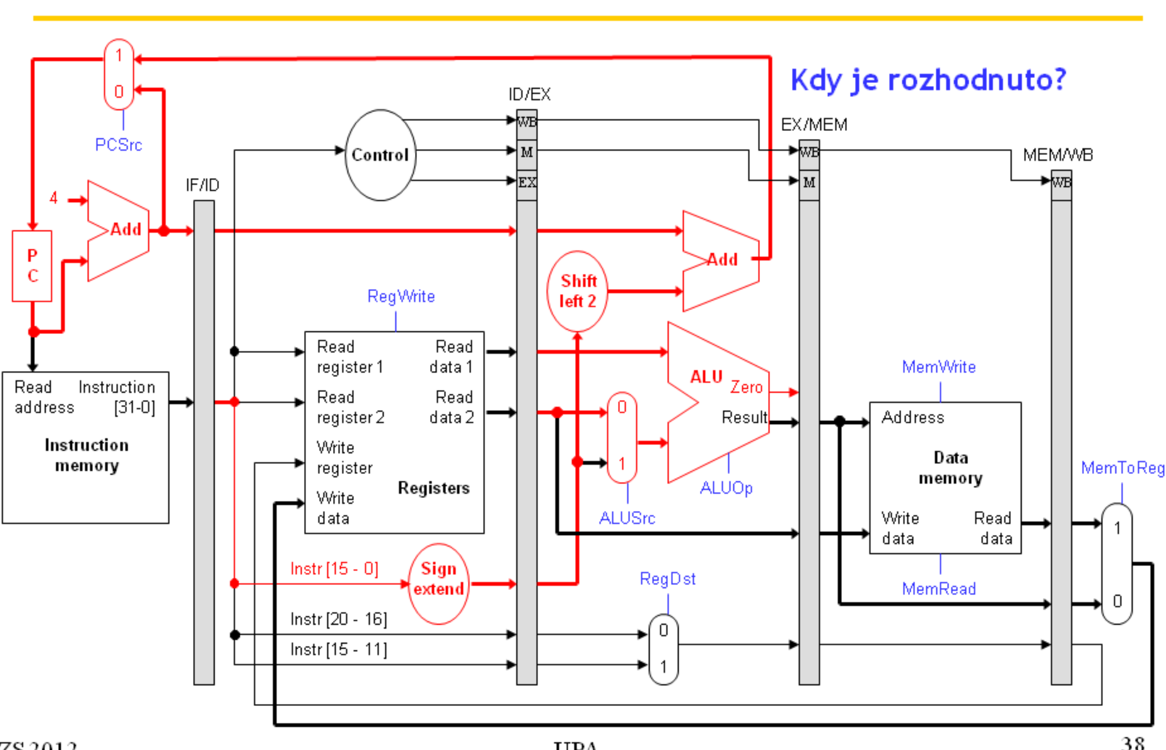
- Kolik vstupů pro bypass musí mít multiplexery ve stupni EX? (Odpověď: 4)
- Které instrukce v příkladu vyžadují pozastavení and/or bypass?

Instrukce	1	2	3	4	5	6	7	8	9	10
lw $3, 0($1)	IF	ID	EX	M1	M2	WB				
add $7, $8, $9		IF	ID	EX	M1	M2	WB			
			IF	ID						
add $5, $7, $3					ID	EX	M1	M2	WB	

ZS 2012 UPA 37

Větvení v původní jednotce s pipeliningem - schéma

Větvení v původní jednotce s pipeliningem

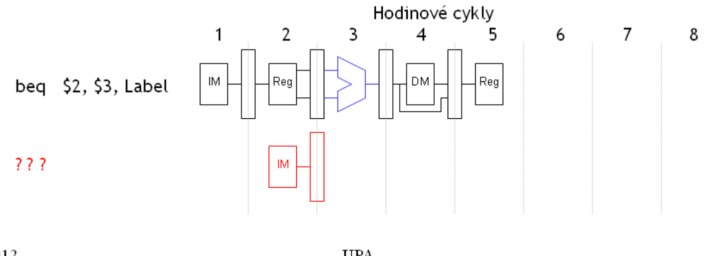


ZS 2012 UPA 38

Podmíněné větvení, řídící hazard

Podmíněné skoky - větvení

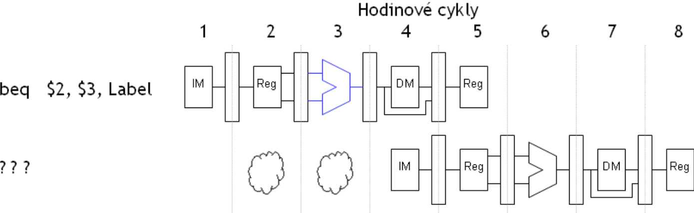
- Největší část výpočtů pro větvení se provádí ve stupni EX.
  - Vypočítává se cílová adresa skoku.
  - Zdrojové registry se komparují v ALU a podle výsledku je nastaven příznak Zero.
- Proto nemůže být rozhodnutí o skoku učiněno před dokončením stupně EX.
  - Potřebujeme ale provést čtení další instrukce a tak zajistit chod pipeline!
  - To vede na takzvaný řídící hazard.



Pozastavení je jen jedno řešení

Pozastavení je jen jedno řešení

- Přesto, pozastavením lze situaci vždy řešit.

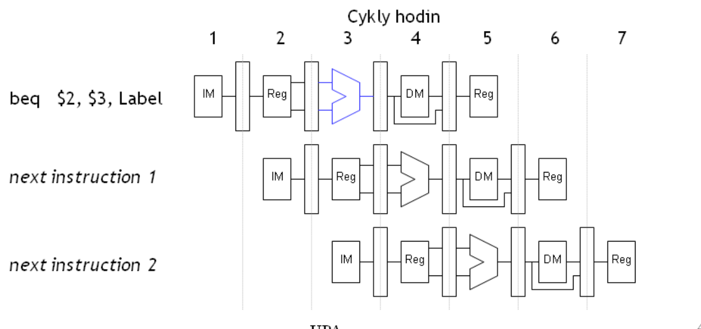


- V tomto případě pozastavíme až do cyklu 4, do té doby, než bude o skoku rozhodnuto.

Predikce skoků, odhad skoku

Predikce skoků

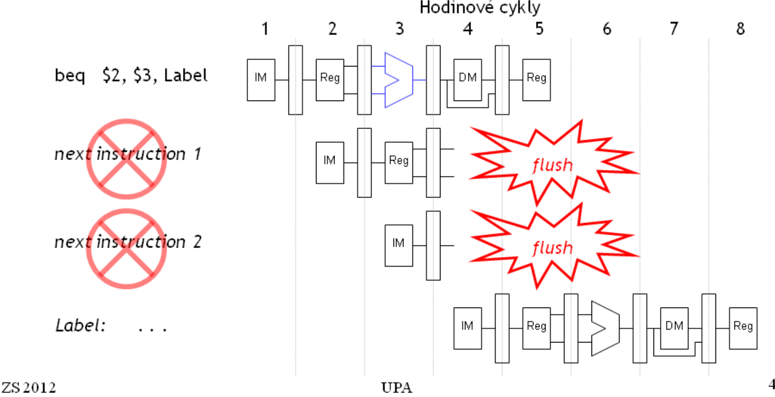
- Jiný přístup se zakládá na *odhadu*, zda se skok bude nebo nebude konat.
  - Z hlediska hardware je jednodušší předpokládat, že se skok *nekoná*.
  - V tomto případě pouze inkrementujeme PC a pokračujeme ve výpočtu.
- Je-li odhad správný, nevzniká problém a pipeline pokračuje plnou rychlostí.



Nezdařená predikce – flush

Nezdařená predikce skoku

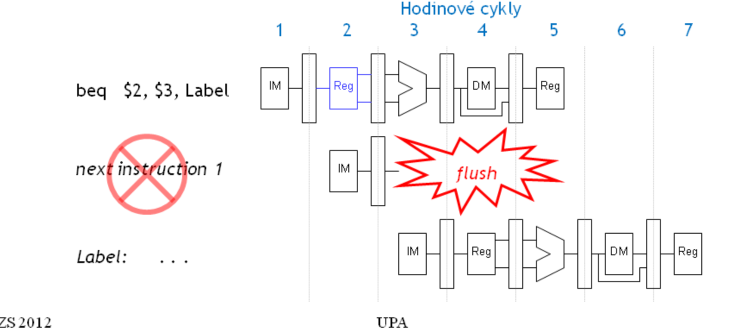
- Je-li náš odhad špatný, došlo k nekorektnímu odstartování dvou instrukcí. Musíme je vyřadit (flush) a začít výpočet na správném místě, tam kam ukazuje adresa cíle skoku (Label).



Implementace větvení

Implementace větvení

- Rozhodnutí o skoku může padnout již o něco dříve ve stupni ID namísto v EX.
  - Instrukční soubor v našem příkladu obsahuje jen instrukci BEQ.
  - Zařadíme malý komparační obvod do stupně ID, po čtení zdrojových registrů.
- Potom při nezdařené predikci stačí „zahazovat" jenom jednu instrukci.



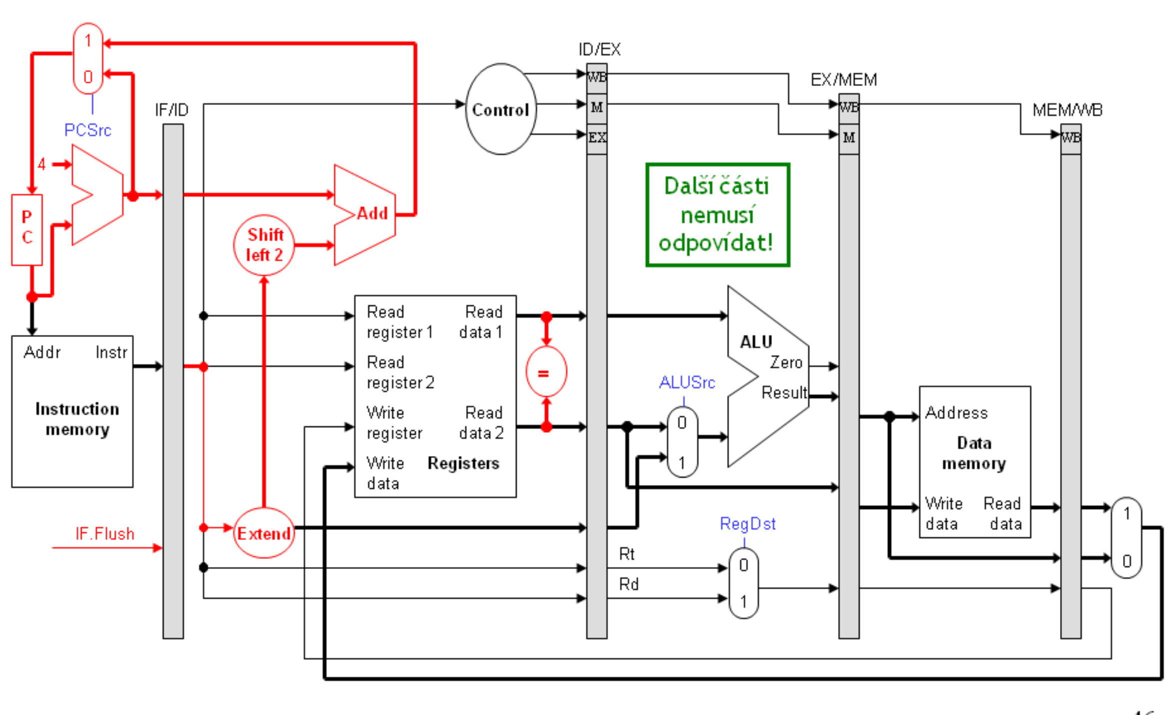
Implementace odhazování - flush

Implementace „odhazování" (flush)

- Musíme „odhodit" jednu instrukci (v jejím stupni IF), jestliže předchozí instrukce je BEQ a její zdrojové registry jsou shodné.
- „Odhození" instrukce ze stupně IF provedeme tak, že ji nahradíme v pipeline registru IF/ID neškodnou instrukcí NOP.
  - MIPS používá sll $0, $0, 0 jako instrukci NOP.
  - Binární prezentace je: 0000 .... 0000.
- Tím je zařazena bublina do pipeline, která reprezentuje zpoždění jednoho cyklu při skokové instrukci.
- Řídící signál IF.Flush, uvedený na dalším snímku implementuje tuto myšlenku, ale diagram neobsahuje žádné další podrobnosti.

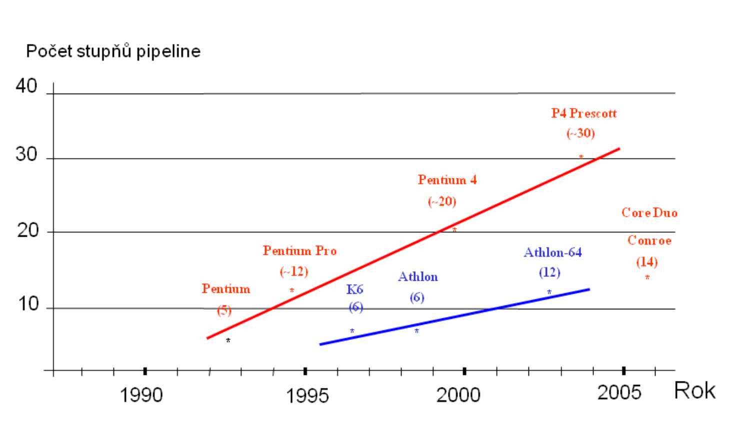
Větvení bez forwardingu a load stalls - schéma

Větvení bez „forwardingu“ a „load stalls“



Časový vývoj hloubky pipeline - graf

Časový vývoj hloubky pipeline



Strukturní hazardy, Datové hazardy, Řídící hazardy

Závěr

- Pipelining je komplikovaný hlavně kvůli třem typům hazardů.
- Strukturní hazardy které pramení z toho, že nemáme dostatečné množství HW pro současné provádění většího počtu instrukcí.
  - Lze je omezit dodáním funkčních jednotek (např. sčítaček, paměti) nebo úpravou stupňů pipeline.
- Datové hazardy mohou nastat, jestliže instrukce přistupuje k registrům, které nebyly včas aktualizovány.
  - Hazardy instrukcí typu R lze odstranit technikou „forwarding".
  - Čtení může způsobit "pravé" hazardy, které pozastaví pipeline.
- Řídící hazardy nastávají když CPU nemůže určit, která instrukce se má načíst jako další.
  - Lze je omezit včasným testováním skoků v pipeline.
  - Úspěšná predikce směru skoku také minimalizuje zpoždění.

Přehled pamětí, fyzická paměť, virtuální paměť, cache paměť

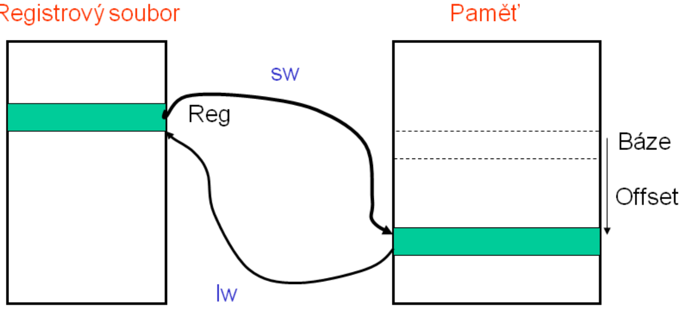
Přehled pamětí

- Uložení dat a instrukcí
- Model - lineární pole 32-bitových slov
- MIPS: 32-bitová adresa => 2^{32} slov
- Doba odezvy je pro každé slovo stejná
- Paměť lze číst/zapsat v 1 cyklu ☺
- Typy pamětí
  - Fyzická: *Instalována v počítači*
  - Virtuální: Disková paměť, která se chová jako hlavní paměť
  - Cache: Rychlá paměť pro dočasné umístění dat nebo instrukcí

Instrukce pro práci s pamětí

Instrukce pro práci s pamětí

- lw reg, offset(base) — Load => přenos paměť do registru
- sw reg, offset(base) — Store => přenos registr do paměti



Úvod do hierarchie paměti

Úvod do hierarchie

- Předpokládejme, že jste navštívili knihovnu a hledáte materiály o historii počítačů.
  - Procházíte mezi policemi a vybíráte knihy, které se týkají vašeho zájmu. Pokládáte je na pracovní stůl.
  - K policím jste šli proto, abyste si přinesli to, co budete brzy potřebovat.
  - A také proto, abyste mohli chvíli nerušeně pracovat, aniž byste se museli znovu zvednout a navštívit police s knihami.

Lokalita – princip lokality, časová lokalita, prostorová lokalita

Lokalita

- Tento příklad znázorňuje ***princip lokality*** – programy přistupují v daném krátkém časovém intervalu k relativně malé části adresního prostoru (stejně jako vy jste přinesli jen malou část knih).
- Existují dva typy lokality …
  - *Časová lokalita* – je-li položka referencována, má tendenci být referencována v krátkém časovém úseku opět. (knihu, kterou jste právě odložili na stůl pravděpodobně zase brzy vezmete do ruky).
  - *Prostorová lokalita* – Je-li položka referencována, budou patrně referencovány i s ní sousedící položky (naleznete-li knihu, je pravděpodobné, že vás mohou zajímat i knihy, ležící v jejím těsném okolí).

Technologie pamětí

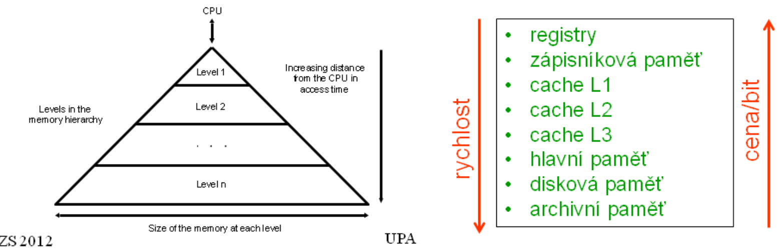
Technologie pamětí

- Programátoři vyžadují od nepaměti obrovské kapacity velmi rychlých pamětí.
- V současné době existuje velká řada nejrůznějších typů pamětí, různé rychlosti, ceny a výkonu. Všechny ale mají jednu tendenci. Čím jsou rychlejší, tím mají vyšší cenu a větší spotřebu.
- A tak namísto pořízení rozsáhlé rychlé paměti za vysokou cenu byl uveden koncept paměťové hierarchie, ve kterém počítače používají paměti různých kapacit a typů.

Technologie pamětí, úrovně, SRAM, DRAM, CACHE, doby přístupu, ceny

Technologie pamětí

- V současné době se ke stavbě hierarchického paměťového systému používají hlavně tři typy pamětí…
  - Hlavní paměť používá technologii DRAM (dynamická RAM).
  - Úrovně blíže k CPU (L1/L2/L3 cache) jsou statické RAM (SRAM). Jsou mnohem dražší, ale také mnohem rychlejší.
  - Největší kapacitu, ale také nejmenší rychlost mají paměti, používané na nejnižší úrovni. (např. harddisky). Tato technologie je velmi levná, ale doba přístupu je dlouhá.



