Úvod do Počítačových Architektur

KIV-FAV

Úvod

Přehled úvodní přednášky

- Úvod do „UPA“
- Přehled kurzu
- Organizační záležitosti

= = = = = = = = = = = = = = =

- Historie výpočetní techniky
- Úrovně abstrakce počítače
- Závěry

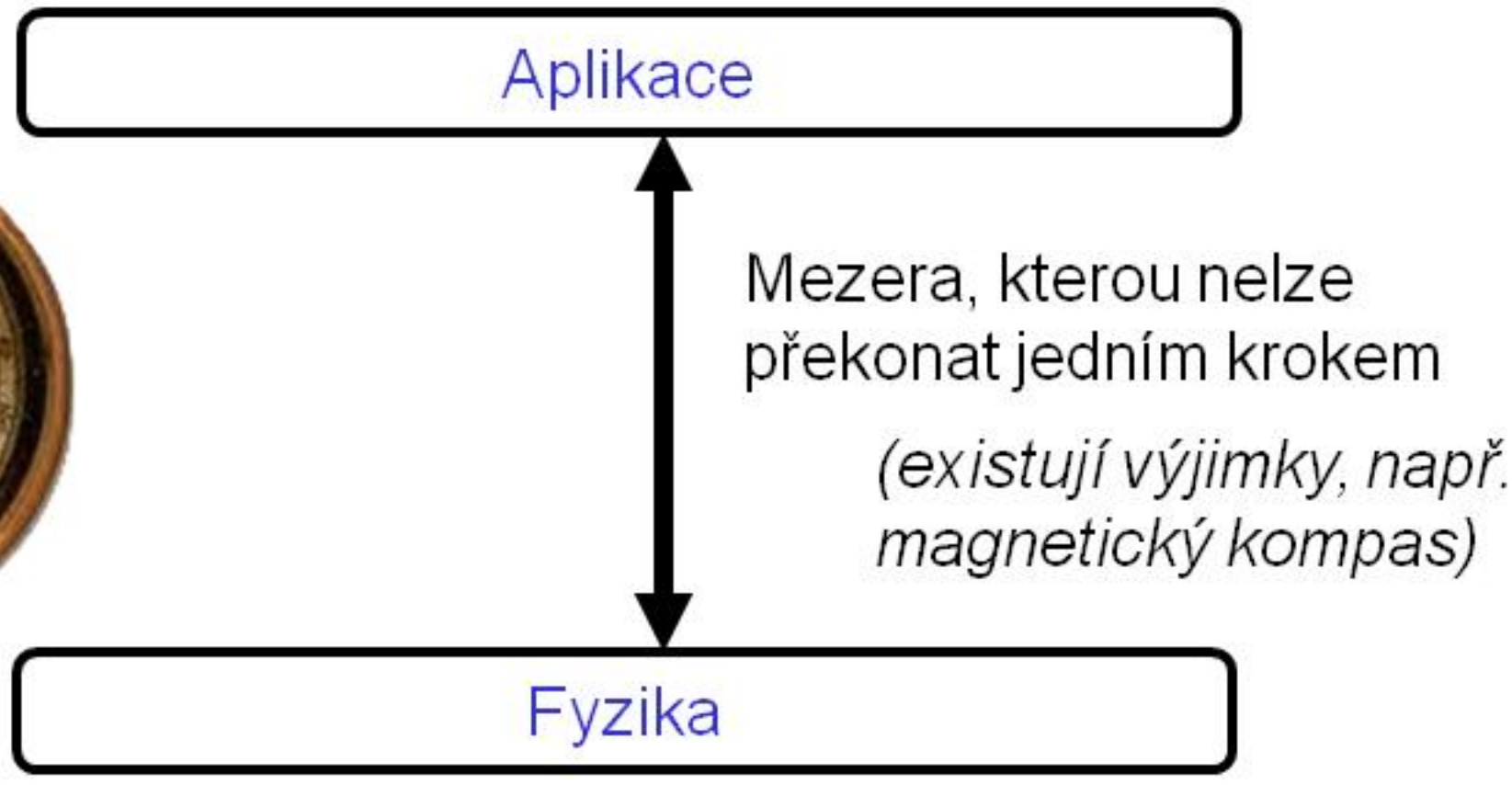
Přehled kurzu

- Technologické základy
- Problematika výkonnosti počítačového systému
- ISA - specifická „Instruction Set Architecture“
- Aritmetika - jak navrhnout ALU
- Konstrukce procesoru pro zvolený soubor instrukcí
- „Pipelining“ pro zlepšení výkonu
- Cache, hlavní a virtuální paměť
- I/O
- Paralelní počítače - přehled

Úvod do UPA

- Rychle se měnící oblast:
  - elektronky -> tranzistory -> IC -> VLSI
  - zdvojnásobení každých 1.5 roku (Moorův zákon):
    - *Kapacita paměti*
    - *Rychlost procesoru (Vlivem pokroku technologie a organizace)*
- Čím se budeme v předmětu UPA zabývat:
  - jak počítače pracují, základy
  - jak analyzovat jejich výkon (popř. jak se to nemá dělat!)
  - problémy ovlivňující moderní procesory (cache, pipeline, paralelizmus)
- Proč studovat tyto problémy?
  - chcete se stát "počítačovým odborníkem"
  - chcete vytvářet software, který dokáže správně využít vlastnosti hardware
  - chcete umět rozhodovat, popř. kvalifikovaně poradit

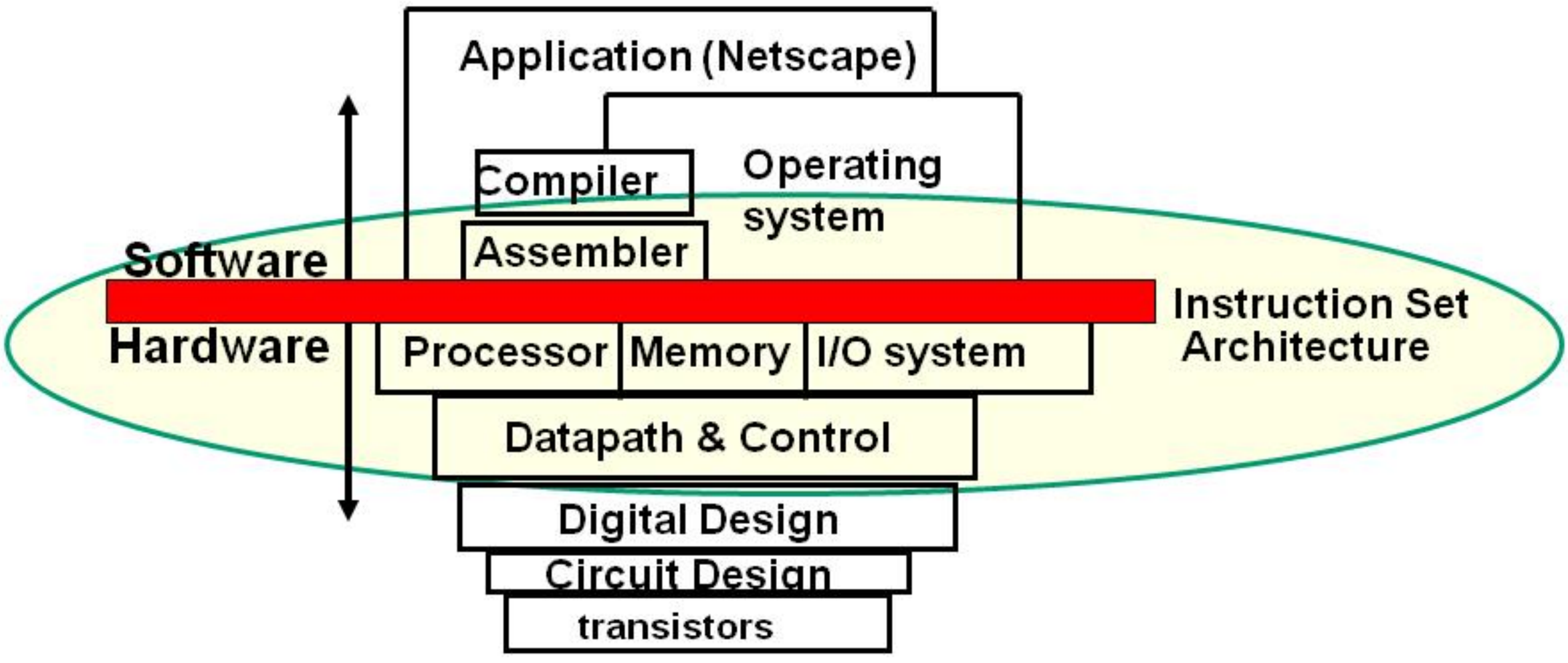
Co je architektura počítače?



V širším pojetí chápeme počítačovou architekturou *návrh úrovní abstrakce*, které dovolují implementovat aplikace (zpracování informace) s efektivním využitím dostupných výrobních technologií.

Výpočetní systém

Kde se budeme pohybovat



Koordinace mnoha *úrovní abstrakce*

O čem bude řeč

- Pět klasických částí počítače (von Neumannova koncepce počítače)
- Data mohou být vše (integer, floating point, znaky): program určuje, o co jde
- Koncepce programu uloženého v paměti: instrukce i data
- Princip lokality, využíván v hierarchickém uspořádání paměti (cache a virtuální paměť)
- Zvyšování výkonu využitím vyšší míry paralelizmu
- Princip abstrakce, používaný v komplexních systémech, budovaných ve „vrstvách“
- Kompilace v. interpretace v jednotlivých vrstvách systému
- Principy a úskalí měření výkonu počítače

Organizační údaje

Přednášky:

- Vlastimil Vavřička, KIV, (vavricka@kiv.zcu.cz)

Cvičení:

- Karel Dudáček ml., KIV (karlos@kiv.zcu.cz)
- Karel Dudáček st., KIV (dudacek@kiv.zcu.cz)

Termíny zkoušek:

- Zkoušky budou organizovány ve třech termínech během zimního zkouškového období

Organizační detaily

Postup při řešení problémů vztahujících se k předmětu:

1. Konzultace s vedoucím cvičení v rámci výuky
2. Návštěva na KIVu v době úředních hodin
3. Mail

Podmínky získání zápočtu z UPA

- Samostatné vypracování zadaných úloh (2 úlohy)
  - Zadání se přidělují při cvičení ve třetím, nejpozději ve čtvrtém týdnu semestru.
  - Odevzdání (a akceptování !) a všech úloh vyučujícímu (vedoucímu cvičení) do konce 13. týdne semestru včetně bude bonifikováno (10 bodů). To představuje 10% bodů z celkového počtu, které lze získat.
- Úspěšné absolvování přehledového testu, který se bude konat v jedenáctém týdnu semestru:
  - znalosti získané na přednáškách
  - úlohy obdobné úkolům, probíraným v rámci cvičení
- Zápočet je možno získat jen do konce zimního zkouškového období

Požadavky ke zkoušce z UPA

Zkouška – písemná

Podmínky pro přihlášení na zkoušku:

- Zápočet – bez zápočtu nebudou studenti zkoušeni

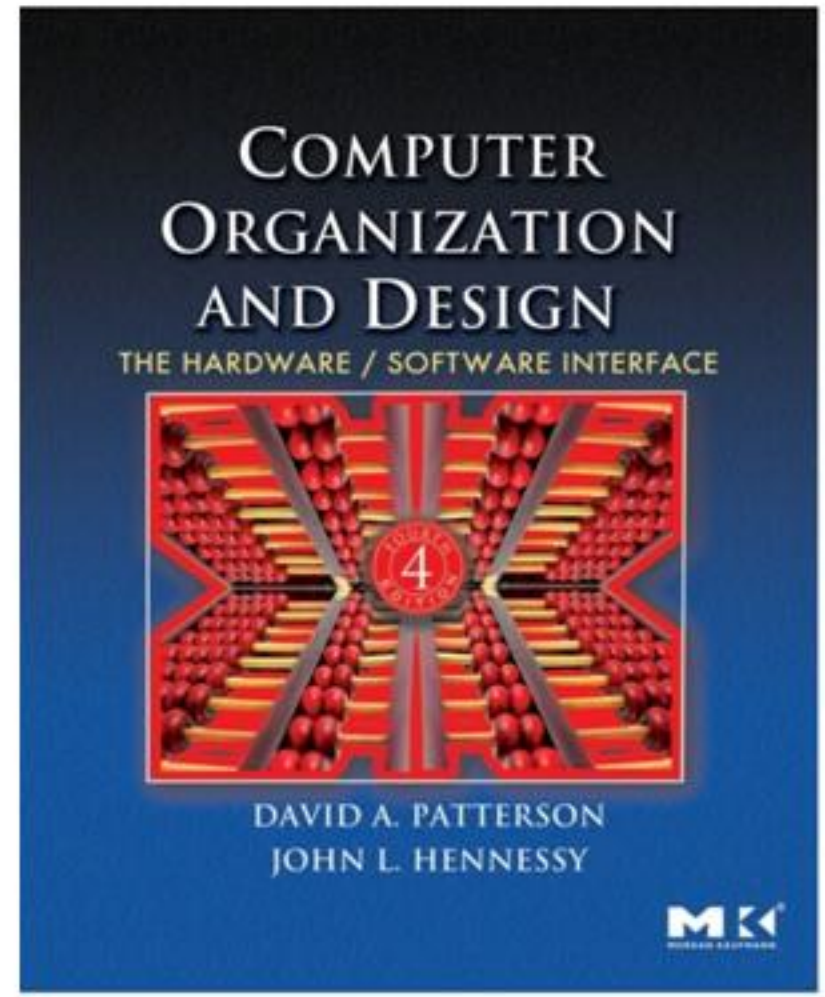
Podmínky pro složení zkoušky:

- Znalost látky v rozsahu přednášek
- Znalost témat (články), která budou určena k nastudování

!! Poznámky z přednášek berte jen jako průvodce ke zvládnutí problematiky, nikoliv jako hlavní učební texty !!

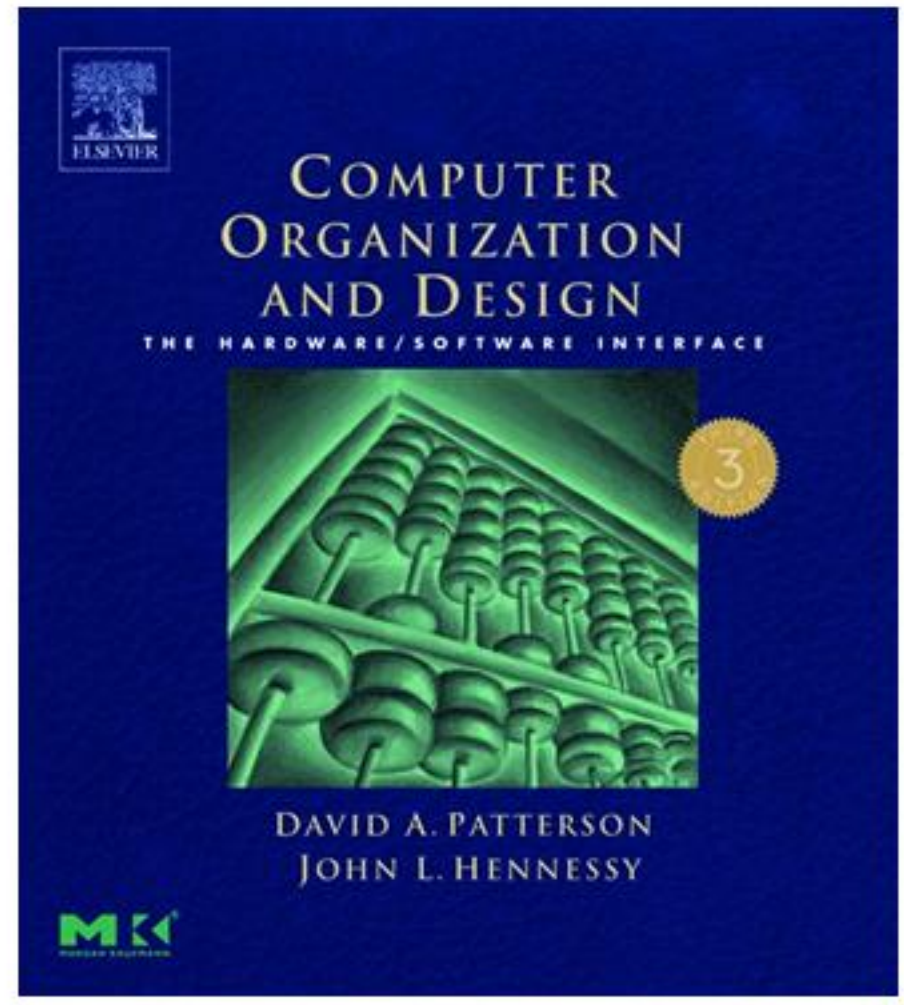
Doporučená literatura

- D. A. Patterson and J. L.Hennessy: Computer Organization and Design: The Hardware Software Interface, 4rd Edition, 2009



Doporučená literatura

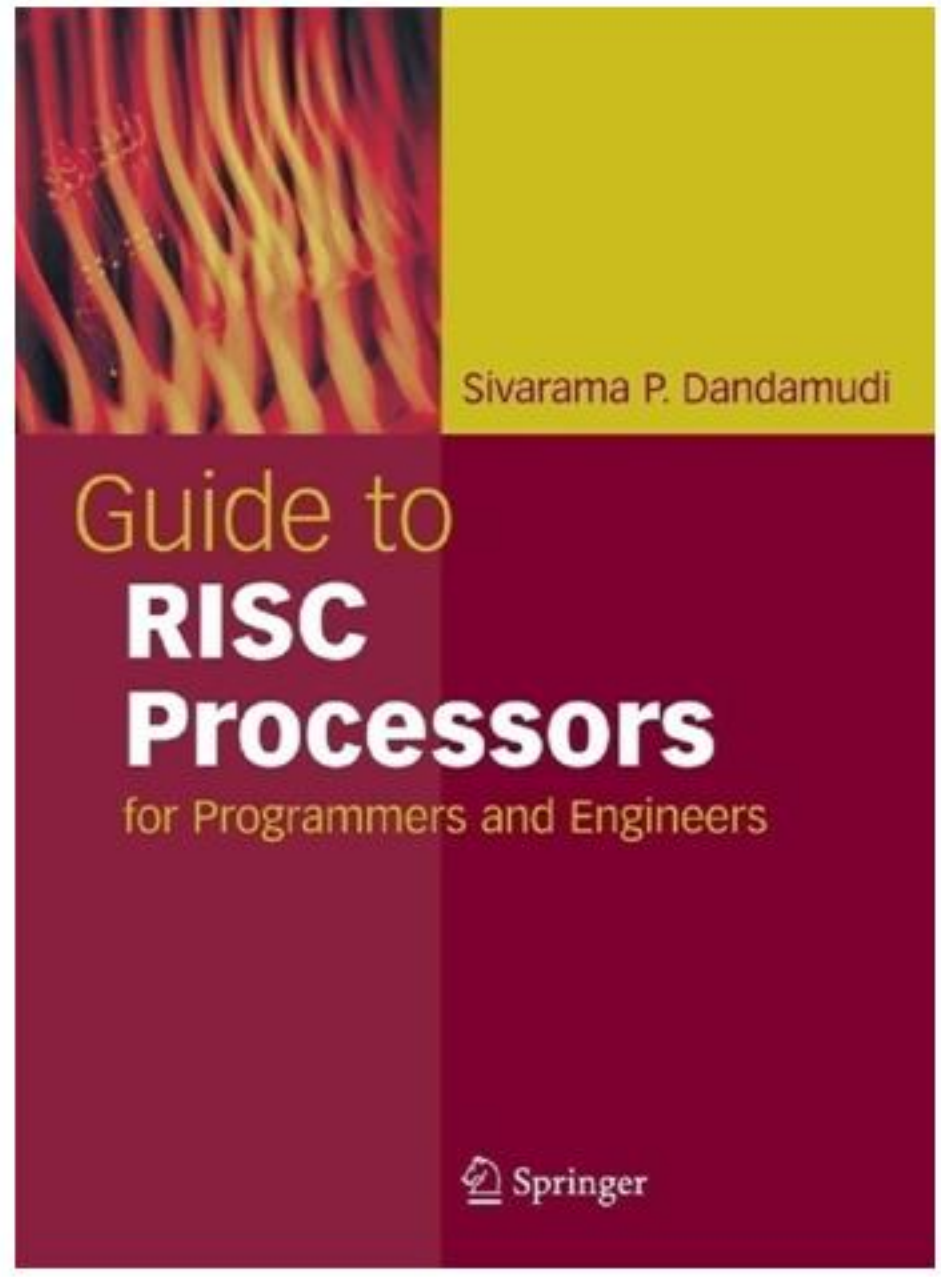
- D. A. Patterson and J. L.Hennessy: Computer Organization and Design: The Hardware Software Interface, 3rd Edition, 2005



Doporučená literatura (pokr.)

Sivarama P. Dandamudi: Guide to RISC Processors for Programmers and Engineers

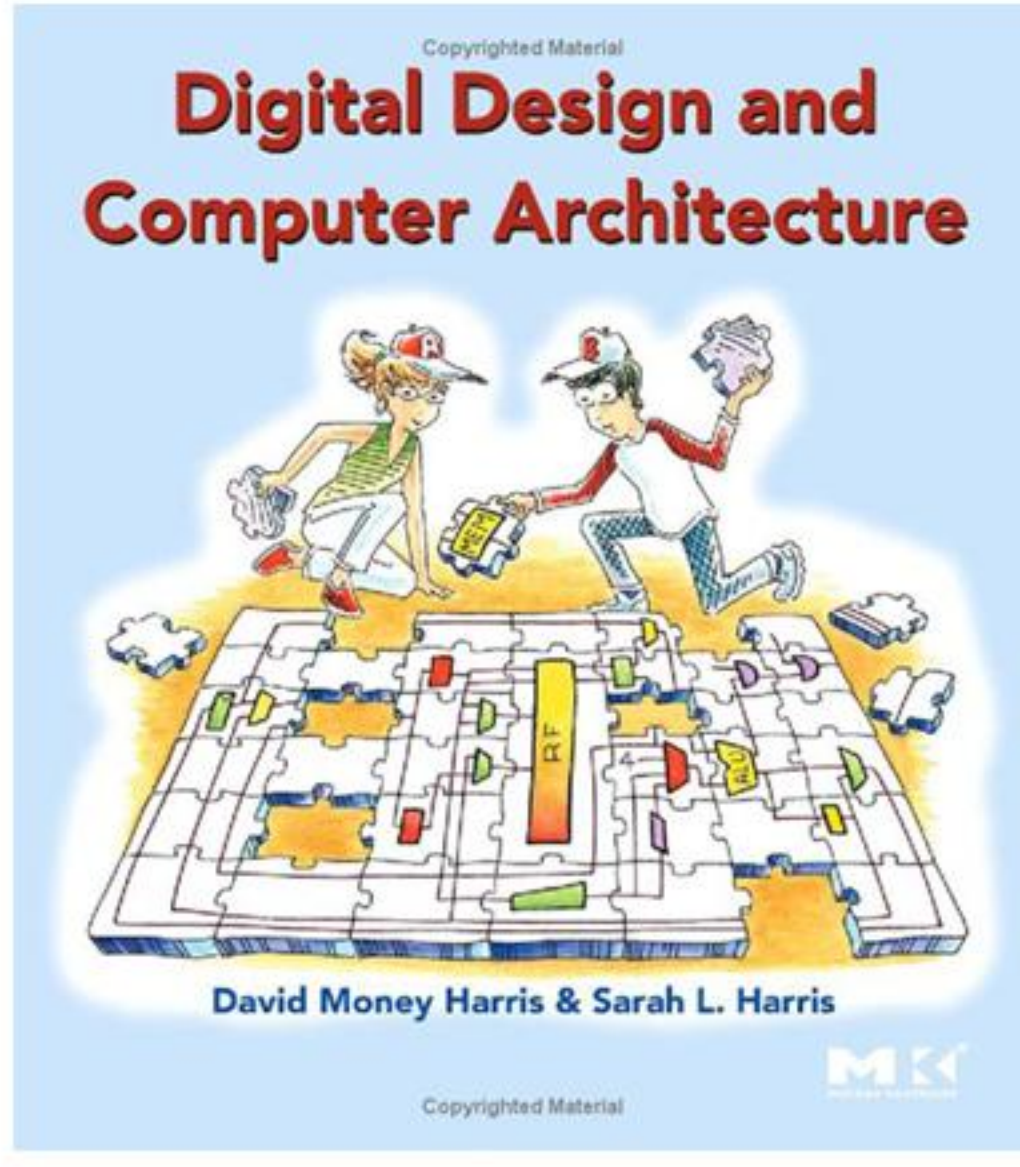
ISBN 0-387-21017-2 (elektronické zdroje ZČU !!!)



Doporučená literatura (pokr.)

David Harris, Sarah Harris: Digital Design and Computer Architecture, Second Edition

ISBN: 978-0-12-394424-5



Doporučená literatura (pokr.)

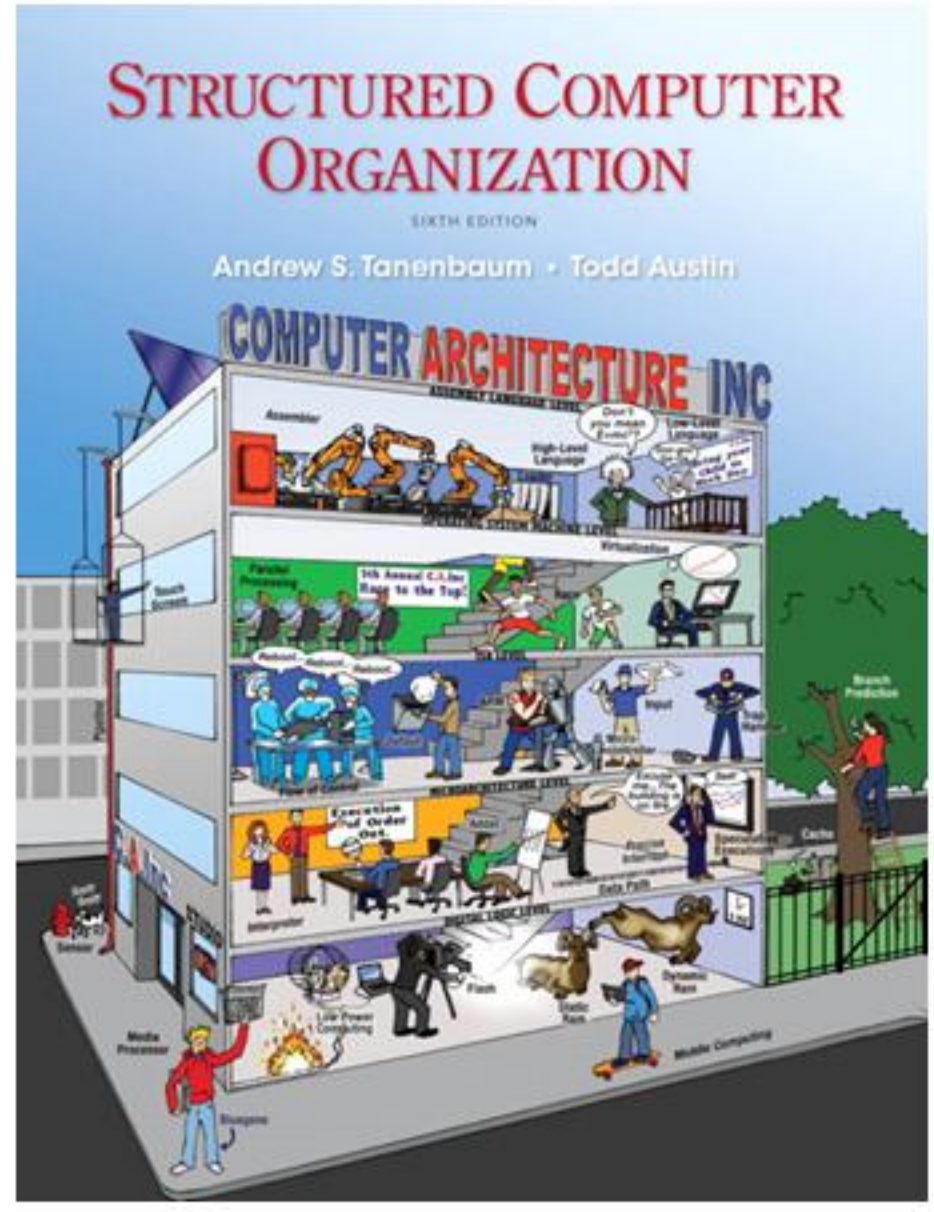
Virgil Bistriceanu, Illinois Institute of Technology:

Fundamentals of Computer Design

(jednotlivé kapitoly)

Doporučená literatura (pokr.)

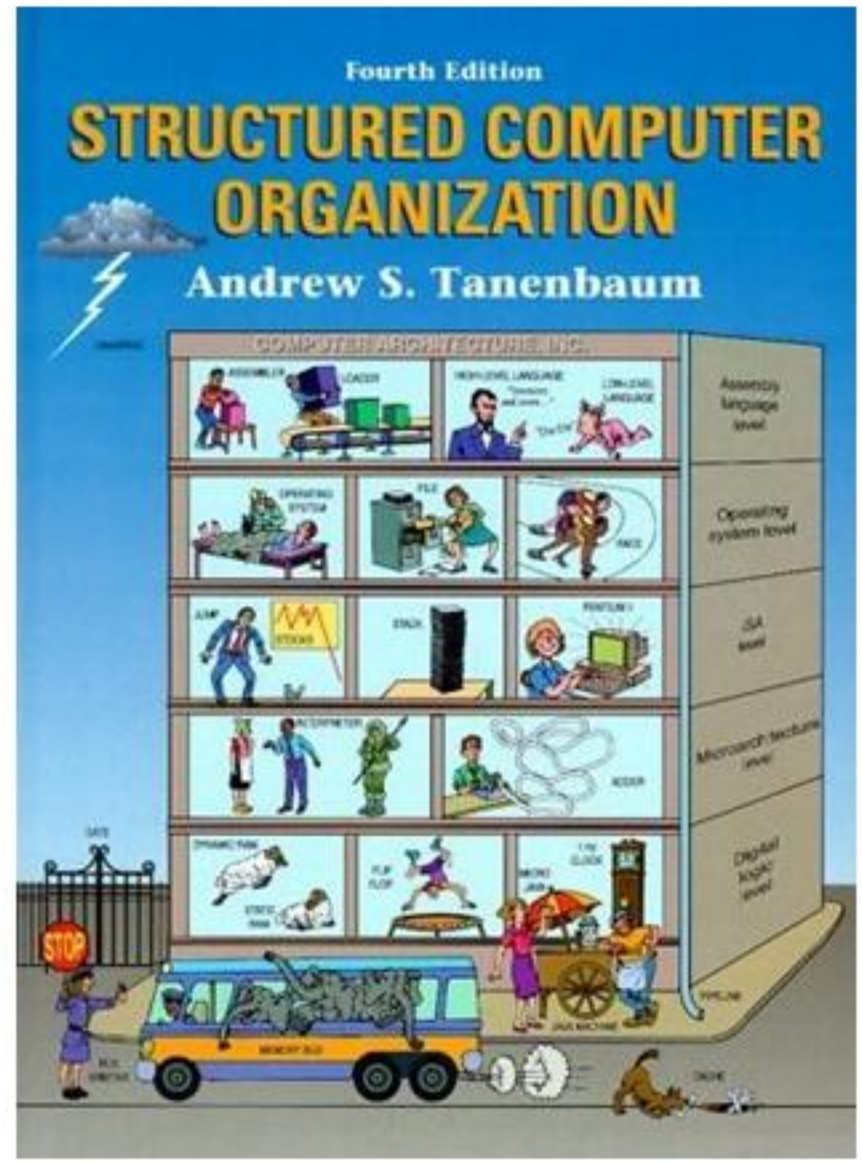
- Tanenbaum, A. S.: Structured Computer Organisation, 6th Edition, Prentice Hall 2012



Doporučená literatura (pokr.)

- Tanenbaum, A. S.: Structured Computer Organisation, 4th Edition, Prentice Hall 1999

(dostupné je i 5. vydání z roku 2005)



Doporučená literatura (pokr.)

Computer Organization and Design Fundamentals

By David Tarnoff - East Tennessee State University

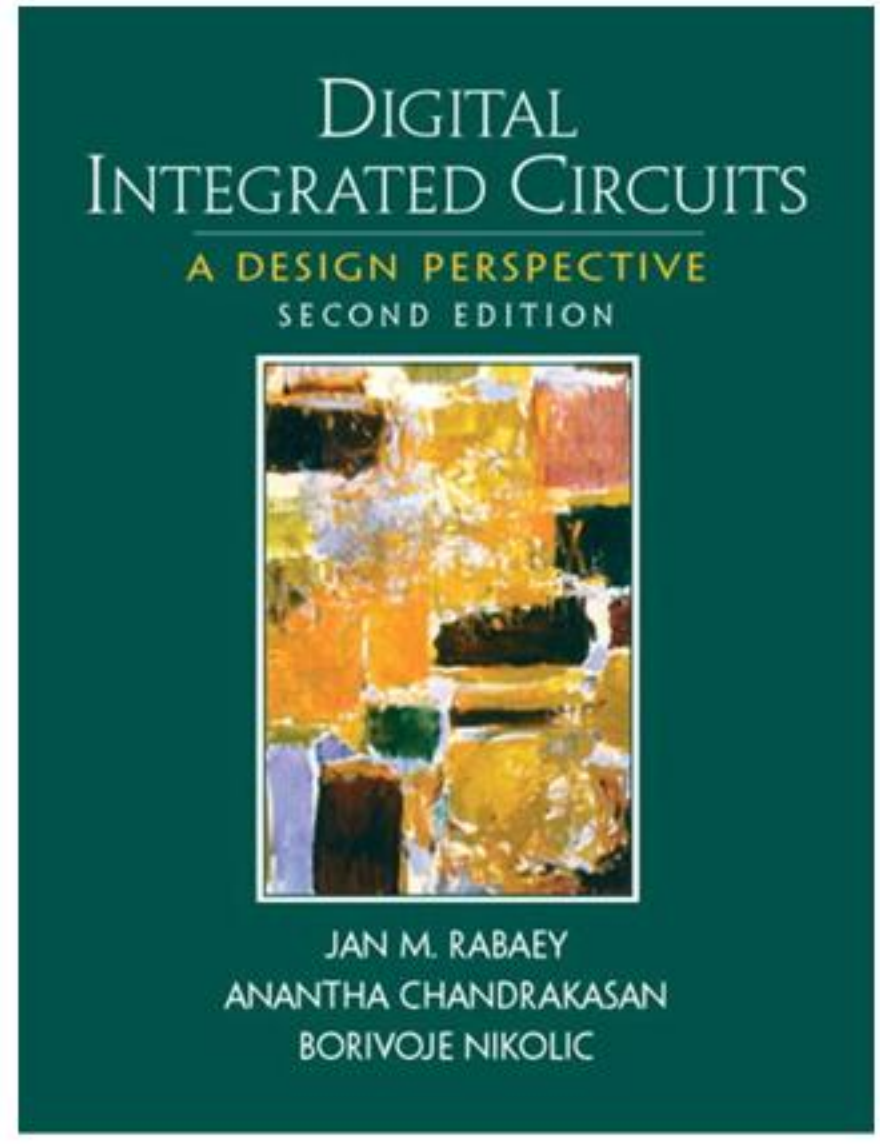
Volně ke stažení !!
– doplňkové čtení

Doporučená literatura (pokr.)

- Digital Integrated Circuits
  Second Edition

  A Prentice-Hall publication by Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic´

http://bwrc.eecs.berkeley.edu/IcBook/index.htm



Doplnění znalostí základů elektroniky

Doporučená literatura (pokr.)