Technologie pamětí

Technologie paměti	Typická doba přístupu	$ za GB v roce 2004
SRAM	0.5–5 ns	$4000–$10,000
DRAM	50–70 ns	$100–$200
Magnetický disk	5,000,000–20,000,000 ns	$0.50–$2

Trendy technologie paměti

Trendy technologie

	Kapacita	**Rychlost (latence)**
Logika:	2x za 3 roky	2x za 3 roky
DRAM:	4x za 3 roky	2x za 10 let
Disk:	4x za 3 roky	2x za 10 let

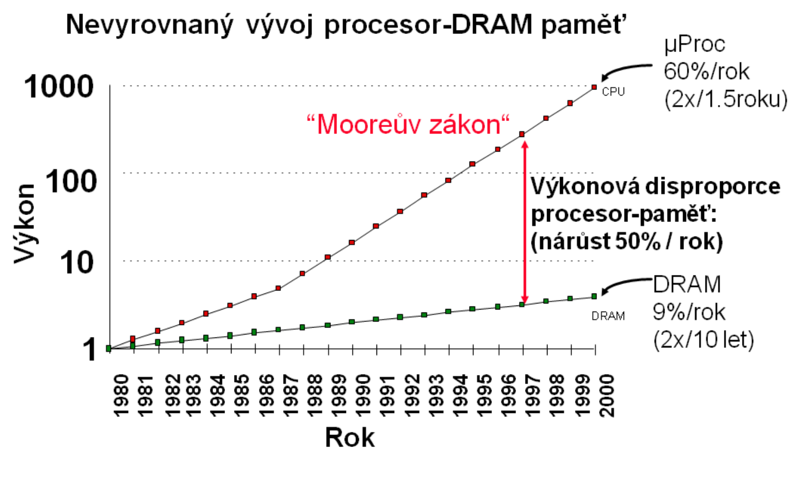
DRAM

Rok	Kapacita	Doba cyklu
1980	64 Kb	250 ns
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns
1998	256 Mb	100 ns
2001	1 Gb	80 ns

1000:1! 2:1!

Jak je to s pamětí, Mooreův zákon, graf

Jak je to s pamětí?



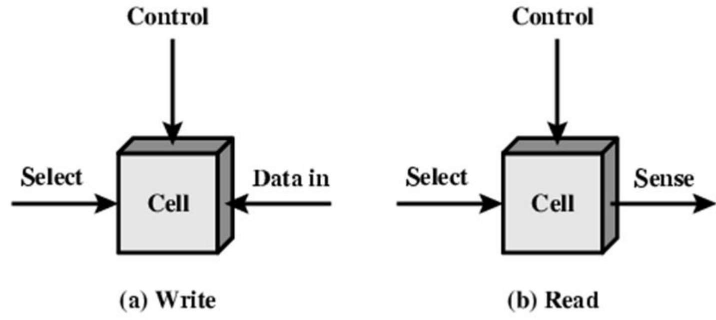
Paměť – SRAM, DRAM

Paměť

- SRAM:
  - Informace je uchována pomocí dvojice invertujících hradel
  - Velmi rychlé, ale větší nároky než DRAM (4 až 6 tranzistorů)
- DRAM:
  - Informace je uchována jako náboj na kondenzátoru (musí se zotavovat)
  - Velmi malá buňka, ale pomalejší než SRAM (poměr 1: 5 až 1:10)

Operace buňky paměť

Operace buňky paměti



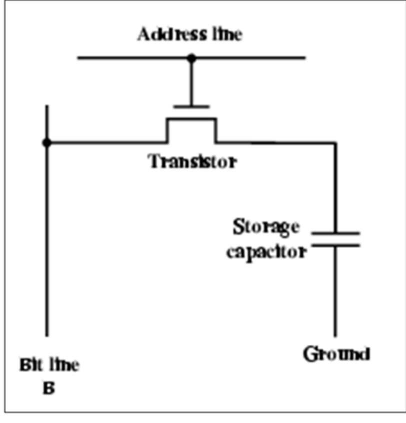
Dynamická paměť, DRAM

Dynamická RAM

- Bity jsou uloženy ve formě náboje na kapacitě
- Náboj se „vytrácí“ vlivem svodových proudů
- Je třeba zotavovat i když je paměť napájena
- Jednoduchá konstrukce
- Malý rozměr „bitu“
- Levnější než statická
- Jsou třeba zotavovací obvody
- Pomalejší
- Hlavní paměť
- V podstatě analogová záležitost
  - Úroveň náboje určuje hodnotu

Struktura DRAM

Struktura dynamické RAM



Operace DRAM – čtení zápis

Operace DRAM

- Adresní linky jsou aktivní při zápisu nebo čtení
  - Tranzistorový spínač sepnut (protéká proud)
- Zápis
  - Napětí bitové linky
    - Vysoká úroveň pro 1 nízká pro 0
  - Pak je aktivována adresní linka
    - Přenos náboje do kondenzátoru
- Čtení
  - Vybere se adresní linka
    - tranzistorový „spínač“ se sepne
  - Náboj kondenzátoru se přivede pomocí bitové linky na čtecí zesilovače
    - Porovnává se s referenční úrovní aby se určila 1 nebo 0
  - Náboj kondenzátoru je třeba obnovit

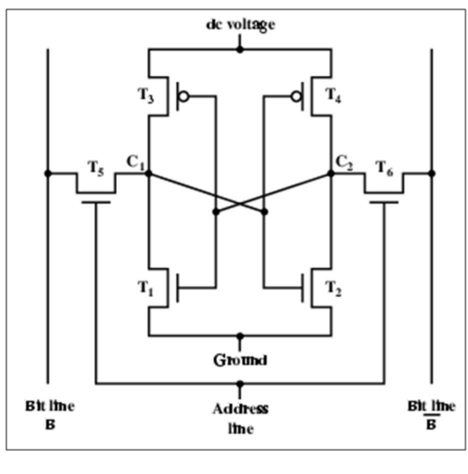
Statická RAM - SRAM

Statická RAM

- Bity jsou uloženy jako stav klopného obvodu
- Svodové proudy nemají vliv
- Není třeba zotavovat
- Složitější konstrukce buňky
- Větší rozměr buňky
- Vyšší cena
- Nejsou třeba zotavovací obvody
- Rychlejší
- Cache
- Čistě digitální princip
  - Využívají se klopné obvody (flip-flops)

Struktura buňka SRAM

Struktura buňky statické RAM



Operace SRAM

Operace statické RAM

- Uspořádání tranzistorů poskytuje stabilní logický stav
- Stav 1
  - C_1 vysoká úroveň, C_2 nízká úroveň
  - T_1 T_4 nevedou, T_2 T_3 vedou
- Stav 0
  - C_2 vysoká úroveň, C_1 nízká úroveň
  - T_2 T_3 nevedou, T_1 T_4 vedou
- Adresní linka a tranzistory T_5 T_6 řídí spínání
  - Zápis – „dopraví“ hodnotu do B & komplement do $\overline{B}$
  - Čtení – hodnota se objeví na lince B

SRAM vs DRAM – statická buňka, dynamická buňka

SRAM kontra DRAM

- Obě ztrácí informaci po vypnutí napájení
  - Pro uchování informace je třeba zachovat napájení
- Dynamická buňka
  - jednoduchá konstrukce, menší
  - Vyšší hustota
  - Levnější
  - Třeba zotavovat
  - Používá se pro stavbu větších paměťových celků
- Statická buňka
  - Rychlejší
  - Cache

Detaily organizace

Detaily organizace

- 16 Mbitový čip může být organizován jako 1M 16 bitových slov
- Organizace „bit per čip“ využívá 16 Mbitové čipy, s 1 bitem 1 v každém slově
- 16Mbit čip lze také organizovat jako pole 2048 x 2048 x 4 bitů
  - Redukuje se počet adresních pinů
    - Multiplexují se adresy řádek a sloupečků
    - 11 pinů pro adresy (2^{11}=2048)
    - Přidáním jednoho pinu adresy se adresovací schopnost zvýší 4 krát

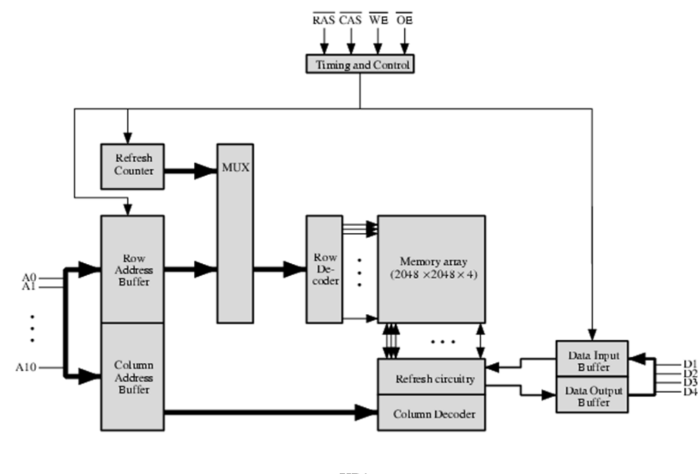
Proces zotavení

Proces zotavení

- Zotavovací obvody umístěny na čipu
- Čip je během zotavení „nedostupný“
- Probíhá po řádcích
- Čtení a zpětný zápis
- Zabírá čas
- Snižuje celkový výkon

Typický 16Mb DRAM (4M x 4) - schéma

Typická 16 Mb DRAM (4M x 4)



Hierarchie paměťového systému – Registry, Cache, Hlavní paměť, Disková paměť, doba přístupu

Hierarchie paměťového systému

1. **Registry**
   - Rychlá interní pamět v procesoru
   - *Rychlost* = 1 cykl hodin CPU *Perzistence* = několik cyklů *Kapacita* ~ 0.1K až 2K bytů
2. **Cache**
   - Rychlá paměť interní *nebo* externí (k procesoru)
   - *Rychlost* = Několik cyklů hodin CPU *Perzistence* = desítky až stovky cyklů *pipeline*, 0.5MB až 2MB
3. **Hlavní paměť**
   - Hlavní paměť obvykle externí vzhledem k procesoru, < 16GB
   - *Rychlost* = 1-10 hodin CPU *Perzistence* = milisekundy až dny
4. **Disková paměť**
   - **Velmi pomalá – *doba přístupu = 1 až 15 milisekund***
   - Použití pro *odkládání dat (instrukcí)* z hlavní paměti

Proč hierarchie

Proč hierarchie?

- Vzhledem k rozdílu v ceně a rychlosti je výhodné vybudovat hierarchii úrovní s rychlejšími pamětmi blíže k procesoru a s levnějšími pamětmi na nižších úrovních.
- Cílem tohoto uspořádání je prezentovat uživateli/programátorovi paměťový systém s kapacitou odpovídající levné technologii, ale s rychlostí, která odpovídá rychlým a drahým paměťovým prvkům.

Organizace hierarchie, blok, cache hit, cache miss

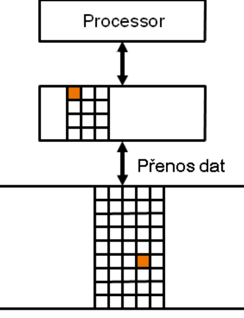
Organizace hierarchie

- U víceúrovňové organizace je každá úroveň organizována jako podmnožina libovolné nižší úrovně. Všechna data obsahuje nejnižší úroveň.
- Data se kopírují mezi sousedními úrovněmi. Přenášejí se po *blocích*. Blok - minimální jednotka informace, která se může nacházet v každé úrovni hierarchie.
- Jsou-li data požadovaná procesorem přítomná v nejvyšší úrovni cache, nastane tzv. *cache hit*. Nejsou-li nalezena, musí být získána z nižší úrovně. Tomu se říká *cache miss*.

Organizace hierarchie

Organizace hierarchie

- Mezi sousedními úrovněmi dochází k výměně informace (přenosu bloků u cache, popř. stránek u VM).



Cache paměť, pojmy, Hit, Miss, Blok, Set, Miss Rate

Cache paměť - pojmy

- **Blok:** Skupina slov **Set:** Skupina bloků
- Hit ☺
  - *Je-li hledán blok B v cache a je nalezen*
  - Hit Time: čas potřebný k nalezení bloku B
  - Hit Rate: četnost přístupů, kdy B je nalezen v cache
- Miss ☹
  - *Je-li hledán blok B v cache a není nalezen*
  - Miss Rate: četnost přístupů, kdy B je hledán v cache a není nalezen
- Příčiny:
  - Špatná strategie výměny bloků
  - Špatná lokalita prováděných programů

Příklad Cache Miss

Příklad: Cache Miss

- Předpokládejme dvouúrovňový systém s cache. Obsahuje hlavní paměť a cache (která je rychlejší a má mnohem menší kapacitu).
- Předpokládejme, procesor vyžaduje data z paměťového místa X_n. Tato data právě nejsou v cache => vzniká miss a data musíme čerpat z hlavní paměti (a kopírovat je do cache).

a. Před referencí Xn
X 4
X 1
X n – 2
X n – 1
X 2
X 3

b. Po referenci Xn
X 4
X 1
X n – 2
X n – 1
X 2
X n
X 3

Příklad cache miss podrobněji – adresa bloku modulo počet bloků v cache

Příklad trochu podrobněji

- Jestliže rozebereme předchozí příklad trochu podrobněji, vznikají dvě otázky…
  - Jak se dozvíme, že data jsou v cache?
  - Jestliže ano, pak kde?
- Tyto otázky můžeme vyřešit současně, jestliže každému slovu v paměti přidělíme slovo v cache, jehož poloha je odvozena od adresy slova v hlavní paměti.
  - Tato organizace se nazývá *přímo mapovaná cache*.
  - Je to jednoduché mapování, u kterého je každému místu v paměti přiřazeno právě jedno místo v cache.
  - Transformační předpis je jednoduchý …
    (Adresa bloku) modulo (# bloků v cache)

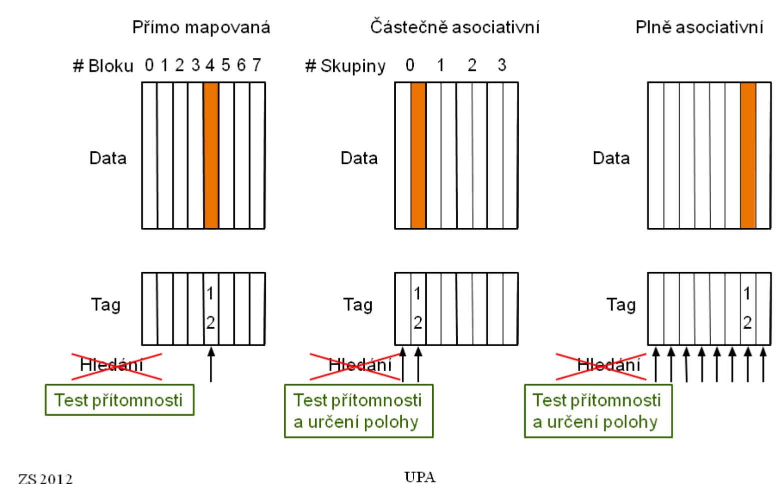
Cache paměť, princip lokality , 3 mapovací schémata

Cache paměť

- Princip *lokality programů*
  - *lw* a *sw* během daného krátkého časového úseku přistupují k velmi malé části paměti
- Cache *obsahuje* kopie úseků paměti => dočasná paměť
  - 3 mapovací schémata: Dán paměťový blok B
    - ***Přímo mapovaná***: cache *jedna k jedné* -> paměťová mapa (B může být obsažen jen v jediném bloku cache)
    - ***Částečně asociativní (vícecestná)***: jeden z *n* bloků cache obsahuje B
      (n … malé číslo, obvykle n=2, 4, 8)
    - ***Plně asociativní***: Kterýkoli blok cache může obsahovat B
- Různá mapovací schémata pro různé způsoby přístupu na data

Tři strategie organizace cache

Tři strategie organizace cache

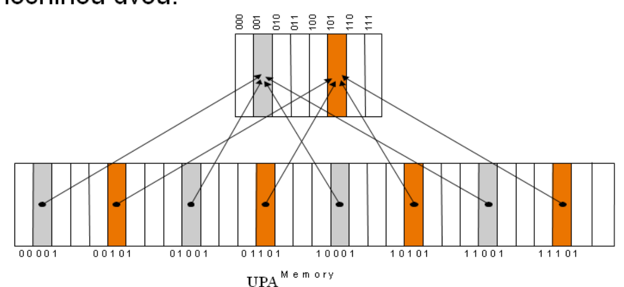


Přímo mapovaná cache

Přímo mapovaná cache

Tato organizace je velmi jednoduchá, protože operaci modulo lze provést jednoduše uplatněním dolních $\log_2$(velikost cache v blocích) bitů adresy.

- cache se adresuje dolní částí adresního pole.
- to je pravda jen tehdy, je-li počet položek v cache mocninou dvou.



Význam pole tag

Význam pole „tag“

- To znamená, že každý blok hlavní paměti se mapuje na jeden jediný blok cache, ale na jeden vstupní bod cache se mapuje větší množství bloků hlavní paměti.
  - Jak můžeme určit, že právě požadovaný blok je umístěný v cache?
- Zajistíme to doplněním souboru *tagů do* cache. Ke každému bloku v cache jednoduše uložíme ještě jeho horní část adresy. Výběr bloku provedeme polem *index* a zkontrolujeme, zda horní část adresy požadovaného bloku je totožná s polem *tag*, uloženým v cache.

Nutnost použití příznaku platnosti

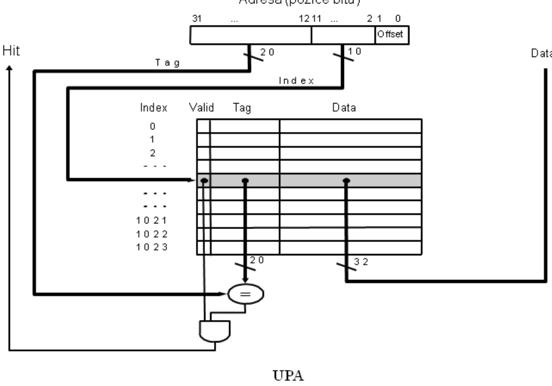
Nutnost použití příznaku platnosti

- Vedle pole *tag* je třeba zaznamenat, zda blok v cache obsahuje platnou informaci.
  - Při startu procesoru bývá cache naplněna náhodnými daty.
  - I po chvíli činnosti, nemusí být obsah všech položek v cache platný.
- Tento problém lze jednoduše odstranit přidáním bitu platnosti (valid bit) ke každému bloku. Není – li tento bit nastaven, nemůže být blok „nalezen“ (nenastane hit).

Příklad přímo mapované cache

Příklad přímo mapované cache

- Příklad přímo mapované cache paměti s 1024 položkami, každá o šířce 1 slovo. Nižší bity jsou použity pro výběr řádky cache.



Čtení z cache

Čtení z cache

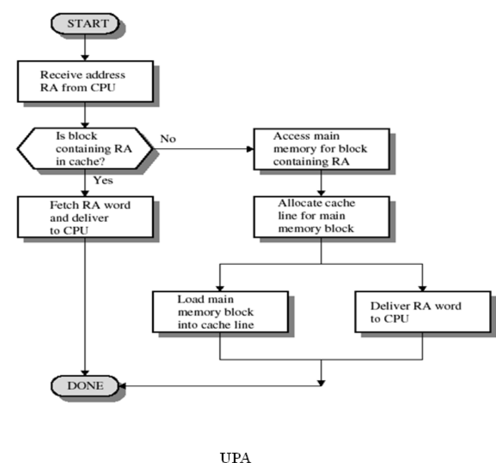
- Co se stane, když procesor vyžaduje data z paměti (operace čtení)?
  - Jedná-li se o „cache hit“, neděje se nic zvláštního, řídící jednotka pouze vyšetřuje příchod signálu „hit“.
  - Jedná-li se o „cache miss“, musí se CPU s tímto problémem zabývat.