- J. Douša, A. Pluháček: Úvod do počítačových systémů. Skripta ČVUT
- Pluháček, A. : Projektování logiky počítače, ČVUT Praha 2000
- Hlavička, J.: Číslicové počítače II, ČVUT Praha 1997
- Shiva, S.G.: Computer Design and Architecture, Dekker 2000
- Blaauw, G.A.: Computer Architecture: Concepts and Evolution, Addison Wesley 1997

Doplňující údaje

- Možnost absolvovat část studia (obvykle semestr) v zahraničí

Pro odlehčení

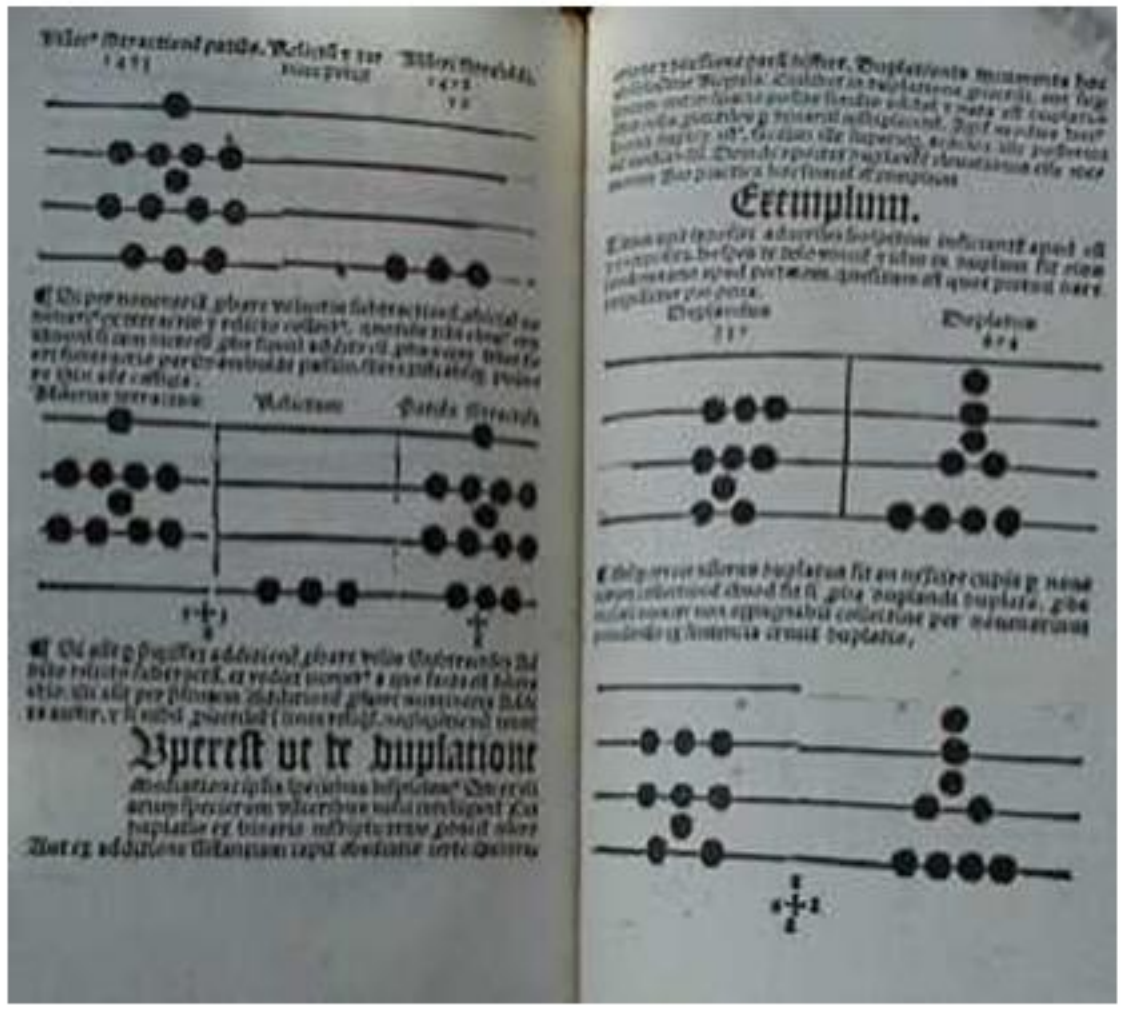
Citát - Steve Meier:

„There are only 10 types of people in the world, those that understand binary and those that don't".

Stručná historie výpočetní techniky (1)

- Abacus - 1100 let př. n. l. Mechanické výpočetní pomůcky. V Číně - " Suan Pan"
- Napierchen Rechenstaebchen - 16. stol.
- Wilhelm Schickardt (1592 - 1635) v roce 1623 návrh počítacího stroje (pro početní úkony +, -, *, /). Stroj se nedochoval, údajně shořel při požáru, zachovaly se pouze nákresy v dopise J. Keplerovi.
- Blaise Pascal (1623 - 1662). V roce 1641 sestrojil do dnešní doby dochovaný sčítací stroj 8-místný Pascaline (Zwinger - Dresden). Celkem bylo vyrobeno více než 50 exemplářů, dochovalo se jen 8. Počítací stroje byly vyrobeny ze dřeva, slonoviny, železa a mědi.

Heinrich Stromer von Auerbach's Algorithmus linealis. Leipzig, 1517.

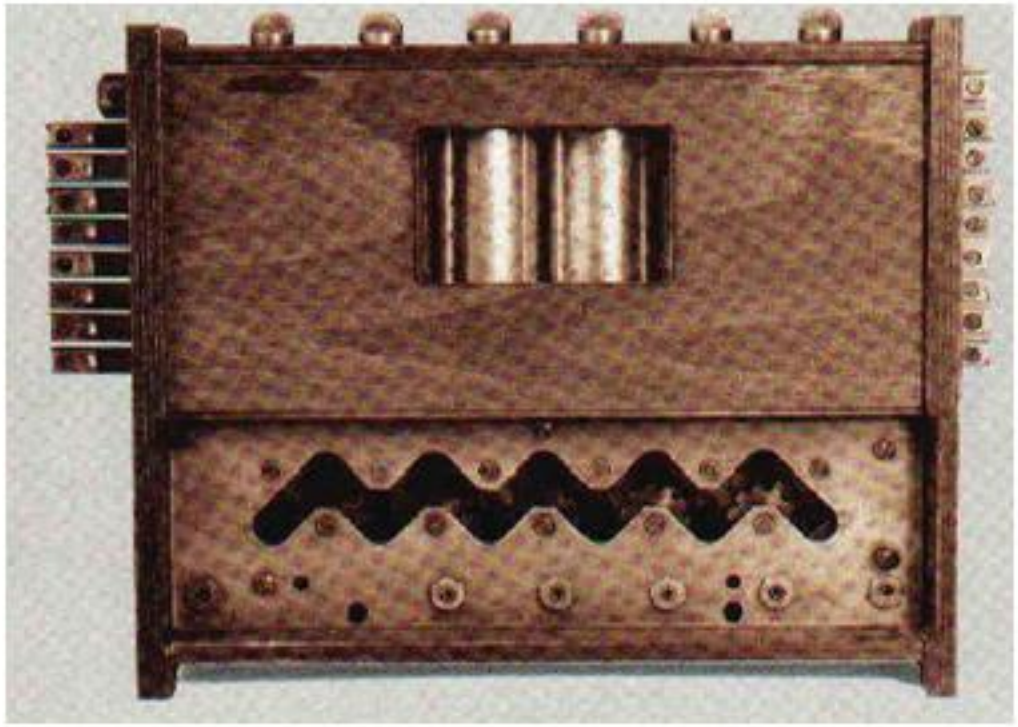


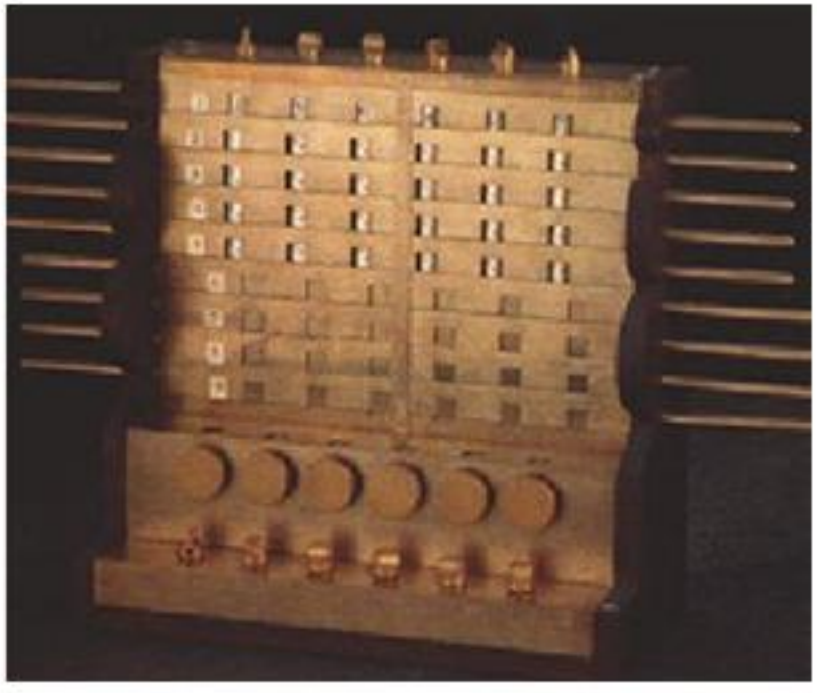
Tento text je jedním z prvních, které byly vytištěny v 15. století a popisovaly použití pomůcky abakus.

Dřevoryt reprezentuje čítače na řádcích abaku.

Popis je věnován sčítání odečítání, násobení a základům aritmetiky.

Wilhelm Schickard

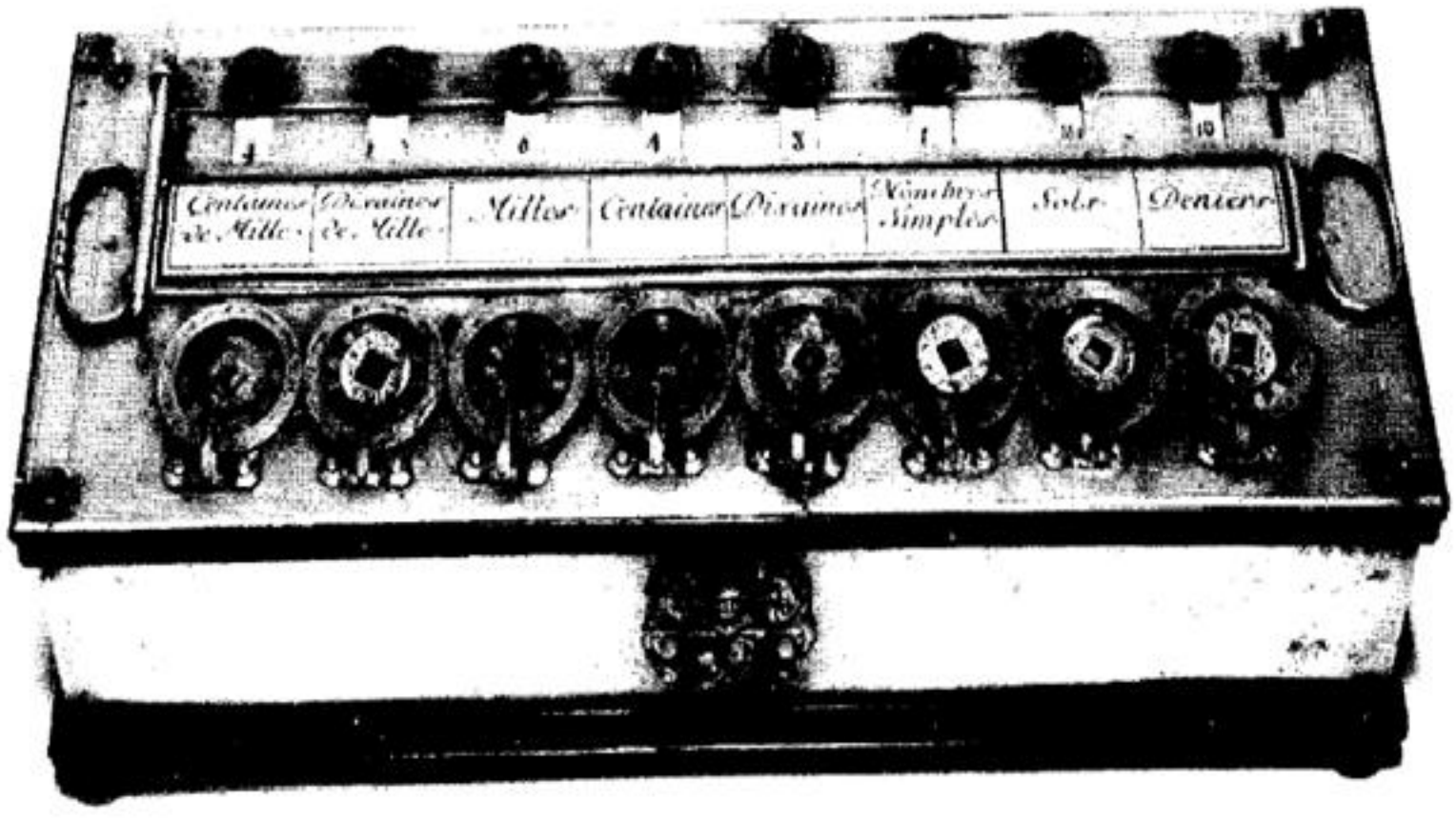




Rok návrhu
1624

Tübingen, (1592 – 1635)

Blaise Pascal

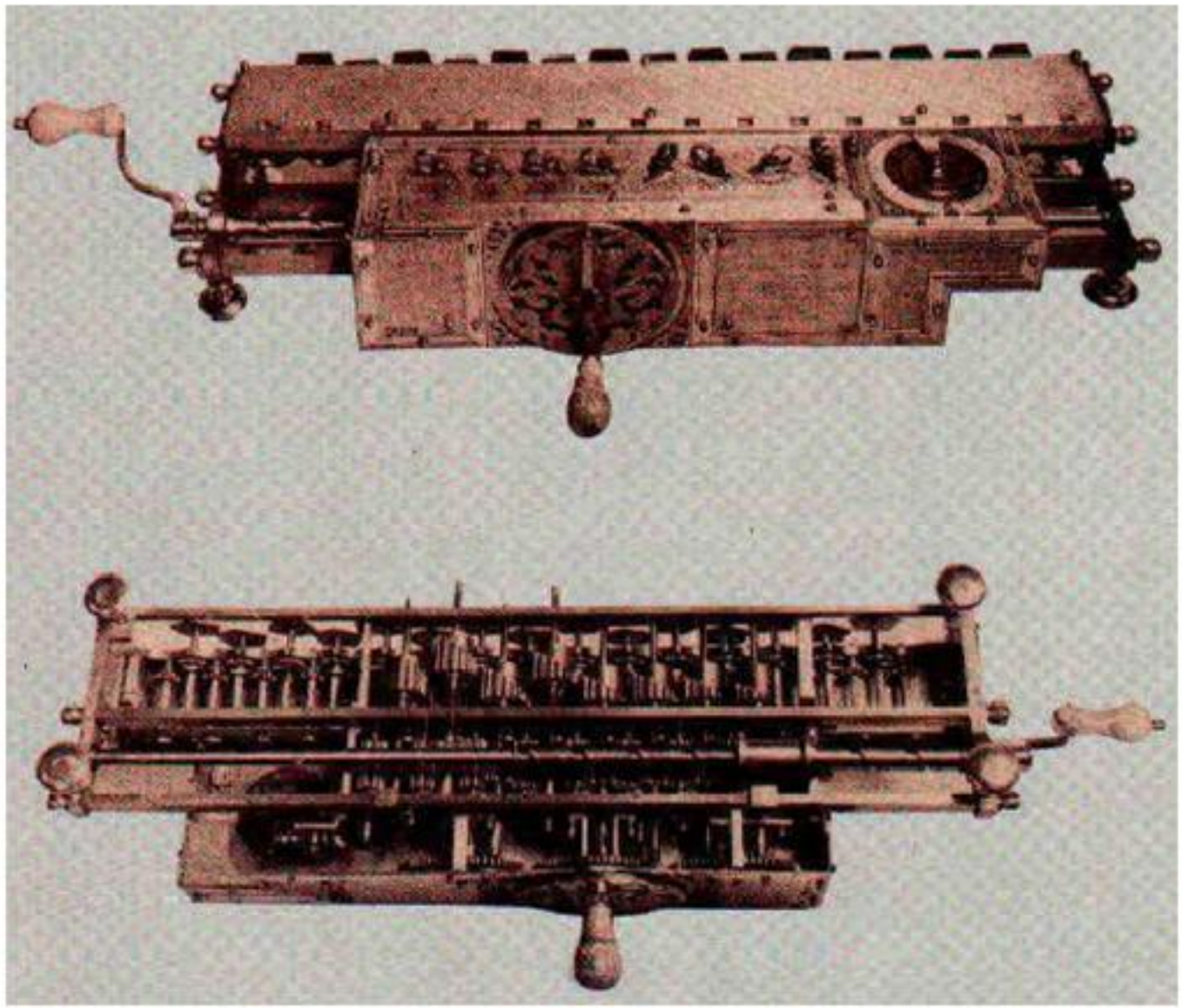


* Clermont-Ferrand, 19. June 1623
+ 1662

Stručná historie výpočetní techniky (2)

- Morland (angličan) - 1660 rozšířil Pascalův stroj o operace násobení a dělení.
- Gottfried Wilhelm Leibnitz (1646 - 1716) zkonstruoval v letech 1671 - 1674 opět dodnes zachovaný počítací stroj, který prováděl všechny čtyři základní početní úkony. Dodnes je uchovaný v muzeu v Hanoveru.
- Seth Partridge - 1650 sestrojil logaritmické pravítko - zástupce analogové techniky.
- Giovanni Poleni z Padovy - pokus o páčkový ruční kalkulační stroj.
- P. M. Hahn – farář z Würtenberku ve spolupráci se švýcarským mechanikem Schusterem sestrojili počítací stroj (+, -, *, /).

Gottfried Wilhelm, Freiherr von Leibnitz

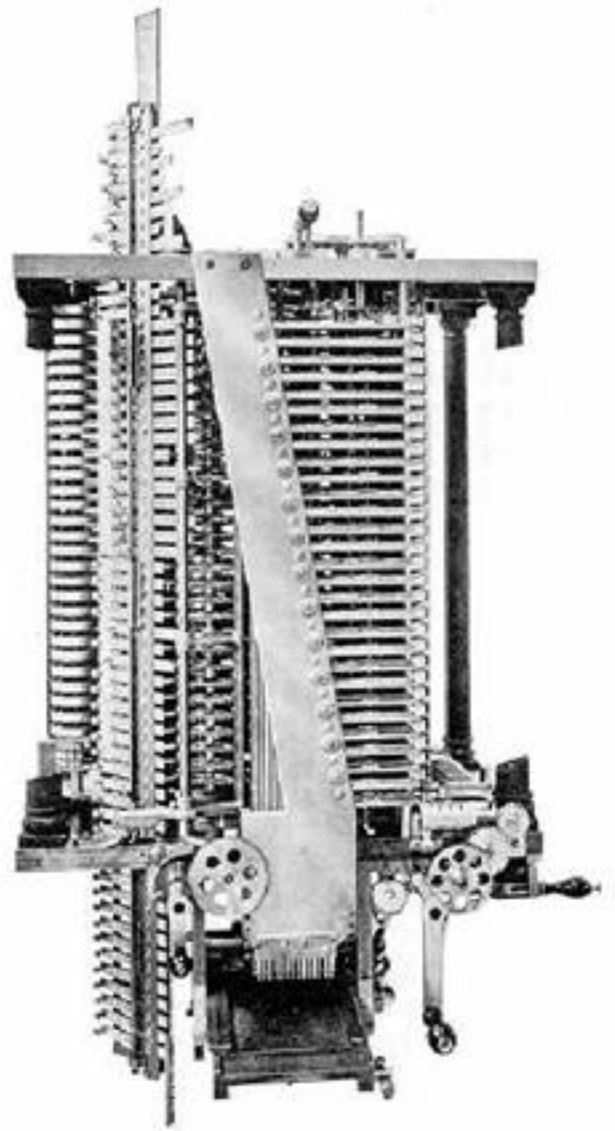


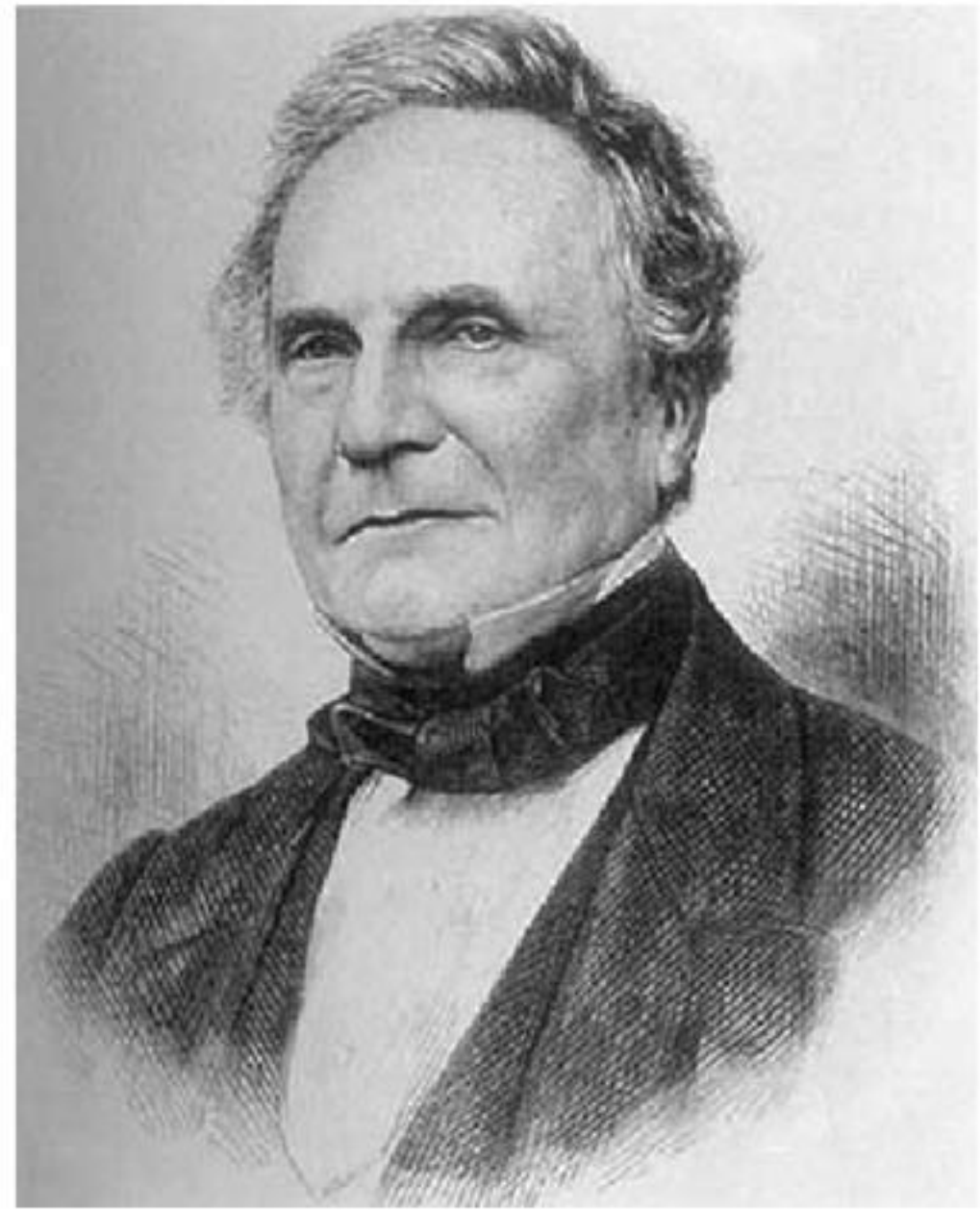
Stručná historie výpočetní techniky (3)

- Jacob Auch 1790 z Vayhingen an der Entz. Stroj zpracovával 8 číslic, prováděl operace +, -, *, /. Jacob Auch pracoval od r. 1798 ve Výmaru jako dvorní mechanik.
- Charles Xavier Thomas (1785 - 1870) zaměstnával mnoho počtářů (byl obchodník). Konstrukce "aritmometru" z r. 1820. Stroj násobil dvě 8 místná čísla za 18 sec. dělení šestnácticiferného čísla osmiciferným trvalo 24 sec. Bylo vyrobeno 1500 kusů !!!
- Olivier Thomas - 1821 Paříž - továrna pro sériovou výrobu počítacích strojů, které byly navrženy podle původní Leibnitzovy konstrukce.
- Charles Babage (1792 - 1871) navrhl mechanický počítací automat, "Difference Engine" (1823). K realizaci nedošlo pro chybějící mechanické prvky.

Charles Babbage

Analytical Engine
(1833-1842)





Charles Babbage

- *Difference Engine* 1823
- *Analytic Engine* 1833

  – Vážný předchůdce moderních digitálních počítačů!

Aplikace

  – Matematické tabulky – astronomie
  – Námořní tabulky – mořeplavba

Základy

  – Libovolnou spojitou funkci lze aproximovat polynomem --- *Weierstrass*

Technologie

  – mechanika – ozubené převody, Jacquardské stavy, jednoduché kalkulátory

Difference Engine

Stroj určený k výpočtu matematických tabulek

Weierstrass:

- Libovolnou spojitou funkci lze aproximovat polynomem
- Každý polynom lze vyčíslit pomocí *diferenčních* tabulek

Příklad:

$f(n) = n^2 + n + 41$

$d1(n) = f(n) - f(n-1) = 2n$

$d2(n) = d1(n) - d1(n-1) = 2$

$f(n) = f(n-1) + d1(n) = f(n-1) + (d1(n-1) + 2)$

vše co potřebujete je sčítačka!

n	0	1	2	3	4
d2(n)			2	2	2
d1(n)		2	4	6	8
f(n)	41	43	47	53	61

Difference Engine

1823
- Babbage' publikoval článek

1834
- Článek si přečetli Scheutz & son ve Švédsku

1842
- Babbage se vzdává myšlenky stroj postavit, pomýšlí již na „Analytic Engine"!
- Babbage svůj stroj nikdy nepostavil. V roce 2002 jej sestrojili pracovníci Science Museum in London, další byl potom postaven v museu Mountain View, California.

1855
- Scheutz vystavuje svůj stroj v Paříži na světové výstavě
- Může počítat polynomy až 6. řádu
- *Rychlost:* 33 až 44 32-ciferných čísel za minutu!

Analytic Engine

1833: Babbageův článek byl publikován

- *koncipován během přestávky v práci na „difference engine“*

Inspirace: *Jacquardské stavy*

- stavy byly řízeny děrnými štítky
  - Soubor štítků s fixními děrami určoval vzor tkaní $\Longrightarrow$ *program*
  - Stejný soubor štítků může být použit k protahování vláken různých barev $\Longrightarrow$ *čísla*

1871: Babbage umírá

- Stroj zůstává nerealizován.

Stručná historie výpočetní techniky (4)

- Charles Babage (1792 - 1871) navrhl další mechanický počítací automat, kde bylo možno nastavit program. “Analytic Engine” (1833). Využil principu tkacích strojů Jacquarda (1805) - děrné štítky. K realizaci nedošlo pro chybějící mechanické prvky.
- Hermann Holerith (1860 - 1929). Z roku 1908 pocházejí první děrnoštítkové stroje. Prováděly jednoduché součty. Místo původního čistě mechanického principu byl využit princip elektromechanický. Po r. 1848 musela rodina odejít do Ameriky.
- William Thomson (lord Kelvin) 1876 objevil princip zpětné vazby.
- Burroughův sčítací stroj, “Čuhelův aritmomet” a “Hlaváčův samočet”. Zmínka v českém časopise “Z říše práce a vědy” z roku 1901.

Burrough's calculator

Burroughův sčítací stroj.

Sčítání dlouhých cifernych řad bývá v obchodech a úřadech nejen prací velmi nesnadnou, duchamornou a zdlouhavou, nýbrž stává se velmi často i zdrojem omylů, které pak nesnadno bývá vyhledati a napraviti.

K zamezení takových vad sestrojeno bylo již mnoho strojů počítacích. V tomto časopise měli jsme již dvakráte příležitost, pojednávati o českých vynálezích tohoto druhu. Bylo to v roč. V. na str. 114., kde jsme popsali Čuhelův arithmet a Hlaváčův samočet.

Dnes předvádíme svým čtenářům nový stroj původu amerického, jejž vynálezce nazval arithmometrem. Zařízen jest v první řadě na sčítání a současné napisování součtu; dá se ho však též užiti k násobení.

Jak obě naše vyobrazení okazují, je to nevelká skřínka, 28 *cm* široká, 32 *cm* vysoká a 38 *cm* dlouhá, která na svém povrchu má 9 řad po 9 klávesích, na zadní stěně pak svitek papíru, na nějž stroj sám si vytiskuje součet. Jak na obr. 2. vidíme, označena jest každá řada klávesů stejným číslem, tak že spodní řada má samé jedničky, druhá samé dvojky atd., až nejvyšší řada samé devítky. Nulových klávesů vůbec tu není; nuly tisknou se samy na příslušných místech bez našeho přičinění.

Klávesy však neoznačují pouze cifernou hodnotu, která na nich jest napsána, nýbrž zároveň místní hodnotu od setin počínajíc až do milionů, což na hořejší desce pro lepší přehled označeno jest kovovými lištami. Chceme-li tudíž strojem napsati na příklad číslo 30.278·56, stiskneme postupně 3 na třetím místě (desetitisíce), 2 na pátém místě (sta), 7 na šestém, 8 na sedmém, 5 na osmém a 6 na devátém místě. Nula na čtvrtém místě objeví se již sama. K otištění tohoto čísla jest pak ještě zapotřebí zatáhnouti kliku na pravé straně skříně do předu.

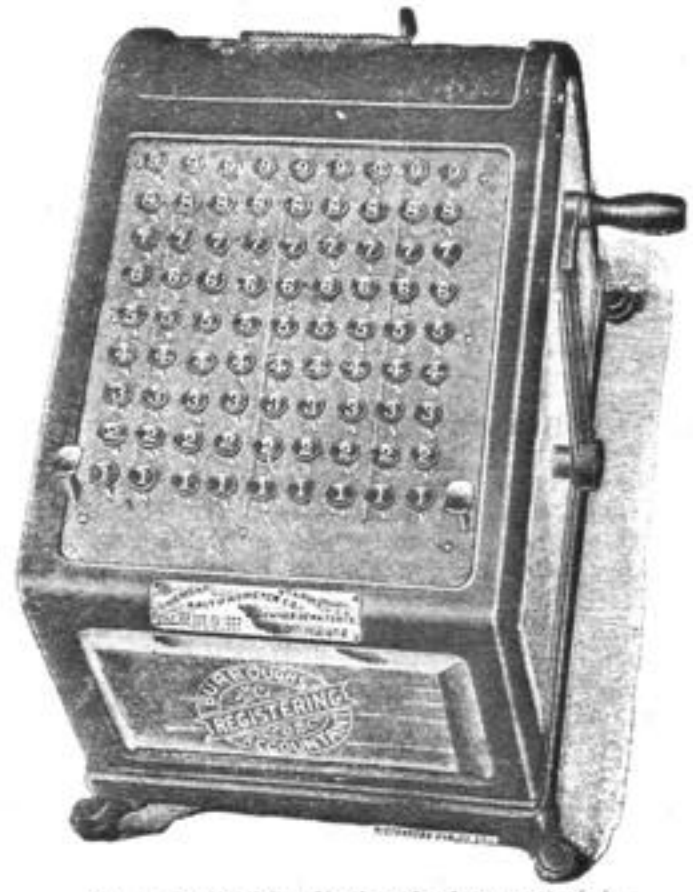
Máme-li totéž číslo několikráte po sobě napsati, stiskneme prázdný kláves na pravé straně (dole) a otočíme klikou tolikráte, kolikráte číslo napsati máme. Této okolnosti užijeme výhodně k násobení. Má-li se na př. číslo 4627 násobiti 26, napíšeme strojem toto číslo šestkráte, pak číslo 46270 dvakráte, načež provedeme součet. Na papíře objeví se pak otisk tento:

```
  4627
  4627
  4627
  4627
  4627
  4627
 46270
 46270
------
120302
```

Pro sčítání různých sčítanců vyhmatáme každého sčítance zvlášť a pomocí kliky učiníme otisky, načež otočením kliky nabudeme bezvadného součtu.

Je-li počítání ukončeno, stiskneme prázdný kláves na levo (dole), čímž se celý stroj do původního (nulového) stavu navrátí. Tohoto

Obr. 1. Burroughův sčítací stroj od strany.



Obr. 2. Burroughův sčítací stroj s hora a od předu.

klávesu lze též užiti k opravám chyb, pokud ještě nejsou vytištěny.

Avšak Burroughův arithmometr provádí také sečítání složitějšího druhu, jak se to při knihování často vyskytuje.

Má-li částečný součet ve stroji setrvati, aby mohl na konec s jinými částečnými součty v konečný součet býti spojen, provede se celý pohyb klikou, aniž by se některý z prázdných klávesů byl stiskl, po té stiskne se kláves sečítací (pravý), pošine se klika ku předu, a dříve než se klika počne zpět pohybovati, pustí se kláves.

Tím upevněn jest součet ve stroji pro další sečítání.

O výhodnosti a výkonnosti tohoto nového stroje počítacího svědčí výmluvně okolnost, že říšský poštovní úřad v Berlíně má již nyní v užívání 116 takových strojů, ačkoli každý stojí 1500 K.

Stručná historie výpočetní techniky (5)

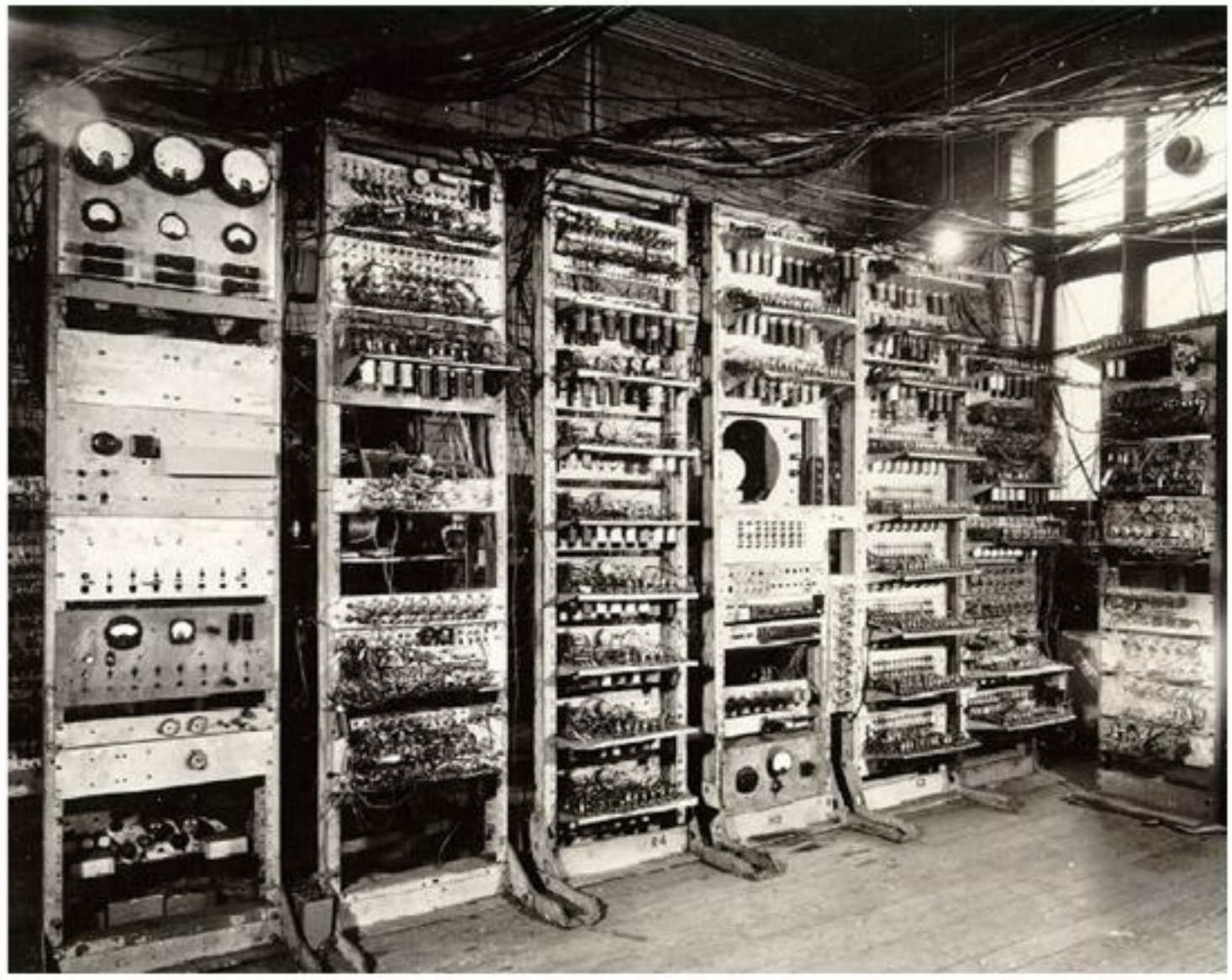
- Udo Knor 1914 předvedl mechanický diferenciální analyzátor (řešení differenciálních rovnic).
- Prof. Konrad Zuse (1910 - 1995) 1941 Německo - sestavil releové počítače Z1, Z2 a Z3. Poslední obsahoval 2500 relé.
- Howard Aiken a další 1941-4 USA - konstrukce počítačů Mark I, Mark II. Do vývoje investoval koncern IBM. Postaven z relé, délka 16 m, vysoký 2.5 m.
- John von Neuman (1903 - 1957) 1947 formuloval princip interního programového řízení.