    Obvyklé řešení:

    CPU se pozastavuje a čeká, dokud nejsou data přenesena z nižší úrovně (z paměti a nebo z cache nižší úrovně).

Diagram čtení z cache

Cache – operace čtení



Cache miss vs zastavení pipeline

Cache miss x zastavení pipeline

- Je-li CPU zastavena, každý registr si musí uchovat svoji hodnotu. Je to jednodušší, než pozastavit pipeline, protože se zastavuje vše a nemusí pokračovat provádění některých instrukcí, jako je tomu v případě pozastavení samotné pipeline.
- Řízení situace„cache miss“, přesun dat z nižší úrovně do cache obstarává vyhrazený řadič.
- Čteme-li z datové paměti, jednoduše zastavíme celý procesor, dokud nejsou data k dispozici.

Instrukční výpadek miss

Instrukční výpadek (miss)

- Uvažujme případ, že nastane miss v instrukční cache. Jaké úkoly bude vyhrazený řadič plnit?
  - Zašle adresu chybějící instrukce (aktuální PC – 4) do paměti.
  - Vyžádá operaci čtení z hlavní paměti a čekání na dokončení čtení.
  - Zapíše položku do cache (uloží datový blok, horní část adresy do pole tag a nastaví příznak platnosti).
  - Restartuje provedení instrukce v prvním kroku. Procesor znovu přečte instrukci (tentokrát ji v cache nalezne).

Přístup do paměti cache příklad

Příklad: Přístup do paměti s cache

- Cache obsahuje 8 bloků. Při startu je prázdná. Budeme sledovat její činnost …

1) Adresa 10110
vstup 110 - miss

Index	Valid	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Příklad: Přístup do paměti s cache

- Cache obsahuje 8 bloků. Při startu je prázdná. Budeme sledovat její činnost …

1) Adresa 10110
vstup 110 - miss
2) Adresa 11010
vstup 010 - miss

Index	Valid	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Memory[10110]
111	N		

Příklad: Přístup do paměti s cache

- Cache obsahuje 8 bloků. Při startu je prázdná. Budeme sledovat její činnost …

1) Adresa 10110
vstup 110 - miss
2) Adresa 11010
vstup 010 - miss
3) Adresa 10110
vstup 110 - hit

Index	Valid	Tag	Data
000	N		
001	N		
010	Y	11	Memory[11010]
011	N		
100	N		
101	N		
110	Y	10	Memory[10110]
111	N		

Příklad: Přístup do paměti s cache

- Cache obsahuje 8 bloků. Při startu je prázdná. Budeme sledovat její činnost …

1) Adresa 10110
   vstup 110 - miss
2) Adresa 11010
   vstup 010 - miss
3) Adresa 10110
   vstup 110 - hit
4) Adresa 10010
   vstup 010 - miss

Index	Valid	Tag	Data
000	N		
001	N		
010	Y	11	Memory[11010]
011	N		
100	N		
101	N		
110	Y	10	Memory[10110]
111	N		

Příklad: Přístup do paměti s cache

- Cache obsahuje 8 bloků. Při startu je prázdná. Budeme sledovat její činnost …

1) Adresa 10110
   vstup 110 - miss
2) Adresa 11010
   vstup 010 - miss
3) Adresa 10110
   vstup 110 - hit
4) Adresa 10010
   vstup 010 - miss

Index	Valid	Tag	Data
000	N		
001	N		
010	Y	10	Memory[10010]
011	N		
100	N		
101	N		
110	Y	10	Memory[10110]
111	N		

Reálné cache – DEC3100, MIPS R2000

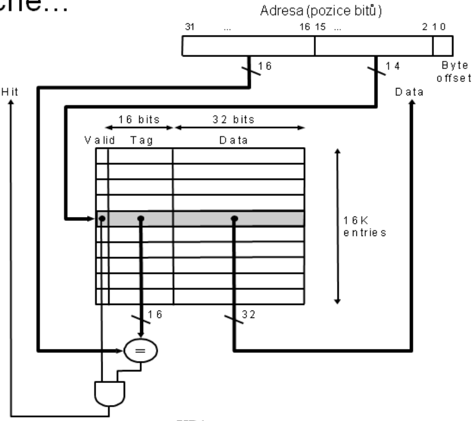
Reálné cache

- Jako reálný příklad uvedeme DEC3100, pracovní stanice s procesorem MIPS R2000.
- Protože čtení instrukce a dat probíhá ve stejném cyklu, používají se oddělené cache paměti pro instrukce a data - I a D cache (každá 64KB).
- Požadavek na čtení je jednoduchý…
  - Zaslání adresy příslušné cache paměti. To se děje buď z PC (I) nebo z ALU (D).
  - Hlásí-li cache miss, zasílá adresu do hlavní paměti. Jakmile paměť informaci přečte, je uložena do cache a zpracování pokračuje.

Cache DEC3100

Cache DEC3100

- Jak pro data tak i pro instrukce je vyhrazena jedna taková cache…



Zápisy do cache – write through, metoda přímého zápisu, nekonzistentní data

Zápisy do cache

- Z předchozího vyplývá, že čtení z paměti, je-li přítomna cache, je jednoduché. Co se ale bude dít, má-li proběhnout zápis?
- Předpokládejme, že zápis proběhne jen do datové cache (hlavní paměť se nezmění). Po tomto zápisu se data v paměti a v cache přestanou shodovat, jsou *nekonzistentní*.
- Nejjednodušší metodou, jak řešit tento problém, je zapsat data do obou ve stejnou dobu. Tato strategie se nazývá metoda přímého zápisu (*write-through)*.

Metoda přímého zápisu – write hit

Metoda přímého zápisu

- Pokud je adresovaná položka v cache (write hit), má smysl aktualizovat kopii v cache a současně zapsat do „originálu“ v hlavní paměti..
- Jak by se měl systém chovat v případě, že se zapisovaná položka v cache nenachází?
  - Nejjednodušším řešením je zapsat data do řádky v cache a současně aktualizovat tag a příznak platnosti (valid bit).
  - Dokonce není třeba zkoumat, zda data jsou nebo nejsou v cache. Zkrátka data zapíšeme do příslušného řádku.

Ztráta výhonu, write through

Ztráta výkonu?

- Uvedený přístup byl implementován u DEC3100.
- Metoda přímého zápisu je velmi jednoduchá – data se jednoduše zapisují do cache. Snadno se implementuje.
- Vychází z pozorování, že u běžných programů představuje operace zápisu jen 10 – 30% ze všech operací s pamětí (čtení, zápis).
- Z hlediska výkonu nedává ale právě nejlepší výsledky.
  - Kdykoliv se procesor provádí zápis, aktualizuje se cache a současně také hlavní paměť. To trvá dlouho vzhledem k relativně pomalé hlavní paměti (dlouhá doba zápisového cyklu).

Zápisový buffer – write buffer

Zápisový buffer

- Redukci výkonu, způsobenou zápisy, lze omezit použitím speciální vyrovnávací paměti – zápisového bufferu (*write buffer*).
  - Zápisový buffer obsahuje data, která čekají na zápis do hlavní paměti.
  - Zápis procesoru probíhá tak, že se zapíše do cache a do zápisového bufferu a pak procesor může pokračovat ve výpočtu.
  - Po ukončení zápisu do paměti se zápisový buffer opět uvolní.
  - Je-li zápisový buffer ještě plný a procesor opět požaduje zápis, musí být pozastaven, dokud se buffer neuvolní.

Selhání zápisového bufferu

Selhání zápisového bufferu?

- Uvedená strategie zlepšuje výkon, ale může selhat.
- Jestliže procesor generuje zápisy rychleji než se mohou provádět zápisy do paměti, zápisový buffer příliš nepomůže.
- I když je četnost zápisů nižší, přesto může docházet k zastavování, pokud se budou zápisy shlukovat. Lze částečně kompenzovat větší hloubkou zápisového bufferu než 1.
- DEC3100 má hloubku zápisového bufferu 4.

Write-back, write back, strategie nepřímého zápisu, metoda nepřímého zápisu

Strategie nepřímého zápisu - Write-Back

- Alternativní přístup je interpretovat všechny zápisové operace jen jako zápisy do cache. Hlavní paměť se aktualizuje až při výměně bloků.
- Tato metoda se nazývá metoda nepřímého zápisu (*write-back*).
- Metoda vede na podstatně složitější implementaci, může ale přinést značné urychlení zápisových operací.

Strategie zápisu bloků – výhody nevýhody

Strategie zápisu bloků

1) Write-Through:

- Zápis dat (a) do cache *a současně* (b) do bloku v hlavní paměti
- **Výhoda:** Případ „Miss" je jednodušší & levnější, protože není třeba zapisovat blok zpět do nižší úrovně
- **Výhoda:** Snazší implementace, je třeba pouze zápisový buffer

2) Write-Back:

- Zápis dat *pouze* do bloku cache. Zápis do paměti pouze při výměně bloků
- **Výhoda :** Zápisy omezeny pouze rychlostí zápisu cache
- **Výhoda :** Podporovány zápisy více slov najednou, efektivní je pouze zápis celého bloku do hlavní paměti najednou

Prostorová lokalita

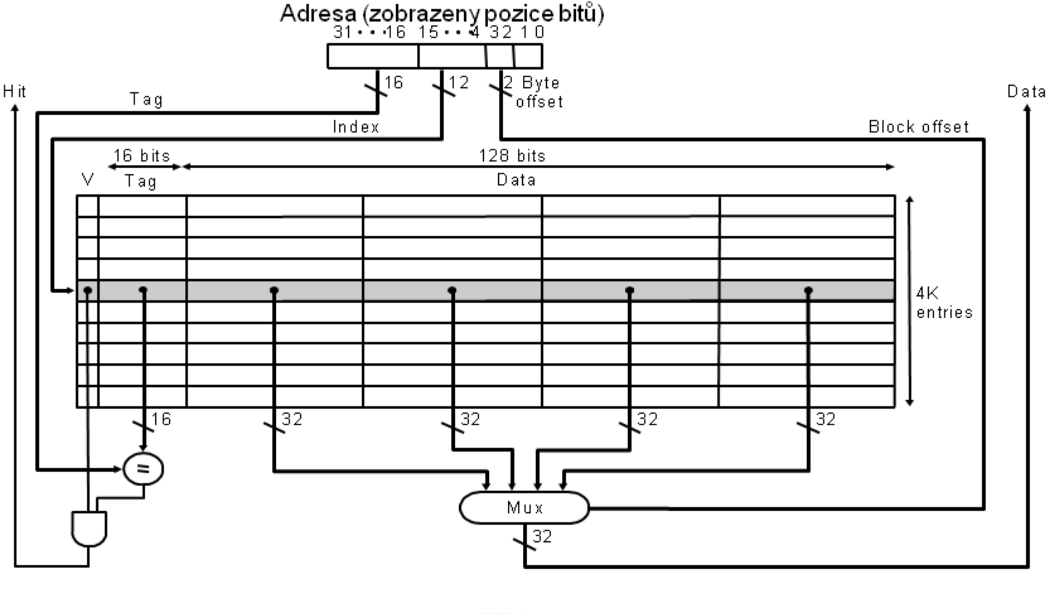
Prostorová lokalita

- Dosud jsme ignorovali prostorovou lokalitu – pozorování, že data, sousedící s právě referencovanými daty mají větší šanci přístup v krátké době.
- Tento princip může být zohledněn tak, že vytvoříme bloky, které mají větší velikost než pouhé jedno slovo.
- Nastane-li výpadek, přečteme kromě požadovaného ještě další slova, ležící „vedle" (ve stejném bloku). Je vysoce pravděpodobné, že budou v zápětí požadována.

Cache s délkou bloku větší než jedno slovo

Cache s délkou bloku větší než jedno slovo

- Struktura cache, která využívá prostorové lokality…



Sémantika cache – bloky s větším počtem slov, tag, blok, ofset v bloku

Sémantika cache - bloky s větším počtem slov

- Předpokládejme, že blok obsahuje 4 slova.
- Dva nejnižší bity použijeme k výběru slova v bloku.
- Dalších 12 bitů vybírá blok (přímo mapovaná cache).
- Horních 16 bitů se používá jako tag.
- Poznámka: Používáme jeden tag pro čtyři slova v paměti…
  - Zlepšuje efektivitu a ukládá se méně pomocné informace (tag), vztažené na jedno slovo.

Tag (16 bitů)	Blok (12 bitů)	Ofset v bloku (2 bity)

Výpadek v cache s víceslovnými bloky – write through

Výpadek v cache s víceslovnými bloky

- Výpadky při čtení jsou stejně jednoduché jako u jednoslovných bloků – jednoduše načteme data z paměti.
- Výpadky při zápisu jsou složitější.
  - Protože každý blok obsahuje více než jedno slovo, nemůžeme jednoduše zapsat tag a datové slovo – ostatní slova nepřísluší stejnému bloku (čtveřici slov).
  - Pro write-through cache, nejsou-li tagy shodné, můžeme načíst celý blok z paměti. Po načtení bloku můžeme zapsat slovo, které způsobilo výpadek do bloku a do paměti.
  - Použitím této strategie způsobuje výpadek při zápisu čtení z paměti (na rozdíl od cache s jedním slovem v bloku).

Cena za více slov v bloku

Cena za více slov v bloku

- Umístíme-li v bloku větší počet slov, četnost výpadků klesá.
- Naopak narůstá cena za načtení nového bloku, protože načítáme více slov.
- Zvětšujeme-li dále velikost bloku, převáží ztráta načítáním velkých bloků a účinnost cache se (podstatně) sníží.

Příklad Cache miss – nenalezena data

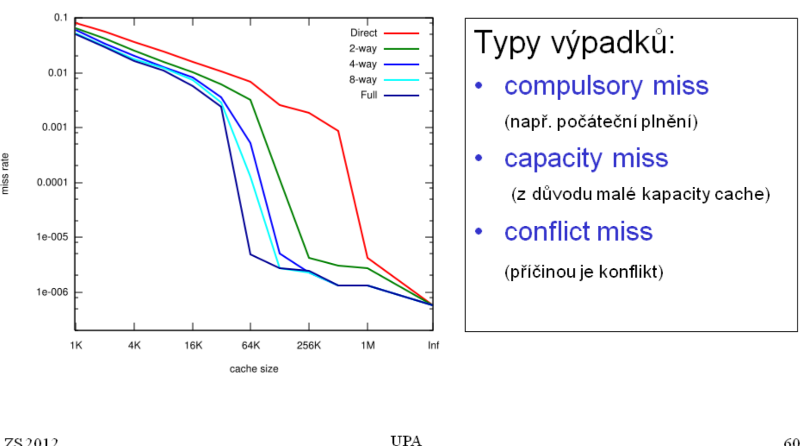
Případ – data v cache nenalezena

***Příklad:* „Miss“ v instrukční cache** (instrukce nebyla v cache nalezena)

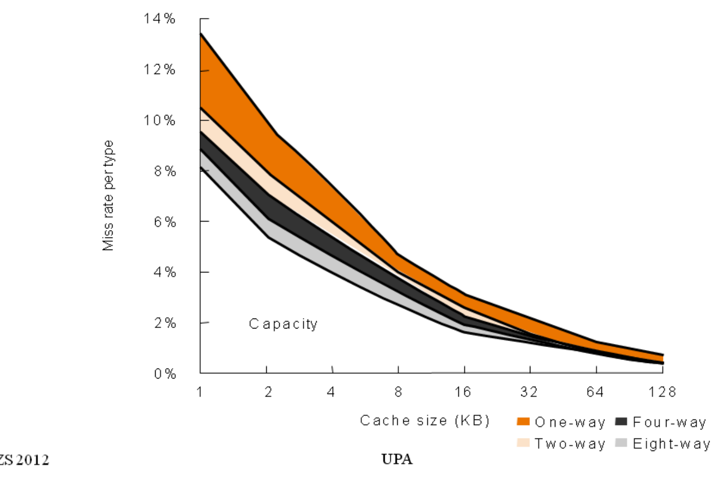
1. Do paměti poslána originální hodnota PC (současný PC – 4)
2. Hlavní paměť provede Read, čekáme na dokončení přístupu
3. Zápis položky do cache (Write) :
   - Data z paměti se zapíší do datového pole cache
   - Zápis horních bitů adresy do příslušného pole Tag
   - Nastavení příznakového bitu Valid Bit ON
4. Restart provedení instrukce ve stupni pipeline IF – instrukce je znovu načtena, nyní již bude v cache nalezena

Četnost výpadků graf, příčiny

Četnost „výpadků“ závisí na organizaci



Četnost výpadků a jejich příčiny



Typy výpadků, povinné, kapacitní, konfliktní

Typy výpadků

- Co způsobuje výpadek bloku? Existují tři příčiny…
  - **Povinné výpadky (Compulsory Misses)** – při prvním přístupu k bloku v cache se blok v cache nenachází.
  - **Kapacitní výpadky (Capacity Misses)** – jestliže cache neobsáhne všechny bloky potřebné k běhu programu. Nastávají, když jsou bloky odklízeny do nižší úrovně a při potřebě opět načítány.
  - **Výpadky kvůli konfliktu (Conflict Misses)** – nastávají u přímo mapované a nebo částečně asociativní cache, kdy různé bloky obsazují stejnou pozici v cache.

Povinné výpadky

Povinné výpadky (Compulsory Misses)

- Generují se v okamžiku prvého přístupu na blok. Nejlépe se redukují zvětšením velikosti bloku.
  - Redukuje se počet referencí k „novým“ blokům pro každý program.
  - Celý program pak obsazuje menší počet bloků.
- Zvětšování má ale za důsledek vyšší cenu za výměnu bloku (cena za miss). Proto je třeba volit kompromis.

Kapacitní výpadky

Kapacitní výpadky (Capacity Misses)

- Výpadky z důvodů kapacitních lze potlačit zvětšováním kapacity cache.
- Zvětšování kapacity s sebou ale zároveň přináší nárůst přístupové doby, což by mohlo vést k poklesu výkonu.

Konfliktní výpadky

Výpadky kvůli konfliktům (Conflict Misses)

- Tento typ výpadků lze redukovat zvýšením asociativity cache.
- Výpadky tohoto typu vznikají tehdy, obsazuje-li nový blok místo jiného, i když je v cache ještě volné místo. Zvětšováním počtu možných pozic pro daný blok se tyto výpadky redukují.
- Opět je třeba volit kompromis, protože zvyšování míry asociativity může způsobovat určitou časovou ztrátu.

Měření výpočtu v cache - vzorec

Měření výkonu cache

- Abychom porozuměli těmto závislostem, je třeba pochopit princip měření.
- Protože čas CPU je rozdělen na hodinové takty věnované výpočtu a hodinové takty strávené čekáním na paměť, vystačíme s jednoduchým vzorcem (který předpokládá, že úspěšné přístupu jsou součástí běžných prováděcích cyklů) ...

CPU time = (CPU execution cycles + Memory-stall cycles) * Clock cycle time

Stall cycle ...“prostoj“

Měření výkonu cache

- Učinili jsme mnoho předpokladů (hlavně ten, že všechna pozastavení pipeline jsou způsobena výpadkem cache). V moderních procesorech vyžaduje přesná predikce výkonu komplexní simulaci procesoru a celého paměťového systému.
- Víme, že přístupy do paměti jsou jednotlivá čtení a zápisy…

Memory-stall cycles = Read-stall cycles + Write-stall cycles

Zpoždění při čtení a zápisu - vzorce

Zpoždění při čtení a zápisu

- Zpoždění při čtení lze snadno určit..

read-stall cycles = reads/program * read miss rate * read miss penalty

- Zpoždění vlivem zápisů se určuje obtížněji. Předpokládáme-li strategii write-through, existují dvě příčiny. Výpadek při zápisu (když blok musí být před zápisem načten) a zpoždění dané činností zápisových bufferů (je-li zápisový buffer zaplněn)…

write-stall cycles = (writes/program * write miss rate * write miss penalty) + write buffer stalls

Zpoždění při zápisu

Zpoždění při zápisu

- Zpoždění při zápisech vlivem zápisových bufferů závisí na jejich časování i na jejich frekvenci. Původní jednoduchá rovnice pak ztrácí na přesnosti. Naštěstí dostatečně dlouhé zápisové buffery a dostatečně rychlá paměť podstatně snižují počet zápisových zpoždění a lze je proto ignorovat (kdyby tomu tak nebylo, znamenalo by to špatný návrh paměťového systému).
- Navíc strategie write-back může přinášet další zpoždění způsobená nutností zápisu bloku zpět do hlavní paměti při výměně.

Zobecněná rovnice výkonu

Zobecněná rovnice výkonu

- Rovnici můžeme zjednodušit, jsou-li „pokuty“ za výpadek při čtení i zápisu stejné (což obvykle platí; doba potřebná k načtení bloku z nižší úrovně)…

memory-stall cycles = memory accesses/program * miss rate * miss penalty

- Lze také psát …

memory-stall cycles = instructions/program * misses/instruction * miss penalty

Výkon v cache, výpočet, příklad

Příklad: Výpočet výkonu cache

- Předpokládejme, že procesor má instrukční cache s četností výpadků 2% a datovou cache s četností výpadků 4%.
- Procesor má CPI = 2.0 bez zpoždění vlivem výpadků paměti a „pokuta“ za výpadek je 40 cyklů za každý.
- Instrukce Load a Store tvoří 36% z celého instrukčního toku. Žádné jiné instrukce přístup do paměti nevyžadují.
- Kolikrát pomalejší je tento systém než ten, u kterého by nevznikaly žádné výpadky cache?

Příklad (řešení):Výpočet výkonu cache

- Počet ztracených cyklů při výpadcích při čtení instrukcí je roven:
  I * 2% * 40 cyklů = 0.80I
- Počet cyklů při výpadcích kvůli datům je roven
  I * 36% * 4% * 40 cyklů = 0.56I
- Proto je celkový počet cyklů při výpadcích roven 1.36I, což je více než jeden cykl na provedenou instrukci.
  - Parametr CPI díky „pokutám“ vzroste na 3.36.
- Poměr výpočetních dob CPU je roven…

Řešení
Proto cache bez výpadků je 1.68 krát rychlejší než reálná cache.

$$\frac{CPUtime_with_stalls}{CPUtime_without_stalls} = \frac{I \times CPI_{stall} \times ClockCycle}{I \times CPI_{perfect} \times ClockCycle} = \frac{CPI_{stall}}{CPI_{perfect}} = \frac{3.36}{2} = 1.68$$

Vztah výkonu procesoru a paměti, pokuty

Vztah výkonu procesoru a paměti

- Zachováme-li vlastnosti paměťového systému a procesor zrychlíme, relativní ztráta se zvýší.
  - Klesne-li CPI, zvýší se vliv „pokuty“ za výpadky.
  - Doba cyklu procesoru se zlepšuje rychleji než paměť (!historie). „Pokuta za výpadek se měří v počtu cyklů CPU na jeden výpadek (miss).
  - Jestliže paměti dvou systémů mají stejné přístupové doby, stroj, jehož CPU má rychlejší hodiny, vykazuje také větší „pokutu“ za výpadek.

Příklad vztah výkonu procesoru a paměti, řešení, výpočet

Příklad: Vztah výkonu procesoru a paměti

- Předpokládejme počítač z předchozího příkladu. Procesor bude pracovat na dvojnásobné frekvenci, „pokuta“ za výpadek cache zůstane stejná.
- Jak rychlý bude celý počítač, předpokládáme-li stejnou četnost výpadků?
- Poznámka:
  Jestliže neuvažujeme vliv cache a paměti, nový stroj bude mít dvojnásobnou rychlost než předchozí.

Příklad: Vztah výkonu procesoru a paměti

- Nová „pokuta“ za výpadek je 80 hodinových taktů.
- Počet cyklů výpadku na instrukci …

  (2% * 80) + 36% * (4% * 80) = 2.75
- Proto nový počítač má CPI = 4.75 v porovnání s CPI = 3.36 pomalejšího stroje.
- Použitím podobného vzorce jako v předchozím případě lze vypočítat relativní výkon…

$$\frac{\text{Výkon(rychlé_hodiny)}}{\text{Výkon(pomalé_hodiny)}} = \frac{I \times CPI_{fast} \times \text{ClockCycle}}{I \times CPI_{slow} \times \frac{\text{ClockCycle}}{2}} = \frac{3.36}{4.75 \times \frac{1}{2}} = 1.41$$

- Počítač s dvojnásobnou hodinovou frekvencí je 1.41 krát rychlejší. Kdyby neexistovaly výpadky cache, byla by jeho rychlost dvojnásobná!

Cache s flexibilnějším umísťováním bloků, přímo mapovaná cache, plně asociativní cache

Cache s flexibilnějším umísťováním bloků

- Dosud jsme uvažovali *přímo mapované cache* - každý blok lze umístit do jediného místa v cache.
- To je jedna krajní varianta ze všech rozličných strategií pro umísťování bloků.
- Opačným extrémem je *plně-asociativní* cache – u tohoto typu mechanizmu cache, blok z hlavní paměti může být umístěn do libovolného místa v cache.
  - Jestliže umožníme umístění bloku do libovolného místa v cache, budeme jej také muset umět v libovolném místě nalézt. Prohledání celé cache.

Cena plně asociativní cache

Cena plně asociativní cache

- Aby takový mechanizmus byl prakticky použitelný, musí hledání bloku ve všech místech probíhat paralelně.
- Každý vstupní bod cache (místo pro blok) obsahuje odpovídající komparátor – tyto komparátory podstatně zvyšují cenu hardware této organizace cache. Uvedená strategie je vhodná pro cache s malým počtem vstupních bodů.

Částečně asociativní cache – n-cestná částečně asociativní cache

Částečně asociativní cache

- Praktičtější řešení představují *částečně asociativní cache*, které představují kompromisní řešení mezi přímo mapovanou a plně asociativní organizací.
- U tohoto typu cache existuje pevný počet míst (2 nebo více) kde může být daný blok umístěn. Existuje-li pro každý blok n možných pozic, nazývá se taková organizace *n-cestná částečně asociativní* cache.
  - n-cestná částečně asociativní cache se skládá z velkého počtu skupin bloků, z nichž každá obsahuje n bloků.
  - Každý blok v paměti se mapuje na určitou skupinu bloků v cache; může být uložen do libovolné pozice v této korespondující skupině bloků.

Vícecestná asociativní cache

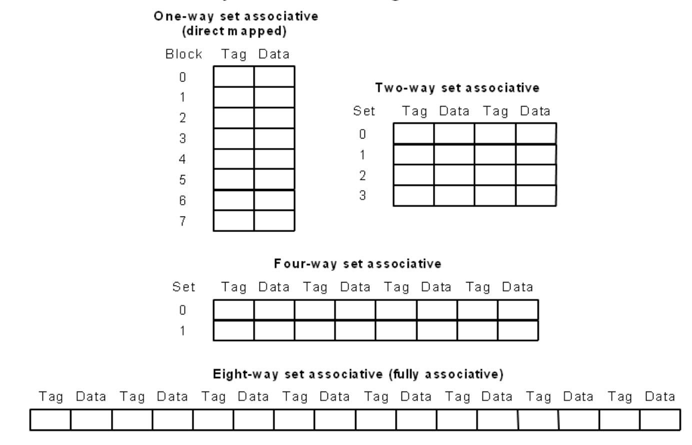
Zpět k vícecestné částečně asociativní cache

- Na každou strategii pro umísťování bloků lze nahlížet jako na variaci vícecestné paměti…
  - Přímo mapovaná cache je vlastně jednocestná částečně asociativní cache; každý vstupní bod uchovává jeden blok (skupina bloků o velikosti 1).
  - Plně asociativní cache s m vstupními body je vlastně m-cestná „částečně“ asociativní cache; má jen jednu skupinu bloků a v ní se může kdekoliv nacházet libovolný blok z hlavní paměti.

Různé organizace paměti

Různé organizace cache

- Cache s 8-bloky můžeme organizovat…



Vlastnosti různých organizací cache – DECStation3100

Vlastnosti různých organizací cache

- Nahlédneme na některé reprezentativní statistiky dvou programů, abychom posoudili, jak zvýšená míra asociativity zlepšuje situaci (organizace podobná DECStation 3100 cache s velikostí bloku 4 slova…

Program	Associativity	Instruction Miss Rate	Data Miss Rate	Eff. Combined Miss Rate
gcc	1	2.0%	1.7%	1.9%
gcc	2	1.6%	1.4%	1.5%
gcc	4	1.6%	1.4%	1.5%
spice	1	0.3%	0.6%	0.4%
spice	2	0.3%	0.6%	0.4%
spice	4	0.3%	0.6%	0.4%

- Zvýší-li se asociativita z 1 na 2, gcc výpadky se redukují o ~20%. Spice již tak nízkou četnost má, tady uvedená změna příliš nepomůže. Změna asociativity ze 2 na 4 již nic nevylepší.

Nalezení bloku u částečně asociativní cache, prohledávání paralelně

Nalezení bloku u částečně asociativní cache

- Podobně jako u přímo mapované cache, každý blok ve vícecestné cache obsahuje tag, který určuje adresu bloku.
- Každá adresa do maměti je rozdělena na tři části…

Tag	Index	Block Offset

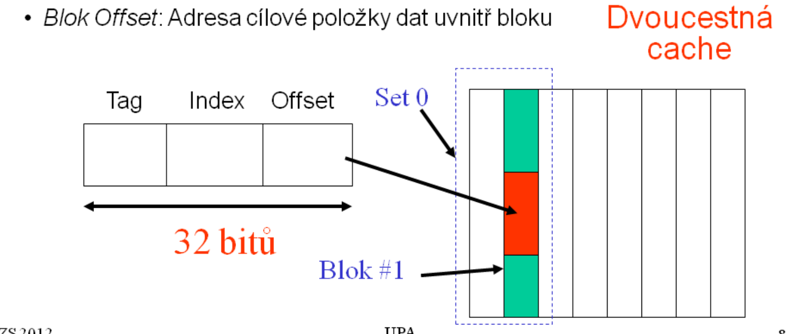
- Hodnota indexu slouží k výběru skupiny, do které blok patří. Tag se komparuje se všemi tagy bloků ve skupině, aby se zjistilo, zda daný vstupní blok obsahuje hledaný blok.
- Rychlost je podstatná – všechny tagy ve vybrané skupině se kontrolují paralelně !!!

Částečně asociativní mapování – výhody nevýhody

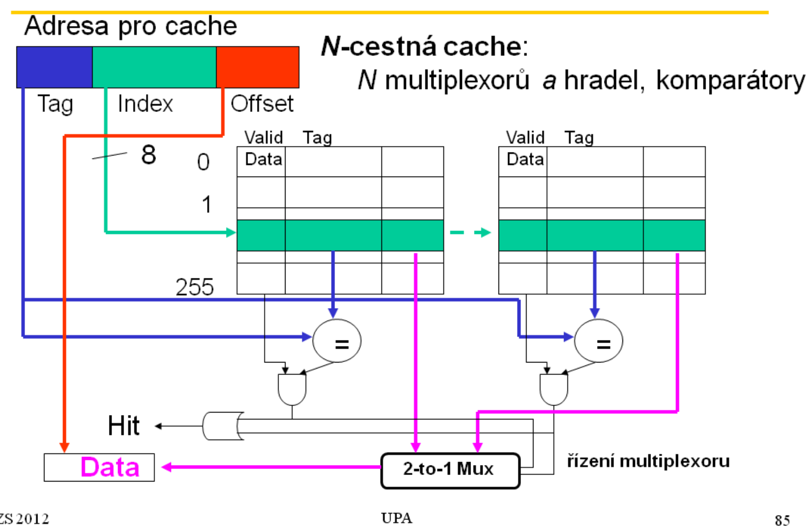
Částečně asociativní mapování

- **Zobecňuje všechna mapovací schémata cache**
  - Předpokládejme, že cache obsahuje *N* bloků
  - 1- cestná částečně asociativní cache: Přímé mapování
  - *M*-cestná částečně asociativní cache: Je-li M = N, pak se jedná o plně asociativní cache
- **Výhoda**
  - Zvyšuje úspěšnost – klesá „miss rate“ (více míst, kde lze nalézt B)
- **Nevýhoda**
  - Narůstá „hit time“ (více míst, kde je třeba hledat B)
  - Složitější hardware

Činnost částečně asociativní cache, dvoucestná cache



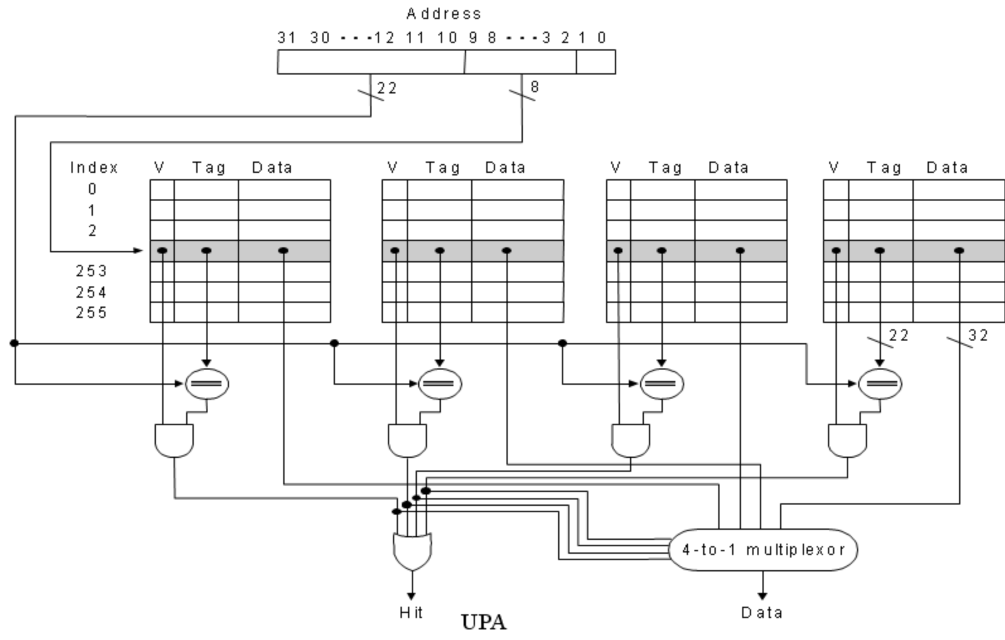
Příklad hardware dvou cestná cache, n-cestná cache - schéma



Nalezení bloku u 4-cestné cache

Nalezení bloku u 4-cestné cache

- Příklad (4-cestná částečně asociativní cache)...



Návrh strategie cache- metriky cena a čas

Návrh strategie cache

- Rostoucí asociativita cache vyžaduje více hardware. Jak se má návrhář vypořádat s volbou strategie umísťování bloků v cache?
- Je třeba zvážit cenu výpadku oproti ceně za vyšší míru asociativity u jednotlivých organizací (přímo-mapovaná, n-cestná a plně-asociativní).
  - Dvě různé metriky: čas a cena
  - Ke které se obrátíte? Rozpočet cache?

Strategie výměny bloků

Strategie výměny bloků

- Nastane-li výpadek u přímo mapované cache, požadovaný blok může být zapsán do jediného místa a blok, který toto místo zaujímá musí být vyměněn.
- U asociativní cache je na výběr, kam bude požadovaný blok zapsán.
  - U plně asociativní cache jsou všechny bloky přítomné v cache kandidáty na výměnu.
  - U n-cestné částečně asociativní cache, všechny bloky korespondující skupiny jsou kandidáty na výměnu.

Algoritmus LRU – least recently used

Algoritmus LRU

- Existuje celá řada strategií jak vybrat blok vhodný pro výměnu (z kandidátů), které lze použít v případě výpadku.
- Velmi často používaným algoritmem je *least recently used (LRU)*.
- Nahrazován je blok, který nejdéle nebyl použit (využívá myšlenku časové lokality).
- LRU strategie se implementuje tak, že sledujeme relativní použití oproti ostatním členům skupiny.

Implementace LRU

Implementace LRU

- U dvoucestné částečně asociativní cache je sledování jednoduché.
  - Ke každému vstupnímu bodu se přidá jeden bit. Kdykoliv je blok referencován, je příslušný bit daného bloku nastaven a odpovídající bit druhého bloku nulován.
  - Nastane-li výpadek a musí dojít k výměně bloků, dojde k ní u bloku, jehož bit je vynulovaný. Bit nového bloku je pak nastaven.
- Situace je poněkud složitější, obsahuje-li skupina více bloků (n = 4, 8).
- Zmíníme se o jiných strategiích výměny. (Jedná se o poměrně složitou problematiku, kterou se nebudeme zabývat do hloubky).

Víceúrovňové cache, L1, L2, L3

Víceúrovňové cache

- Všechny moderní počítače používají cache paměti.
- Většina novějších počítačů používá víceúrovňové cache paměti. Dnešní systémy mívají obvykle dvě úrovně cache (L1 a L2).
- Výkonné systémy dokonce 3 úrovně (přidána L3) – Pentium 4 Extreme Edition má 2MB L3 cache.
- Podívejme se na funkci takového víceúrovňového systému.

Hierarchie cache

Hierarchie cache

- V takovém systému se do L2 cache vykoná přístup, jestliže nastane výpadek v L1 cache.
- Do L3 cache vykoná přístup, jestliže nastane výpadek v L2 cache.
- Teprve v případě, že dojde k výpadku na všech úrovních cache, vykoná se přístup k nejpomalejší technologii - k hlavní paměti.
- V této hierarchii L1 cache je nejmenší a nejrychlejší ze všech pamětí. L2 cache je větší, ale pomalejší. L3 cache je ještě větší a ještě pomalejší. Hlavní paměť je největší a nejpomalejší ze všech.