Prof. Konrad Zuse

Plankalkül, erste Programmiersprache

(*1910 - +1995)

Manchester - Mark 1

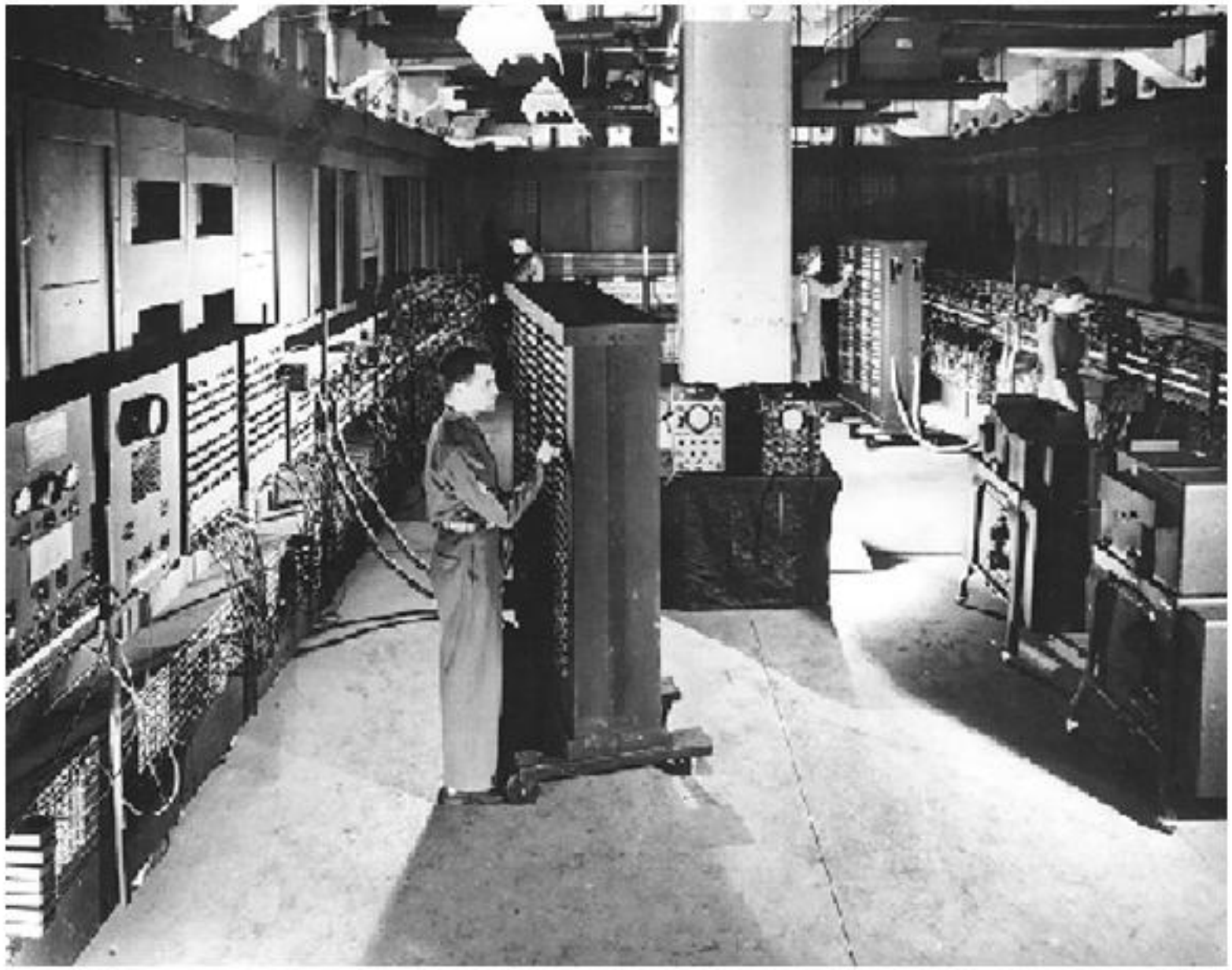


Stručná historie výpočetní techniky (6)

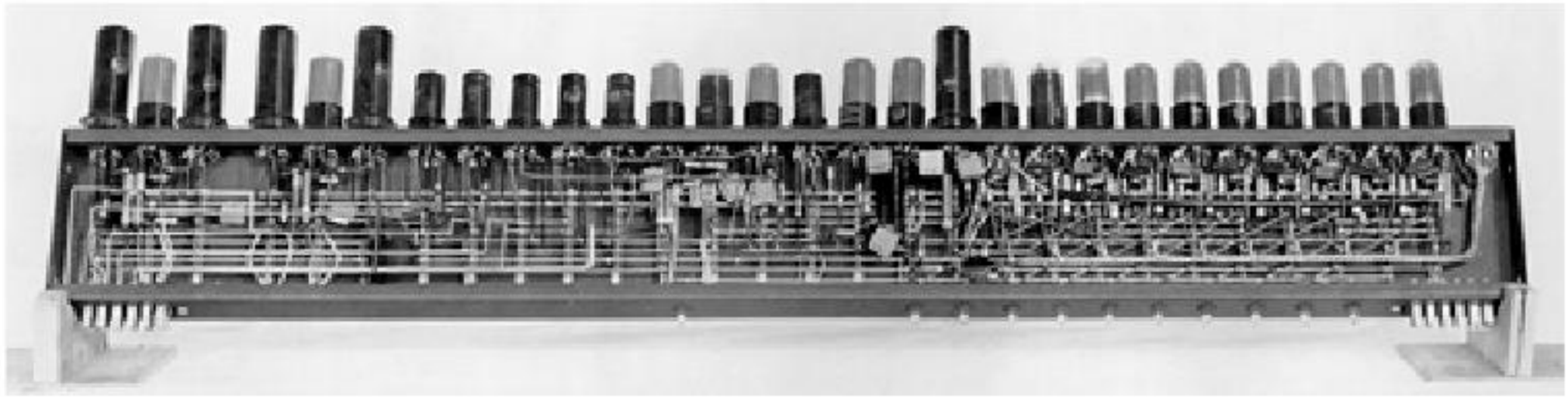
- Eniac
  - Kompletovaný r. 1946 na Moore School of Engineering (University of Pensylvania) pod vedením Eckerta a Mauchly. Jedná se o první elektronický počítač.
  - Obsahoval: 18000 elektronek a 500 relé.
  - Měl 20 registrů a 312 slov ROM.
  - Určen pro výpočet balistických trajektorií.
  - Rychlost 5000 op./sec.
  - Aritmetické operace se prováděly sériově, v desítkové soustavě. Každá číslice byla uložena v kruhovém registru, složeném z deseti klopných obvodů.
  - Neprogramoval se, řešená úloha byla pevně propojena.
  - Používal se 10 let.

Von Neuman zde působil jako konzultant a ovlivnil konstruktéry Eckerta a Mauchlyho, aby použili princip interního programového řízení v dalším projektu. V té době strávil léto ve Philadelphii také Wilkes, který postavil pak v roce 1949 na universitě v Cambridge (U.K.) EDSAC.

ENIAC

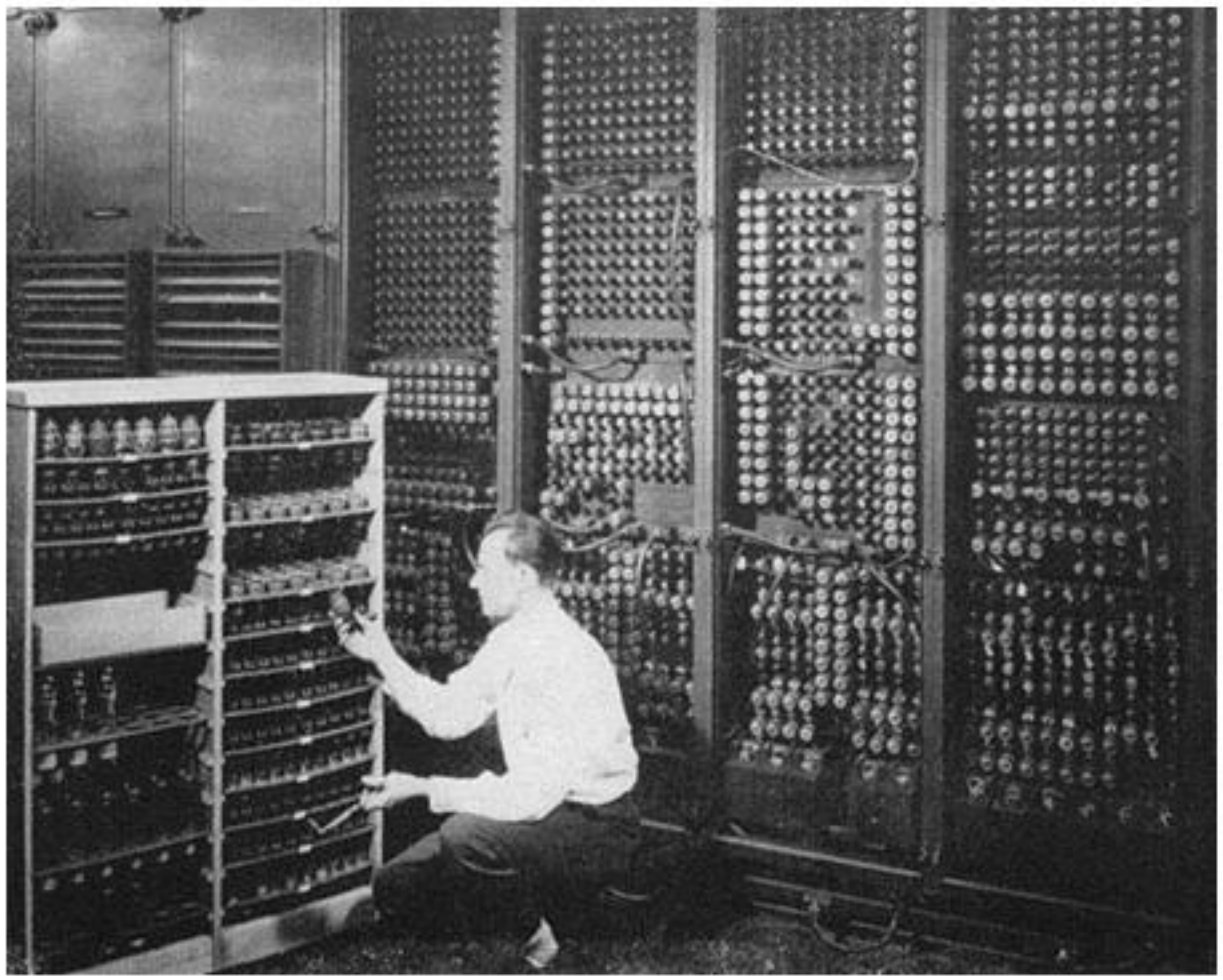


Eniac – jednotka akumulátoru



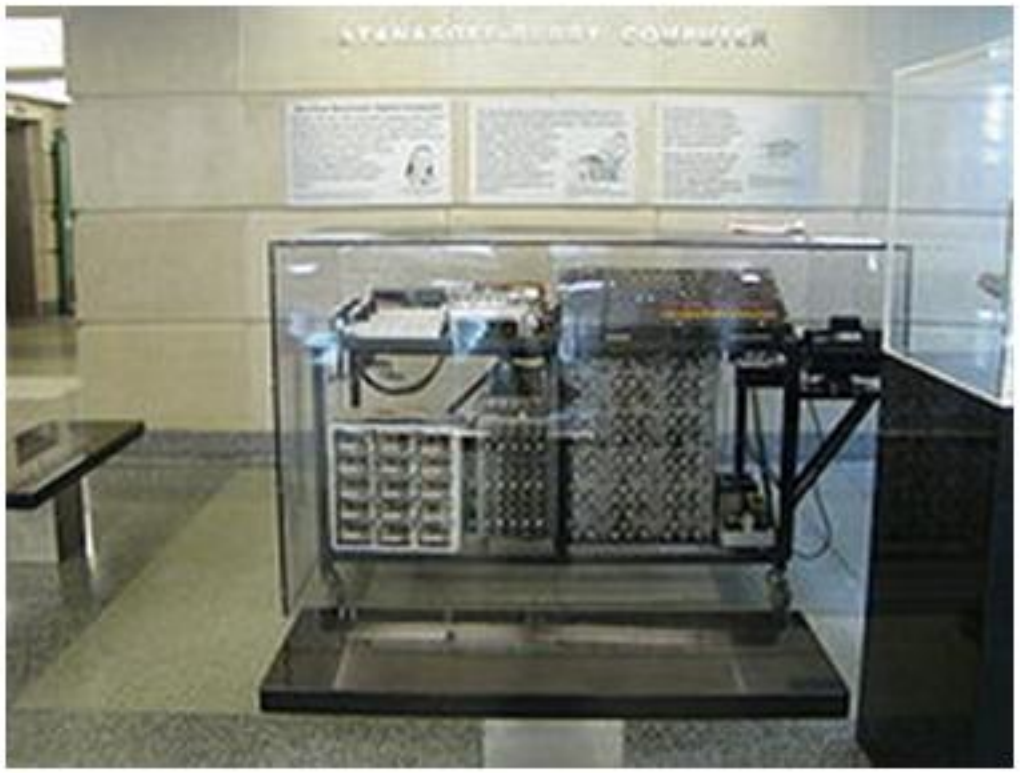
ENIAC

- ENIAC (Electronic Numerical Integrator and Calculator): Jeden z prvních electronických počítačů pro obecné použití (general purpose)
  - Eckert a Mauchly, ve firmě U. Penn, založené U.S. Army
  - Byl zprovozněn během II. sv. války



Náhrada jedné z 18,000 elektronek

Linear Equation Solver



1930's:

- Atanasoff staví the Linear Equation Solver (Iowa State University)
- Měl 300 elektronek!
- Speciální binární digitální kalkulátor
- Dynamická RAM (hodnoty uloženy na kondenzátorech se zotavením)

Aplikace:

- Lineární a integrální diferenciální rovnice

Předchůdce:

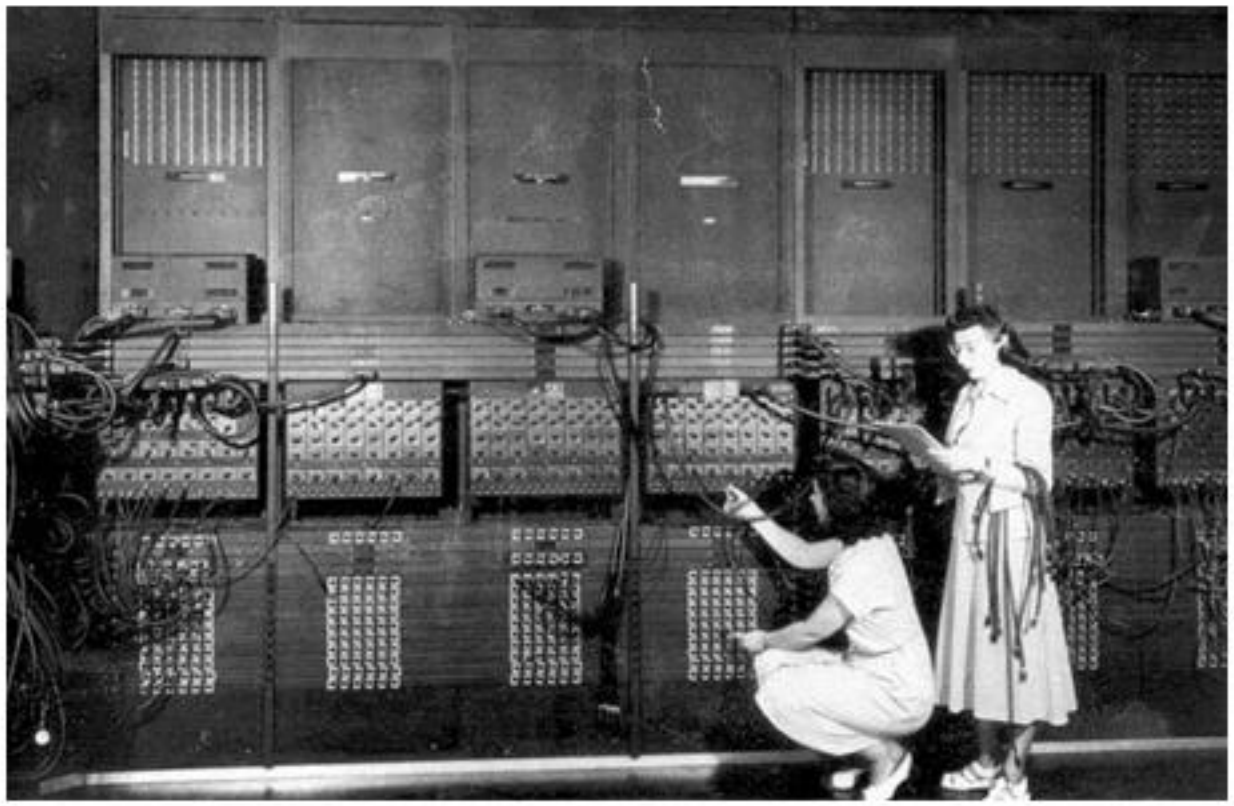
- Vannevar Bush's Differential Analyzer
  --- *analogový počítač*

Technologie:

- Elektronky a elektromechanická relé

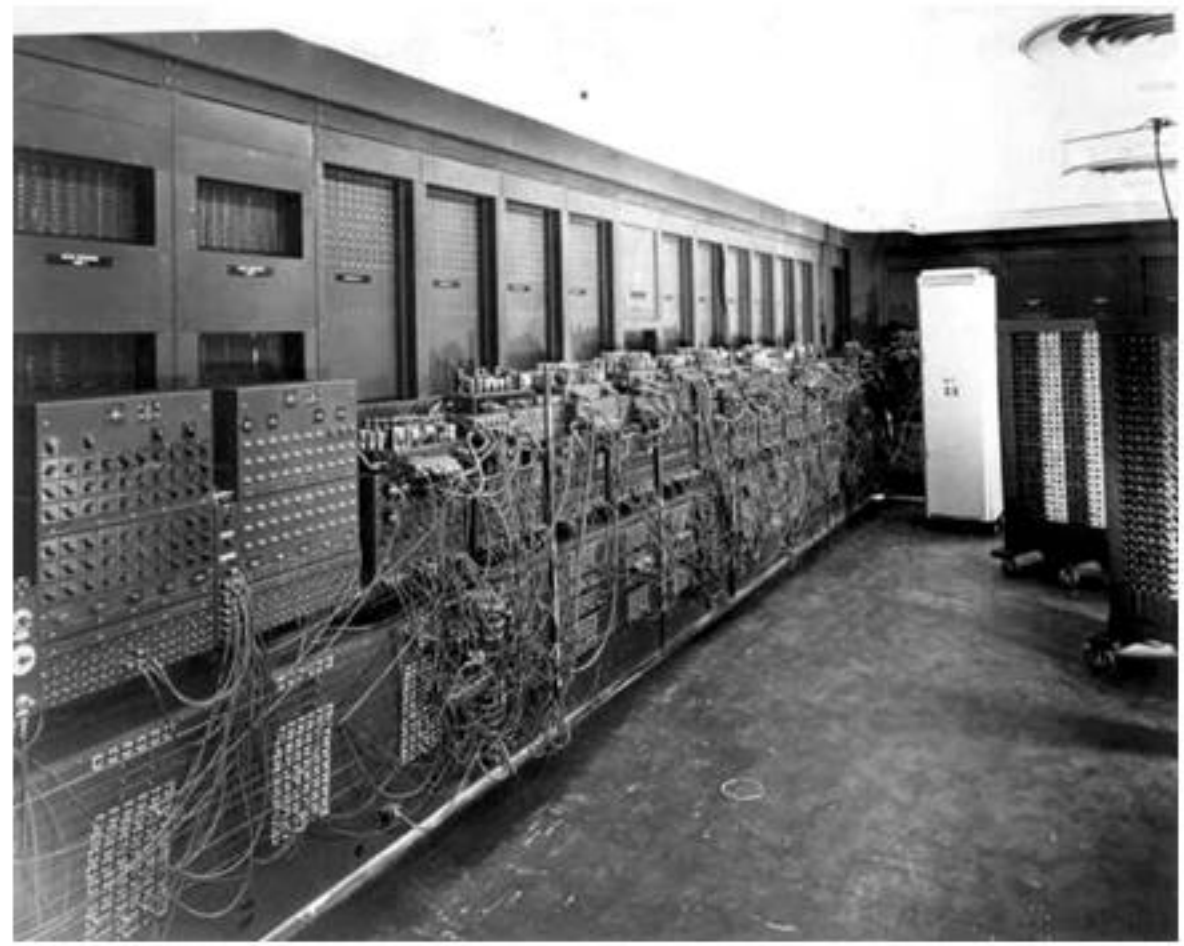
Atanasoff rozhodl, že nejvhodnějším režimem výpočtu bude využití elektronické implementace binárních cifer.

ENIAC



Vytváření programu (Wiring a program)

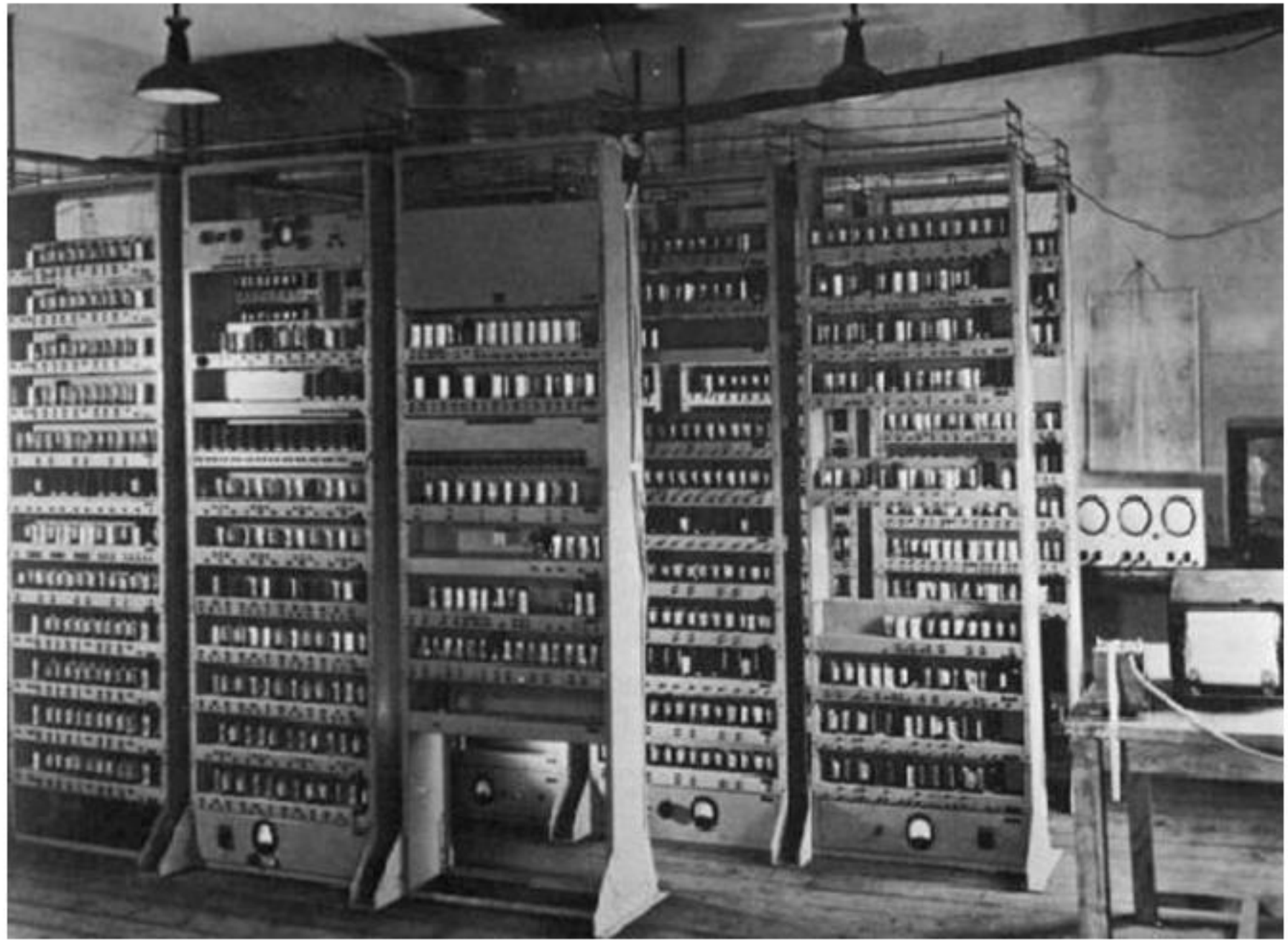
Naprogramovaný počítač (A wired program)



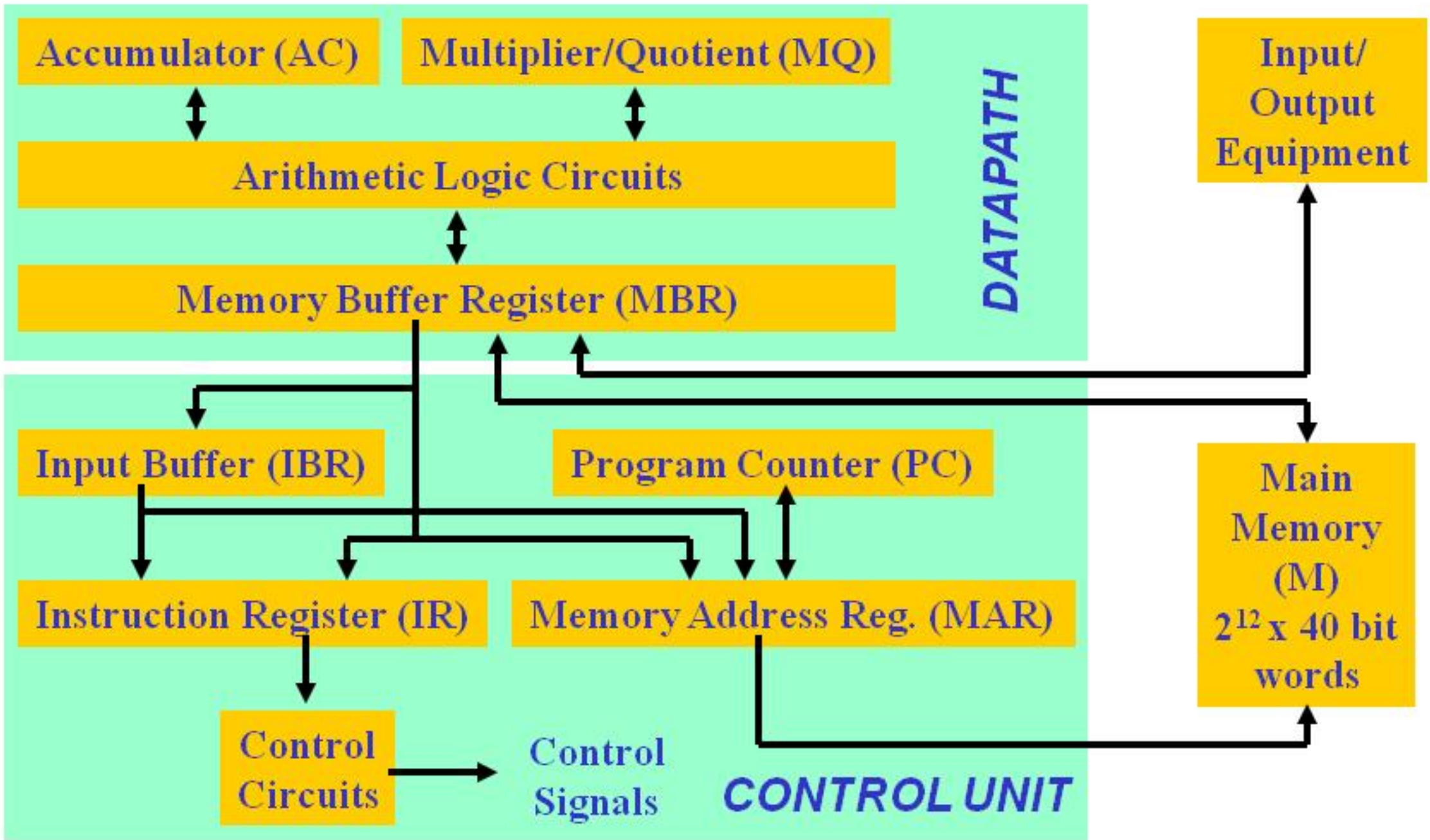
Stručná historie výpočetní techniky (7)

- Edsac
  - První počítač, který pracoval s programem, uchovaným v paměti.
  - Primární paměť měla kapacitu 1024 slov, realizovanou jako rtuťovou zpožďovací linku.
  - Dále bubnovou sekundární paměť o kapacitě 4600 slov.
  - Projekt Edsac byl zastaven, když oba tvůrci odešli ze školy a založili vlastní společnost
- IAS
  - Von Neumann a Goldstein sestavují počítač na Institute for Advanced Study.
  - Stejnojmenný počítač obsahuje známých 5 bloků.
  - Počítač má 64 instrukcí orientovaných na práci s akumulátorem. Paměť měla 4096 slov po 40 bitech.
  - Každé slovo obsahovalo 2 instrukce.