Hit time

Uvažujeme-li „Hit Time“

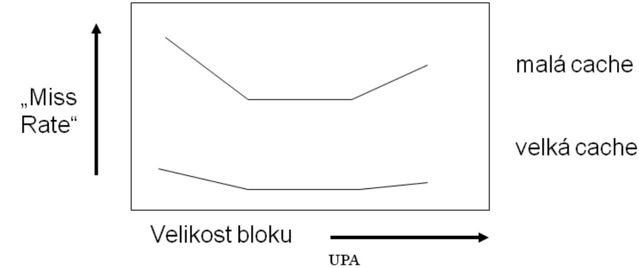
- Dosud jsme opomíjeli „hit time“, množství času, které cache potřebuje ke zjištění, že se jedná o hit.
- Tato doba není nulová, protože cache musí porovnat svůj obsah s požadavkem, zjistit, zda je požadované slovo přítomno v cache. Z tohoto důvodu hit time narůstá, zvětšuje-li se kapacita cache.
- V určitém bodě tento nárůst převládne nad vlivem zlepšující se četnosti výpadků cache. (větší neznamená vždy lepší!).

Cena za Cache miss

Cena za „Cache Miss“

***Příklad:* „Miss“ v instrukční cache** (instrukce v cache nenalezena)

- **Pozorování:** Větší bloky se načítají pomaleji
- **Operace:** Větší bloky využívají lépe *prostorovou lokalitu*
- **Co vybrat?** **Řešení: Cache co možná největší, to omezí vliv velikosti bloku**



Návrh víceúrovňové cache

Návrh víceúrovňové cache

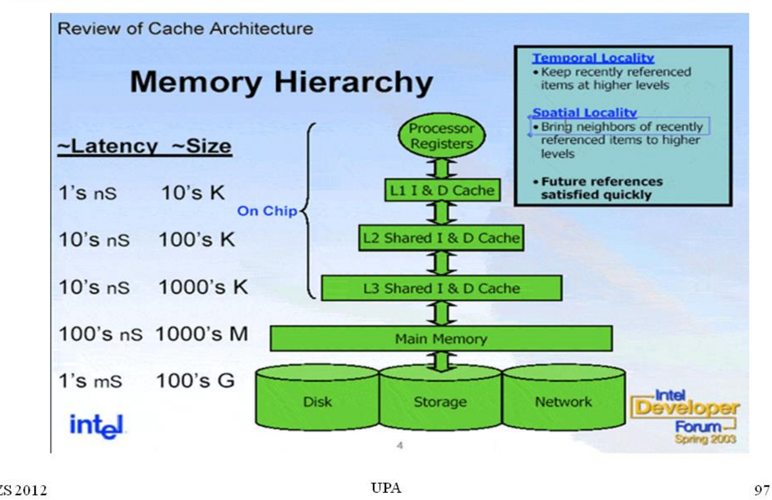
- U víceúrovňových cache může být L1 cache malá, aby se minimalizoval parametr „hit time“, zatímco L2 cache může být velká tak, aby byla příznivá četnost výpadků (!nízká). Tím se maskuje poměrně dlouhá přístupová doba do hlavní paměti.
  - Cena za miss v L1 cache se drasticky redukuje přítomností L2 cache, což dovoluje implementovat L1 poměrně malou (! ale rychlou !) s větší četností výpadků.
  - Doba přístupu L2 cache není tolik rozhodující, protože ovlivňuje „pokutu“ za miss v L1 víc, než přímo L1 „hit time“ nebo hodinovou frekvenci CPU.

Příklad Pentium 4 Extreme Edition

Příklad: Pentium 4 Extreme Edition

- Pentium 4 Extreme Edition má tříúrovňovou cache…
  - 8KB L1 cache – Jedná se o 4-cestnou částečně asociativní cache s přenosovou rychlostí 23295 MB/sec. při čtení
  - 512KB L2 cache – Jde o 8-cestnou částečně asociativní cache s přenosovou rychlostí 12920 MB/sec. při čtení.
  - 2MB L3 cache – Jde o 8-cestnou částečně asociativní cache s přenosovou rychlostí 6522 MB/sec. při čtení.

Příklad: Pentium 4 Extreme Edition



Virtuální paměť virtual memory

Nová látka: Virtuální paměť (VM)

Pozorování: *Cache vyrovnává rychlosti procesoru a hlavní paměťi*

…Hlavní paměť je „cache" pro diskovou paměť…

Upřesnění:

- VM umožňuje efektivní a bezpečné sdílení paměti mezi více programy (podpora multiprogramového zpracování)
- VM odstraňuje potíže s malým rozsahem fyzické paměti
- VM zjednodušuje *zavádění* programů podporou *relokace*

Historie: Nejdříve byla vyvinuta VM, teprve pak cache.

Cache je založena na technologii VM, ne naopak.

Virtuální paměť pojmy, TLB, stránky, fyzická adresa, virtuální adresa, mapování, translace

Virtuální paměť - pojmy

Stránka (Page): *Blok* ve virtuální paměti (cache = blok, VM = page)

Výpadek stránky: Neúspěch při operaci MemRead (cache = miss, VM = výpadek stránky)

Fyzická adresa: Adresa mířící do fyzické paměti

Virtuální adresa: Určena CPU, odpovídá *velkému* adresnímu prostoru, je transformována pomocí HW + SW na fyzickou adresu

Mapování paměti nebo **translace adresy**:

Proces transformace virtuální adresy na fyzickou adresu

Translation Lookaside Buffer (TLB):

Zvyšuje efektivitu procesu transformace adresy

Fyzická vs virtuální paměť

Fyzická vs. virtuální paměť

- **Fyzická paměť**
  - *Instalována v počítači:* ~ 512MB – 4 GB RAM v PC
  - Limitována velikostí a spotřebou
  - *Potenciální rozšíření limitováno velikostí adresního prostoru*
- **Virtuální paměť**
  - **Jak uložit 2^{32} *slov* do vašeho PC?**
  - **Nikoliv jako RAM – nedostatek prostoru nebo napájení**
  - Nechť CPU „věří“ , že má k dispozici 2^{32} slov
  - Hlavní paměť pak pracuje jako pomalejší cache
  - *Stránka* - jednotka pro přesun obsahu paměti na/z disk(u)
  - Jednotka Memory Management Unit využívá tabulku stránek (adresovací systém) při řízení přesunu stránek z/do hlavní paměti

Motivace virtuální paměti

Motivace

- Předpokládejme soubor programů, běžících na počítači současně.
  - Celková kapacita paměti, kterou tyto programy potřebují může být mnohemmvětší, než je celá kapacita hlavní paměti stroje.
  - V daném okamžiku se využívá jenom malý zlomek celé kapacity hlavní paměti.
  - Hlavní paměť musí obsahovat pouze aktivní části všech programů – působí vlastně jako cache.
  - Aby takový systém pracoval úspěšně, musíme zaručit, aby každý program prováděl čtení/zápis jen v oblasti, která přísluší právě jemu (separace).

Virtuální paměti a adresní prostor, systém virtuální paměti

Virtuální paměti a adresní prostor

- Jakmile je program přeložen, nevíme jaké programy budou zpracovávány současně (navíc se toto může při každém spuštění měnit).
- Proto je každý program překládán jakoby pro práci ve svém vlastním *adresním prostoru*. To je oddělený úsek paměti, dostupné jen tímto programem.
- *Systém virtuální paměti* (VM) počítače implementuje translaci z adresního prostoru programu do fyzického adresního prostoru.

Další motivace virtuální paměti

Další motivace

- Další motivací pro virtuální paměť je možnost, aby i jediný program mohl překročit svou velikostí kapacitu fyzické paměti.
- Paměť, která se nepoužívá, může být přenesena na disk (*swapped* to disk) a opět načtena zpět, pokud systém virtuální paměti rozhodne (podobně jako je tomu mezi cache a hlavní pamětí).
- Virtuální paměť automaticky spravuje dvě úrovně hierarchie paměti, reprezentované hlavní/fyzickou pamětí a sekundární pamětí (obvykle disk).

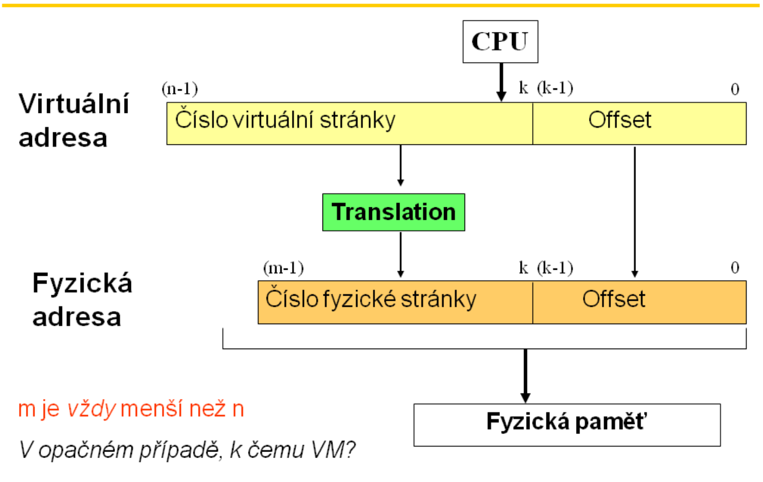
Overlaye

Overlaye

- Před příchodem mechanizmu virtuální paměti (~ 1960 Manchester) museli programátoři přímo organizovat využívání sekundární paměti tak, že rozdělili svůj program na úseky, které pak střídavě přenášeli pro zpracování do hlavní paměti počítače.
- Program byl rozdělen na části nazývané „overlaye“, které byly programem (nazývaným overlay manager) načítány do paměti a zpět na disk.
  – Každý overlay byl typicky modul, obsahující program i data.
- Součet všech „overlayů“ v paměti musel být menší než velikost fyzické paměti.
- To byl podstatný nedostatek a překážka pro každého programátora.

Činnost virtuální paměti schéma

Činnost virtuální paměti



m je *vždy* menší než n

V opačném případě, k čemu VM?

Cena virtuální paměti, latence, redukce ceny vm, chytré softwarové algoritmy

„Cena“ virtuální paměti

- **Výpadek stránky:** Načtení dat z *disku* ☹ **(*latence* - milisekundy)**
- **Redukce „ceny“ VM :**
  - » Velikost stránek, aby se amortizovala dlouhá doba přístupu na disk
  - » Plně asociativní mechanizmus umísťování stránek, aby se snížila *četnost výpadků.*
- Obsluha výpadku stránek v software, protože mezi přístupy na disk je dost času, v porovnání s cache *(která je mnohonásobně rychlejší)*
- Použít *chytré softwarové algoritmy* pro umísťování stránek (je čas!)
  - » **Náhodné** umístění stránek **(Random)** se nepoužívá
  - » **Least Recently Used** je mnohem obvyklejší varianta

Terminologie, výpadek stránky, virtuální adresa, fyzická adresa, dynamická transformace adresy, translace adresy

Terminologie

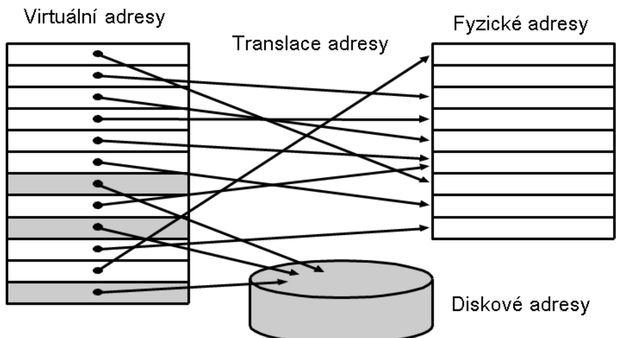
- Virtuální adresní (logický) prostor se dělí na *stránky*.
- VM miss se nazývá *výpadek stránky*.
- CPU vytváří *virtuální adresu* která je transformována kombinací hardware a software na *fyzickou adresu*, která je pak použita k přístupu do hlavní paměti.
- Tento proces se nazývá *dynamická transformace adresy* nebo *translace adresy*.
- Současné fyzické hlavní paměti jsou typicky dynamické paměti, jako sekundární paměť se používají magnetické paměti nebo flash technologie.

Relokace, realokace - schéma

Relokace

- VM také zjednodušuje načítání programů s *relokací*. Mapuje virtuální adresy užívané programem na jiné fyzické adresy – to dovoluje umístit program do libovolného místa v hlavní paměti.
- Všechny současné systémy VM také relokují program jako soubor stránek. Pro načtení programu se pak nepotřebuje souvislý prostor v hlavní paměti; potřebujeme pouze dostatečný počet stránek ve fyzické paměti a vlivem systému VM jsou tyto stránky spojeny do souvislého prostoru (v logickém prostoru), i když tomu tak ve fyzické paměti není.

Relokace



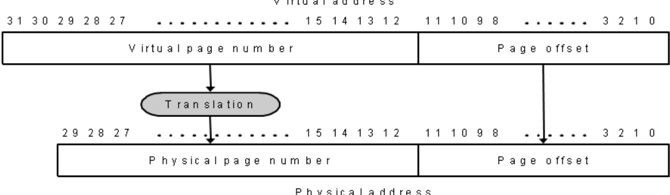
- Tento program obsazuje spojitý prostor ve virtuálním adresním prostoru, nikoliv však ve fyzickém.

Translace adresy, převod adresy, příklad translace adresy

Translace adresy

- Jak se provádí translace adresy z virtuálního adresního prostoru do fyzického?
- Každá adresa se rozdělí na pole „virtuální číslo stránky“ a pole „offset“ (adresa na stránce).
  - Virtuální číslo stránky se transformuje na fyzické číslo stránky.
  - Offset zůstává bez změny.
- Počet stránek virtuálního prostoru není obvykle roven počtu stránek fyzické paměti, bývá mnohem větší.
  - Velký počet virtuálních stránek (v porovnání s fyzickými) vytváří iluzi téměř neomezeného virtuálního prostoru.

Příklad translace adresy



- Velikost stránky je 2^{12} = 4K.
- Počet fyzických stránek je 2^{18}.
- Počet virtuálních stránek je 2^{20}.
- Fyzická paměť může mít nejvýše kapacitu 1GB, zatímco virtuální adresní prostor je 4GB.

Úvahy při návrhu VM, page fault, výpadek stránek, write through, write back

Úvahy při návrhu

- Většina virtuálních paměťových systémů je navržena tak, aby se omezily výpadky stránek (page fault) – každý výpadek znamená přístup na disk, jehož zpracování trvá miliony cyklů procesoru.
- To vede při návrhu na celou řadu klíčových rozhodnutí…
  - Velikost stránky by měla být volena tak, aby se redukovala dlouhá doba přístupu. Typická velikost stránek je dnes 32K a 64K. Toto rozhodnutí je motivováno prostorovou lokalitou.

Úvahy při návrhu

- Redukce četnosti výpadků stránek má prioritu číslo jedna. Podporuje ji volnost při umísťování stránek.
- Výpadky stránek jsou typicky ošetřovány v software - pro umísťování stránek mohou být použity sofistikované algoritmy, jejichž použití je na úrovni hardware obtížné (v porovnání s výpadkem bloku u cache je více času).
- Techniky Write-though pro VM nelze použít, protože zápis trvá příliš dlouho. Systémy VM používají techniku Write-back.

Segmentace

Segmentace

- Alternativou pro stránkování je *segmentace*.
- Virtuální adresa se člení na dvě části - číslo segmentu a offset (adresa uvnitř segmentu).
- Každé číslo segmentu odkazuje na segmentový registr, který obsahuje fyzickou adresy segmentu. K tomuto pointeru je přičten offset a tak získáme aktuální adresu do fyzické paměti.
- Segmenty jednoho systému mohou mít různou velikost.
- Segmentace rozděluje adresu do dvou logicky oddělených částí, zatímco stránkování dělá hranici mezi číslem stránky a offsetem neviditelnou pro programátory a pro kompilátor.

Nyní zpět na stránkování…

Umístění stránky

Umístění stránky

- Jak se umístí stránka do fyzické paměti?
- Umístění stránky je možné do libovolného místa a kam a kdy se stránka do hlavní paměti zapíše určuje operační systém.
  - To zajišťuje sofistikovaný algoritmus, součást OS, který sleduje využití stránky a naplánuje její přenos do vnější paměti, pokud se stránka delší dobu nepoužívá.
  - Volné umístění stránek poskytuje OS velkou flexibilitu při umísťování nové stránky.
  - Redukuje také četnost výpadků stránek.

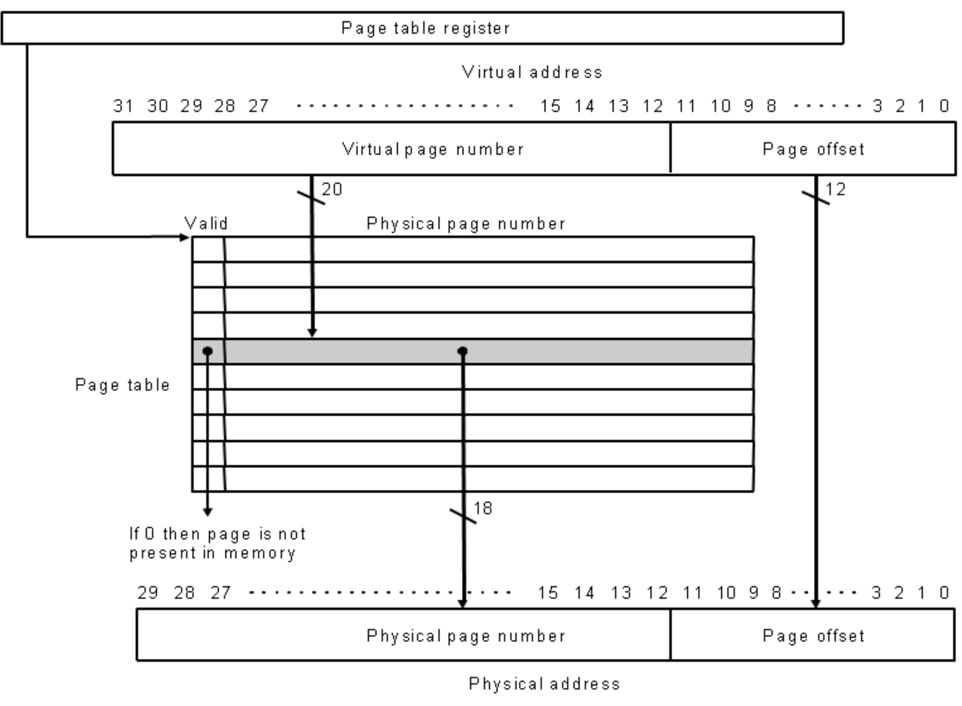
Nalezení stránky ve fyzické paměti, tabulka stránek page table

Nalezení stránky ve fyzické paměti

- Jak je stránka nalezena, když se již nachází v hlavní paměti?
- Úplné prohledávání (analogie plně asociativní cache) není praktické.
- Ve virtuální paměti je na stránky přistupováno pomocí úplné tabulky, která indexuje paměť a nazývá se *tabulka stránek* (*page table*).
- Tabulka stránek je umístěna v paměti, vybírá se z ní pomocí čísla virtuální stránky a nalezená položka obsahuje index stránky do fyzické paměti.
  - Každý program může mít svoji vlastní tabulku pro svůj virtuální adresní prostor.

Tabulka stránek schéma

Tabulka stránek



Tabulka stránek, registr stránkové tabulky, bit platnosti

Tabulka stránek

- Aby mechanizmus pracoval, musíme být schopni nalézt tabulku stránek. Proto je adresa počátku tabulky uložena do *registru stránkové tabulky* (*page table register*) programu. Sama tabulka leží v souvislé oblasti paměti.
- Stránková tabulka úplně mapuje virtuální adresní prostor a obsahuje mapování pro každou možnou virtuální stránku. Nepotřebujeme tagy.
- Bit platnosti má stejnou funkci jako v cache - je-li nulový, platná stránka není ve fyzické paměti a nastává výpadek stránky.

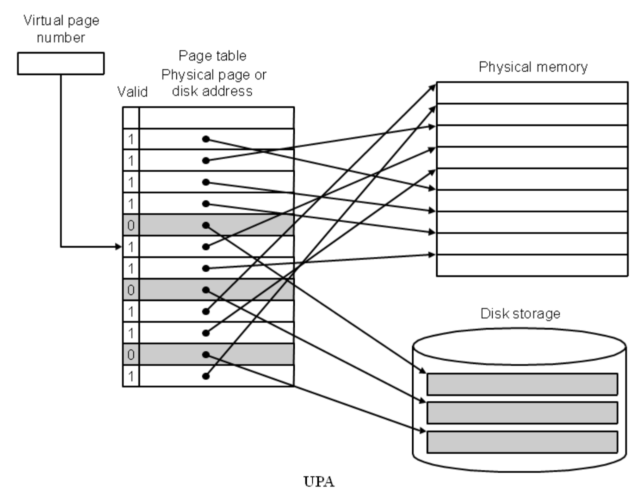
Ošetření výpadku stránky, rozdíl oproti cache

Ošetření výpadku stránky

- Co se stane při výpadku stránky?
- Operačnímu systému je předáno řízení, ten musí najít stránku na disku a umístit ji do hlavní paměti.
  (rozdíl oproti cache !)
- Předání řízení a výměna se zajišťuje s využitím výjimek.
- Přirozeně je nutné udržovat informaci o poloze každé stránky na disku.

Výpadek stránky a tabulka stránek

Výpadek stránky a tabulka stránek



Výpadek stránky a disk

Výpadek stránky a disk

- Nevíme dopředu, kdy se bude stránka v paměti vyměňovat – OS obvykle vytváří stránku na disku pro všechny stránky procesu, když jej vytváří (*proces* je běžící verze programu).
- Ve stejné době vytváří datovou strukturu pro uložení každé virtuální stránky na disku.
  - V našem příkladu stránková tabulka obsahuje jak fyzickou adresu v paměti, tak adresu na disku.
  - Může to ale být i oddělená datová struktura, která existuje vedle tabulky stránek.

Výpadek stránky a disk – page descriptor, kroky

Výpadek stránky a disk

- Položka v tabulce stránek se nazývá *page descriptor*. Obsahuje celou řadu polí, která obsahují informace o dané stránce a využívají se pro organizaci výměny stránek.
  1. Aktivita stránky (jednobitový příznak)
  2. Pointer na počátek stránky ve fyzické paměti
  3. Poloha stránky ve vnější paměti
  4. Dirty bit (příznak modifikace) – (jednobitový příznak)
  5. Pole pro algoritmus výměny stránek (např. LRU)
  6. Pole ochrany proti nedovolenému přístupu
  7. …

Implementace LRU

Implementace LRU

- Předpokládejme, že systém VM používá algoritmus (LRU). OS vytvoří datovou strukturu, obsahující údaje, který proces užívá které fyzické stránky. Tato struktura také udržuje historii o přístupech na stránky, která je pak využita algoritmem LRU.

- Často se jedná jen o aproximaci (skutečný LRU by vyžadoval aktualizaci při každém přístupu).
  – Každá stránka používá referenční bit, který se nastaví po každém přístupu na stránky (R nebo W). OS periodicky nuluje tyto referenční bity – to dovoluje zjistit, na které stránky byl učiněn přístup během určitého časového intervalu.

Velikost tabulky- výpočet

Velikost stránkové tabulky

- Zabývejme se tím, jaké velikosti může tabulka nabýt.
- Předpokládejme, že máme 32-bitovou virtuální adresu, 4K stránky a 4 byty se vyžadují na jeden vstupní bod do tabulky stránek => můžeme spočítat celkovou velikost tabulky stránek…

$$PageTableEntries = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$PageTableSize = 2^{20} PageTableEntries \times 2^{2} \frac{bytes}{PageTableEntries} = 4MB$$

- Potřebujeme tedy 4MB paměti pro každý aktivní program. Tento přístup se nejeví příliš realistický.

Omezení velikosti tabulky stránek

Omezení velikosti tabulky stránek

- Existuje celá řada metod, jak omezit velikost stránkové tabulky…
- Nejjednodušší technikou je použít registr, který je naplněn limitem velikosti tabulky pro každý proces.
  - Jestliže počet virtuálních stránek narůstá a překračuje limit, roste i tabulka stránek. To dovoluje přizpůsobit velikost tabulky požadavkům procesu.
  - Toto uspořádání vyžaduje, aby virtuální adresní prostor expandoval jenom jedním směrem.

Složitější techniky, zásobník, halda

Složitější techniky

- Uvedená metoda není právě optimální. Potřeba nalezení efektivnějších přístupů.
  - Většina systémů obsahuje dvě oblasti, které musí expandovat, *zásobník* a *„halda“ (heap)*.
  - Některé systémy používají dvě oddělené rostoucí tabulky. Jednu, která se zvětšuje směrem nahoru a druhou, která se zvětšuje směrem dolů.
  - Využívají se dvě stránkové tabulky segmentů (jedná se o trochu jiný model segmentace, než o kterém již byla řeč). Nejvyšší bit adresy určuje která tabulka bude použita.
  - Toto uspořádáni pracuje velmi dobře ve většině případů. Neosvědčilo se v případě, kdy byly do virtuálního adresního prostoru činěny jednotlivé přístupy, namísto přístupů do kontinuální množiny adres…

Víceúrovňové stránkové tabulky

Víceúrovňové stránkové tabulky

- Jinou metodu představují víceúrovňové stránkové tabulky.
  - První úroveň mapuje velké bloky fixní velikosti (také někdy nazývané segmenty). Tabulka první úrovně se také nazývá segmentová tabulka.
  - Segmentová tabulka udává, zda se nějaké stránky daného segmentu nacházejí v paměti. Když ano, obsahuje pointery na dané stránky segmentu.
  - Umožňuje to „řídké“ využití virtuálního adresního prostoru (násobné segmenty). Není třeba alokovat celou stránkovou tabulku. Systém se osvědčil u velmi velkých adresních prostorů s požadavky na nespojitou alokaci paměti.
  - Přístup do takové paměti je komplikovanější.

Stránkování stránkovaných tabulek,

Stránkování stránkových tabulek

- Většina moderních systémů dovoluje, aby byly *stránkovány* i *stránkové tabulky*.
- Využívá se stejného mechanizmu virtuální paměti. Stránkové tabulky jsou rovněž umístěny ve virtuálních adresních prostorech.
- Mohou ale vznikat nekončící rekurzivní posloupnosti výpadků stránek.
- Řešení jsou hardwarově-závislá a složitá. Proto jsou obvykle stránkové tabulky umístěny v adresním prostoru OS (kernel memory), v části paměti, která je stále v paměti přítomná a nepodléhá stránkování.

Zápis do paměti, write back, write through

Zápis do paměti

- Systémy virtuální paměti používají strategii write-back pro zápis do paměti.
  - Rozdíl v době přístupu mezi cache a hlavní pamětí jsou desítky cyklů.
  - Rozdíl doby přístupu do hlavní paměti a na disk jsou miliony cyklů.
  - Strategie write-through je nepoužitelná.

Zápisy do paměti a dirty bit

Zápisy do paměti a dirty bit

- Přináší další výhodu pro systém virtuální paměti.
  - Zápisy stránky na disk představují z hlediska cyklu procesoru velmi dlouhou dobu.
  - Vyplatí se uchovávat informaci, zda je nutné provést zpětný zápis stránky na disk v případě její výměny, nebo zda lze tuto diskovou operaci vynechat.
  - K tomu je určen *dirty bit*. Tento jednobitový příznak je nastaven v případě, že během přítomnosti stránky v hlavní paměti je na ni učiněn zápis. V případě výměny stránek, nebylo-li na stránku zapisováno, znamená to, že stránka v hlavní paměti je totožná s její kopií v hlavní pamětí. Výměna pak spočívá jen v načtení nové stránky.

Rychlá transformace adresy – využití principu lokality

Rychlá transformace adresy

- Tabulky stránek jsou uloženy v hlavní paměti – to znamená, že přístup do paměti trvá dvojnásobnou dobu (jeden přístup pro získání fyzické adresy, druhý pro data).
- Klíčem pro zlepšení výkonu je *využití principu lokality* referencí do paměti – je-li jednou transformována adresa určité virtuální stránky, je velmi pravděpodobné, že výsledek transformace bude v zápětí opět třeba. Časová i prostorová lokalita.

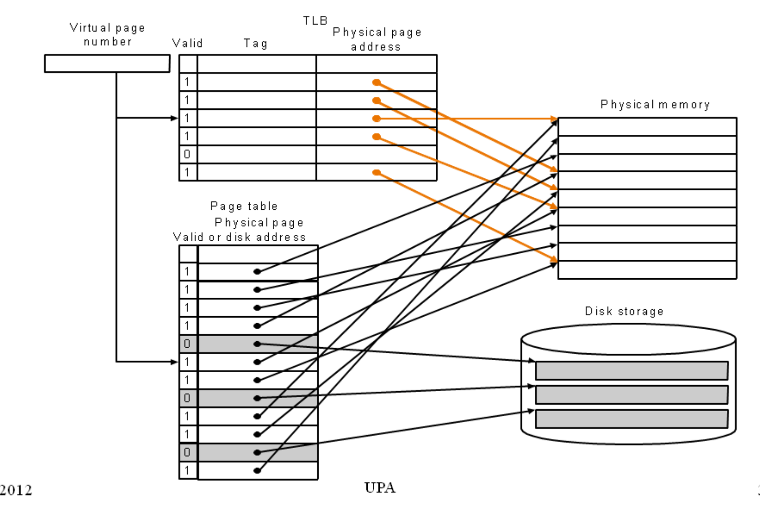
Translation lookaside buffer - TLB

Translation-Lookaside Buffer (TLB)

- Většina moderních systémů používá speciální cache, ve které jsou uloženy výsledky naposledy prováděných transformací adresy – nazývá se *translation-lookaside buffer* (TLB).
- TLB je cache, která uchovává mapování stránek – každý vstup uchovává virtuální číslo stránky (tag) a číslo fyzické stránky (data uložená v této cache).
- Při přístupu do paměti se nejdříve provede přístup do TLB, nezačíná se přístupem do stránkové tabulky. Ta je použita jen v případě, že hledaná stránka není v TLB nalezena. Proto musí TLB obsahovat také různé informace, jako např. pointer na fyzickou stránku, dirty bit, atd.

Virtuální paměť a TLB - schéma

Virtuální paměť a TLB



Paměťové reference s TLB

Paměťové reference s TLB

- Při každé referenci se nejdříve hledá číslo virtuální stránky v TLB.
  - Při nalezení je zároveň získáno číslo fyzické stránky, nastaví se příznak reference a jedná-li se o zápis, nastaví se i dirty bit.
  - Při výpadku je nutno rozhodnout, jestli se jedná o výpadek stránky nebo jenom o výpadek TLB.
    - Jestliže je stránka přítomna v paměti, výpadek TLB indikuje, že translační informace v TLB chybí. V tomto případě CPU načte znovu translační informaci ze stránkové tabulky do TLB a výpočet pokračuje.
    - Není-li stránka v paměti, výpadek TLB znamená výpadek stránky. CPU výjimkou startuje obsluhu OS a stránka je načtena z disku do paměti.

Výpadek v TLB a výměna

Výpadek v TLB a výměna

- Výpadek TLB je mnohem častější než výpadek stránky, protože TLB má mnohem méně položek, než je stránek ve fyzické paměti.
- Potom, co nastane výpadek TLB a stránka je načtena, je třeba rozhodnout, která položka v TLB bude nahrazena. Návrháři využívají různé míry asociativity pro TLB.
  - Některé systémy mají TLB malé a plně asociativní, protože to zvyšuje úspěšnost. Protože TLB má malou kapacitu, není cena za plnou asociativitu tak vysoká.

Výpadky v TLB a výměna – quick and dirty

Výpadky TLB a výměna

  - Jiné systémy mají velké TLB s malou nebo dokonce nulovou mírou asociativity.
- Není jednoduché vybrat optimální variantu.
  - U plně asociativní TLB je použití hardwarového LRU rozsáhlé a nepraktické.
  - Nelze použít ani složitější algoritmus v rámci OS, protože výpadky TLB jsou mnohem častější než výpadky stránek.
- Proto mnoho systémů provádí náhodný výběr položky TLB, určené pro výměnu (quick and dirty).
- K algoritmům výměny se ještě vrátíme.

O výměně v TLB – write back

O výměně v TLB …

- Položky v TLB se nemění (se dvěma výjimkami). Výměna položky není náročná, mnoho se nestane.
  - Protože každá položka TLB má dirty bit a reference bit, pouze tyto dva bity je třeba kopírovat zpět do stránkové tabulky v okamžiku, kdy se položka z TLB odstraňuje.
  - Nic jiného se z TLB zachraňovat nemusí. Strategie write-back je efektivní, protože četnost výpadků TLB je celkem nízká.

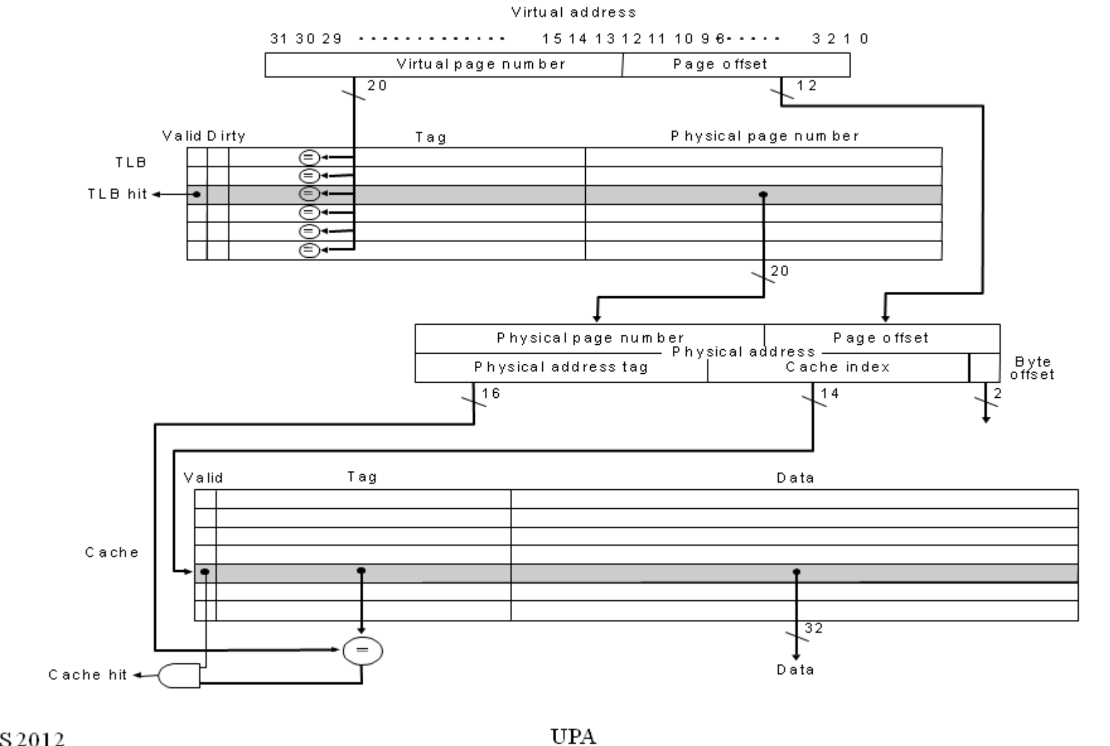
Reálné systémy virtuální paměti – MIPS R2000/DECStation 3100

Reálné systémy virtuální paměti

- Budeme se zabývat reálnou TLB a virtuální pamětí MIPS R2000/DECStation 3100.
  - Systém virtuální paměti používá stránky o velikosti 4K.
  - Adresní prostor je 32-bitů (číslo virtuální stránky je široké 20 bitů).
  - Fyzická adresa má stejnou velikost.
  - TLB má 64 položek, je plně asociativní a je sdílená pro data i instrukce. Každá položka zabírá 64 bitů a obsahuje 20-bitový tag, korespondující fyzické číslo stránky, bit platnosti, dirty bit a dvojici dalších administračních bitů.

Schéma DECStation 3100 TLB a virtuální paměť

DECStation 3100 TLB a virtuální paměť



Výjimka – výpadek TLB

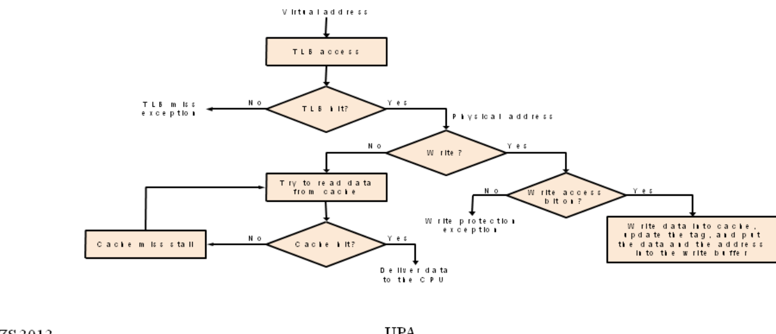
Výjimka - výpadek TLB

- Nastane-li výjimka výpadku TLB, hardware uloží číslo stránky reference do speciálního registru a generuje výjimku.
- OS zahájí obsluhu výpadku. Handler výpadku TLB zjistí ve stránkové tabulce uložené virtuální číslo stránky a registr stránkové tabulky.
  - Pravý výpadek stránky nastane, neobsahuje-li stránková tabulka fyzickou adresu.
- Užitím speciálního souboru instrukcí MIPS, OS aktualizuje TLB zapsáním fyzické adresy ze stránkové tabulky do nové položky TLB.

Přístup do paměti u DECStation 31000

Přístup do paměti u DECStation 3100

- Hardware spravuje také index, který indikuje doporučenou položku TLB k výměně - index je vybrán náhodně.
- Obrázek znázorňuje celou proceduru přístupu do paměti pro DECStation 3100…



Interakce v paměťové hierarchii, cache indexována, fyzické adresy, tag, index, virtuální adresy

Interakce v paměťové hierarchii

- Jak je vidět, nejlepší případ nastává, jestliže je virtuální adresa transformována pomocí TLB a zaslána do cache, kde jsou požadovaná data nalezena.
- V nejhorším případě nastane všude výpadek, v TLB, v tabulce stránek i v cache.
- V takovém systému jsou všechny adresy transformovány na fyzické ještě před přístupem do cache.
- Cache je *indexována* a bloky označeny položkou *tag* podle *fyzické adresy*. To znamená, že jak *tag* tak *index* je odvozen podle *fyzické adresy* a nikoliv podle *virtuální*.

Interakce v paměťové hierarchii – virtuálně indexována , virtuální tag

Interakce v paměťové hierarchii

- Alternativním řešením je indexovat cache pomocí virtuální adresy (*virtuálně indexována* a bloky označeny *virtuálním „tagem“*).
- V tomto uspořádání neprobíhá transformace adresy při běžných přístupech do cache, protože k adresování cache se používá virtuální adresa..
- Nastane-li výpadek cache paměti (cache miss), pak je virtuální adresa transformována na fyzickou, takže pak může být načten blok z hlavní paměti do cache..
- Výše uvedený návrh je složitější.