EDSAC, University of Cambridge, UK, 1949

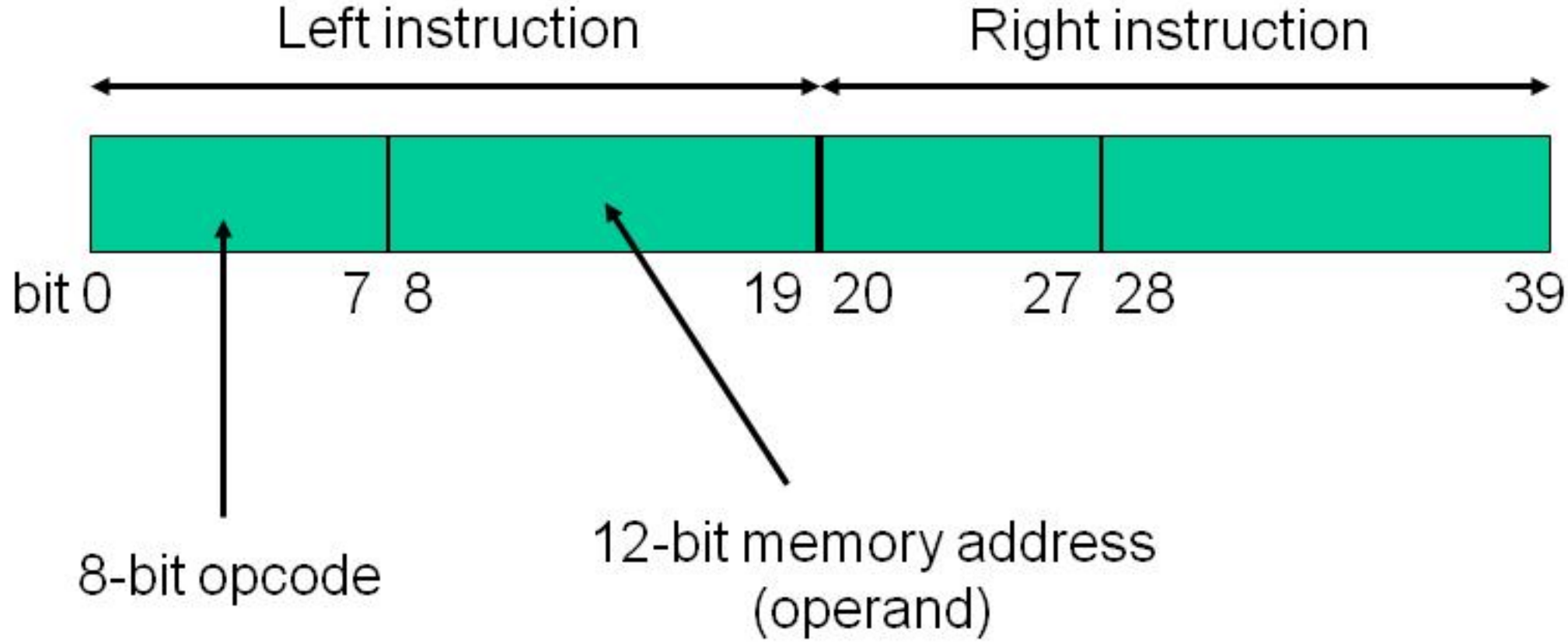


Organizace počítače IAS



Strojní jazyk IAS

40-bitové slovo, obsahující dvě strojní instrukce



Ref: **J. P. Hayes, *Computer Architecture and Organization*, New York: McGraw-Hill, 1978.**

Instrukce přenosu dat (IAS)

Instruction	*Opcode*	*Description*
• LOAD MQ	00001010	AC←MQ
• LOAD MQ,M(X)	00001001	MQ←M(X)
• STOR M(X)	00100001	M(X)←AC
• LOAD M(X)	00000001	AC←M(X)
• LOAD – M(x)	00000010	AC← – M(X)
• LOAD |M(X)|	00000011	AC←|M(X)|
• LOAD – |M(X)|	00000100	AC← – |M(X)|

Instrukce nepodmíněného skoku

Instruction	*Opcode*	*Description*
• **JUMP M(X,0:19)**	**00001101**	**next instruction M(X,0:19)**
• **JUMP M(X,20:39)**	**00001110**	**next instruction M(X,20:39)**

Instrukce podmíněného skoku

Instruction	*Opcode*	*Description*
• JUMP +M(X,0:19)	00001111	IF AC≥0, then next instruction M(X,0:19)
• JUMP +M(X,20:39)	00001110	IF AC≥0, then next instruction M(X,20:39)

Aritmetické instrukce IAS

Instruction	*Opcode*	*Description*
• ADD M(X)	00000101	AC←AC+M(X)
• ADD |M(X)|	00000111	AC←AC+|M(X)|
• SUB M(X)	00000110	AC←AC–M(X)
• SUB |M(X)|	00001000	AC←AC–|M(X)|
• MUL M(X)	00001011	AC,MQ←MQxM(X)
• DIV M(X)	00001100	MQ,AC←MQ/M(X)
• LSH	00010100	AC←AC x 2
• RSH	00010101	AC←AC / 2

Instrukce modifikace adresy

	Instruction	*Opcode*	*Description*		
•	STOR M(X,8:19)	00010010	M(X,8:19)	←	AC(28:39)
•	STOR M(X,28:39)	00010011	M(X,28:39)	←	AC(28:39)

Sčítání dvou čísel na IAS

- Předpokládejme, že čísla jsou uložena v paměti na adresách 100 a 101
- Výsledek – suma se má uložit na adresu 102

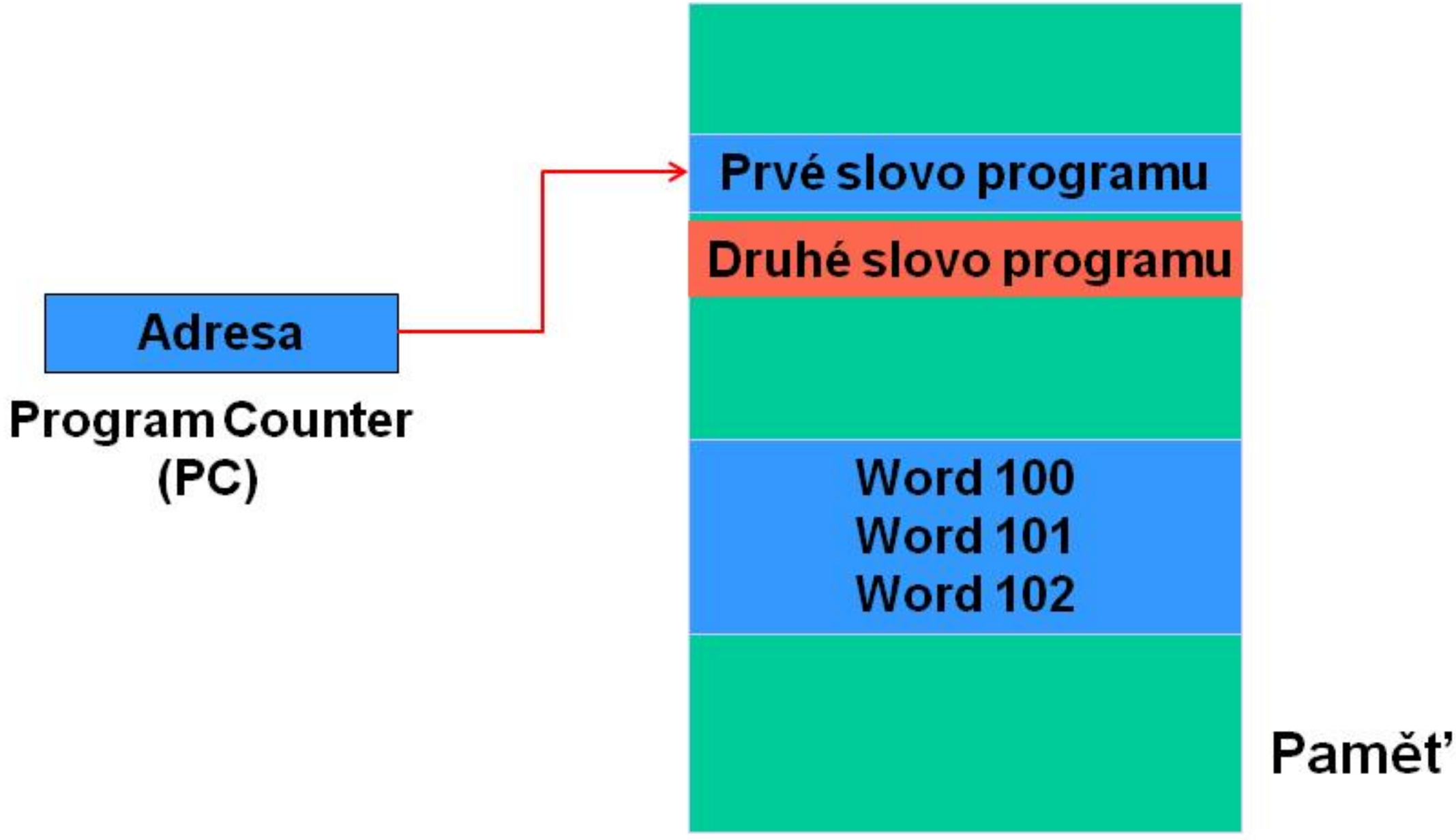
Instruction	*Opcode*	*Description*
LOAD M(100)	00000001	AC←M(100)
ADD M(101)	00000101	AC←AC+M(101)
STOR M(102)	00100001	M(102)←AC

Strojní kód IAS

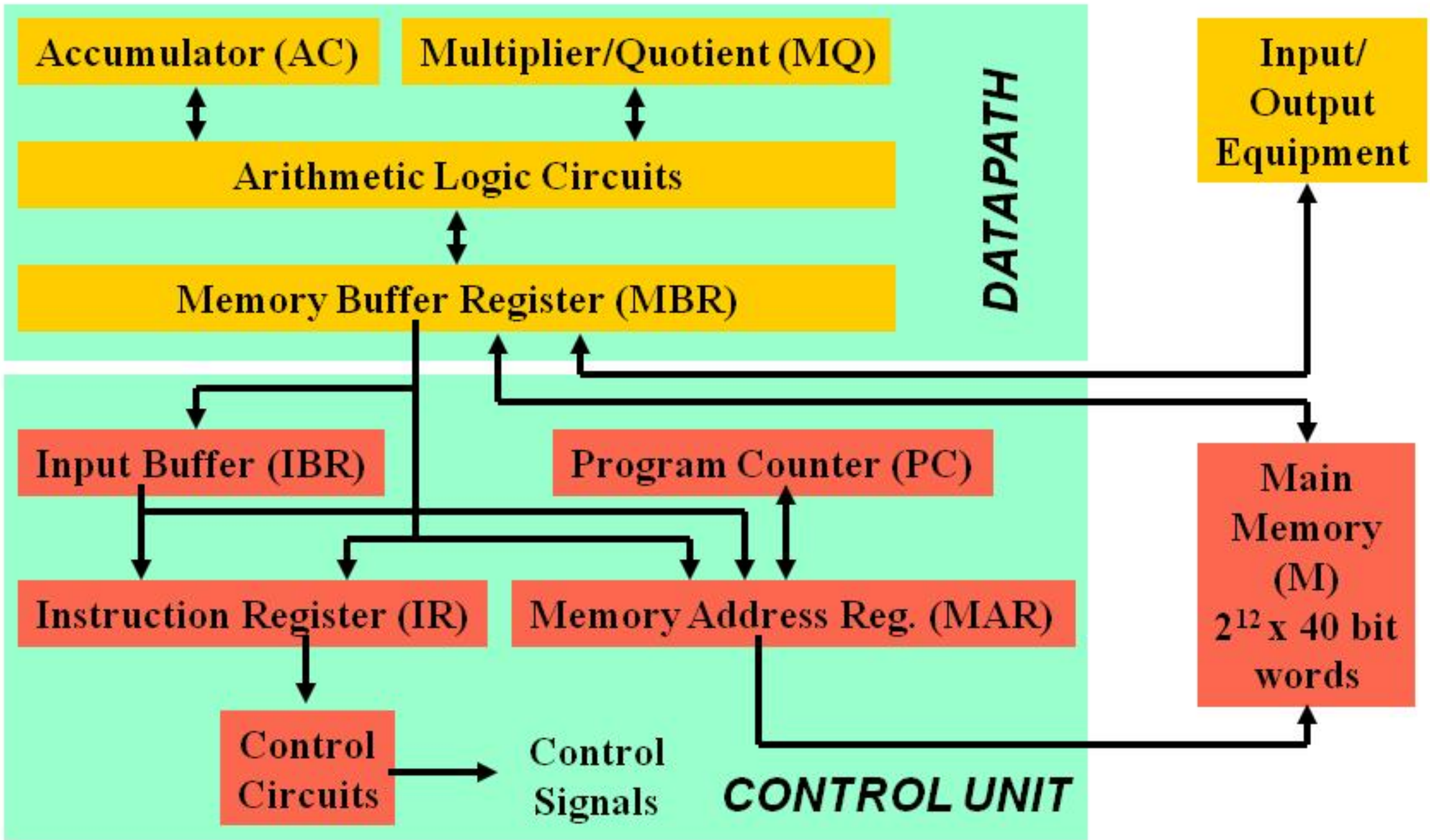
00000001	000001100100	00000101	000001100101

00100001	000001100110	00000000	000000000000

Program uložený v paměti



Provedení programu



Instrukční cykly IAS

- Strojní kód uložen v paměti na po sobě jdoucích adresách.
- Startovací adresa se uloží do čítače adres instrukcí (PC).
- Start programu: MAR ← PC
- Čtení paměti: IBR ← MBR ← M(MAR), fetch cycle
- Levá instrukce se zapíše do IR a adresa 100 do MAR
- Čtení paměti: AC ← M(100), fetch cycle
- Pravá instrukce se zapíše do IR a adresa 101 do MAR
- Čtení paměti a operace součtu: AC ← AC + M(101), execution cycle
- PC ← PC + 1

Instrukční cykly IAS (pokr.)

- MAR ← PC
- Čtení paměti: IBR ← MBR ← M(MAR), fetch cycle
- Levá instrukce se zapíše do IR a adresa 102 do MAR
- MBR ← AC
- Operace zápisu do paměti

Stručná historie výpočetní techniky (8)

Rychle se rozbíhají další projekty:

- Illiac - University of Illinos
- Johniac - RAND Corporation
- Maniac - Los Alamos
- Weizac - Weizman Institute in Israel

Stručná historie výpočetní techniky (9)

- Whirlwind - MIT (1947-1951)
  - Slovo 16 bitů, 5 bitů OP.C.
  - 27 instrukcí
  - 2048 slov
  - 20000 op/s., r. 1951 byla paměť na principu paměť. elektronek nahrazena feritovou jádrovou pamětí o kapacitě 2K slov a dobou cyklu 8 us

!!! MTBF byla 20 minut !!!

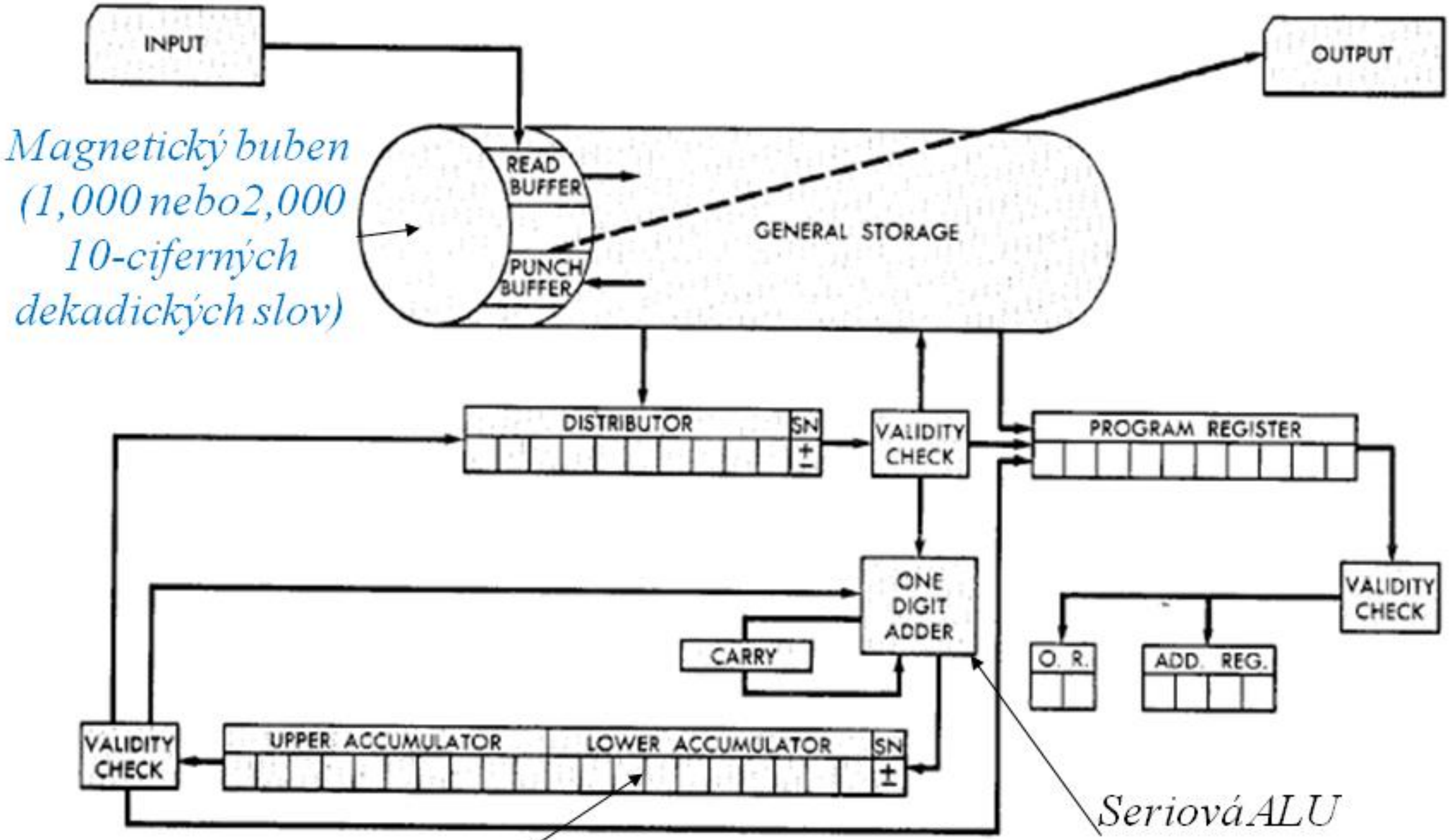
Eckert a Mauchly opustili universitu v Pensylvanii a začali stavět malý binární počítač Binac. Dva procesory, výsledek se porovnával kvůli spolehlivosti. Většinou se neshodly.

Binac (1950) - Nepracoval podle programu v paměti a nikdy ho neodladili.

Stručná historie výpočetní techniky (10)

- Univac I (1951)
  - Magnetická páska 128 char/inch, 1.44 Mbit max.
  - čtení vpřed i vzad
  - používal desítkovou soustavu, 12 cifer
  - instrukce obsahovala 1 adresu, dvě tvořily slovo
  - pamět – rtuťová zpožďovací linka
- Univac II
  - Kompatibilní s Univac I
  - používal feritovou paměť

IBM 650 (1953-4)



Magnetický buben (1,000 nebo 2,000 10-ciferných dekadických slov)

Seriová ALU

20-ciferný akumulátor

[From 650 Manual, © IBM]

IBM 650 z hlediska programátora

Bubnový stroj se 44 instrukcemi

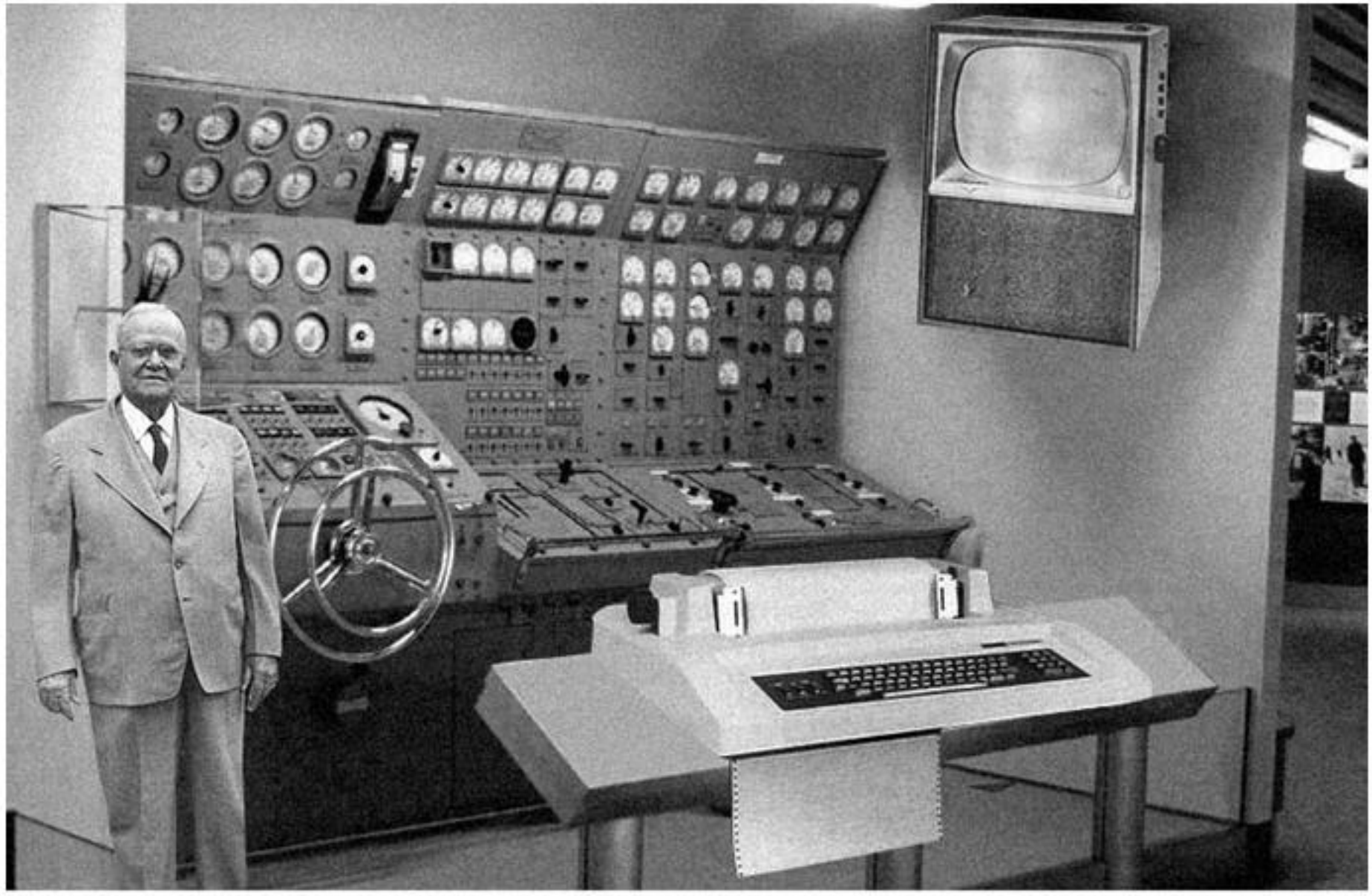
- Instrukce: 60 1234 1009
  - "Přečti obsah buňky 1234 do *distributoru*; dále do *horní části akumulátoru*; vynuluj *dolní část akumulátoru* a pak jdi na adresu 1009 pro další instrukci."

Dobří programátoři optimalizovali umístění instrukcí na bubnu tak, aby se redukovala latence bubnové paměti!

Stručná historie výpočetní techniky (11) - počítače druhé generace

- 1955 to 1964
- Elektronky byly nahrazeny tranzistory
- Magnetické jádrové paměti
- Floating-point aritmetika
- Jazyky vyšší úrovně: ALGOL, COBOL a FORTRAN
- Systémový software: kompilátory, knihovny podprogramů, batch processing
- Příklad: IBM 7094

Představy o PC z let minulých



Scientists from the RAND Corporation have created this model to illustrate how a "home computer" could look like in the year 2004. However the needed technology will not be economically feasible for the average home. Also the scientists readily admit that the computer will require not yet invented technology to actually work, but 50 years from now scientific progress is expected to solve these problems. With teletype interface and the Fortran language, the computer will be easy to use and only

Problém kompatibility u IBM

Na počátku 60-tých let, měla firma IBM
4 nekompatibilní řady počítačů!

701 → 7094
650 → 7074
702 → 7080
1401 → □ 7010

Každý systém měl svůj vlastní:

- instrukční soubor
- I/O systém a sekundární paměť:
  magnetické pásky, bubny a disky
- asemblery, kompilátory, knihovny,...
- svoje vlastní zajištění na trhu, ...

⇒ *IBM 360*

IBM 360 : Premisy návrhu

Amdahl, Blaauw and Brooks, 1964

- Návrh musí být podřízen požadavkům na růst a rozvoj následných řad počítačů
- Obecná metoda pro připojování periferií
- Celkový výkon – *počet odpovědí za měsíc* lépe než *počet bitů za mikrosekundu* ⇒ *programovací prostředky*
- Stroj se musí sám kontrolovat, bez manuálních zásahů
- Vestavěná *hardwarová kontrola chyb* a *detekční prostředky* pro zkrácení doby poruchy
- Jednoduché sestavení systémů s redundantními I/O zařízeními, pamětmi atd. pro dosažení „*fault tolerance*“
- Řešení některých problémů vyžaduje *floating point* zobrazení delší než 36 bitů

IBM 360: *Univerzální registrový (GPR) stroj*

- Stav procesoru
  - 16 univerzálních 32-bitových registrů
    - *mohou být využity jako indexregistry nebo jako báze*
    - *registr 0 má zvláštní vlastnosti*
  - 4 FP 64-bitové registry
  - Program Status Word (PSW)
    - *PC*, podmínkové kódy, *řídící bity*
- 32-bitový stroj se 24-bitovou adresou
  - ale žádná instrukce neobsahovala 24-bitovou adresu!
- Datové formáty
  - 8-bitové byty, 16-bitová půlslova, 32-bitová slova, 64-bitová dvojslova

IBM 360 je příčinou proč má byte dnes 8-bitů!

IBM 360: Počáteční implementace

	Model 30	. . .	*Model 70*
Storage	8K - 64 KB		256K - 512 KB
Datapath	8-bit		64-bit
Circuit Delay	30 nsec/level		5 nsec/level
Local Store	Main Store		Transistor Registers
Control Store	Read only 1μsec		Conventional circuits

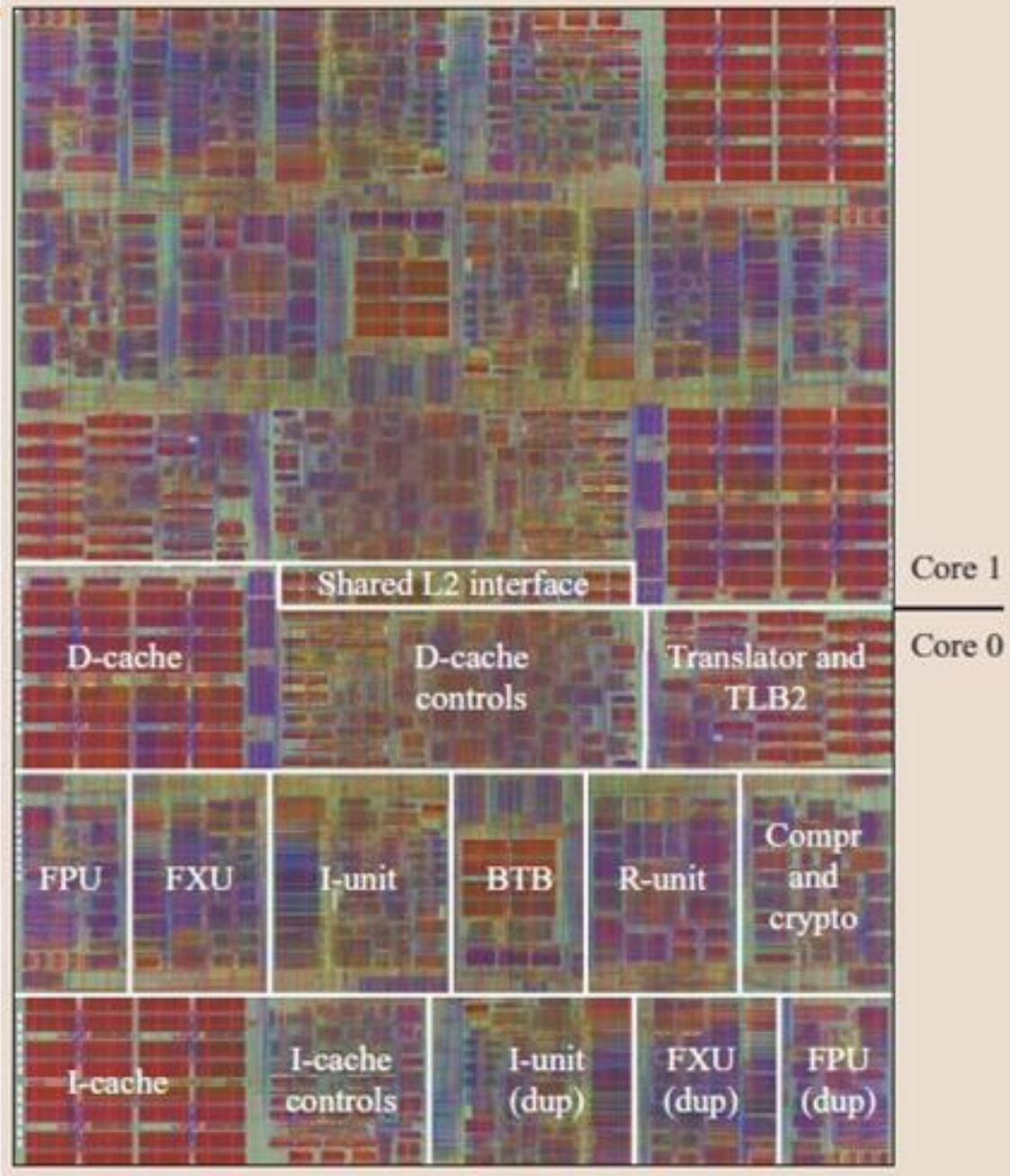
Architektura instrukčního souboru IBM 360 (ISA) úplně překryla rozdíl mezi jednotlivými modely.

Mezník: První pravá ISA navržená jako přenositelný hardware-software interface!

S mírnými modifikacemi přežívá dodnes!

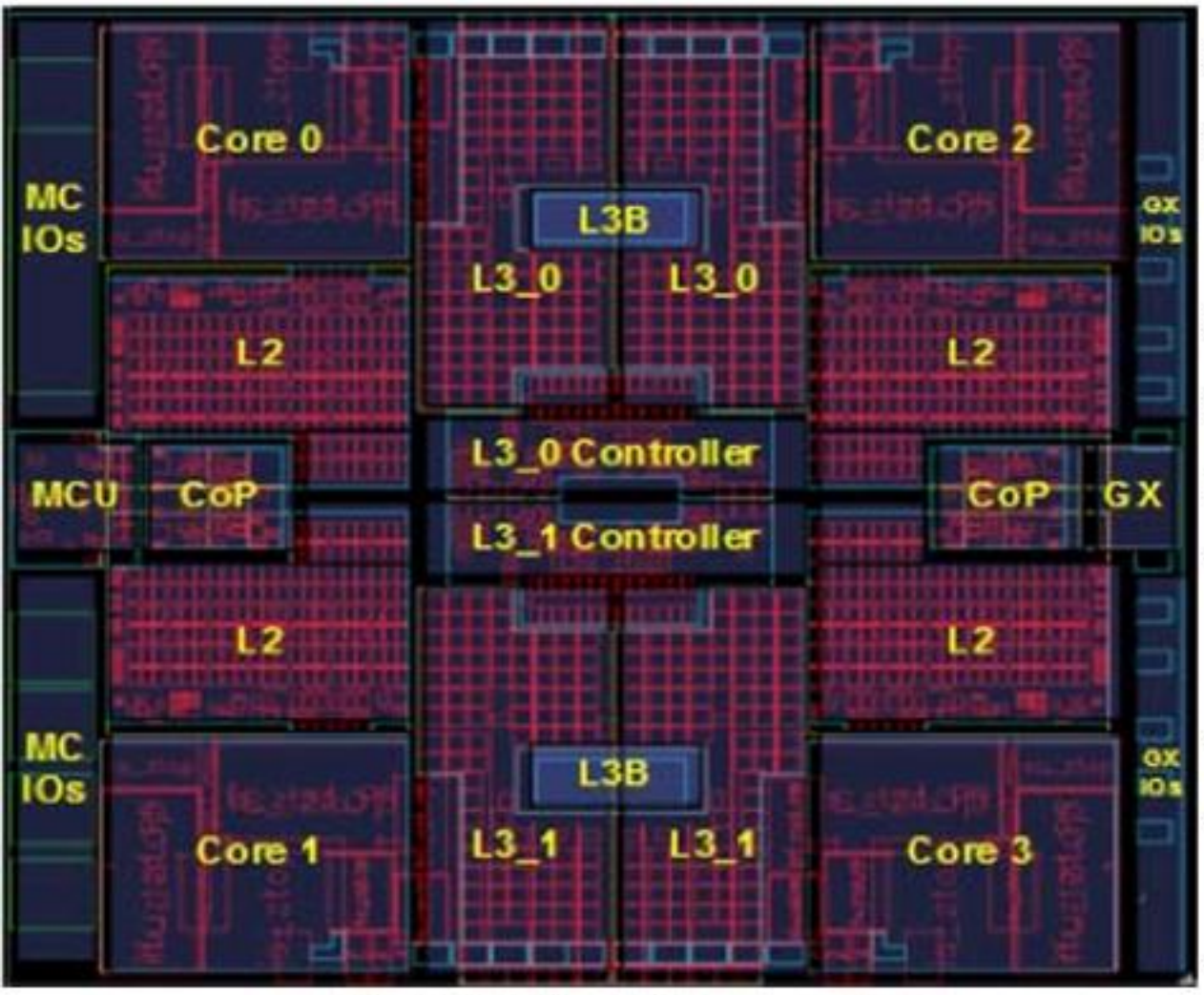
IBM 360: Po 40 letech… Mikroprocesory řady z990

- 64-bitové virtuální adresování
  - originálně u S/360 bylo 24-bitové a u S/370 byla 31-bitová extenze
- Návrh se dvěma jádry
- Dvouproudová superskalární architektura
- 10-stupňová CISC pipeline
- Out-of-order přístupy do paměti
- Redundantní datové cesty
  - každá instrukce se provádí ve dvou paralelních jednotkách a výsledky se porovnávají
- 256KB L1 I-cache, 256KB L1 D-cache na chipu
- 32MB sdílená L2 unifikovaná cache externí
- 512-entry L1 TLB + 4K-entry L2 TLB
  - rozlehlý TLB pro podporu násobných virtuálních strojů
- 8K-entry Branch Target Buffer
  - rozlehlý buffer pro podporu větvení programů
- Až 64 procesorů (48 viditelných zákazníkovi) v jednom stroji
- 1.2 GHz v IBM 130nm SOI CMOS technologii, 55W pro obě jádra



[IBM Journal R&D, 48(3/4), May/July 2004]

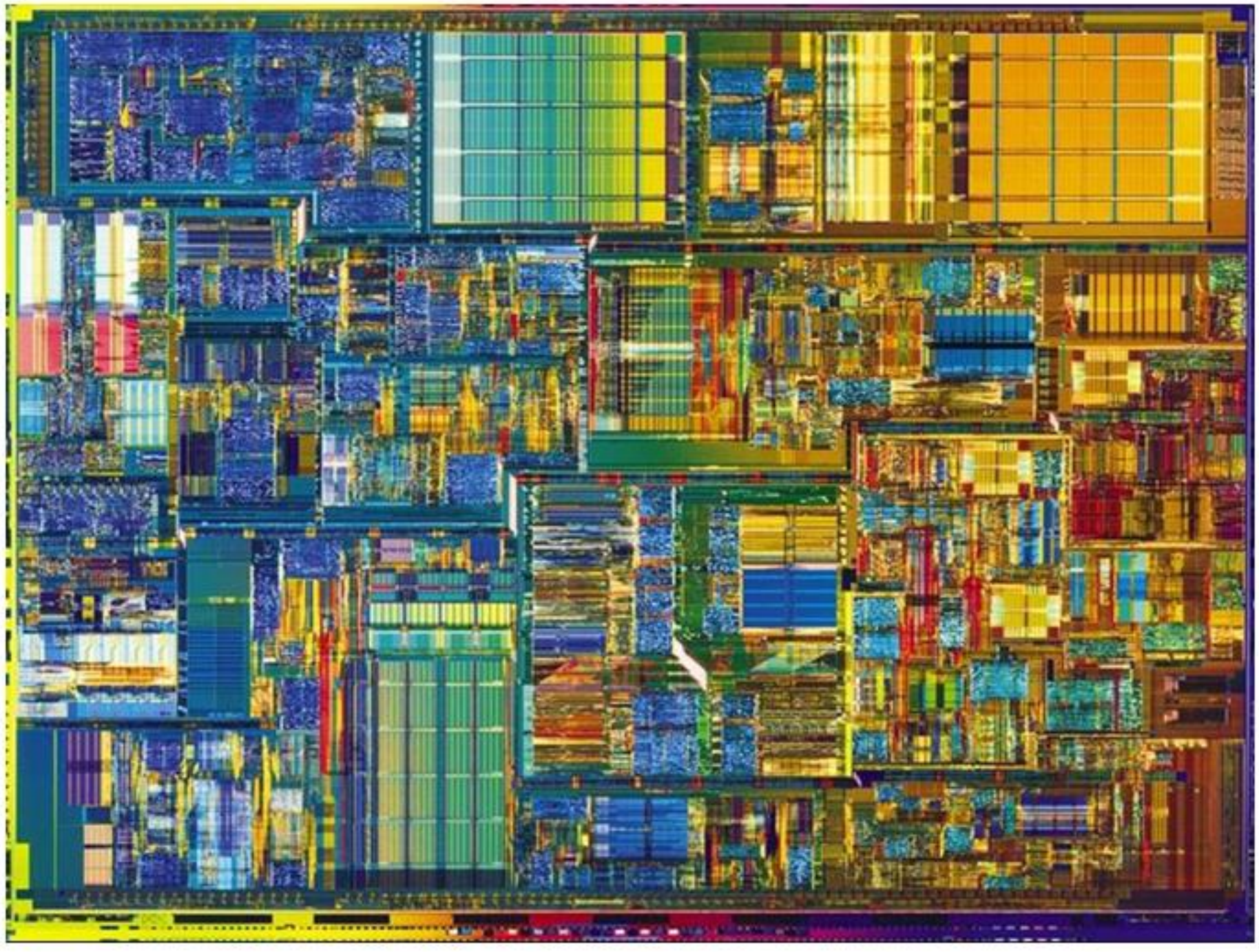
IBM 360: Po 47 letech… Mikroprocesory řady z11