Interakce v paměťové hierarchii – hit miss možnosti

Interakce v paměťové hierarchii

- Podíváme-li se na vztah tří hardwarových jednotek a jejich interakci během přístupu do paměti, můžeme zjistit možné kombinace situací „hit“ a „miss“…

Cache	TLB	VM	Možné? Jak?
miss	hit	hit	možné – tabulka stránek se ale netestuje, je-li TLB hit
hit	miss	hit	TLB miss, položka ve stránkové tabulce nalezena, cache dává hit
miss	miss	hit	TLB miss, položka ve stránkové tabulce nalezena, cache dává miss
miss	miss	miss	TLB vykazuje miss, výpadek stránky, cache dává také miss
miss	hit	miss	nemůže nastat – v TLB nemůže nastat hit, pokud stránka není v paměti
hit	hit	miss	nemůže nastat – v TLB nemůže nastat hit, pokud stránka není v paměti
hit	miss	miss	nemůže nastat – data nemohou být v cache není-li stránka v paměti

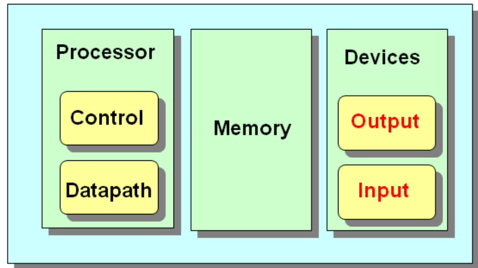
Ošetření výpadků v TLB

Ošetření výpadků v TLB

- Byl uveden obecný případ transformace virtuální adresy na fyzickou.
- Proces je velmi jednoduchý, pokud nastane TLB hit. Nastane-li miss v TLB, je situace složitější a zaslouží podrobnější rozbor.
- Zopakujme, že TLB miss může indikovat dvě možnosti…
  - Stránka je přítomna v paměti a pouze je třeba vytvořit chybějící přístupový bod v TLB pomocí tabulky stránek.
  - Stránka není přítomna v paměti a pak je třeba předat řízení OS, který se musí vypořádat s výpadkem stránky.

Základní části počítače – metriky I/O systému

Opakování: Základní části počítače



- Důležité metriky I/O systému
  - Výkon
  - Rozšiřitelnost
  - Spolehlivost
  - Cena, velikost, váha

Vstupní a výstupní zařízení, hlediska

Vstupní a výstupní zařízení

- I/O zařízení jsou velmi různorodá z hlediska
  - Chování – vstup, výstup nebo paměť
  - Partner – člověk nebo stroj
  - Rychlost přenosu dat – špičková rychlost, se kterou jsou data přenášena mezi I/O zařízením a hlavní pamětí nebo procesorem

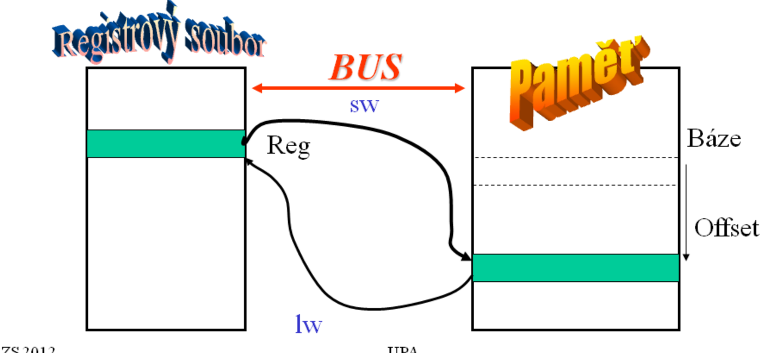
Zařízení	Chování	Partner	Rychlost přenosu dat (Mb/s)
Klávesnice	vstup	člověk	0.0001
Myš	vstup	člověk	0.0038
Laserová tiskárna	výstup	člověk	3.2000
Grafický display	výstup	člověk	800.0000-8000.0000
Síť/LAN	vstup nebo výstup	stroj	100.0000-1000.0000
Magnetický disk	paměť	stroj	240.0000-2560.0000

o 8 řádů rozdílná rychlost

Konfigurace sběrnice, Kanál, Řadič, Vysílač/přijímač

Konfigurace sběrnice

- Vysílač/Přijímač: Zdroj/Příjemce dat
- Kanál: Cesta pro data nebo instrukce
- Řadič: Master řídí akce sběrnice



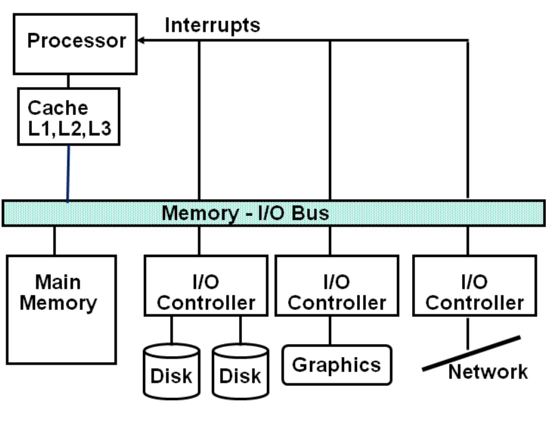
Fyzické sběrnice, paralelní sběrnice, sériové sběrnice, simplexní, duplexní

Fyzické sběrnice

- **Paralelní sběrnice**
  - *N* paralelních vodičů: ≡ *N*
  - Výhoda: Rychlé Nevýhoda: Složitost
- **Sériové sběrnice**
  - Jedna přenosová cesta
  - Obvykle *kroucená dvoulinka* (diferenciální vstupy)
  - Může být *stíněná* (koaxiální kabel, příklad Ethernet)
  - Simplexní komunikace: Jednosměrná
  - Duplexní komunikace: Obousměrná
  - Problémy: *Kolize (duplex), chyby, výpadky*
  - Výhoda: Jednoduchost Nevýhoda: Pomalé (???)

Typický I/O systém – schéma

Typický I/O systém



Přehled I/O zařízení a sběrnic – typy organizace I/O, polled, Interrupt Driven, DMA

Přehled I/O zařízení a sběrnic

- Přenos dat a instrukcí
- Paralelní nebo sériové sběrnice
- Různé I/O protokoly
- I/O zařízení se velmi liší
- Rovnice výkonu sběrnice
- Typy organizace I/O
  - Polled: Specifické testy zařízení
  - Interrupt-Driven: Přenos na žádost
  - DMA: Rychlé blokové I/O přenosy

Typy organizace I/O, polled, Interrupt-Driven Direct Memory Access DMA

Typy organizace I/O

1. **Polled**
   - Programová kontrola stavu I/O zařízení a následné plánování činnosti volných I/O zařízení
   - *Dotaz na stav (status)* (busy, wait, idle, dead…)
2. **Interrupt-Driven**
   - Obsluha I/O na požadavek, vyjádřený interruptem
   - *Vyžaduje **frontu** na uložení čekajících I/O žádostí*
3. **Direct Memory Access (DMA)**
   - Přímý přenos dat mezi I/O zařízením a pamětí
   - *Velmi rychlé, používá se pro velká množství dat, blokové přenosy,* (např., disky, video, audio)

Komunikace I/O zařízení a procesoru, jak procesor ovládá I/O zařízení, jak procesor komunikuje s I/O zařízeními, pooling interrupt driven

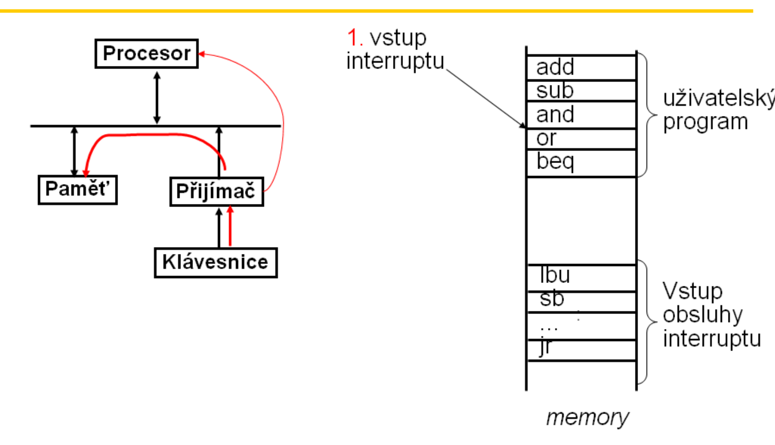
Komunikace I/O zařízení a procesoru

- Jak procesor ovládá I/O zařízení
  - Speciální I/O instrukce
    - Specifikují jak zařízení, tak vlastní příkaz
  - I/O mapované do paměťového prostoru
    - Část adresního prostoru (obvykle horní adresy) je přiřazena I/O zařízením
    - Čtení a zápis na tato paměťová místa se interpretují jako příkazy pro I/O zařízení
    - Čtení a zápis na tyto adresy může provádět jen OS
- Jak procesor komunikuje s I/O zařízeními
  - „Polling" – procesor periodicky testuje stav I/O zařízení aby zjistil, zda nepotřebuje obsluhu
    - Procesor zajišťuje všechno řízení – vykonává veškerou práci
    - Velká ztráta času procesoru vlivem rozdílných rychlostí
  - „Interrupt-driven" I/O – I/O zařízení způsobí interrupt a tím oznámí procesoru nutnost obsluhy

ZS 2012 UPA 9

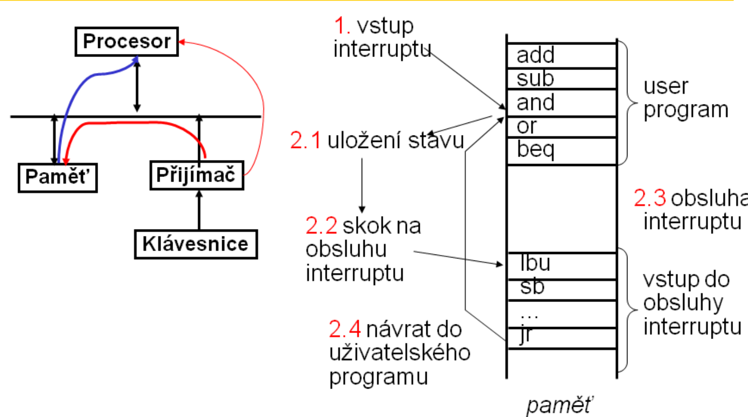
Vystup s využitím interruptů

Vstup s využitím interruptů



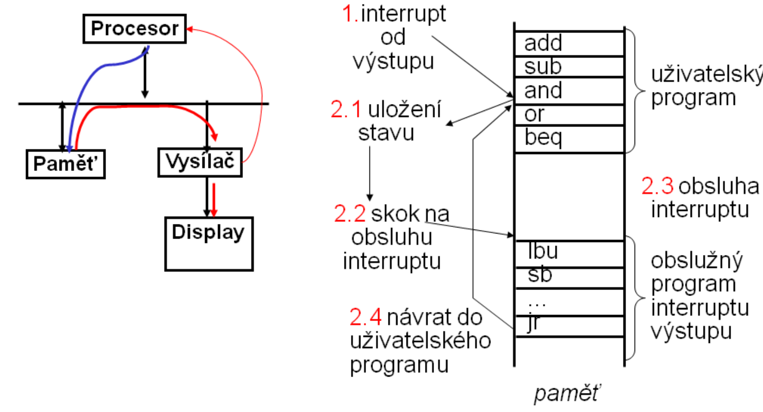
ZS 2012 UPA 10

Vstup s využitím interruptů



ZS 2012 UPA 11

Výstup s využitím interruptů



I/O s využitím interruptů, asynchronní událost, výhody, nevýhody

I/O s využitím interruptů

- I/O interrupt je asynchronní událost, nezávislá na prováděných instrukcích
  - Není propojena s žádnou instrukcí a neomezuje žádné instrukci v provedení
    - Můžete vlastně vybrat vlastní okamžik, kdy interrupt obsloužit
- S využitím I/O interruptů
  - Způsob, jak identifikovat zařízení, které způsobilo interrupt
  - Mohou být různé stupně důležitosti (organizace pomocí priorit)
- Výhody využití interruptů
  - Procesor nemusí kontinuálně testovat události I/O; pokračování uživatelského programu je pozastaveno jenom během přesunu I/O dat z/do uživatelské paměti
- Nevýhoda – je třeba speciální hardware
  - Generace interruptu (I/O zařízení), jeho detekce a uložení nutné informace tak, aby pak výpočet zase mohl po obsluze interruptu pokračovat.

Přímý přístup do paměti DMA, DMA controller

Přímý přístup do paměti (DMA)

- Pro zařízení s velkou přenosovou rychlostí (např. disky) by interruptový režim pro I/O spotřeboval příliš mnoho cyklů procesoru
- DMA – I/O controller má schopnost přenášet data přímo z/do paměti bez účasti procesoru
  1. Procesor inicializuje DMA přenos dodáním I/O adresy zařízení, typu operace, která se má vykonat, adresou do paměti zdroje/určení a počtem bytů, které se mají přenést
  2. I/O DMA řadič řídí celý přenos (i tisíce bytů), přitom soupeří o sběrnici
  3. Když je přenos DMA ukončen, I/O řadič generuje interrupt pro procesor. Tím oznamuje ukončení požadované přenosové operace.
- V jednom systému může pracovat větší počet DMA zařízení
  - Procesor a I/O DMA řadiče soupeří o cykly sběrnice a o paměť

Problém zastaralých dat, problém koherence, snooper, flushing

Problém „zastaralých" dat

- V systémech s cache pamětmi může existovat více kopií jedné položky, jedna v cache, druhá v hlavní paměti
  - Při čtení DMA (z disku do paměti) – procesor použije stará (již neplatná) data, jestliže požadovaná data mají svoji kopii v cache
  - Při zápisu DMA (z paměti na disk) a strategii write-back v cache –I/O zařízení dostane stará data pokud došlo k modifikaci dat v cache a daný blok nebyl dosud zapsán do hlavní paměti
- Problém koherence lze řešit:
  1. Směrováním všech I/O aktivit přes cache – drahé a velmi redukuje celkový výkon
  2. Jestliže OS selektivně zneplatňuje cache pro I/O čtení a vynucuje zpětné zápisy pro I/O zápis (flushing)
  3. Nutnost hardware pro selektivní zneplatnění nebo vynucení zápisu do hlavní paměti (hardware snooper)

I/O a operační systém, sdílené I/O zdroje

I/O a operační systém

- Operační systém vytváří interface mezi I/O hardwarem a programovými požadavky na I/O
  - Kvůli ochraně sdílených I/O zdrojů, nemůže uživatelský program komunikovat přímo s I/O zařízením
- Proto musí být OS schopen dávat příkazy I/O zařízením, obsluhovat přerušení, která I/O zařízení generují, provádět vyvážený přístup ke sdíleným I/O zdrojům a plánovat I/O požadavky tak, aby se optimalizoval výkon celého systému
  - I/O interupty způsobují přechod od zpracování uživatelských procesů k procesům OS

Připojení I/O systému, sběrnice, propustnost, šířka pásma, Délka sběrnice, počet zařízení, maximální rychlost sběrnice omezena

Připojení I/O systému

- Sběrnice je sdílená komunikační linka (jednoduchý soubor vodičů, které slouží k propojení subsystémů), propojující celou řadu rozdílných zařízení s různými latencemi a značně rozdílnými přenosovými rychlostmi.
  - Výhody
    - Univerzální – nová zařízení lze snadno přidávat a lze je používat ve více systémech, které používají stejný typ (standard) sběrnice
    - Nízká cena – soubor vodičů je sdílen pro vytvoření mnoha spojů
  - Nevýhody
    - Tvoří „úzké místo" komunikace – šířka pásma sběrnice omezuje maximální propustnost I/O
- Maximální rychlost sběrnice je omezena
  - Délkou sběrnice (!závisí také na typu - sériová x paralelní)
  - Počtem zařízení připojených na sběrnici

Charakteristika sběrnice, řídící linky ,datové linky, master slave, transakce na sběrnici

Charakteristika sběrnice

- Řídicí linky
  - Signály žádosti (request) a zpětného hlášení (acknowledge)
  - Indikace typu informace na datových linkách
- Datové linky
  - Data, adresy a komplexní příkazy
- Transakce na sběrnici se skládá z
  - Master vysílá povel (command) a adresu – žádost (request)
  - Slave přijme (nebo vyšle) data – akce
  - Definováno vzhledem k paměti
    - vstup – vstup dat z I/O zařízení do paměti
    - výstup – výstup dat z paměti do I/O zařízení

Typy sběrnic – sběrnice procesor-paměť (proprietární), I/O sběrnice, Blackplane bus

Typy sběrnic

- **Sběrnice procesor-paměť** (proprietární)
  - Krátká a vysoká rychlost
  - Přizpůsobena paměťovému systému, aby se maximalizovala přenosová rychlost procesor-paměť
  - Optimalizována pro přenosy bloků do cache
- **I/O sběrnice** (průmyslový standard, např., SCSI, USB, Firewire)
  - Obvykle je delší a pomalejší
  - Musí přizpůsobit velmi rozdílná I/O zařízení
  - Připojena ke sběrnici procesor-paměť a nebo ke sběrnici typu „backplane bus“
- **Backplane bus** (průmyslové standardy, např., ATA, PCI, PCIexpress)
  - „Backplane“ je propojovací struktura spojená s chassis
  - Používá se jako sběrnice, propojující I/O sběrnice a sběrnici procesor-paměť

Synchronní a asynchronní sběrnice, handshaking, skew, ReadReq, Ack, DataRdy

Synchronní a asynchronní sběrnice

- **Synchronní sběrnice** (např. sběrnice procesor-paměť)
  - Zahrnuje hodiny mezi řídící linky a má fixní protokol pro komunikaci, který je vztažený k hodinám
  - Výhoda: Obsahuje málo logiky a dosahuje vysoké rychlosti
  - Nevýhody:
    - Každé zařízení, komunikující na sběrnici musí používat stejnou frekvenci
    - Aby se zamezilo „skew“ hodin, nemůže být dlouhá, má-li pracovat rychle
- **Asynchronní sběrnice** (např. I/O sběrnice)
  - Není taktována, proto vyžaduje handshaking protokol a přídavné řídící linky (ReadReq, Ack, DataRdy)
  - Výhody:
    - Může zahrnout široké spektrum zařízení a přenosových rychlostí
    - Větší délka, aniž vzniknou problémy se „skew“ hodin nebo se synchronizací
  - Nevýhoda: nižší rychlost

Fyzické vs logické I/O sběrnice, rekonfigurovatelné sběrnice

Fyzické vs. logické I/O sběrnice

- **Fyzické sběrnice**
  - *Instalováno v počítačích:* sběrnice ISA, PCI, AGP pro grafické karty
  - Limitována velikost a spotřeba el. výkonu
  - *Výkon omezen šířkou sběrnice, rychlostí řadiče*
- **Logické I/O (rekonfigurovatelné sběrnice)**
  - Používají fyzické sběrnice, které mohou být různě konfigurovány
  - Vhodné pro maximální využití sběrnice
  - CPU se chová jako kdyby měla variabilní sběrnice
  - Rekonfigurace sběrnic – přizpůsobení aktuálním I/O požadavkům
  - Vyžaduje komplexní řadič sběrnice a plánovací software

Příklad výkonnosti I/O systému, výpočet

Příklad výkonnosti I/O systému

- Zátěž disku představuje 64 KB čtení a zápisů, kde uživatelský program provede 2.10^5 instrukcí na jednu diskovou I/O operaci a
  - procesor s výkonem 3.10^9 instr/s a průměrně 10^5 instrukcí OS na provedení jedné diskové I/O operace

Maximální rychlost diskových I/O operací (# I/O/sec) procesoru je rovna:

$$\frac{\text{Instr execution rate}}{\text{Instr per I/O}} = \frac{3 \times 10^9}{(2 + 1) \times 10^5} = 10{,}000 \text{ I/O's/s}$$

- Sběrnice I/O-paměť dosahuje přenosovou rychlost 1000 MB/s

Každá disková I/O čte nebo zapisuje 64 KB, takže maximální rychlost I/O sběrnice je:

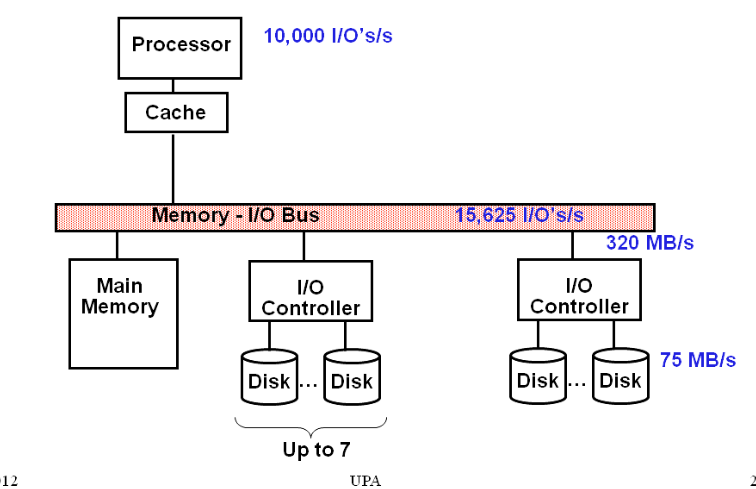
$$\frac{\text{Bus bandwidth}}{\text{Bytes per I/O}} = \frac{1000 \times 10^6}{64 \times 10^3} = 15{,}625 \text{ I/O's/s}$$

- Diskové SCSI I/O kontroléry DMA s přenosovou rychlostí 320 MB/s, které obsluhují až 7 disků/kontrolér
- Diskové jednotky s přenosovou rychlostí při read/write 75 MB/s a střední přístupovou dobou latence 6 ms

Jaká je maximální dosažitelná rychlost I/O a jaký je počet disků a SCSI kontrolérů potřebných pro dosažení této rychlosti?