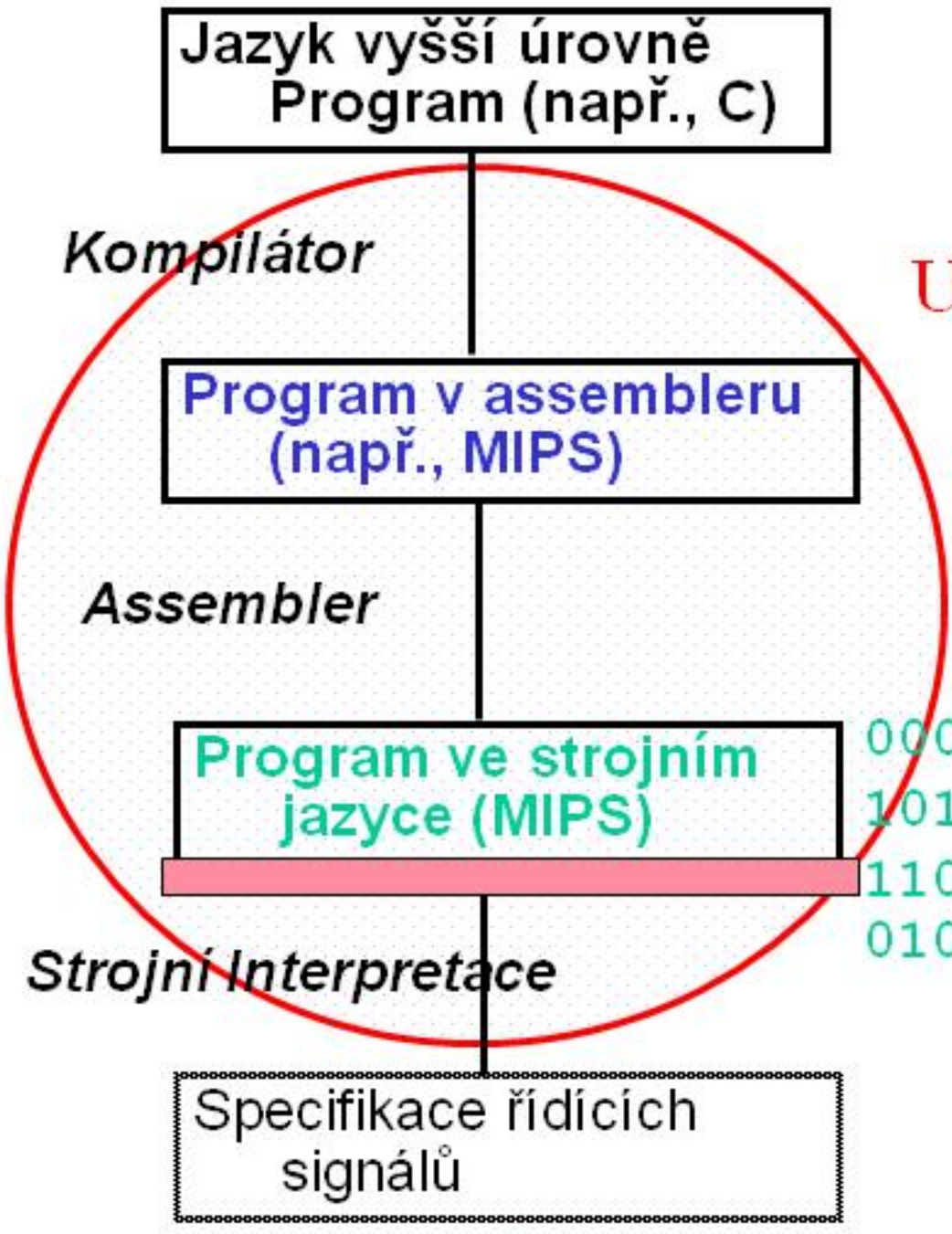
[IBM, HotChips, 2010]

- 5.2 GHz v IBM 45nm PD-SOI CMOS technologii
- 1.4 miliardy tranzistorů na 512 mm^2
- 64-bitové virtuální adresování
  - originalně S/360 měl 24-bitovou adresu, S/370 měl rozšíření na 31-bitů
- Čtyřjádrový návrh
- Three-issue out-of-order superscalar pipeline
- Out-of-order přístupy do paměti
- Redundantní datové cesty
  - every instruction performed in two parallel datapaths and results compared
- 64KB L1 I-cache, 128KB L1 D-cache na čipu
- 1.5MB privátní L2 unifikovaná cache pro každé jádro na čipu
- Na čipu 24MB eDRAM L3 cache
- Scales to 96-core multiprocessor with 768MB of shared L4 eDRAM

„Nedávná minulost“ u firmy Intel – Pentium 4



Úrovně reprezentace

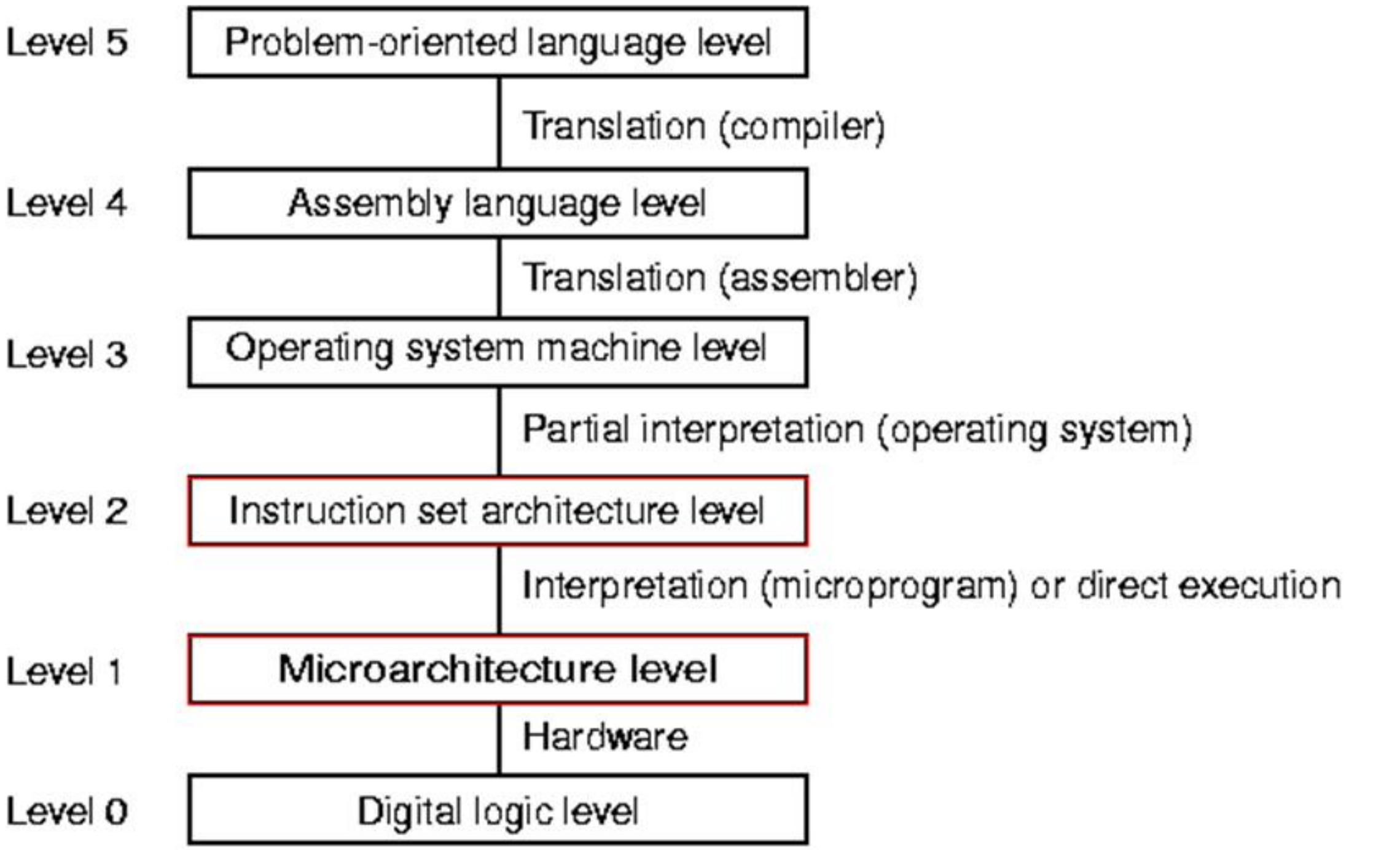



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

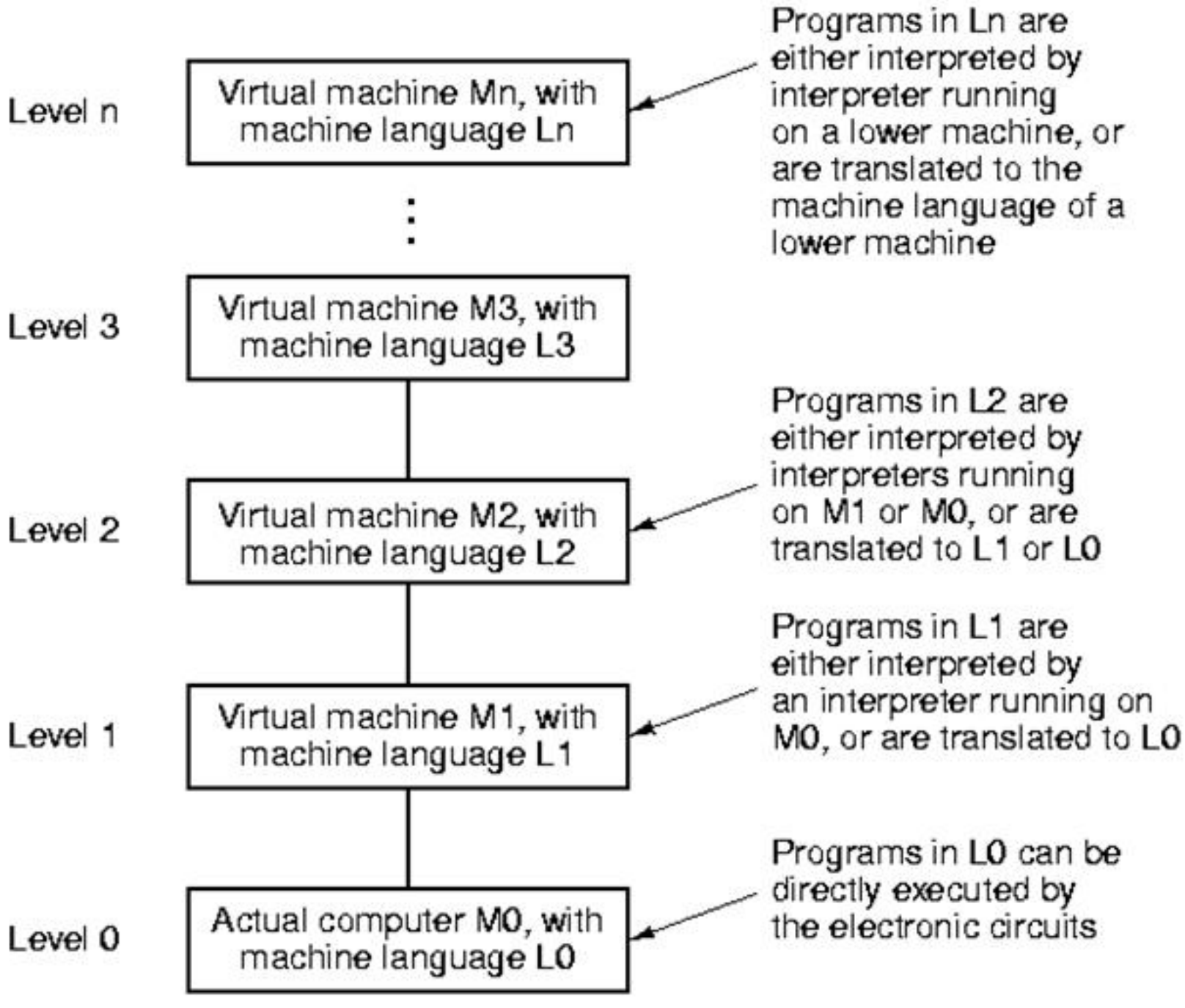
```
lw $to,  0($2)
lw $t1,  4($2)
sw $t1,  0($2)
sw $t0,  4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Vrstvy počítačového systému



Víceúrovňový stroj



Hierarchie výstavby počítačového systému

1. Samotný hardware
2. Mikroprogramování
3. **Hardware / software interface**
   - Jednoduchá ISA
   - CISC
   - RISC
   - .....
4. Operační systém
5. Kompilátory

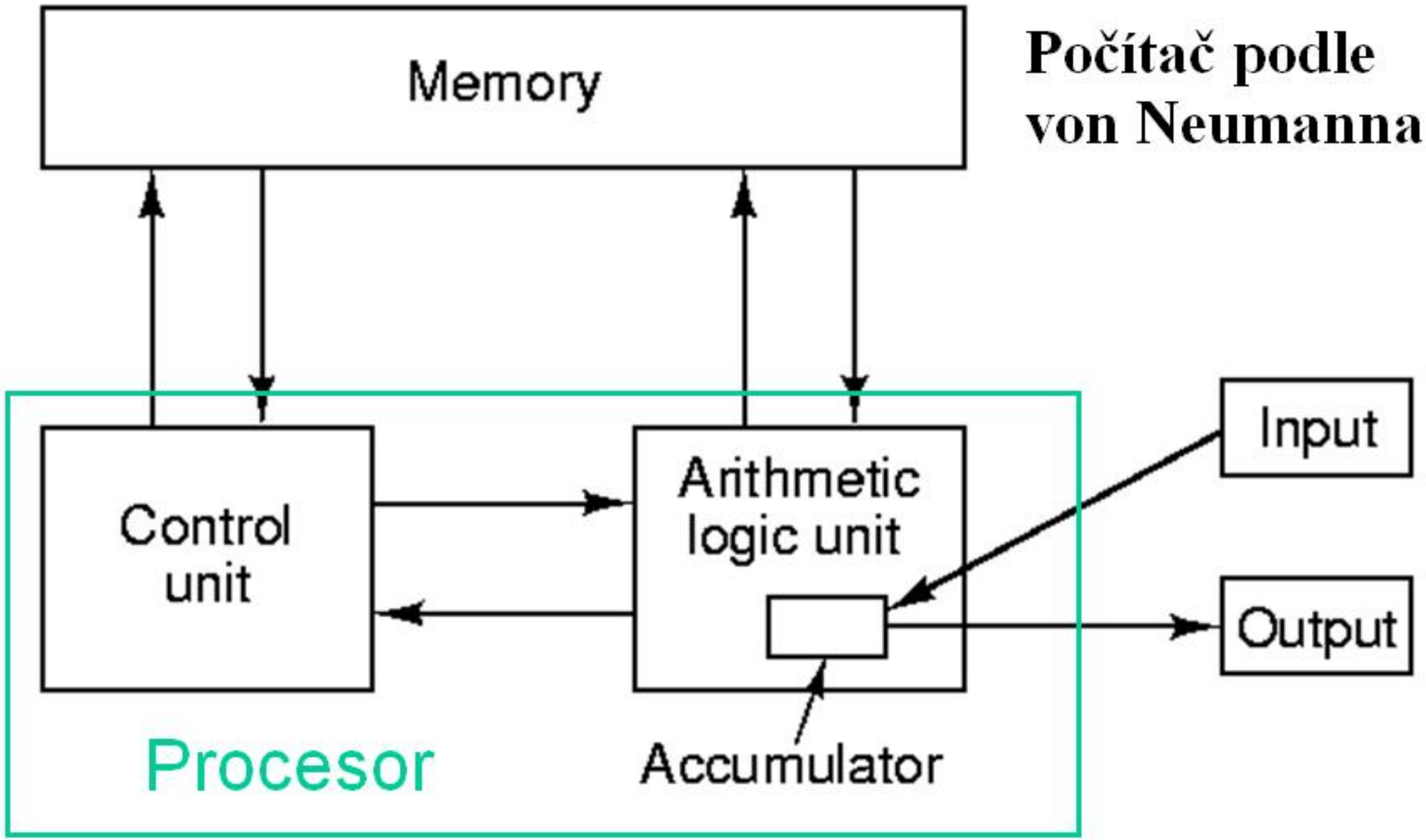
Principy návrhu

CISC versus RISC

RISC:

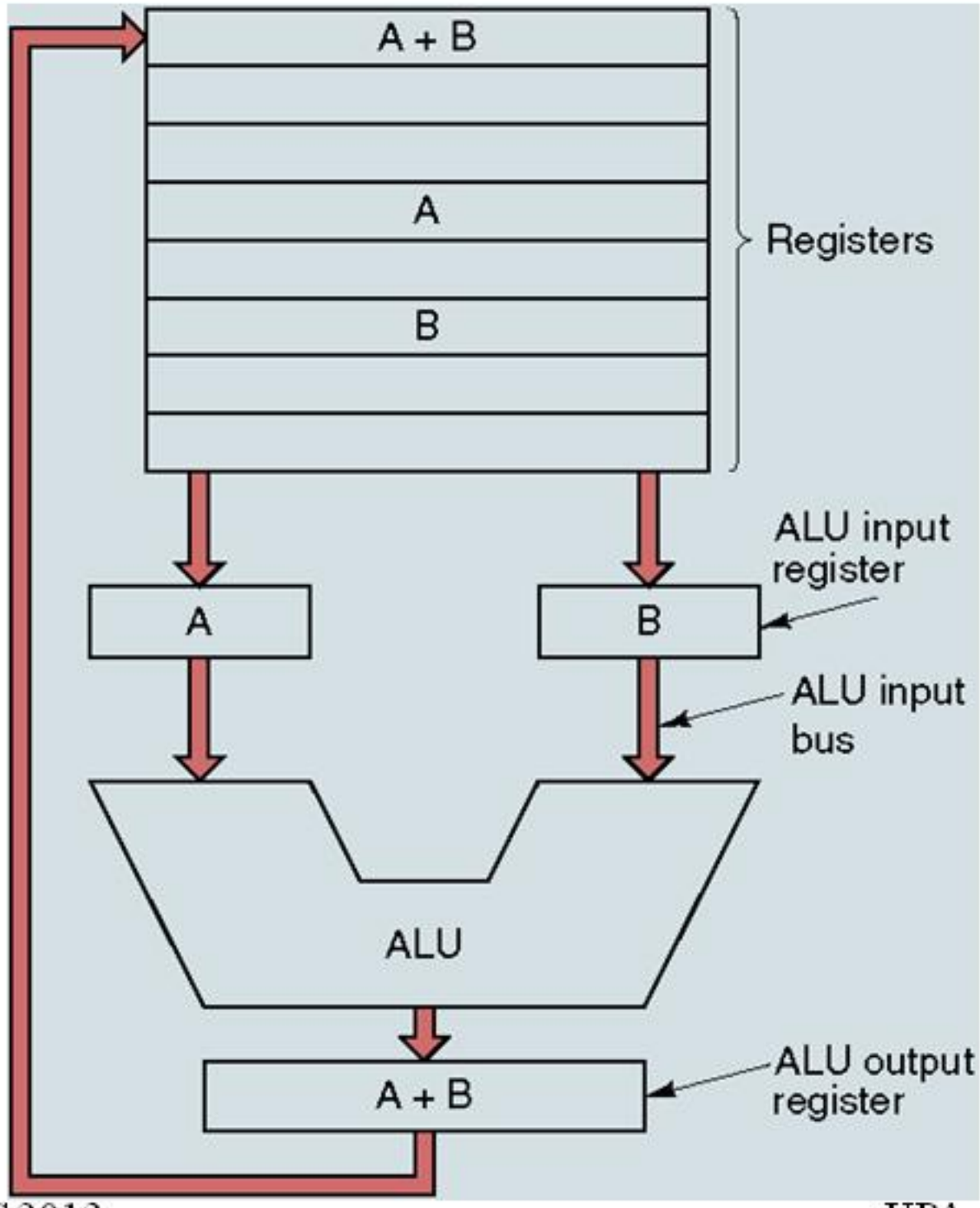
- Instrukce jsou přímo prováděny hardwarem
- Maximální průchodnost instrukcí (ILP)
- Jednoduché instrukce (snadné dekódování)
- Přístup do paměti jen instrukcemi load/store
- Velké množství registrů
- Pipelining

Organizace počítače



Počítač podle von Neumanna

Datové cesty – současná „klasika“



⇄ Memory ⇄ I/O

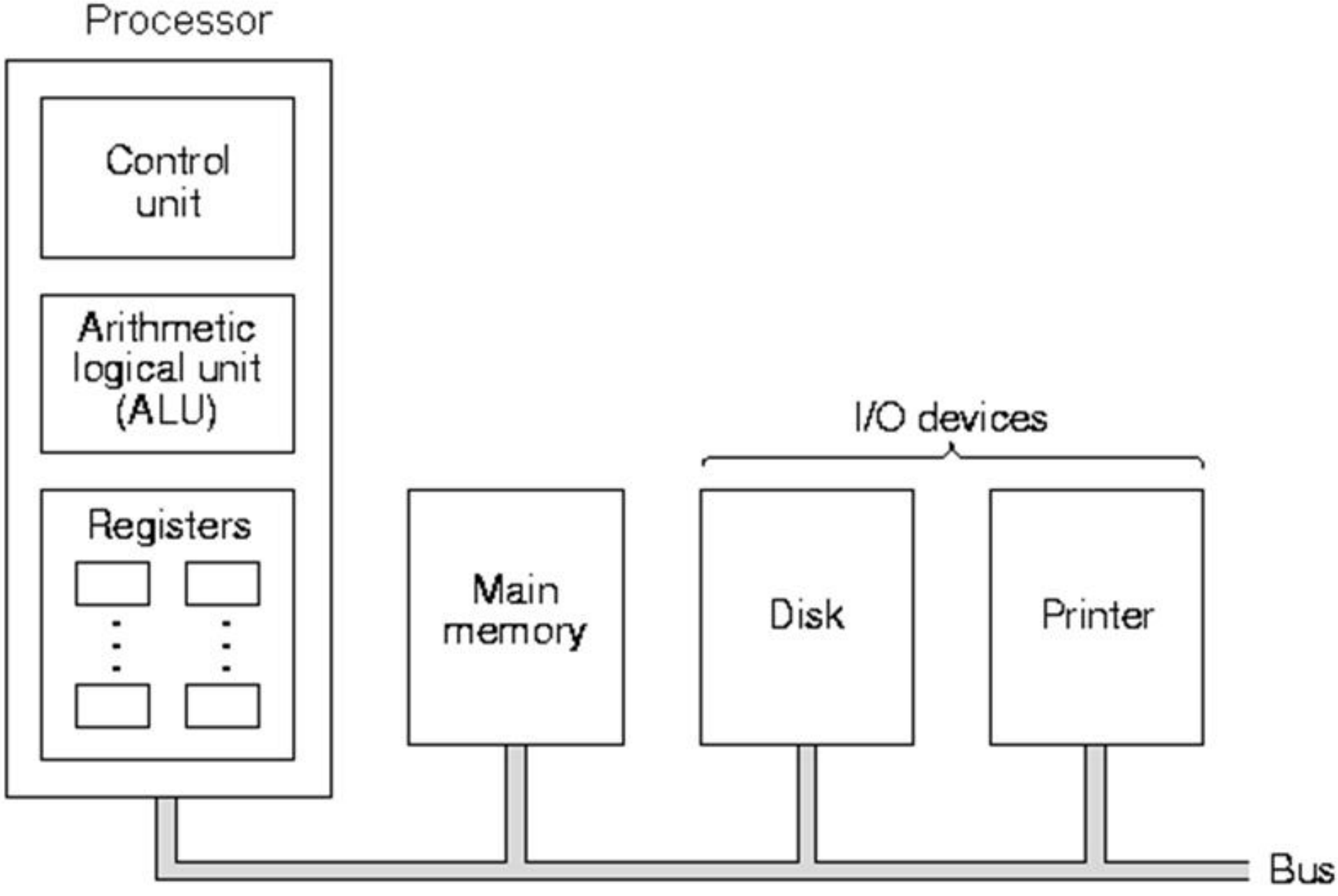
- Registry a ALU
- “Pomocné registry”
  - jen v hardware
  - neobjevují se v programu
- „Jádro procesoru“
  - určuje rychlost
  - určuje výkon

Cyklus zpracování instrukce

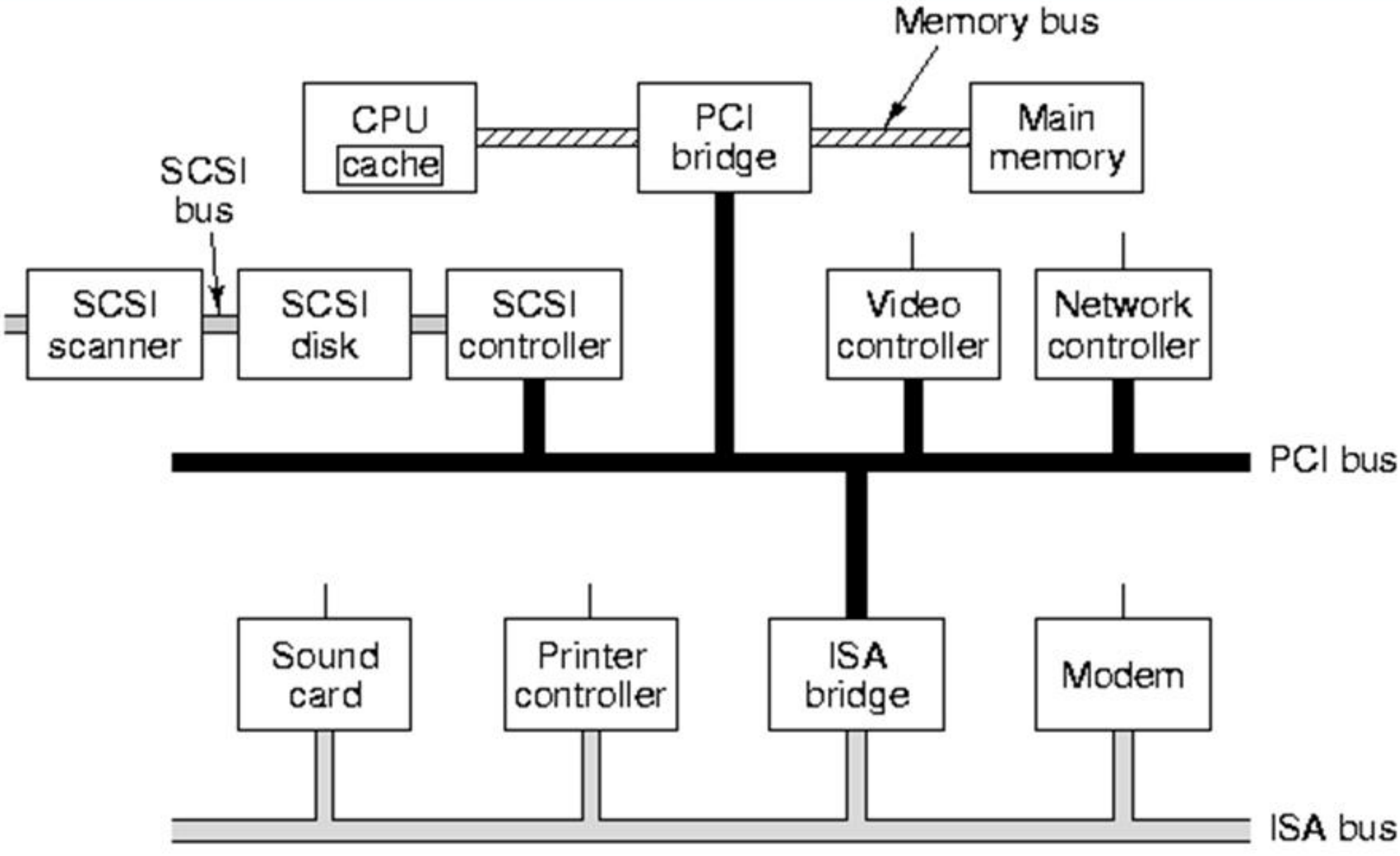
Fáze provádění instrukce:

1. Přečtení instrukce z paměti podle adresy v PC do IR
2. Aktualizace PC, aby ukazoval na další instrukci
3. Určení typu instrukce (dekódování) :
   registr/registr nebo registr/paměť …
4. Výpočet adres operandů (adresní režimy)
5. Načtení operandů do pomocných registrů
6. Provedení vlastní operace prováděcí jednotkou
7. Zápis výsledku do registru(ů), popř. do paměti
8. Pokračování v bodu 1. – zpracování další instrukce

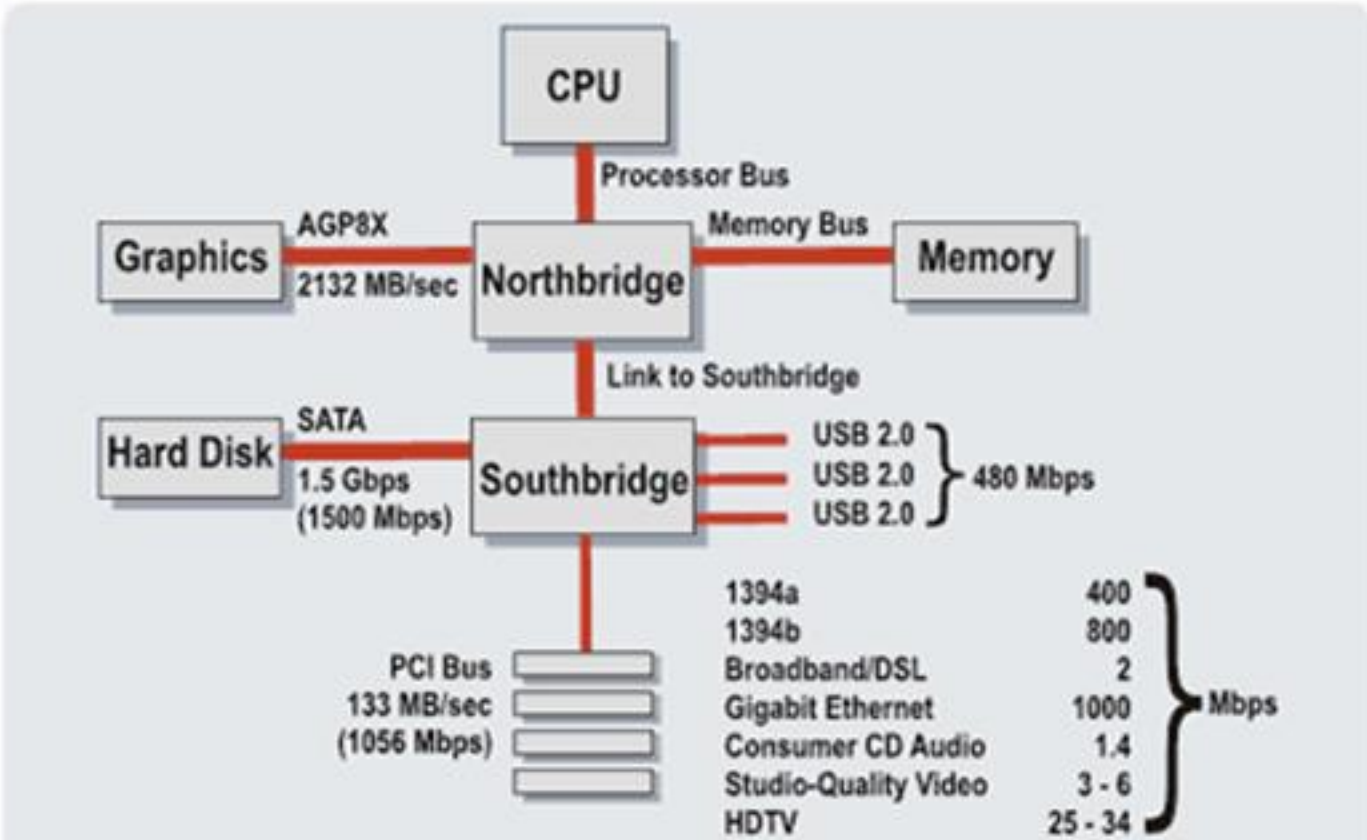
Počítač se sběrnicovou koncepcí

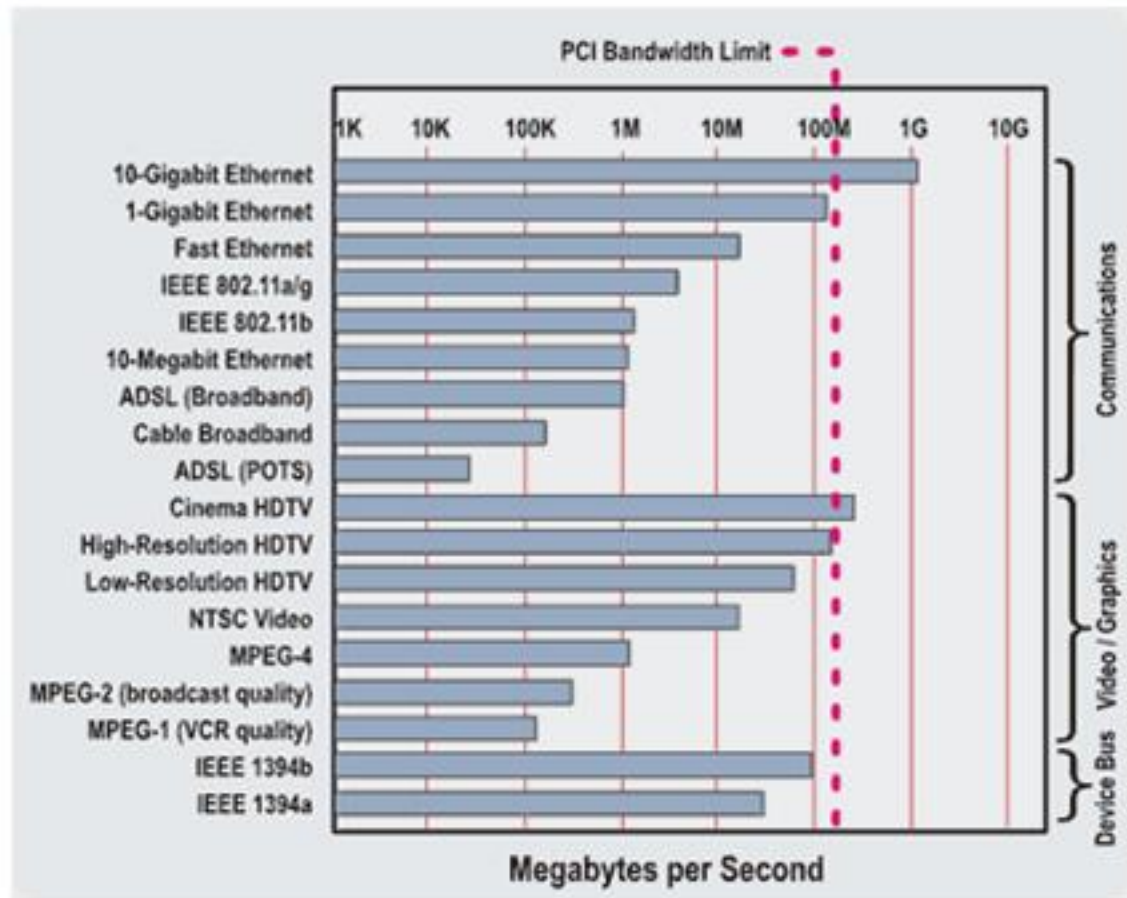


Anatomie PC (kolem r. 1996)

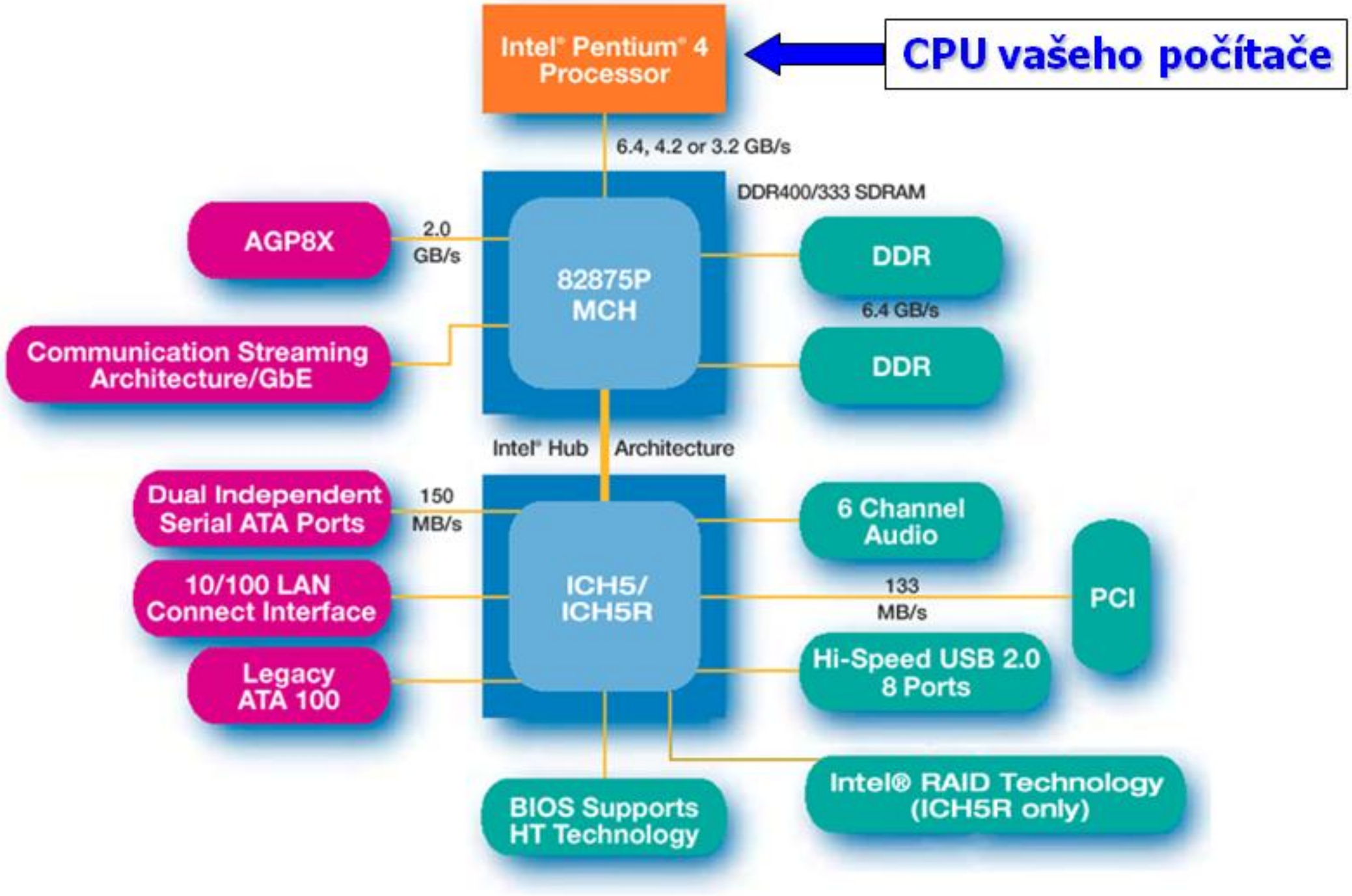


Anatomie PC (kolem r. 2006)

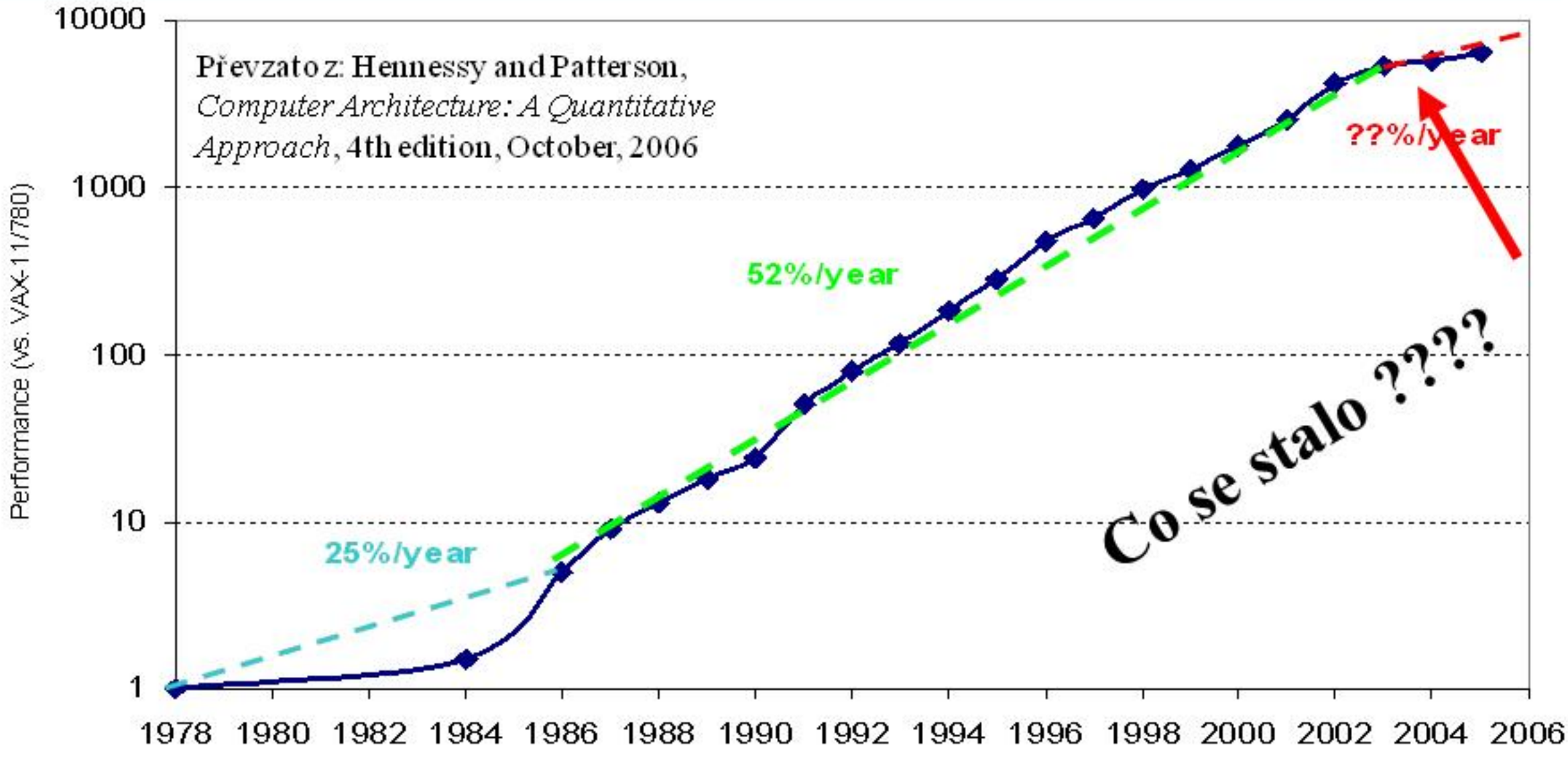




Anatomie PC (kolem r. 2006)

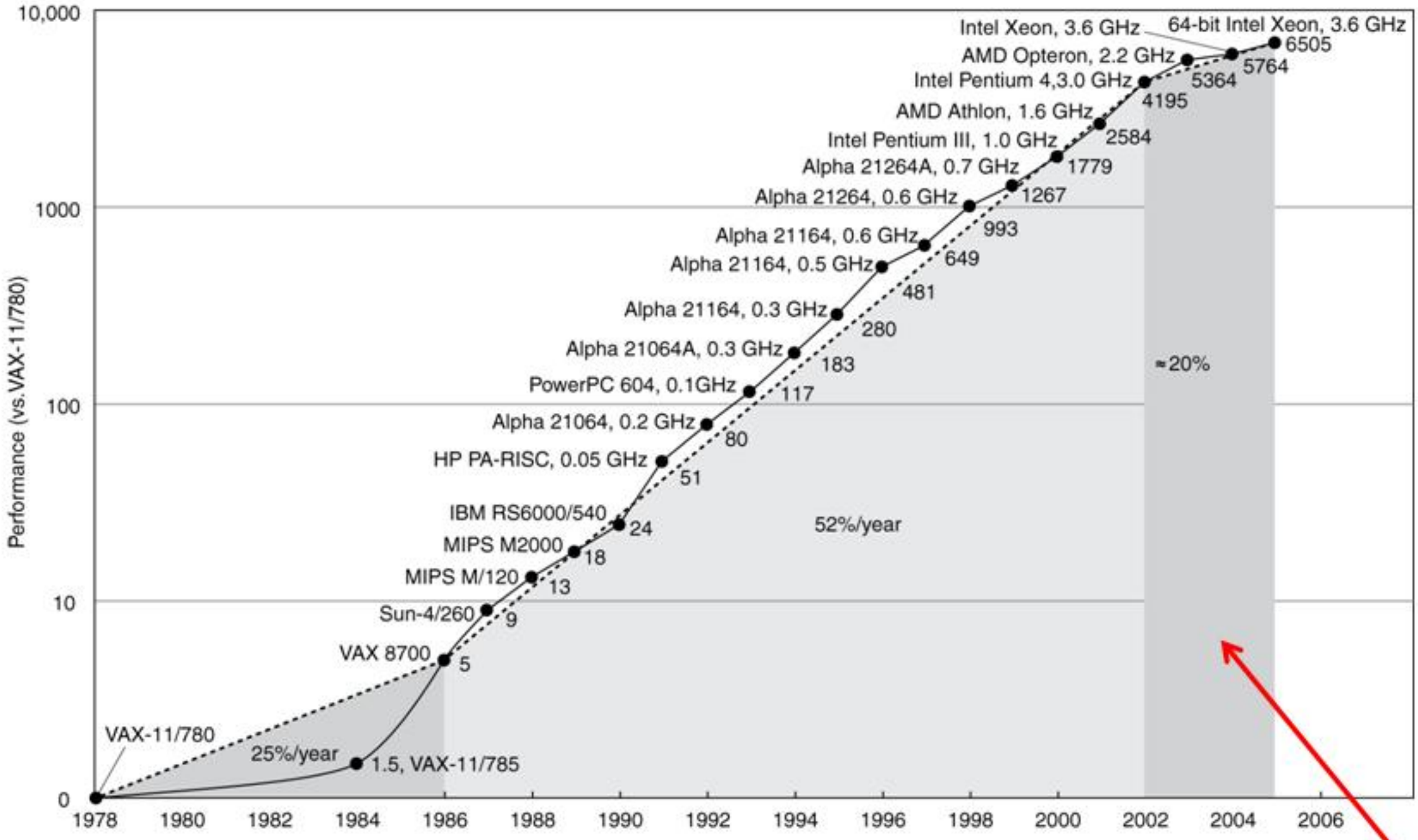


Výkon jednoprocesorového počítače



- VAX : 25%/rok od r. 1978 do r. 1986
- RISC + x86: 52%/rok od r. 1986 do r. 2002
- RISC + x86: ??%/rok od r. 2002 do současnosti

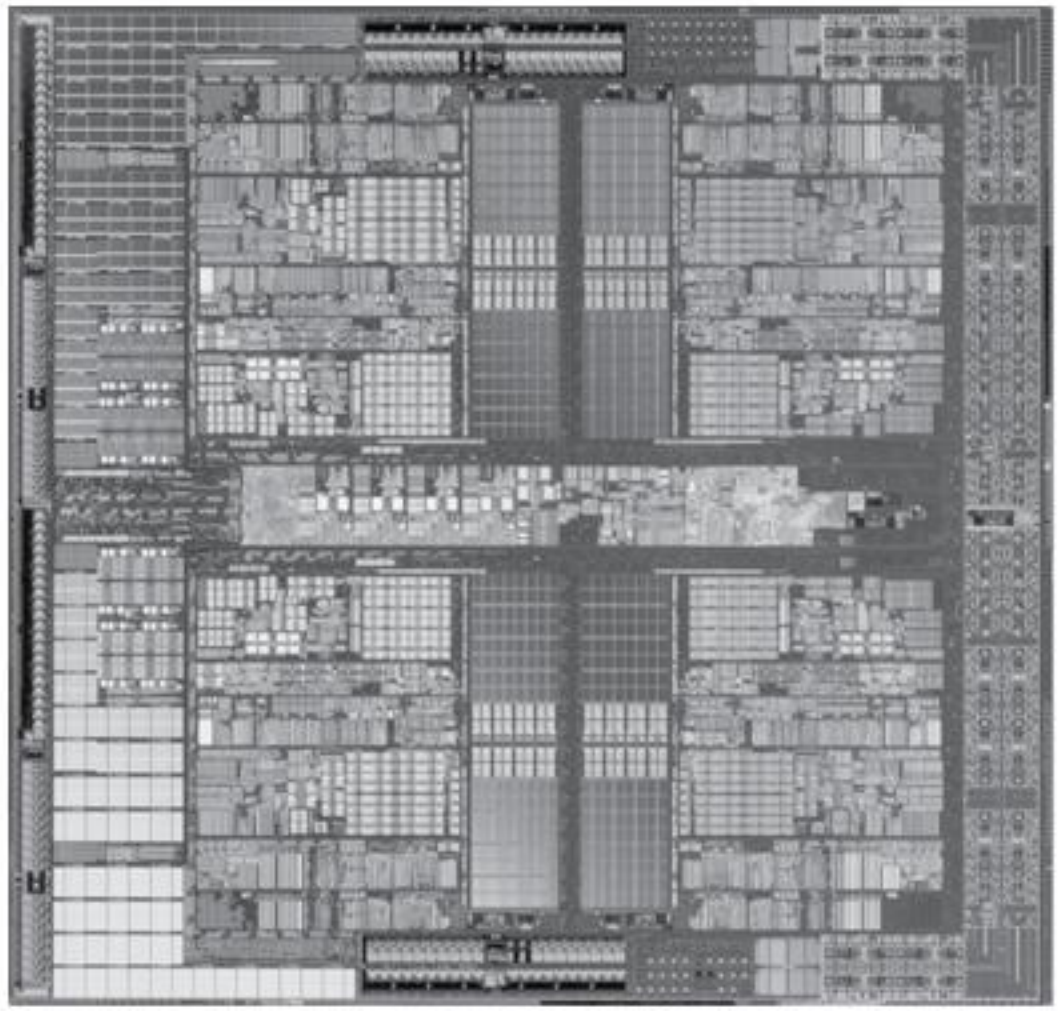
Výkon jednoprocesorového počítače

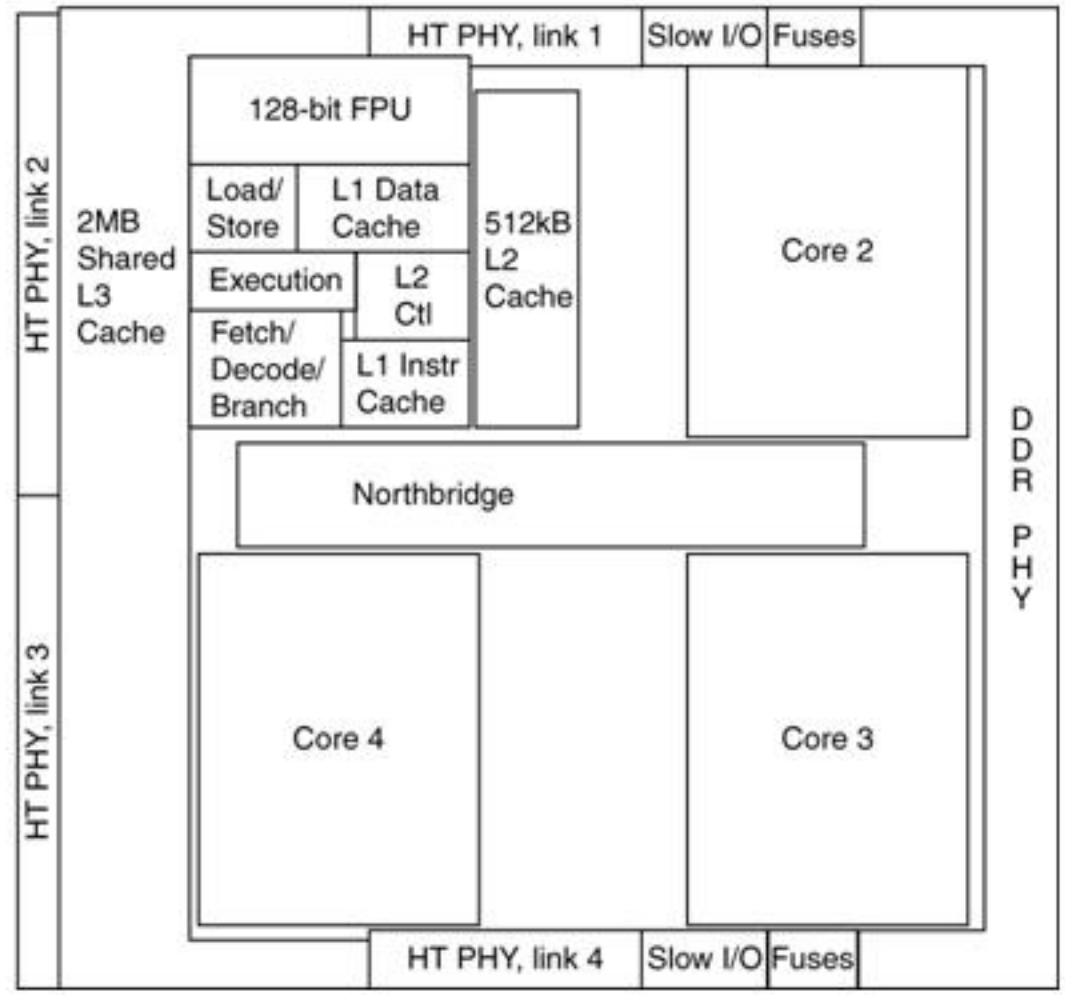


Omezení způsobené ztrátovým výkonem, úrovní paralelizmu a latencí paměti

Vícejádrové procesory

- AMD Barcelona: 4 procesorová jádra (2007)





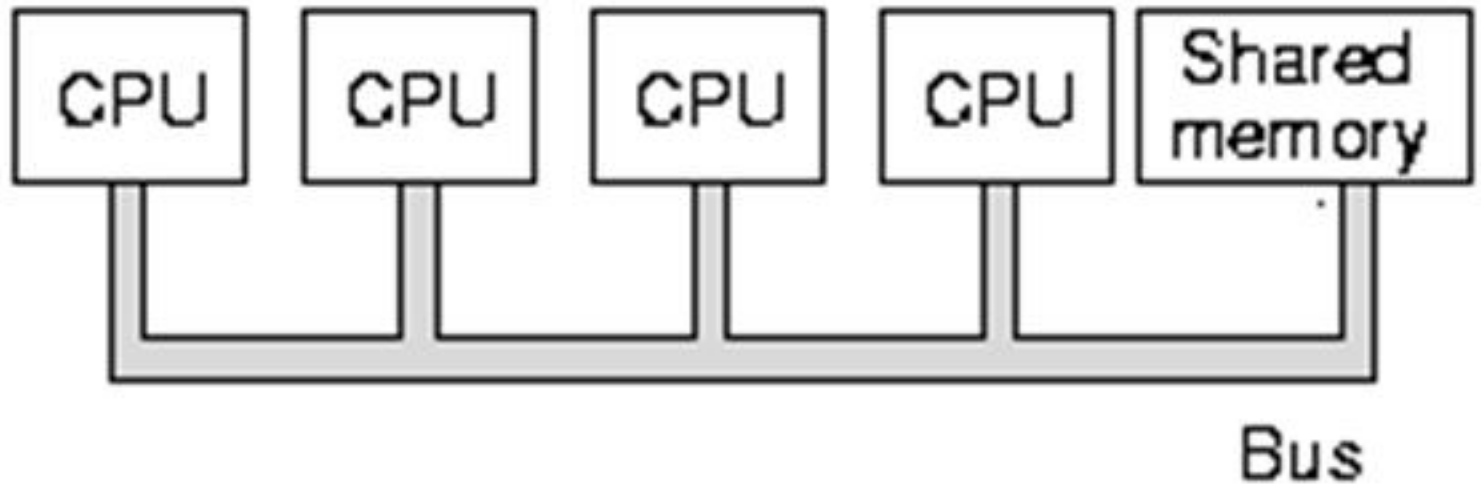
Zvyklosti v počítačové architektuře

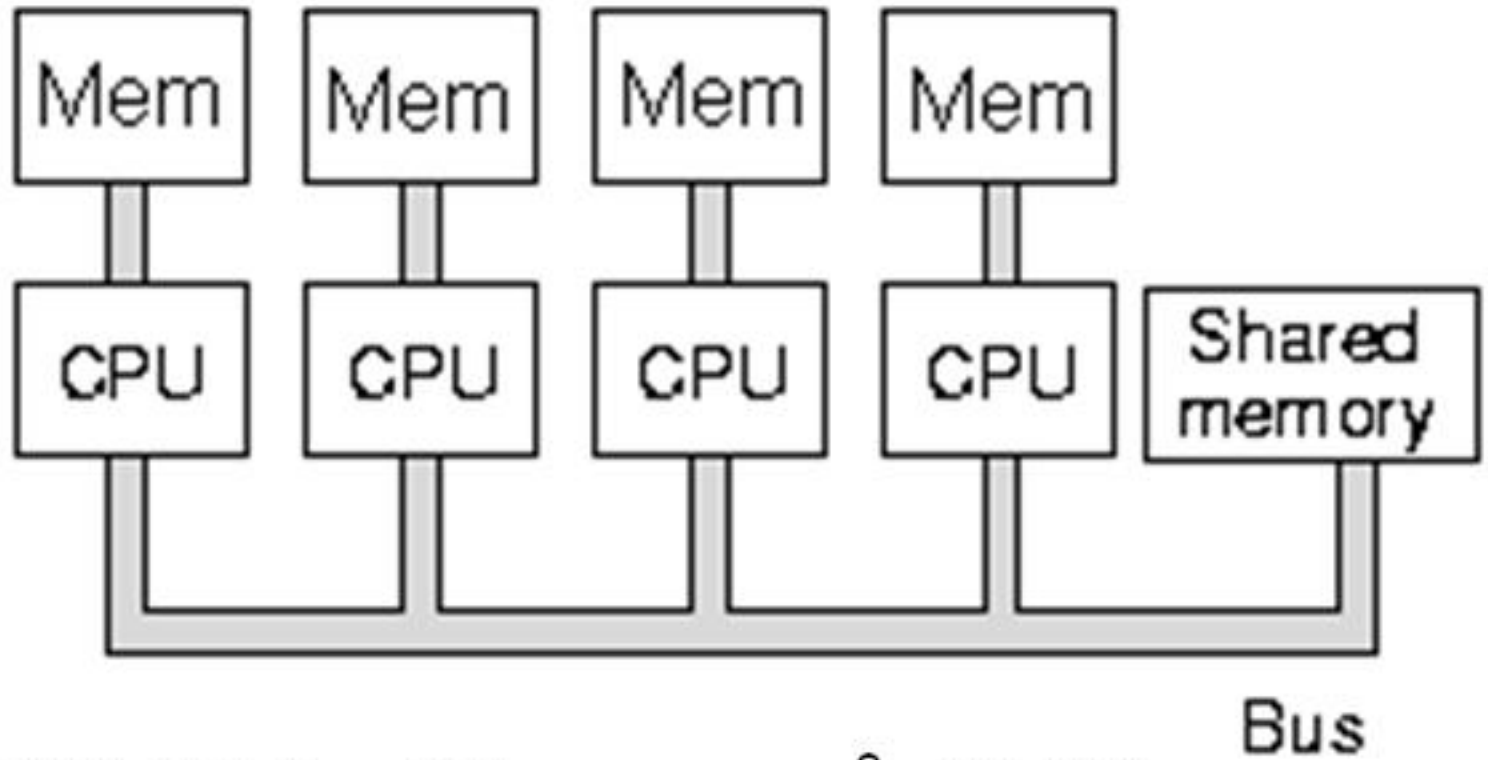
- Dříve: Napájení je neomezené, drahý tranzistor
- Nyní: "Power wall" výkon je „drahý", tranzistory nestojí téměř nic
  (Na čip lze umístit víc, než je možno „uchladit")
- Dříve : Dostatečný nárůst paralelismu instrukční úrovně vlivem kompilátorů a inovací (pipelining, superskalární systémy, out-of-order, spekulace, VLIW, ...)
- Nyní : "ILP wall" zákon klesající výtěžnosti investice HW pro ILP
- Dříve : Násobení je pomalé, přístup do paměti je rychlý
- Nyní : "Memory wall" pomalá paměť, rychlé násobení
  (200 cyklů hodin pro přístup do DRAM, 4 hodinové cykly pro násobení)
- Dříve : Výkon jednoprocesorového systému 2x za 1.5 roku
- Nyní : Power Wall + ILP Wall + Memory Wall = Brick Wall
  - Výkon jednoprocesorového systému nyní 2x za 5(?) let

  ⇒ Viz změny v návrhu čipu: násobná "jádra"
  (2x procesorů na čip za ~ 2 roky)
- Více jednodušších procesorů dává efektivněji větší výkon

ILP ... **I**nstruction **L**evel **P**arallelism

Multiprocesory





IBM ASCI White: 8K procesorů, 13 Tflops

Třídy počítačů

- Osobní počítače
  - Navrženy tak, aby poskytovaly dobrý výkon pro jednoho uživatele za nízkou cenu užívajícího obvykle tzv. 3. party software. Zahrnují grafický displej, klávesnici, myš atd.
- Servery
  - Využívány pro zpracování programů více uživatelů současně, typicky přístupné pouze přes síť. Větší důraz je kladen na spolehlivost a (často), bezpečnost
- Superpočítače
  - Třída serverů s vysokou cenou, ale také vysokým výkonem, se stovkami až tisíci procesorů, terabajty paměti a petabajty úložného prostoru, které se používají pro „high-end“ vědecké a inženýrské aplikace
- „Embedded“ počítače (procesory)
  - Počítač umístěný v „jiném“ zařízení, používaný obvykle pro spouštění jedné předem určené aplikace

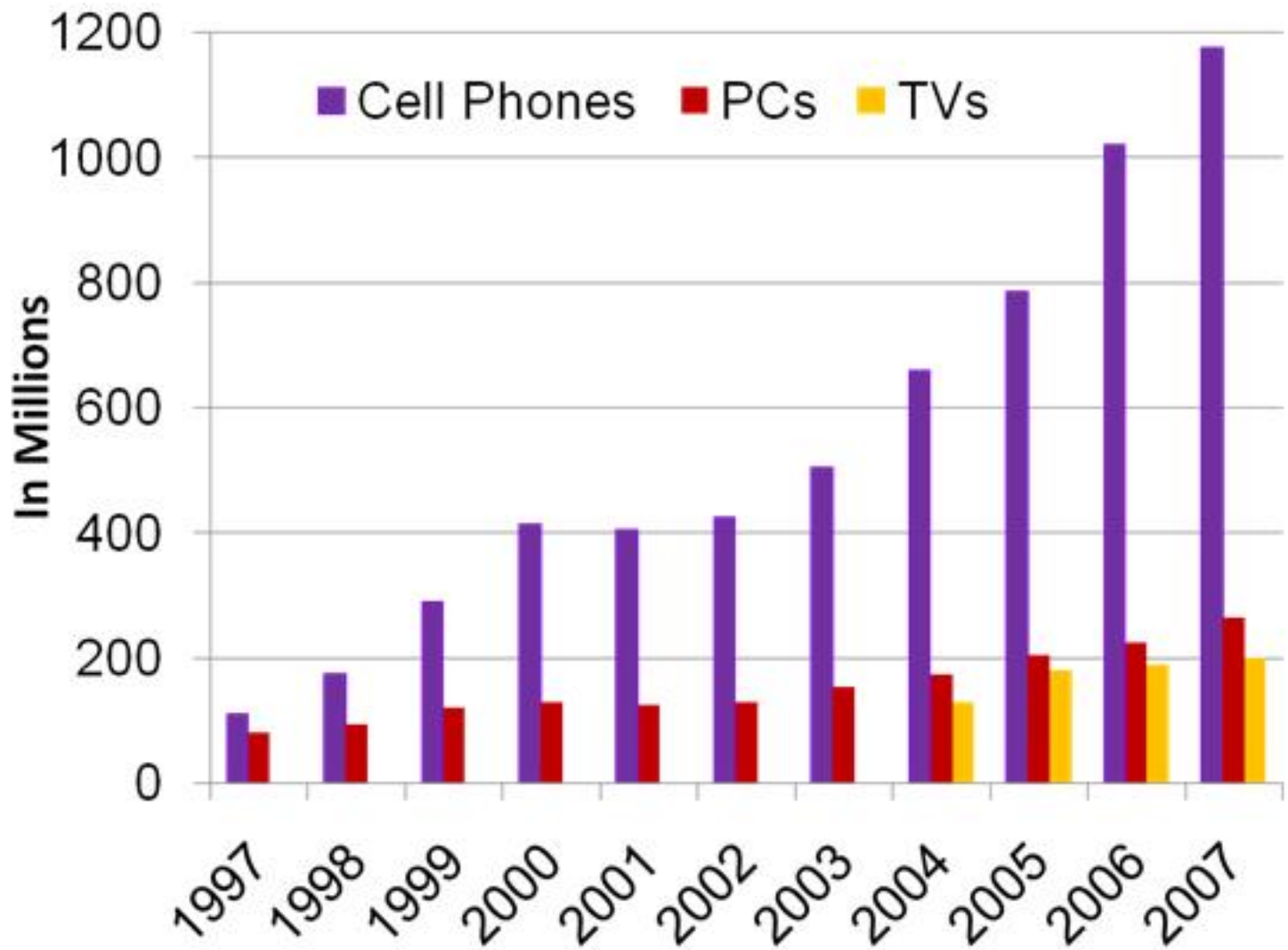
Opakování: Některé základní definice

- 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. !

Nárůst prodeje mobilů („embedded“)

nárůst aplikací třídy „embedded“ >> nárůst PC



- Kde lze nalézt tzv. „embedded“ procesory?

Charakteristika „embedded“ processorů