Příklad diskového I/O systému schéma

Příklad diskového I/O systému



Příklad výkonnosti I/O systému – výpočet, přenosová rychlost disku, doba operace disku

Příklad výkonnosti I/O systému (pokračování)

Procesor je nejslabším místem, nikoliv sběrnice

– disková mechanika se šířkou pásma pro operace (read/write) 75 MB/s a střední dobou latence přístupu 6 ms (seek + rotace)

Doba I/O operace disku (read/write) = seek + rotational time + transfer time = 6ms + 64KB/(75MB/s) = 6.9ms

Proto každý disk může provést 1000ms/6.9ms = 146 I/O's za sekundu.
Saturování procesoru vyžaduje 10,000 I/O's za sekundu nebo-li 10,000/146 = 69 disků

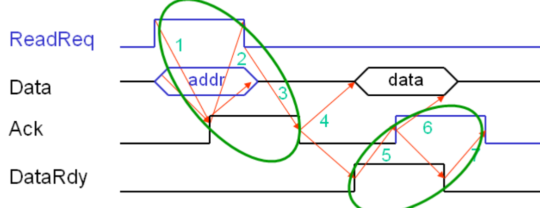
Pro výpočet počtu SCSI diskových kontrolérů potřebujeme znát střední přenosovou rychlost jednoho disku, abychom určili, zda můžeme připojit maximální počet 7 disků na SCSI kontrolér a že diskový kontrolér nebude saturovat sběrnici IO-paměť během DMA přenosu.

Přenosová rychlost disku = (velikost bloku)/(doba přenosu) = 64KB/6.9ms = 9.56 MB/s
Proto 7 disků nebude saturovat ani SCSI kontrolér (s maximální přenosovou rychlostí 320 MB/s), ani sběrnici I/O-paměť (1000 MB/s). To znamená – budeme potřebovat 69/7 tedy 10 SCSI kontrolérů.

Protokol asynchronní sběrnice handshaking, výstup dat z paměti na I/O zařízení, ReadReq, Data, Ack, DataRdy

Protokol asynchronní sběrnice (handshaking)

- Výstup (read) dat z paměti na I/O zařízení



Zařízení I/O oznamuje požadavek nastavením ReadReq a vydáním addr na datových linkách.

1. Paměť dostane ReadReq, převezme addr z datových linek a aktivuje Ack
2. Zařízení I/O reaguje na Ack, uvolňuje ReadReq a datové linky
3. Paměť zjistí sestupnou hranu ReadReq a deaktivuje Ack
4. Když paměť dokončí čtení, umístí data na datové linky a aktivuje DataRdy
5. Zařízení I/O zjistí DataRdy, sejme data z datových linek a aktivuje Ack
6. Paměť zjistí aktivní Ack, uvolní datové linky a deaktivuje DataRdy
7. Zařízení I/O reaguje na sestupnou hranu DataRdy deaktivací Ack

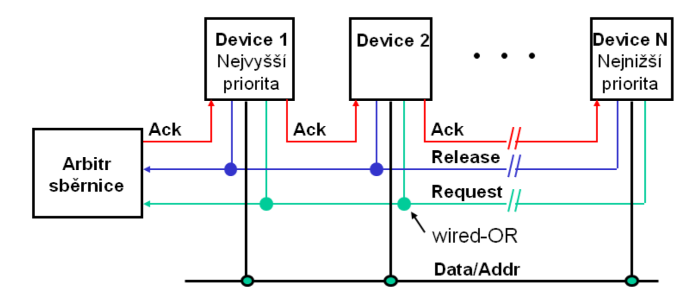
Arbitrace sběrnice, Bus priority, Fairness, Daisy chain, Centralizovaná, Distribuovaná arbitrace, s detekcí kolize

Arbitrace sběrnice

- Sběrnici může být schopno současně řídit větší množství zařízení => nutnost arbitrovat požadavky
- Arbitrační schémata se snaží zohledňovat:
  – Bus priority – zařízení nejvyšší priority by mělo být obslouženo nejdříve
  – „Fairness" – i zařízení s nejnižší prioritou nesmí být odříznuto od sběrnice
- Arbitrační mechanizmy sběrnic lze dělit do čtyř tříd:
  – „Daisy chain" arbitrace (jinak „postupná obsluha") – další snímky
  – Centralizovaná, paralelní arbitrace – další snímky
  – Distribuovaná arbitrace (self-selection) – každé zařízení, které se uchází o sběrnici vydává identifikační kód na sběrnici
  – Distribuovaná arbitrace s detekcí kolize – zařízení využije sběrnici pokud je volná a nastane-li kolize (protože i jiná zařízení se mohou rozhodnout stejně), potom zařízení pokus opakuje později (Ethernet)

Daisy Chain arbitrace – schéma, výhody nevýhody

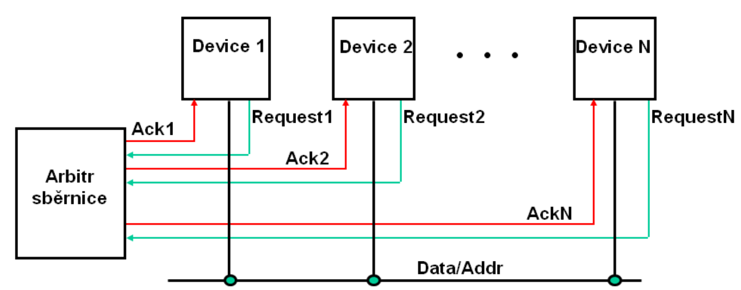
„Daisy Chain“ arbitrace



- Výhoda: jednoduchost
- Nevýhody:
  - Nelze zajistit „spravedlnost“ – zařízení nízké priority mohou být trvale blokována
  - Pomalé – „daisy chain“ signál omezuje rychlost procesu přidělování

Centralizovaná paralelní arbitrace – schéma, výhody nevýhody

Centralizovaná paralelní arbitrace



- Výhody: flexibilní, může zajistit „spravedlnost“
- Nevýhody: složitější hardware arbitru
- Používáno hlavně u všech sběrnic procesor-paměť a u rychlých I/O sběrnic

Šířka pásma sběrnice - ovlivnění

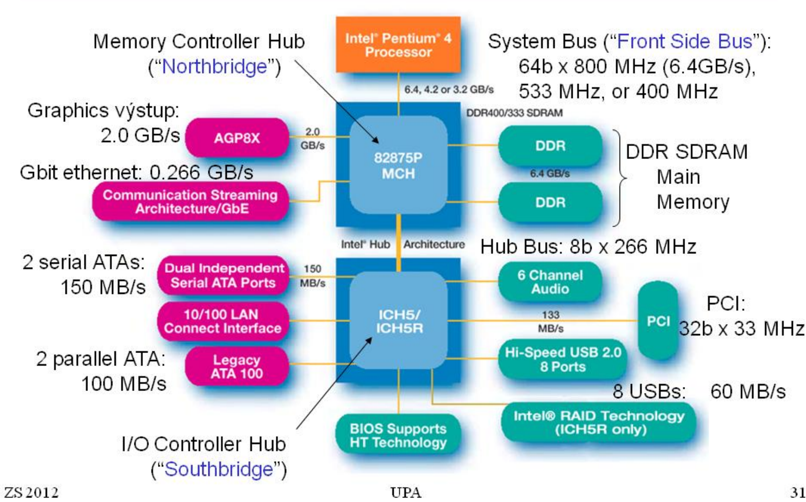
Šířka pásma sběrnice

- Šířku pásma sběrnice ovlivňuje:
  - Zda se jedná o asynchronní nebo synchronní sběrnici a dále pak charakteristika použitého protokolu
  - Šířka sběrnice
  - Zda sběrnice podporuje blokové přenosy nebo nikoliv

	Firewire	USB 2.0
Typ	I/O	I/O
Datové linky	4	2
Časování	Asynchronní	Synchronní
Max. # zařízení	63	127
Max. délka	4.5 metru	5 metrů
Špičkový přenosový výkon (šířka pásma)	50 MB/s (400 Mbps) 100 MB/s (800 Mbps)	0.2 MB/s (low) 1.5 MB/s (full) 60 MB/s (high)

Příklad sběrnice Pentia 4, Pentium 4

Příklad: Sběrnice Pentia 4



Sběrnice ve vývoji, široké synchronní, úzké asynchronní

Sběrnice ve vývoji

- Výrobci dříve přecházeli od asynchronních sběrnic k synchronním, dnes od *širokých* synchronních k *úzkým* asynchronním sběrnicím.
  - Odrazy na vedeních a skew hodin nedovolují zvyšovat hodinové frekvence u sběrnic, kde se používá 16 až 64 paralelních vodičů (nad ~400 MHz). Proto výrobci přecházejí na úzké jednosměrné s vysokou hodinovou frekvencí (~2 GHz)

	PCI	PCIexpress	ATA	Serial ATA
Celkový # vodičů	120	36	80	7
# datových linek	32 – 64 (obousměrné)	2 x 4 (jednosměrné)	16 (obousměrné)	2 x 2 (jednosměrné)
Hodiny (MHz)	33 – 133	635	50	150
Špičkový BW (MB/s)	128 – 1064	300	100	375 (3 Gbps)

Rozměry kabelů ATA, PATA, SATA

Rozměry kabelů ATA

- Kabely pro sériovou sběrnici ATA (červená) jsou mnohem tenší než paralelní kabely ATA (zelená)

Měření výkonu I/O, šířka pásma I/O, propustnost, doba odezvy I/O, latence

Měření výkonu I/O

- Šířka pásma I/O (propustnost) – množství informace, která prochází vstupem (výstupem) a propojovací strukturou (např. sběrnicí) k procesoru/paměti (I/O zařízení) za jednotku času
  1. Kolik dat můžeme přesunout v systému za určitou dobu?
  2. Kolik I/O operací můžeme provést za jednotku času?
- Doba odezvy I/O (latence) – celková doba nutná k provedení vstupní nebo výstupní operace
  - Zvláště důležitá metrika pro systémy pracující v reálném času
- Mnoho aplikací vyžaduje obojí – vysokou propustnost a krátkou dobu odezvy

Výkon I/O systému, návrh I/O systému

Výkon I/O systému

- Návrh I/O systému aby vyhověl požadavkům na šířku pásma a/nebo požadavkům na latence
  1. Nalezení „nejslabší“ linky v I/O systému – prvek, který způsobuje omezení
     - Procesor a paměťový systém?
     - Propojovací struktura (např., sběrnice) ?
     - I/O kontroléry ?
     - I/O zařízení sama o sobě ?
  2. (Re)konfigurace nejslabšího prvku tak, aby byly splněny požadavky na šířku pásma a/nebo požadavky na latence
  3. Určení požadavků na zbylé části a jejich (re)konfigurace tak, aby byly splněny požadavky na šířku pásma a/nebo požadavky na latence

Výkonové parametry sběrnice, šířka pásma, propustnost, četnost poruch a cena, četnost chyb a cena

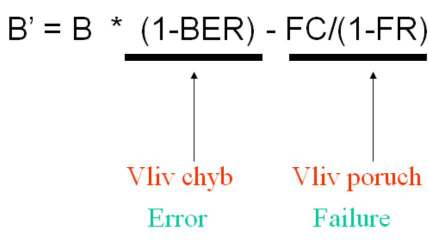
Výkonové parametry sběrnice

- Šířka pásma nebo propustnost
  - *Kolik dat lze přenést sběrnicí za jednotku času* **(jednotka = bity za sekundu)**
- Četnost poruch a cena
  - Četnost: ~ Pravděpodobnost poruchy sběrnice
  - *Cena: Kolik stojí restart*
    - *Sběrnice se musí* ***zotavit*** *(cykly sběrnice, x bitů na cykl)*
    - *Opakování vadných paketů (bity, které se musí opakovat)*
- Četnost chyb a cena
  - Podobné jako četnost poruch

Rovnice výkonu sběrnice, předpoklady, výpočet aktuální šířky pásma, BER, FR, FC

Rovnice výkonu sběrnice

- Předpoklady
  - Šířka pásma (B), četnost bitových chyb (BER)
  - Četnost poruch (FR), cena poruch (FC)
- Výpočet aktuální šířky pásma (B')



Výkon sběrnice příklady, nominální šířka pásma, četnost poruch, četnost bitových chyb, cena poruchy

Výkon sběrnice - příklad #1

- Předpoklady
  - Nominální šířka pásma: 32 MHz, 32 bitů paralelně
  - Četnost poruch = 10^{-4} ; Cena poruchy = 0.2 Mbps
  - Četnost bitových chyb = 10^{-6}
- Výpočet aktuální šířky pásma (B')

B' = B * (1-BER) - FC/(1-FR)
= 32bitů * (32 * 10^6 Hz) * (0.999999)
- (0.2 x 10^6 bps/ 0.9999)
= 1.023999 Gbps – 0.20002 Mbps
= 1023.799 Gbits/sec = 0.02% snížení

Výkon sběrnice - příklad #2

- Předpoklady
  - Nominální šířka pásma: 32 MHz, 32 bitů paralelně
  - Četnost poruch = 10^{-1} ; Cena poruchy = 0.2 Mbps
  - Četnost bitových chyb = 10^{-6}
- Výpočet aktuální šířky pásma (B')

B' = B * (1-BER) - FC/(1-FR)
= 32bitů * (32 * 10^6 Hz) * (0.999999)
- (0.2 x 10^6 bps/ 0.9)
= 1.023999 Gbps – 0.2222 Mbps
= 1023.7768 Gbitů/sek = 0.03% snížení

Výkon sběrnice - příklad #3

- Předpoklady
  - Nominální šířka pásma: 32 MHz, 32 bitů paralelně
  - Četnost poruch = 10^{-1} ; Cena poruchy = 0.2 Mbps
  - Četnost bitových chyb = 10^{-3}
- Výpočet aktuální šířky pásma (B')

B' = B * (1-BER) - FC/(1-FR)

= 32bitů * (32 * 10^6 Hz) (0.999)

- (0.2 x 10^6 bps/ 0.9)

= 1.022976 Gbps – 0.2222 Mbps

= 1023.7538 Gbps = 0.23% snížení

Výkon sběrnice realita, problémy

Výkon sběrnice - realita

- Problémy (BW … BandWidth)
  - Kolize na sběrnici: Pakety používají stejný HW
  - Simplex: Méně kolizí, Duplex: Více kolizí
- Některé praktické výsledky testů

PCI: 32 bitů paralelně na 32 MHz (128MB/s)

Nominální BW = 1K MHz ~ 1GHz

Simplex: 70 - 80% of BW

Duplex: 20 - 40% of BW

např., duplex => 25 to 50 MB/s

Sběrnice Aplikace => požadavky, Aplikace, složitost, Reálně, důsledek

Sběrnice: Aplikace => Požadavky

- Aplikace
  - Obrázky: M*N pixelů v rámci, K bitů na pixel
  - Video: F rámců za sekundu
- Složitost = **O**(M*N*K*F) bitů za sekundu

Reálně: M, N = 1024, K = 24 bpp, F = 30 fps

MNKF = 1M (720) = ***720 Mbitů/sec***

Simplex: **720 Mbps / 0.7 => B = 1.03 Gbps**

Duplex: **720 Mbps / 0.3 => B = 2.4 Gbps**

Důsledek: Pro zpracování obrazů jsou třeba *velmi rychlé* sběrnice

Technologie sběrnic, současné, rozšiřuje se, Fiber Optic

Technologie sběrnic

- Současné:
  - Měděné vodiče, cesty na PCB (2-8 GHz)
  - Koaxiální, Fiber optic Internet (2 Gbps - …)
- Rozšiřuje se:
  - **Optický přenos volným prostorem**: 20+ Gbps
    - BW omezena šířkou pásma by vysílače/přijímače
    - Problémy s atmosférickými vlivy rozptyl & absorpce
  - **"Vše opticky"**
    - Nelze doslova – vždy je třeba nějaké elektronika a optoelektronika
    - Rychlost omezena šířkou pásma použité elektroniky
- V dohlednu: Fiber Optic (rychlé, levné)

I/O ovlivňuje přenos dat, maximální výkon I/O systému, rekonfigurovatelná sběrnice

Závěr

- I/O ovlivňuje přenos dat:
  - Dělením toku dat do bloků nebo paketů
  - Vysíláním datových paketů sériově po sběrnici
  - Udržováním I/O sběrnice na plném výkonu (obsazena) pro dosažení *maximálního výkonu I/O systému*
- Rekonfigurovatelné sběrnice jsou výhodné, protože:
  - Různé aplikace mají různé nároky na I/O
  - Konfigurací může být sběrnice přizpůsobena těmto požadavkům