Největší třída počítačů zahrnující nejširší škálu aplikací a výkonů

- Často mají minimální požadavky na výkon. Příklad?
- Často mají přísné omezení na cenu. Příklad?
- Často mají přísné omezení na spotřebu energie. Příklad?
- Často mají nízkou toleranci k selhání. Příklad?

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*

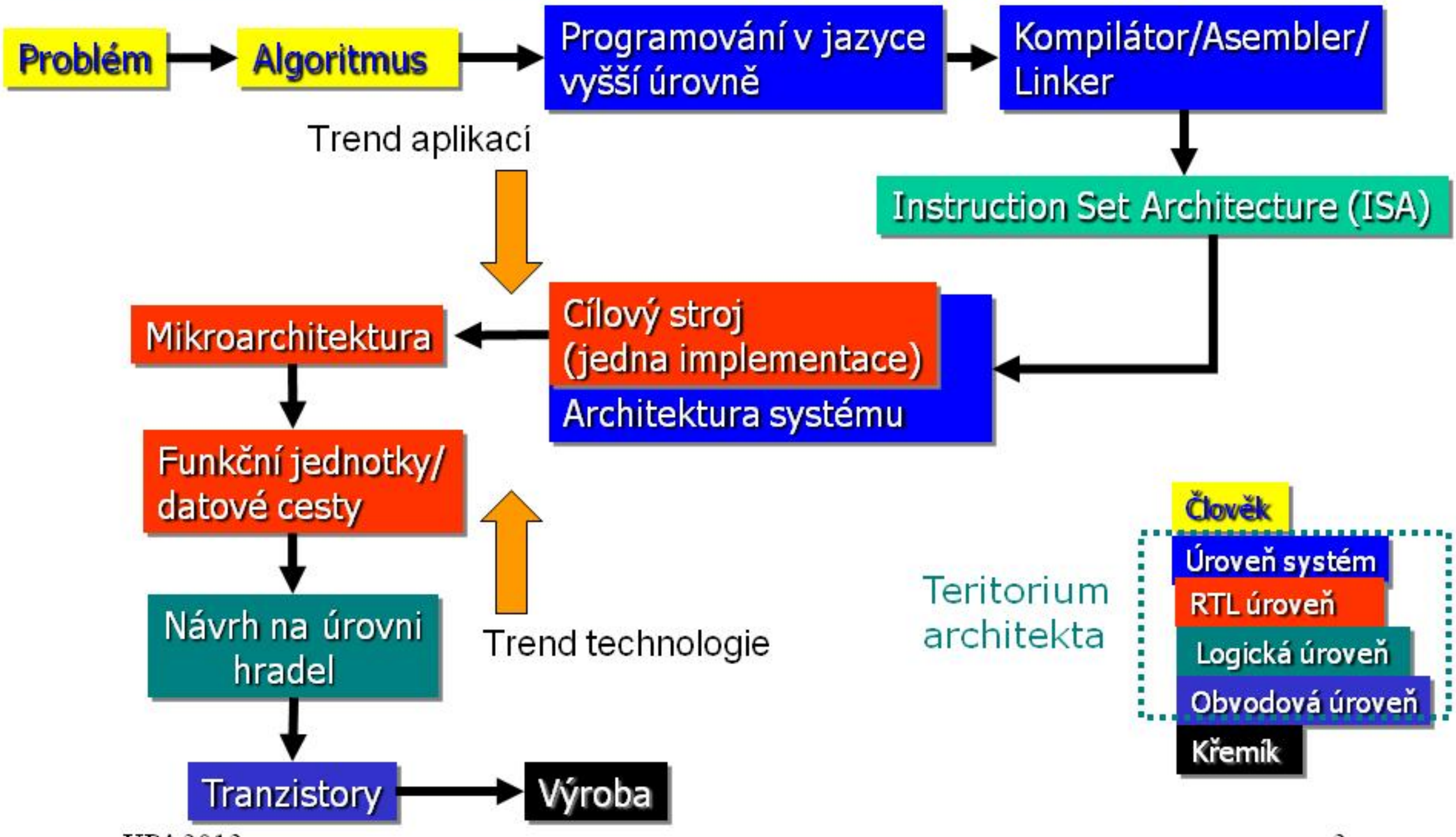
Úvod do organizace počítače

Trendy technologie, základy digitální logiky

Opakování (předchozí přednáška)

- Pět základních částí počítače
- Principy *abstrakce* systému, budovaného po vrstvách
- „*Flexibilní*“ *data*: program určuje interpretaci dat
- Koncepce „*program v paměti*“*:* Instrukce jsou také data
- *Princip lokality*: hierarchie paměťového systému
- Vyšší výkon využitím *paralelizmu*
- Kompilace vs. interpretace

Rozklad problému



Přehled (dnešní přednáška)

- Generace počítačů
- Technologie ⇔ aplikace – vzájemná podpora vývoje
- Trendy technologie
  - Hardware
  - Software
- Mooreův zákon
- Základy digitální logiky
  - Základní elektronické prvky a jejich chování
  - Pravdivostní tabulky
  - Operace

Generace počítačů

- **Gen-0**: Mechanické počítače (BC do počátku 1940)
- **Gen-1**: Elektronky (1943-1959)
- **Gen-2**: Tranzistory (1960-1968)
  - John Bardeen, Walter Brattain a William Shockley
- **Gen-3**: Integrované obvody (1969-1977)
  - průkopník Jack Kilby (1958)
- **Gen-4**: VLSI (1978-současnost)
- *Gen-5*: Optické systémy?
  Kvantové systémy?

Mezníky ve vývoji počítačů

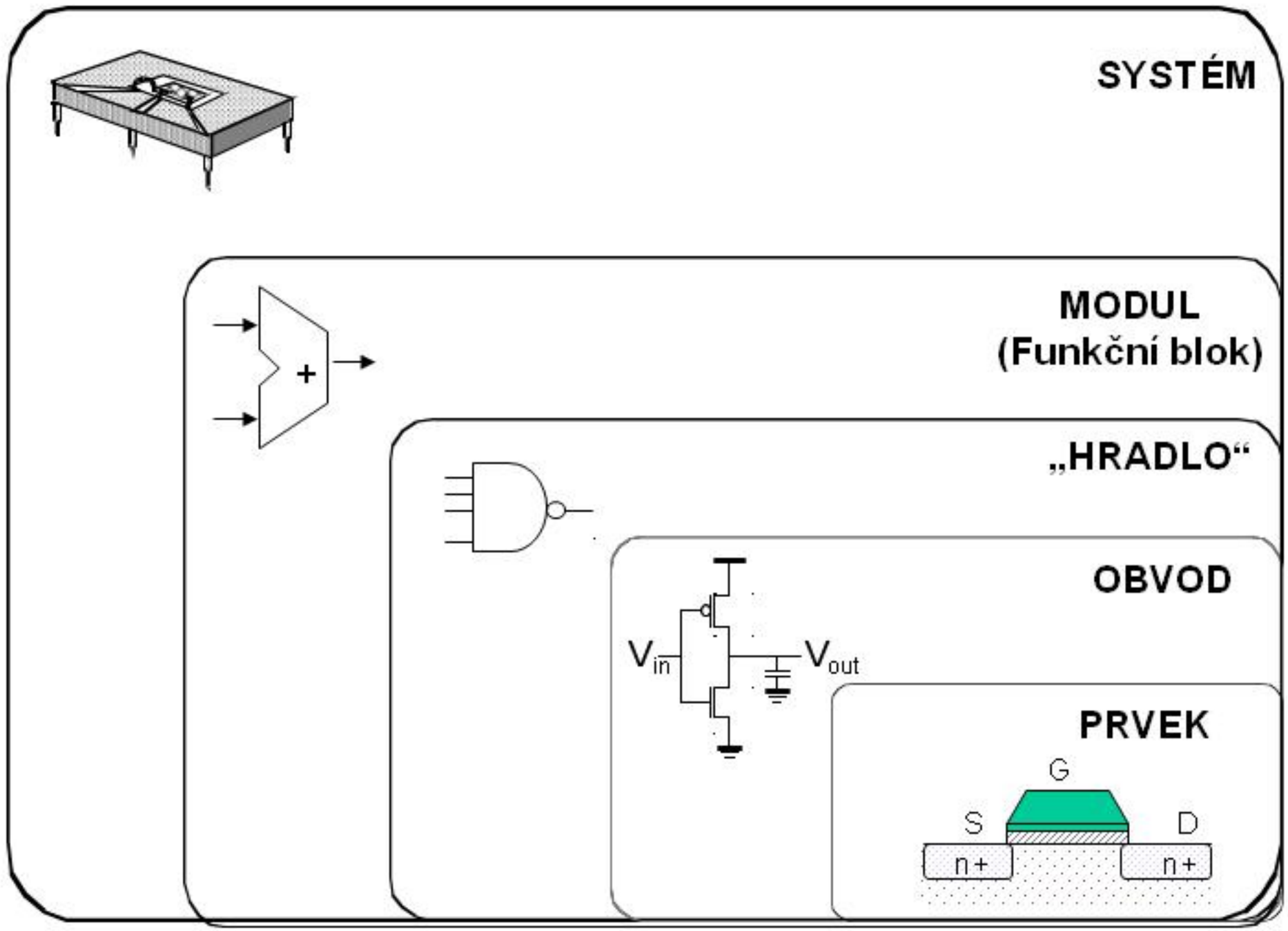
1800s	Analytical Engine	Babbage	První číslicový počítač
1936	Z1	Zuse	První releový stroj
1943	COLOSSUS	British gov't	První elektronický počítač
1944	Mark I	Aiken	První univerzální počítač
1946	ENIAC I	Eckert/Mauchley	Začátek moderní počítačové historie
1949	EDSAC	Wilkes	První počítač s programem v paměti
1952	IAS	Von Neumann	Vznik „tradiční" organizace
1960	PDP-1	DEC	První minipočítač
1964	360	IBM	Počítačové rodiny, architektura
1964	6600	CDC	První superpočítač pro VT výpočty
1974	8080	Intel	První procesor na čipu
1974	CRAY-1	Cray	První vektorový superpočítač
1981	IBM PC	IBM	Éra PC
1985	MIPS	MIPS	První komerční RISC procesor
1990	RS6000	IBM	První superskalární mikroprocesor
2000	ASCI White	IBM	Jeden z nejvýkonnějších počítačů

Trendy technologie

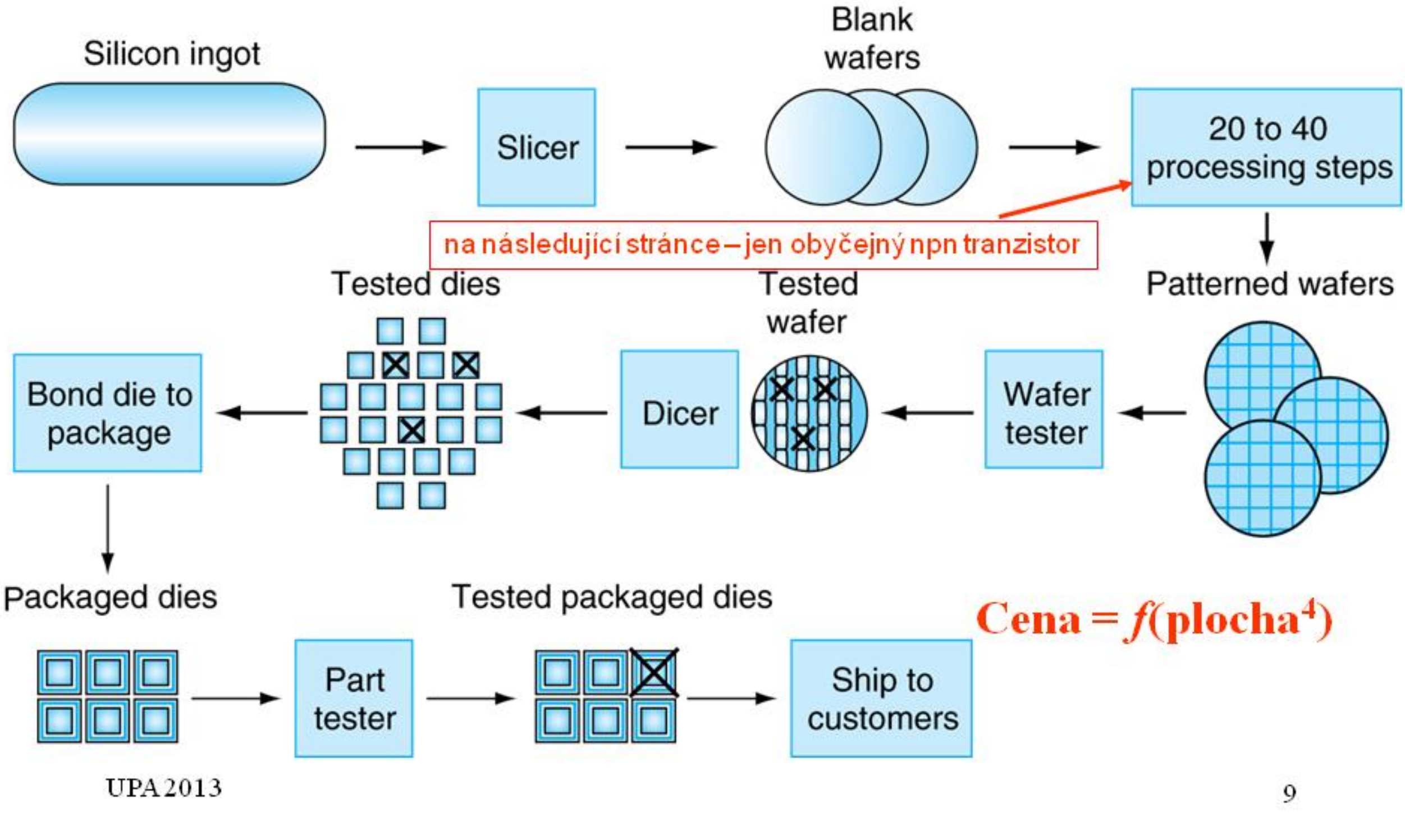
- Technologie ⇔ aplikace - podpora (**virtuální kruh**)
  - Rychlé CPU, nedostatek požadavků na aplikace
  - Požadavky současných aplikací
    - E-komerční servery
    - Databázové servery
    - Pracovní stanice
    - Narůstající poptávka po technologii „mobile computing“
- Technologie
  - Kompilátory
  - Křemík

  } ISA a organizace počítače

Opakování: Návrh úrovní abstrakce

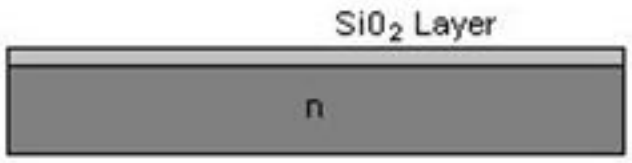


Výroba IC

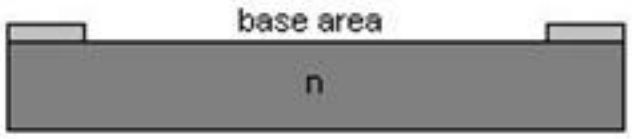


Cena = f(plocha^4)

Postup výroby planárního tranzistoru npn



(a) a. Vytvoření vrstvy SiO_2



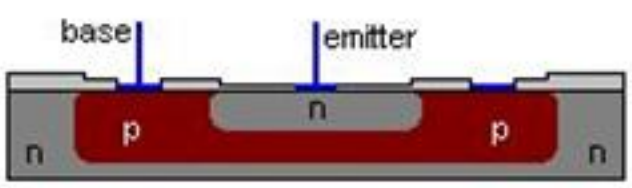
(b) b. Odleptání vrstvy SiO_2 pro bázi

(c) c. Difuze a další oxidace

(d) d. Odleptání vrstvy SiO_2 pro emitor

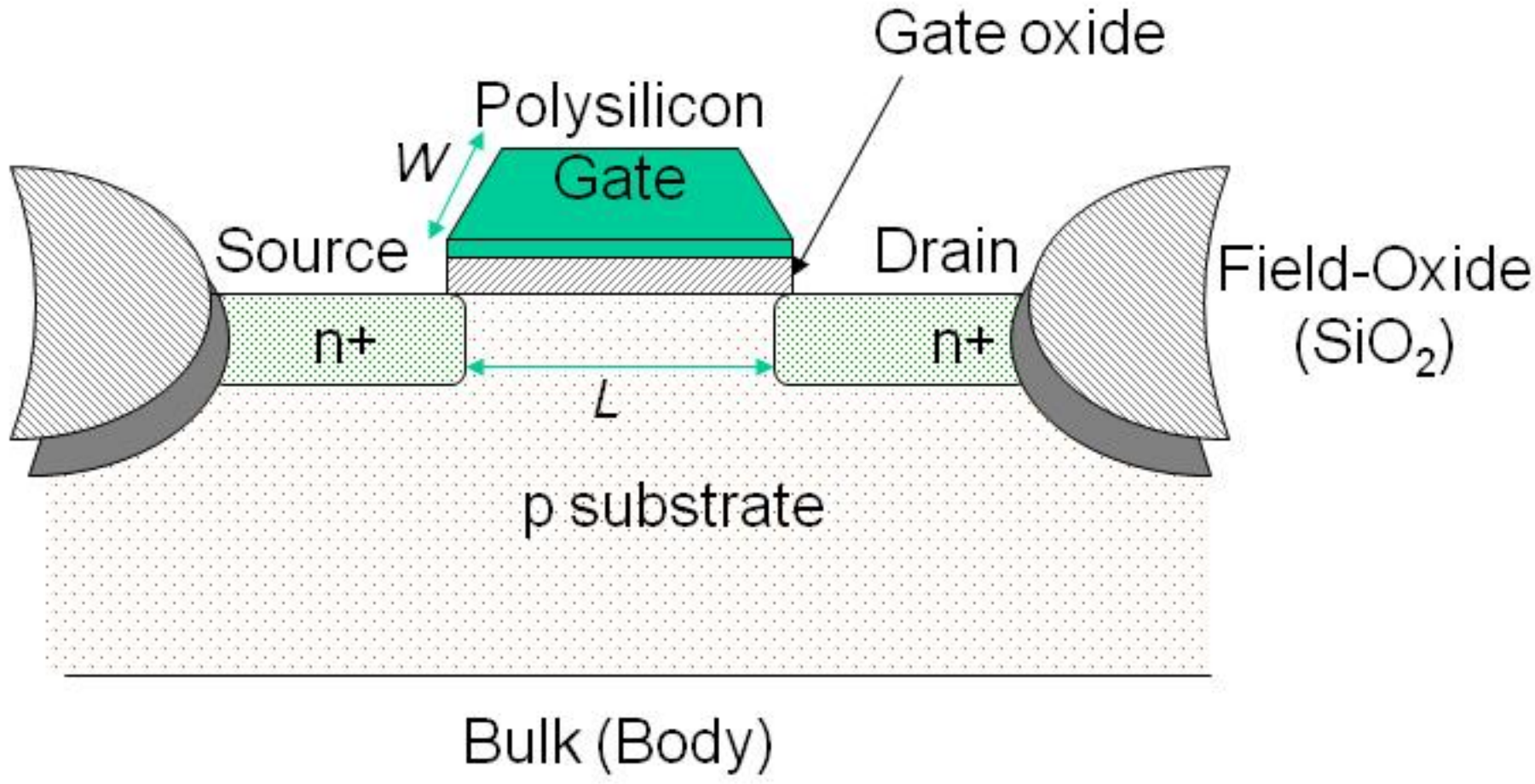
(e) e. Difuze a další oxidace

(f) f. Odleptání vrstvy SiO2 pro bázi a emitor



(g) g. Nakontaktování báze a emitoru

MOS tranzistor



Základní metriky návrhu

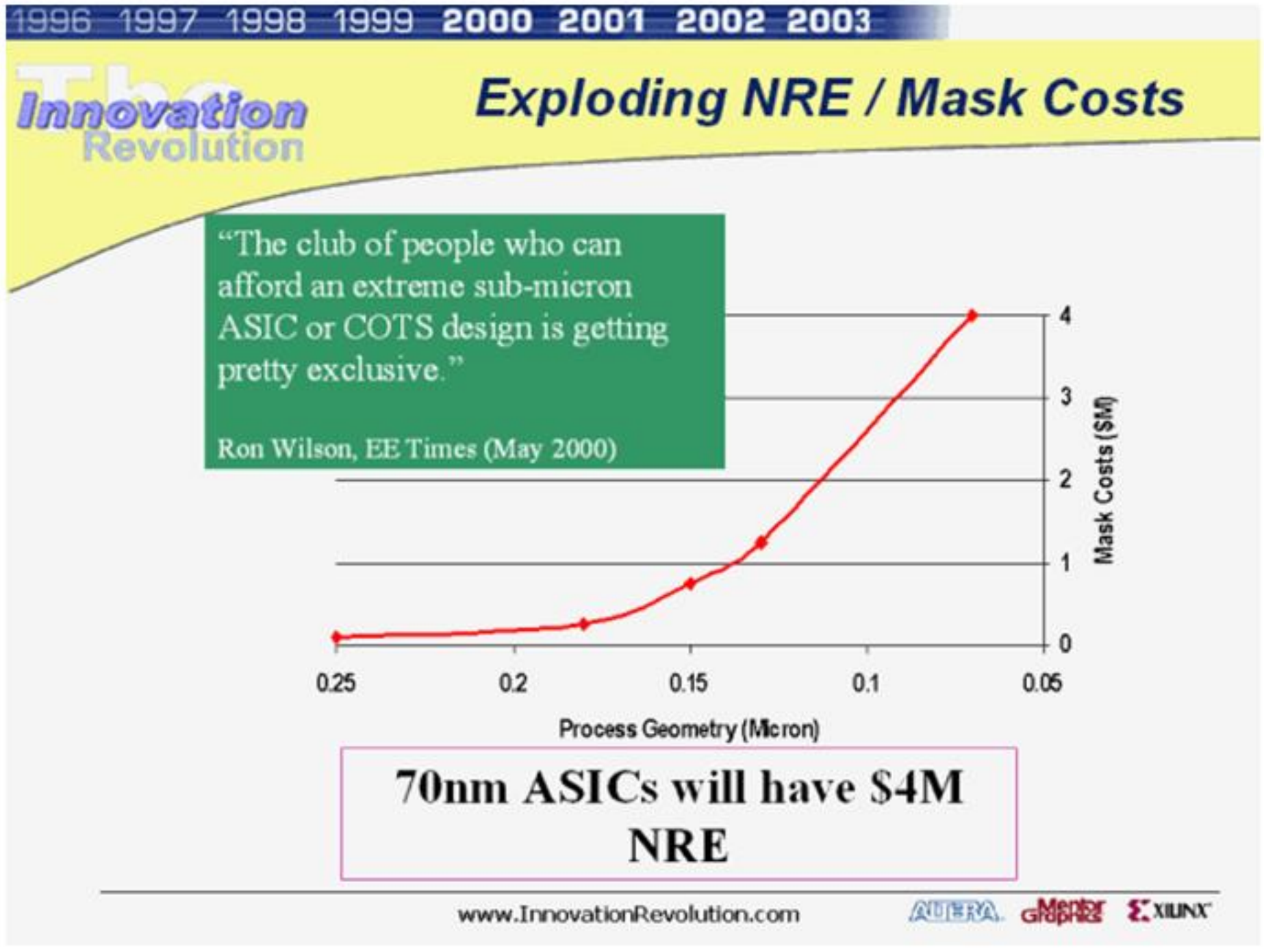
- Funkčnost
- Cena
  - Konstantní náklady - návrhové prostředky, infrastruktura
  - Variabilní náklady – cena vlastního obvodu, zapouzdření, testy
- Spolehlivost, robustnost
  - Odstup šumu
  - Šumová imunita
  - MTBF
- Výkonnost
  - Rychlost (zpoždění)
  - Spotřeba energie
- Doba potřebná pro uvedení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

Cena integrovaného obvodu

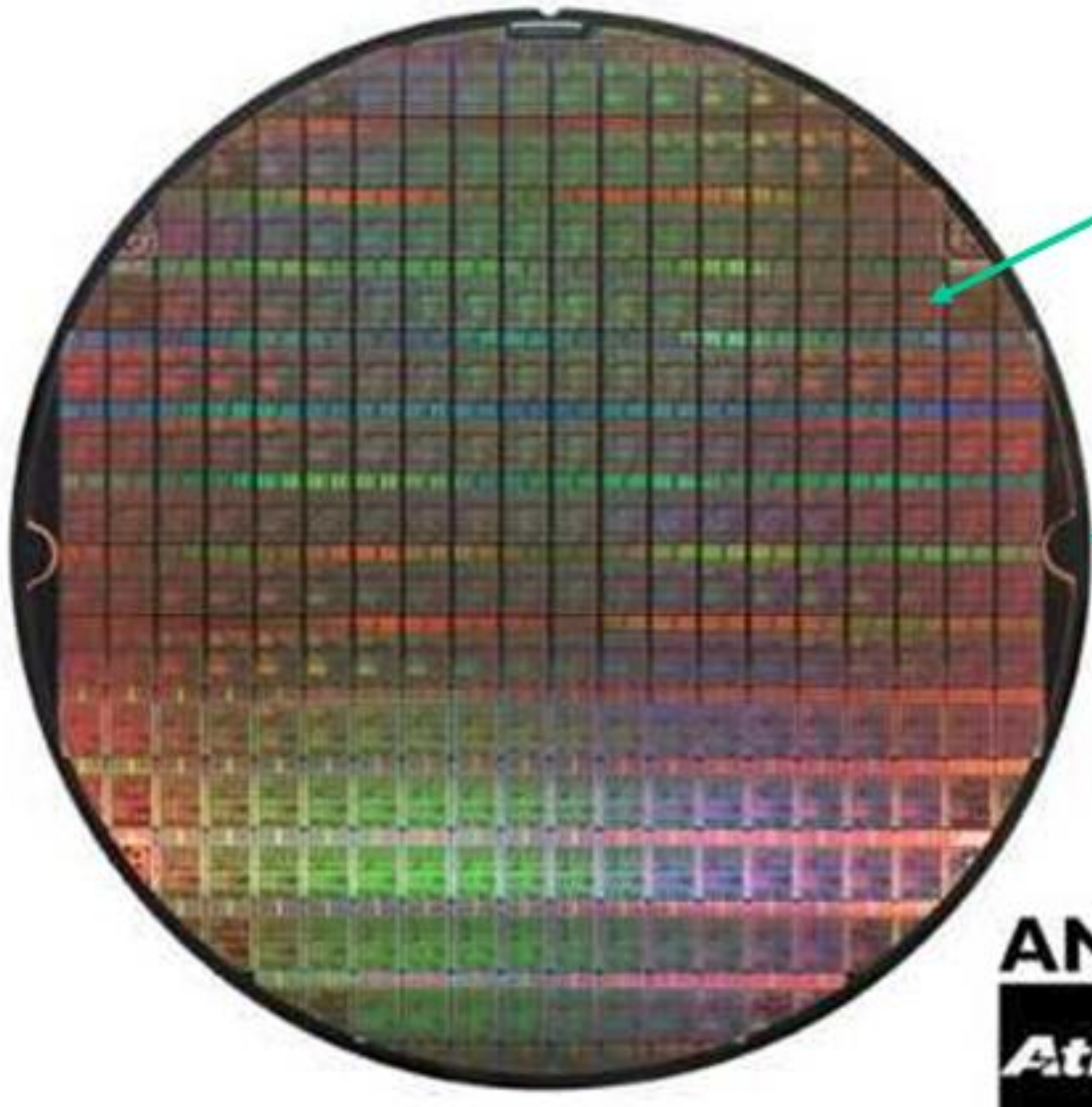
- Konstantní náklady
  - Fixní náklady pro vytvoření návrhu
    - vlastní návrh
    - verifikace návrhu
    - generování masek
  - Ovlivněno složitostí návrhu a produktivitou návrhářů
  - Jsou více významné pro malé objemy výroby
- Variabilní náklady – úměrné objemu výroby
  - zpracování křemíku
   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

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

Konstantní náklady narůstají



Křemíkový plát (wafer)



die

(křemíkový plátek, na kterém je vyroben jeden chip)

wafer

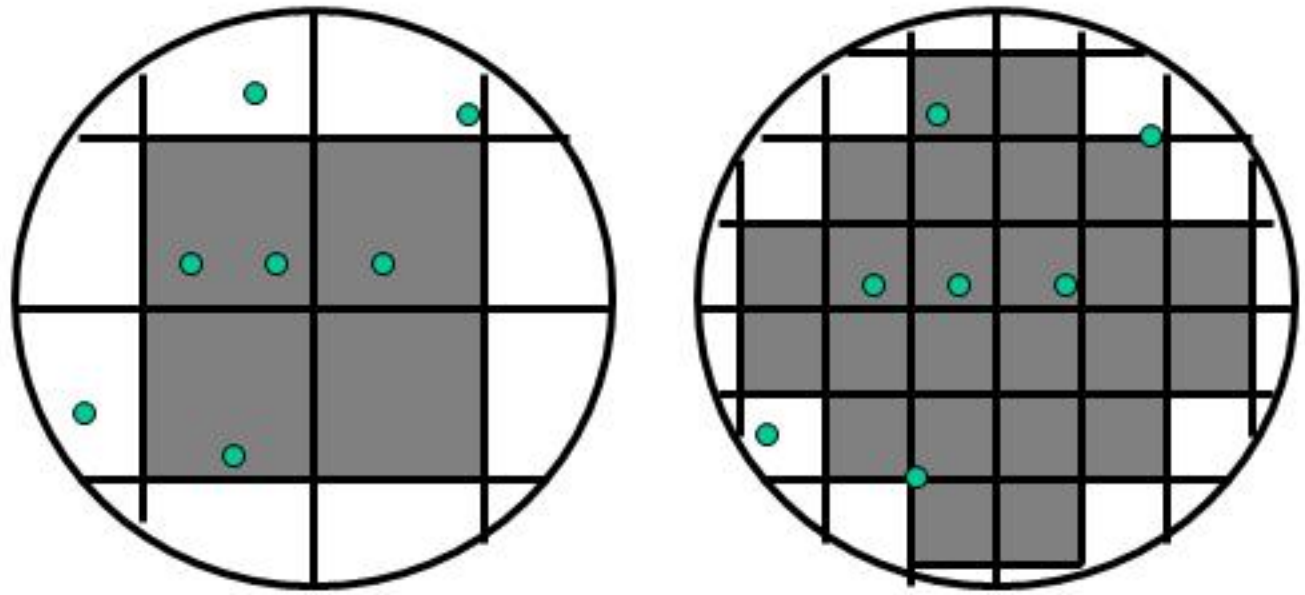
Převzato z: http://www.amd.com

Variabilní náklady

$$\text{variabilní náklady} = \frac{\text{cena die + cena testování die + cena zapouzdření}}{\text{finální výtěžnost při testování}}$$

$$\text{cena die} = \frac{\text{cena waferu}}{\text{počet die na waferu} \times \text{výtěžnost die}}$$

$$\text{počet die na waferu} = \frac{\pi \times (\varnothing\text{waferu}/2)^2}{\text{plocha die}} - \frac{\pi \times \varnothing\text{waferu}}{\sqrt{2} \times \text{délka hrany die}}$$



$$\text{výtěžnost die} = (1 + (\#\text{ defektů na jednotku plochy} \times \text{plocha die})/\alpha)^{-\alpha}$$

Příklad výtěžnosti

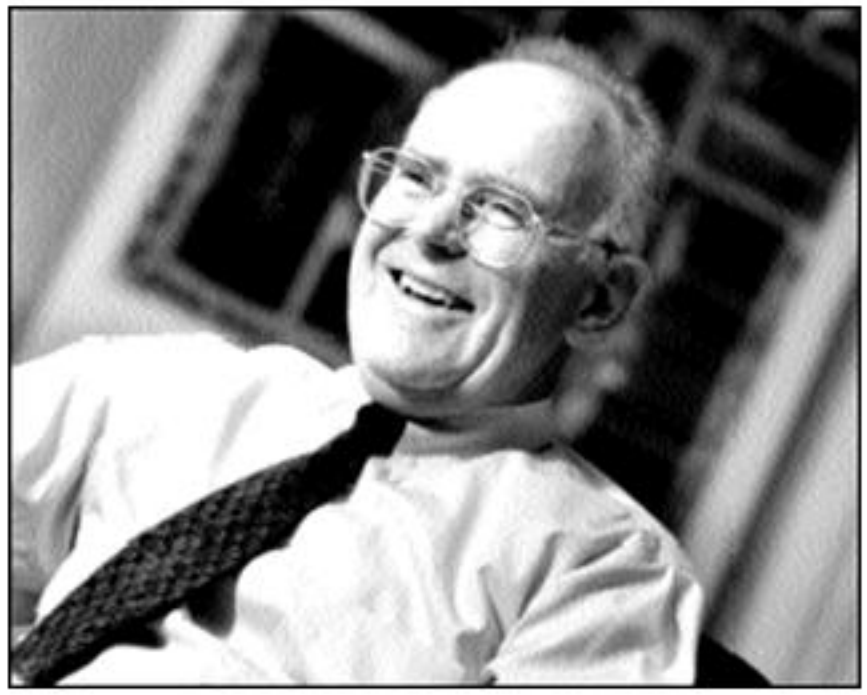
- Příklad
  - průměr waferu 12 palců, plocha die 2.5 cm², 1 defekt/cm², $\alpha = 3$ (závisí na složitosti procesu)
  - 252 die/wafer (nezapomeňte, wafery jsou kulaté, kdežto die čtvercové)
  - výtěžnost die 16%
  - 252 x 16% = výtěžnost pouze 40 die/wafer !

- Cena je funkcí plochy die
  - úměrná třetí až čtvrté mocnině plochy die

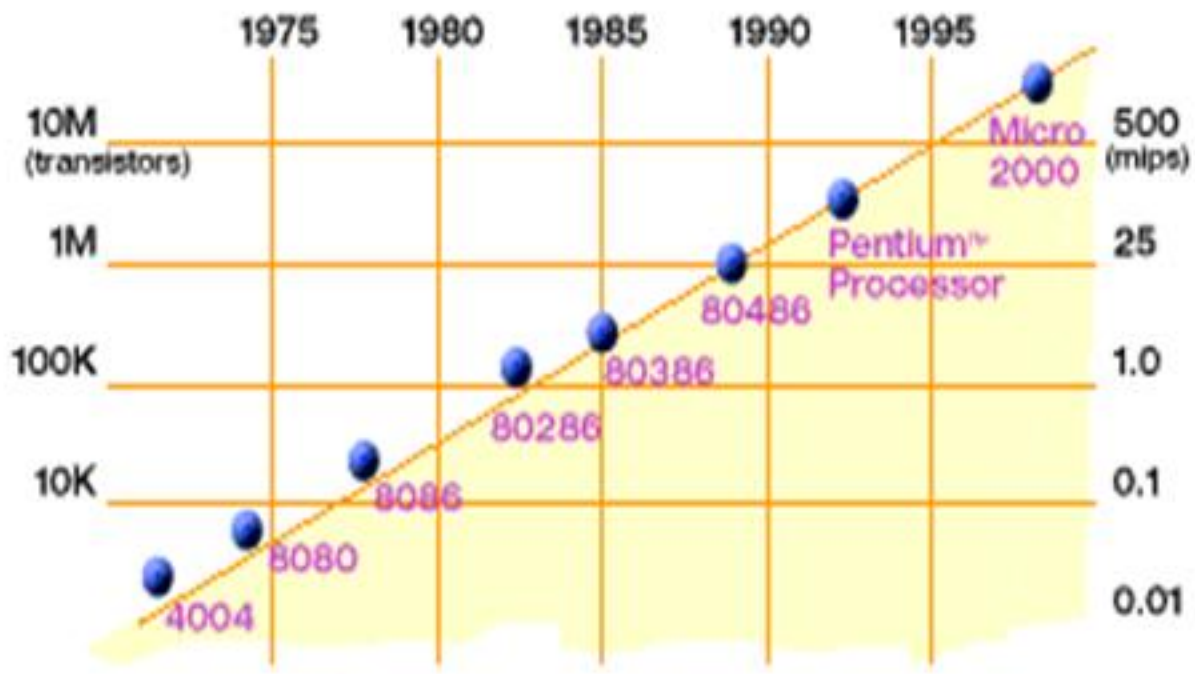
Příklad metrik ceny (kolem 1994)

Chip	Metal layers	Line width	Wafer cost	Defects/ cm^2	Area (mm^2)	Dies/ wafer	Yield	Die cost
386DX	2	0.90	$900	1.0	43	360	71%	$4
486DX2	3	0.80	$1200	1.0	81	181	54%	$12
PowerPC 601	4	0.80	$1700	1.3	121	115	28%	$53
HP PA 7100	3	0.80	$1300	1.0	196	66	27%	$73
DEC Alpha	3	0.70	$1500	1.2	234	53	19%	$149
Super SPARC	3	0.70	$1700	1.6	256	48	13%	$272
Pentium	3	0.80	$1500	1.5	296	40	9%	$417

Mooreův zákon



Gordon Moore (spoluzakladatel Intelu) předpověděl v r. 1965, že hustota tranzistorů polovodičového čipu se zdvojnásobí každých 18 měsíců.

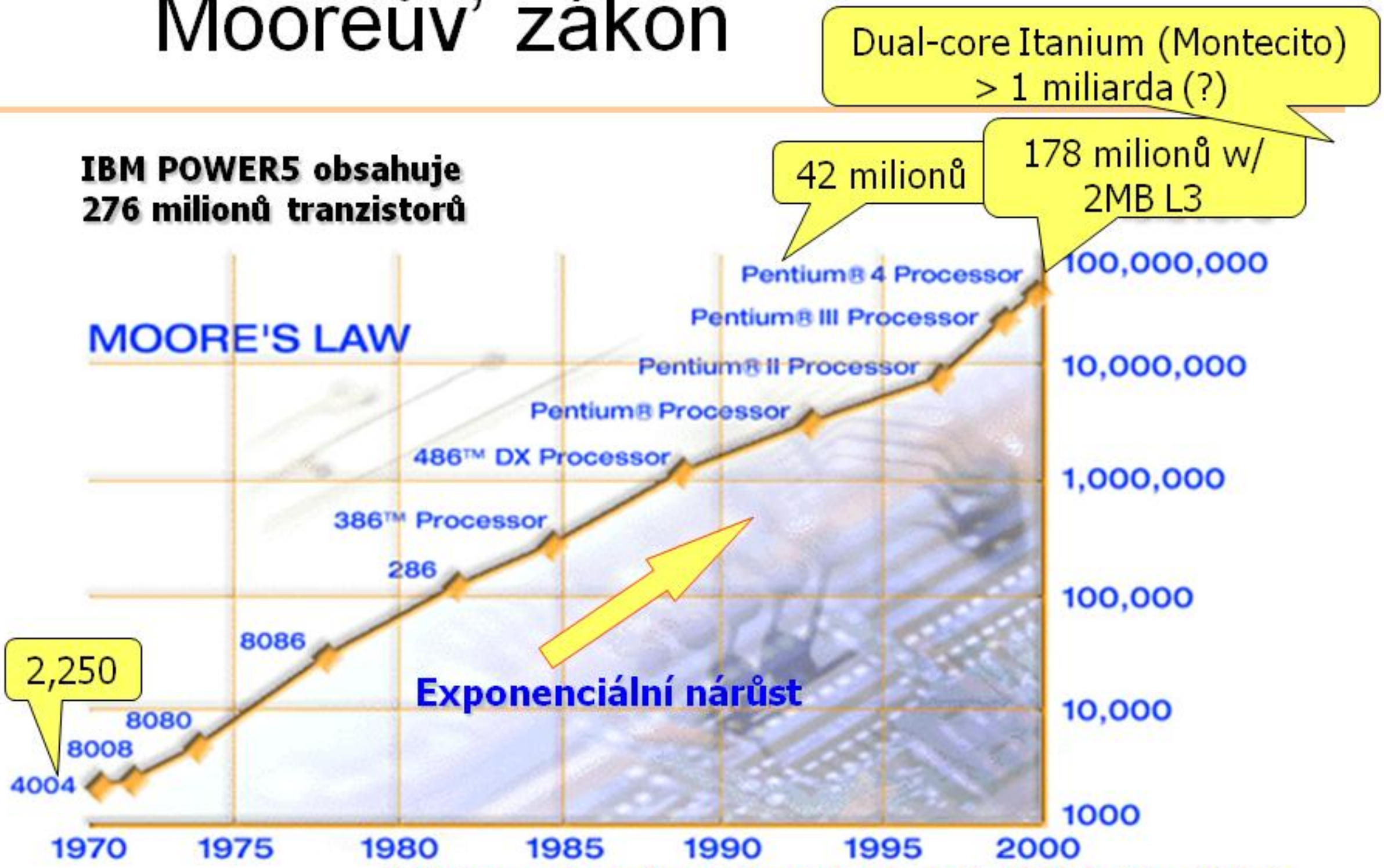


Mooreův zákon platil velmi dlouhou dobu

Hustota souvisí s rychlostí. (Jak je známo, činit odhady o rychlosti počítačů je obtížné.)

Mooreův zákon

IBM POWER5 obsahuje 276 milionů tranzistorů



Počet tranzistorů se zdvojnásobuje každých 18 měsíců

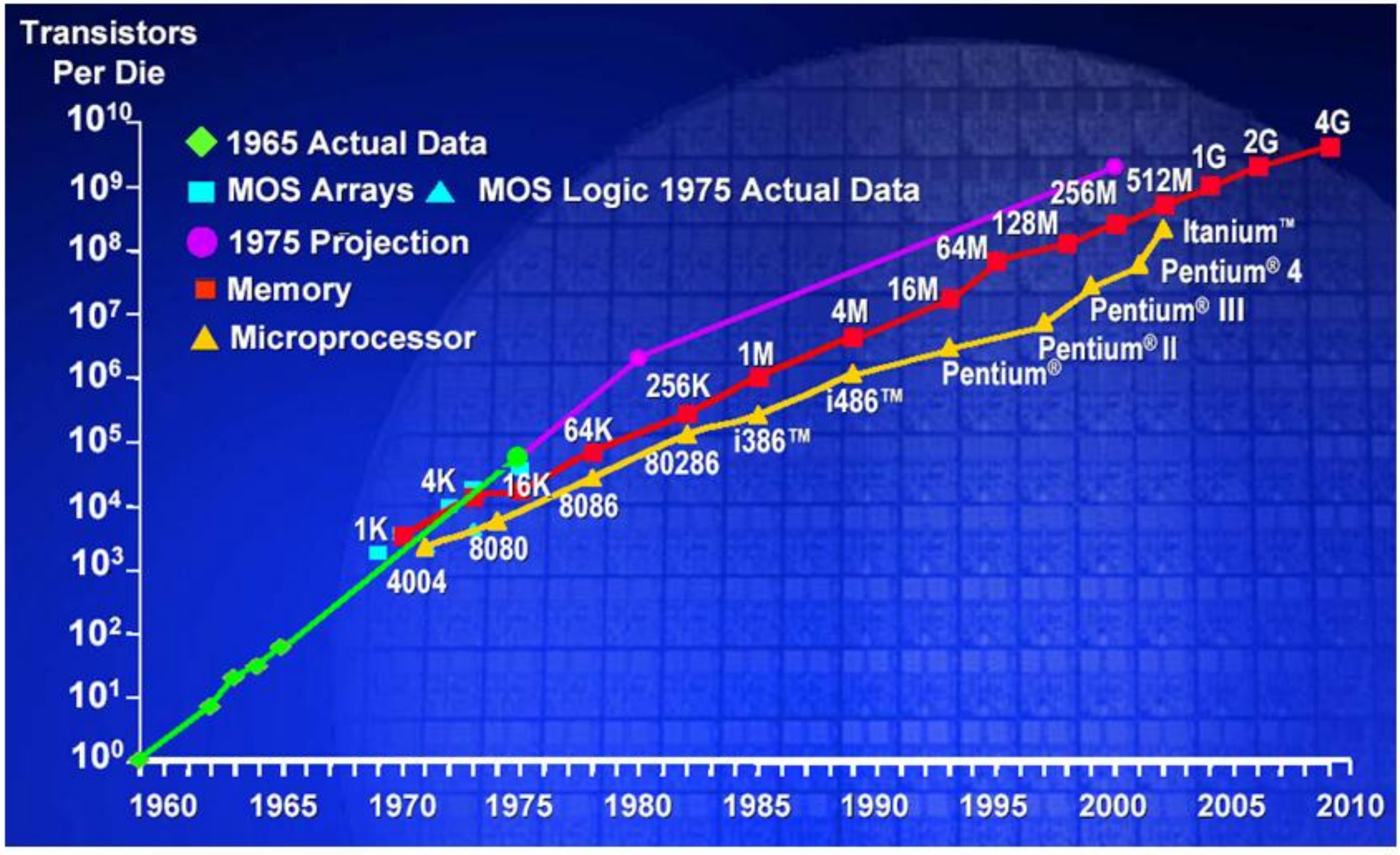
— Gordon Moore, spoluzakladatel Intelu

Trendy technologie

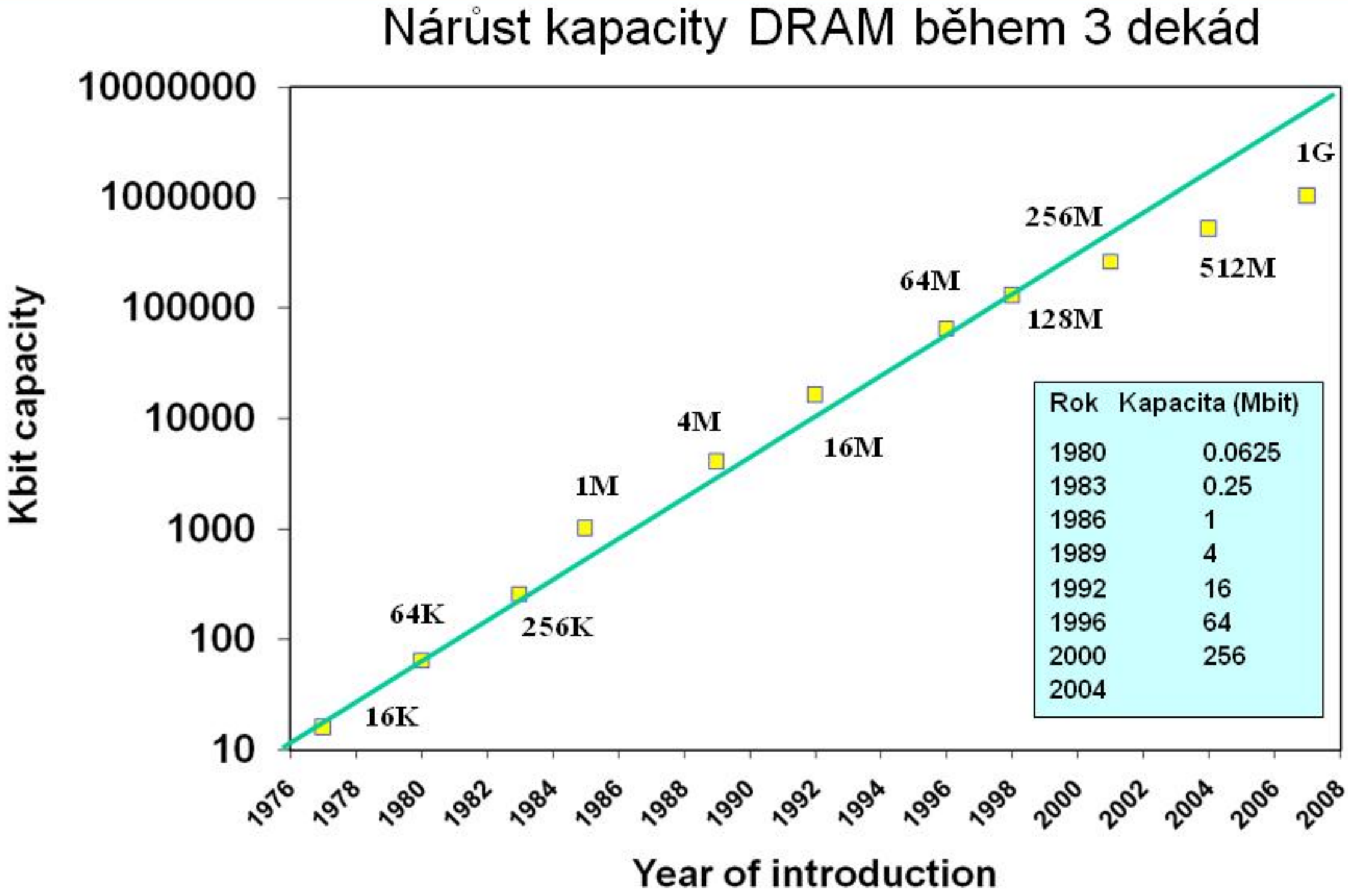
Rok	2004	2006	2008	2010	2012
Char. velikost (nm)	90	65	45	32	22
Intg. kapacita (BT)	2	4	6	16	32

- Zajímavá fakta o 45nm tranzistorech
  - 30 milionů se jich vejde na hlavičku špendlíku
  - 2,000 jich lze umístit přes lidský vlas
  - Kdyby cena aut klesala stejnou rychlostí od roku 1968 jako cena tranzistoru, stálo by dnešní auto kolem 1 centu

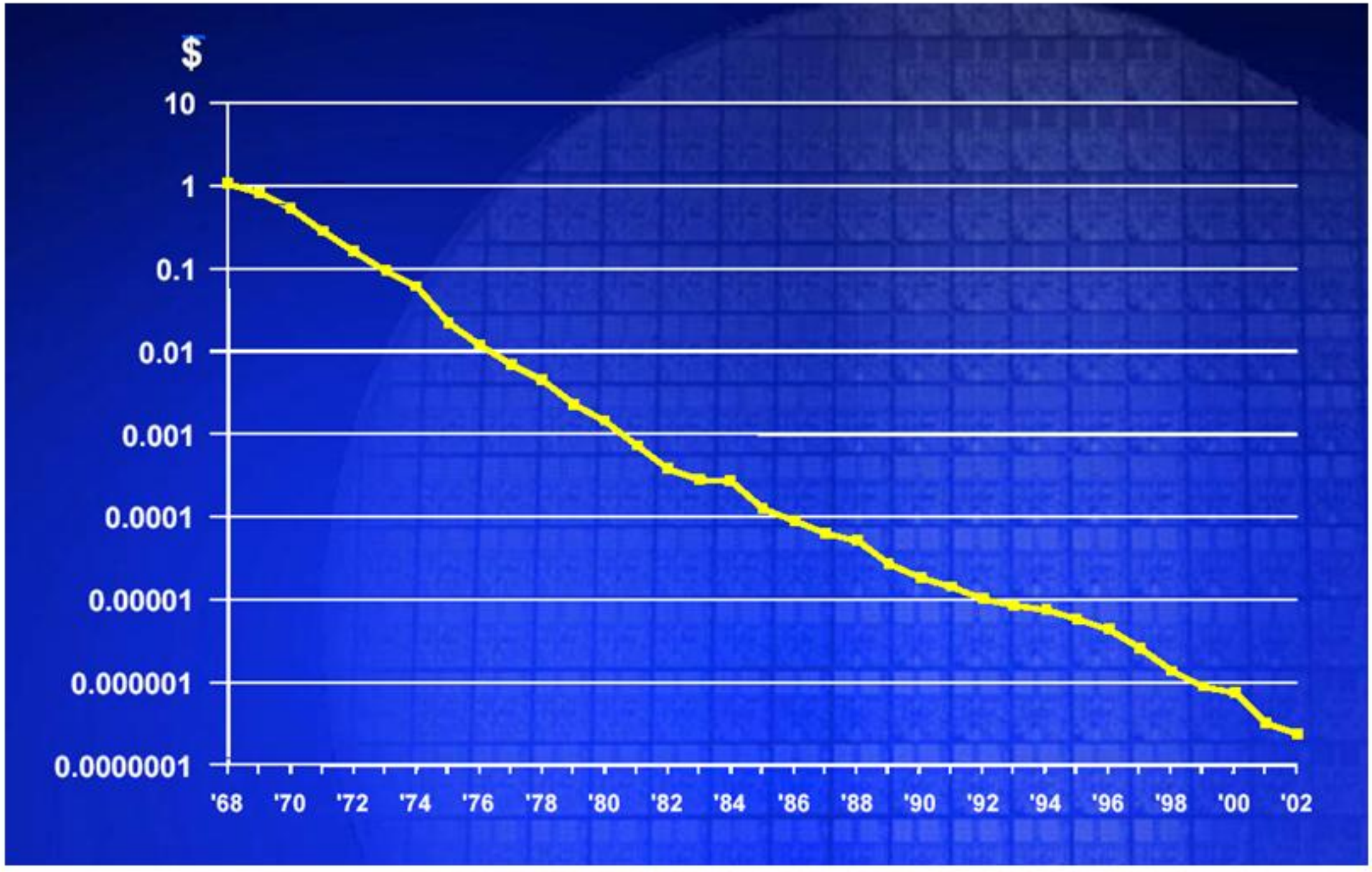
Kapacita integrovaných obvodů



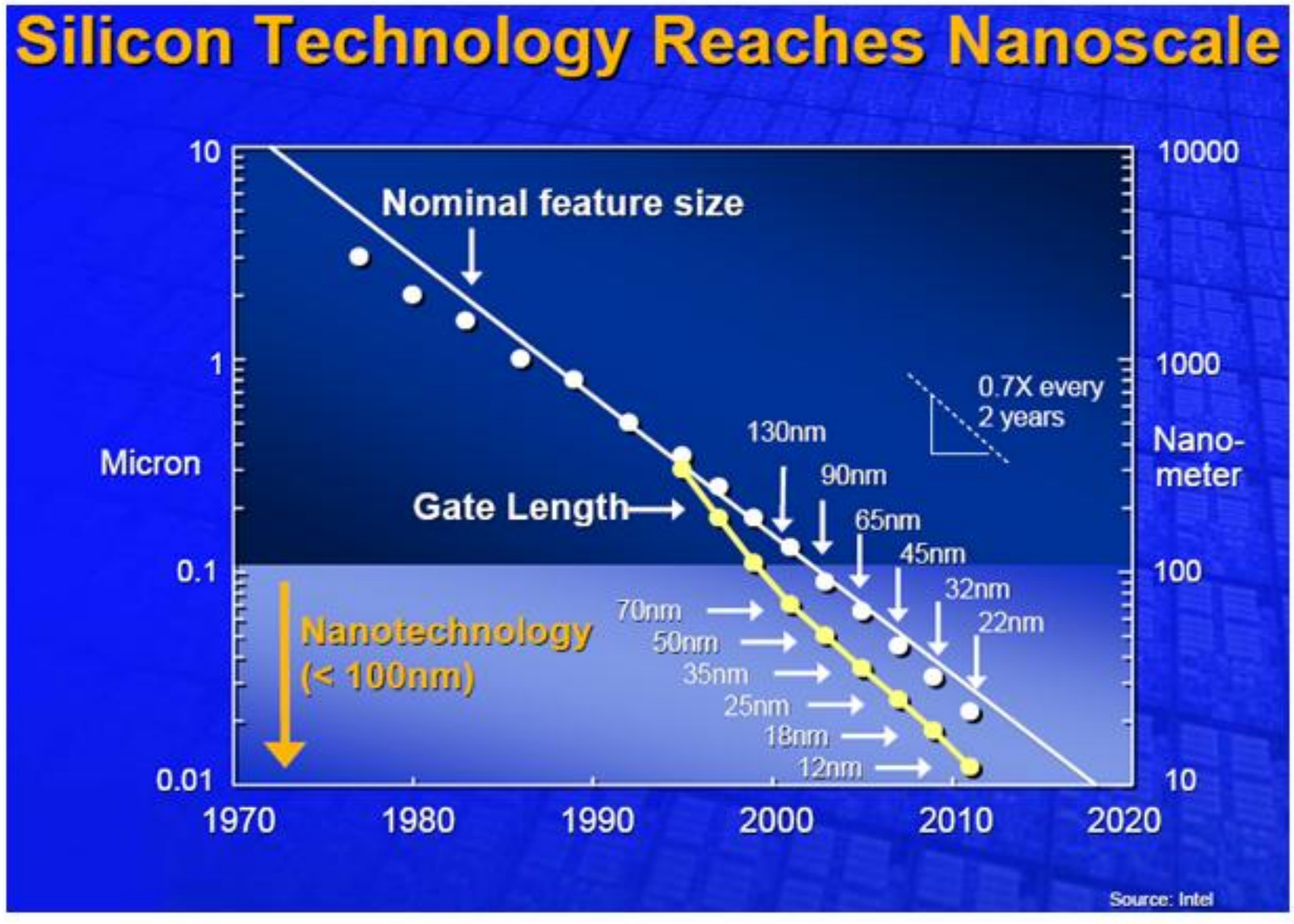
Další příklad Mooreova zákona



Průměrná cena tranzistoru

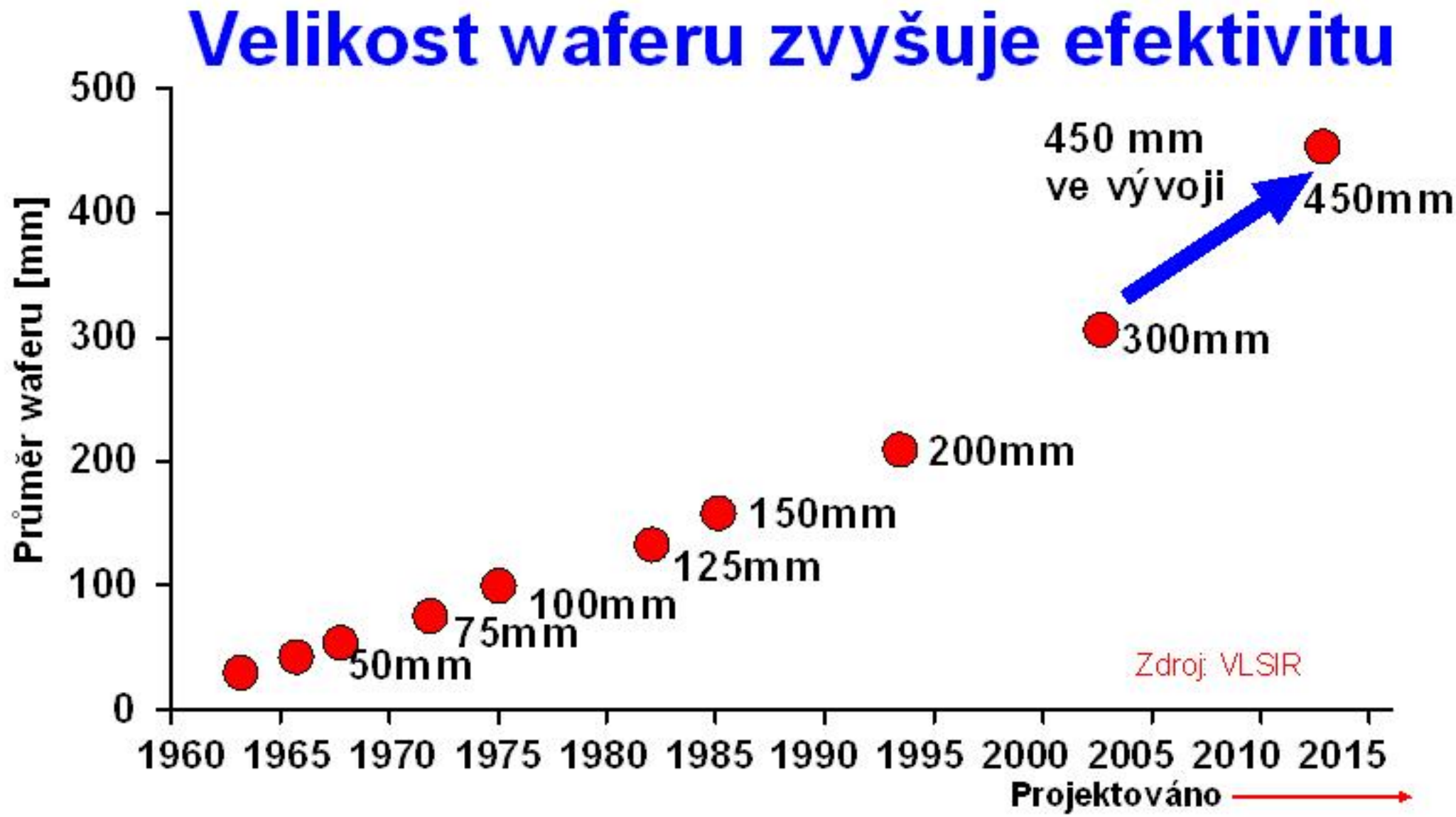


Vývoj technologie křemíku



Velikost prvku klesá na 70 % každých 18 až 24 měsíců

Velikost waferu

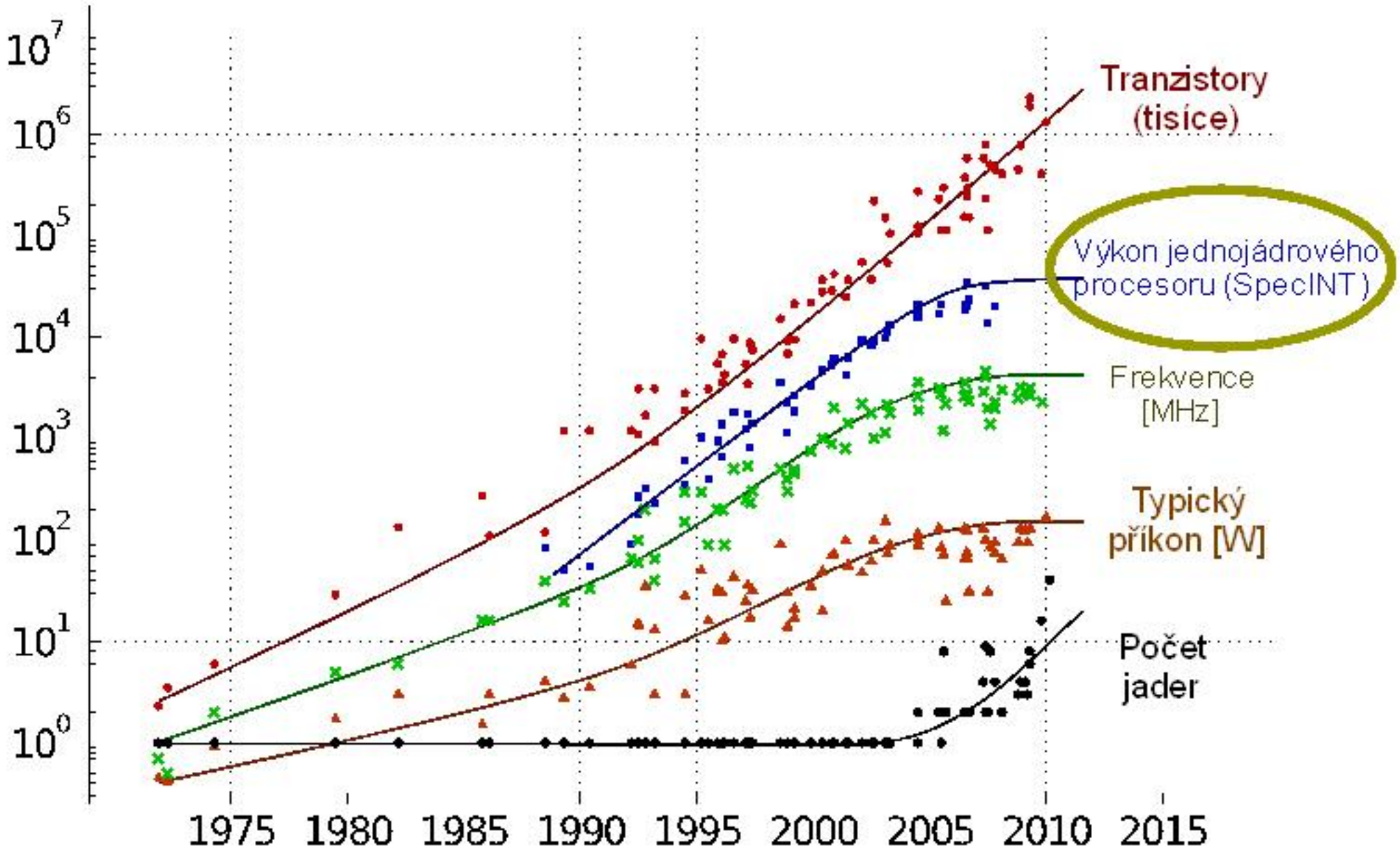


Intel CPU

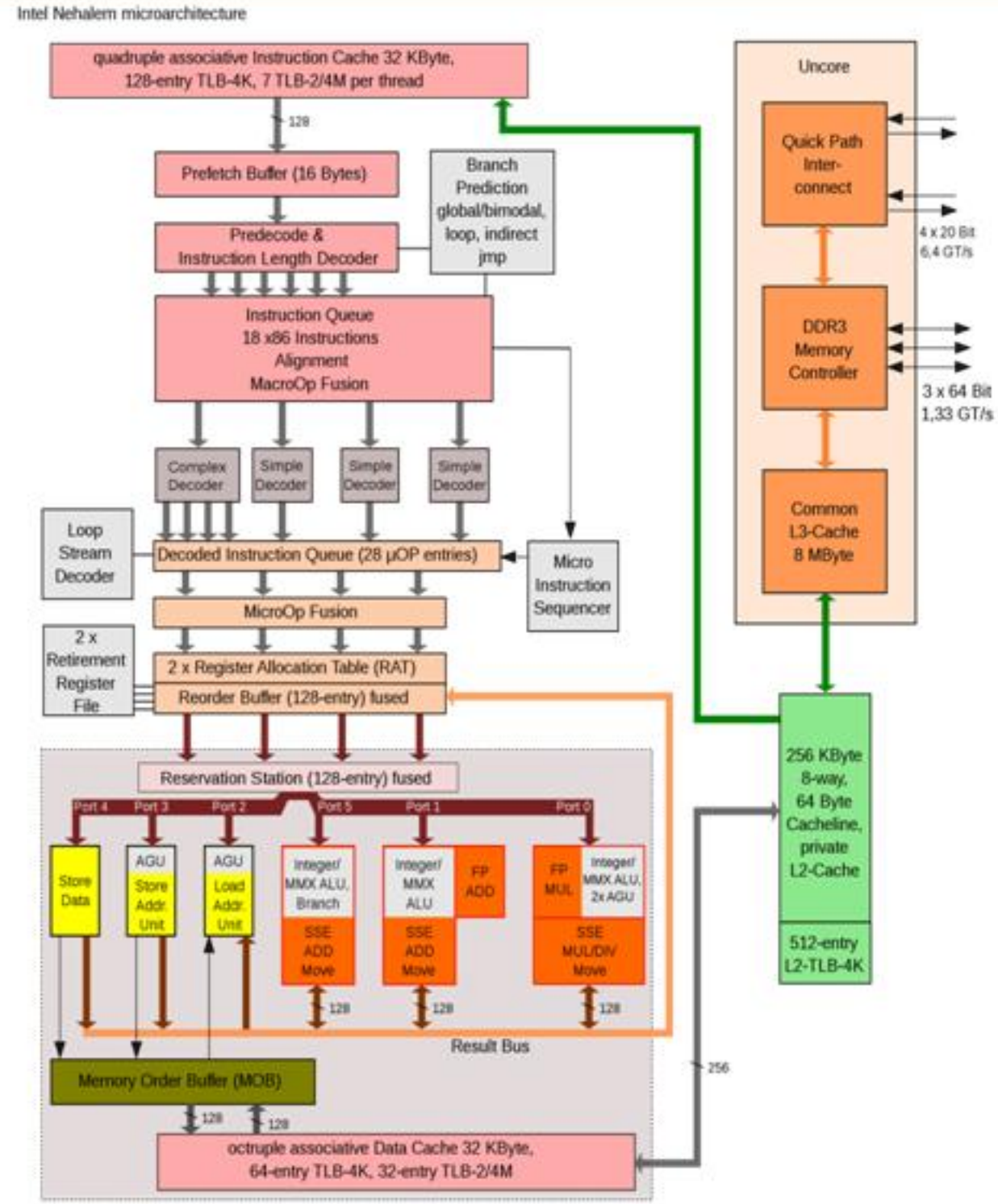
Chip	Date	MHz	Transist	Memory	Notes
4004	4/1971	0.108	2,300	640	First microprocessor on a chip
8008	4/1972	0.108	3,500	16 KB	First 8-bit microprocessor
8080	4/1974	2	6,000	64 KB	First general-purpose CPU on a chip
8086	6/1978	5-10	29,000	1 MB	First 16-bit CPU on a chip
8088	6/1979	5-8	29,000	1 MB	Used in IBM PC
80286	2/1982	8-12	134,000	16 MB	Memory protection present
80386	10/1985	16-33	275,000	4 GB	First 32-bit CPU
80486	4/1989	25-100	1.2M	4 GB	Built-in 8K cache memory
Pentium	3/1993	60-233	3.1M	4 GB	Two pipelines; later models had MMX
Pentium Pro	3/1995	150-200	5.5M	4 GB	Two levels of cache built in
Pentium II	5/1997	233-400	7.5M	4 GB	Pentium Pro plus MMX

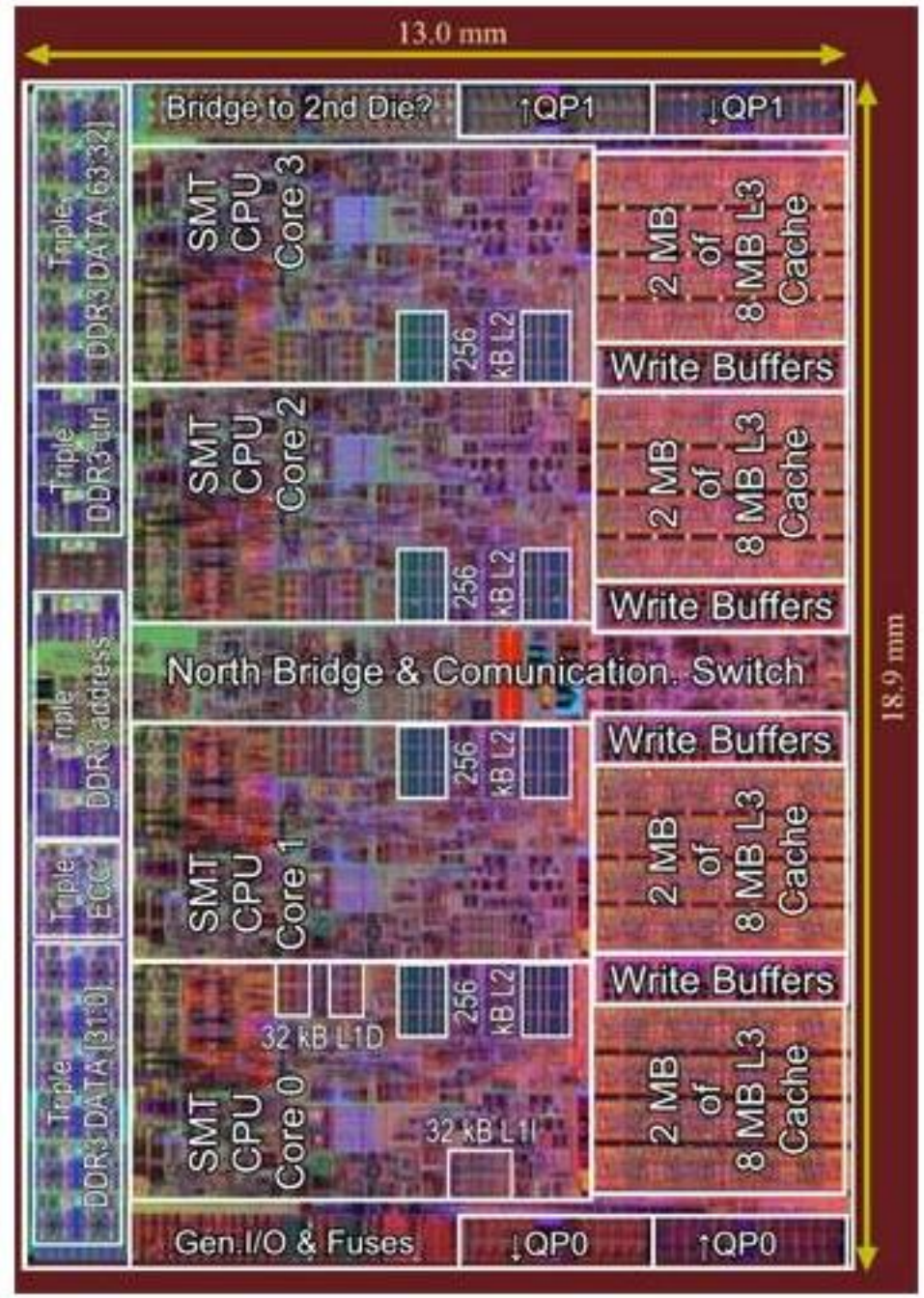
Pentium III – 800 MHz, 4GB Memory // Pentium 4 – 2+GHz, 4GB

Konec zvyšování výkonu jednojádrových procesorů



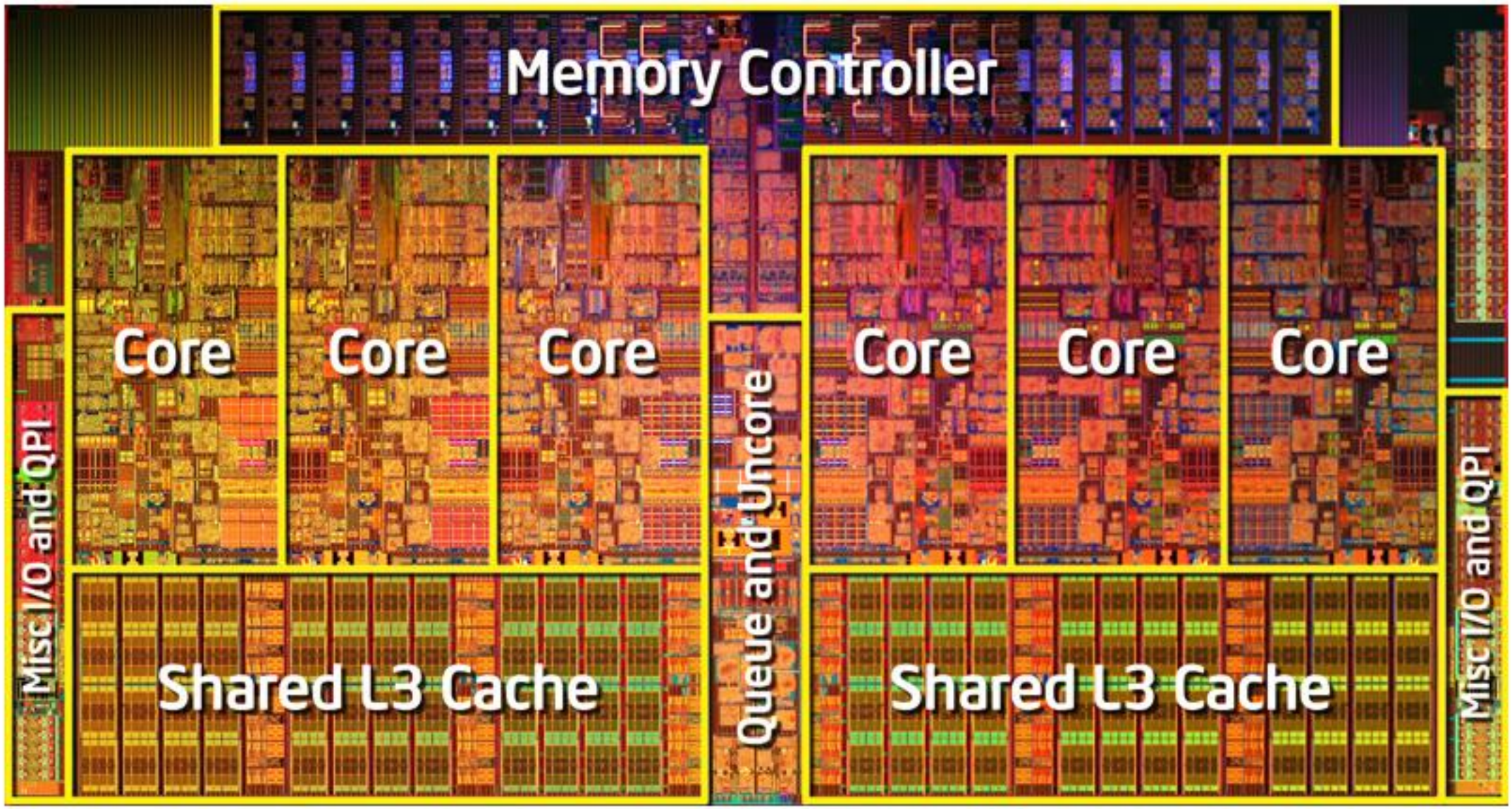
Mikroarchitektura Nehalem





Nehalem die

Westmere die (6 jader)



Trendy technologie hardware

- **Procesor**
  - 2x větší rychlost každých 1.5 roku
    100x větší výkon za poslední dekádu
- **Paměť**
  - DRAM kapacita: 2x / 2 roky; **64x** kapacita za dekádu
  - Cena za bit: zlepšení asi o 25% za rok
- **Disk**
  - kapacita: > 2x kapacita každý 1.0 rok
  - Cena za bit: zlepšení asi o 100% za rok (1/2 cena)
  - **120x kapacita** v poslední dekádě
- Nové jednotky! *Giga* (10^9) *Tera* (10^{12})

Fyzikální limity Mooreova zákona

- Limity vlivem nutných izolačních vzdáleností **(2-3nm)**
- Kvantové tunelové efekty => *crosstalk*
- Jak moc zmenšovat? **(0.02 micron / 2nm = 10x)**
- O kolik rychlejší? Rychlost = k * plocha
  - o 3 až 4 řády rychlejší (10^3- 10^4)
  - 2.6 GHz nyní => 5 THz až 10 THz ???
- ***Kdy?*** **(během příštích 10-15 let …)**

Dospěje vývoj počítačů ke svým limitům?

Možnosti:

- Rychlejší procesory, algoritmy využívající stávající technologii
- Nárůst šířky pásma sběrnic, které dodávají procesoru data
- Nalezení kompaktnějších způsobů kódování dat v procesu zpracování
- Neustávající zlepšování výkonu procesorů, pamětí a komunikace
- Technologie ⇔ aplikace – jedno podporuje rozvoj druhého
  - Kompilátory
  - Křemík
- Bude Mooreův zákon platit navždy? ☹/☺
  - Objevují se „skeptické“ názory, že éra platnosti Mooreova zákona je u konce.

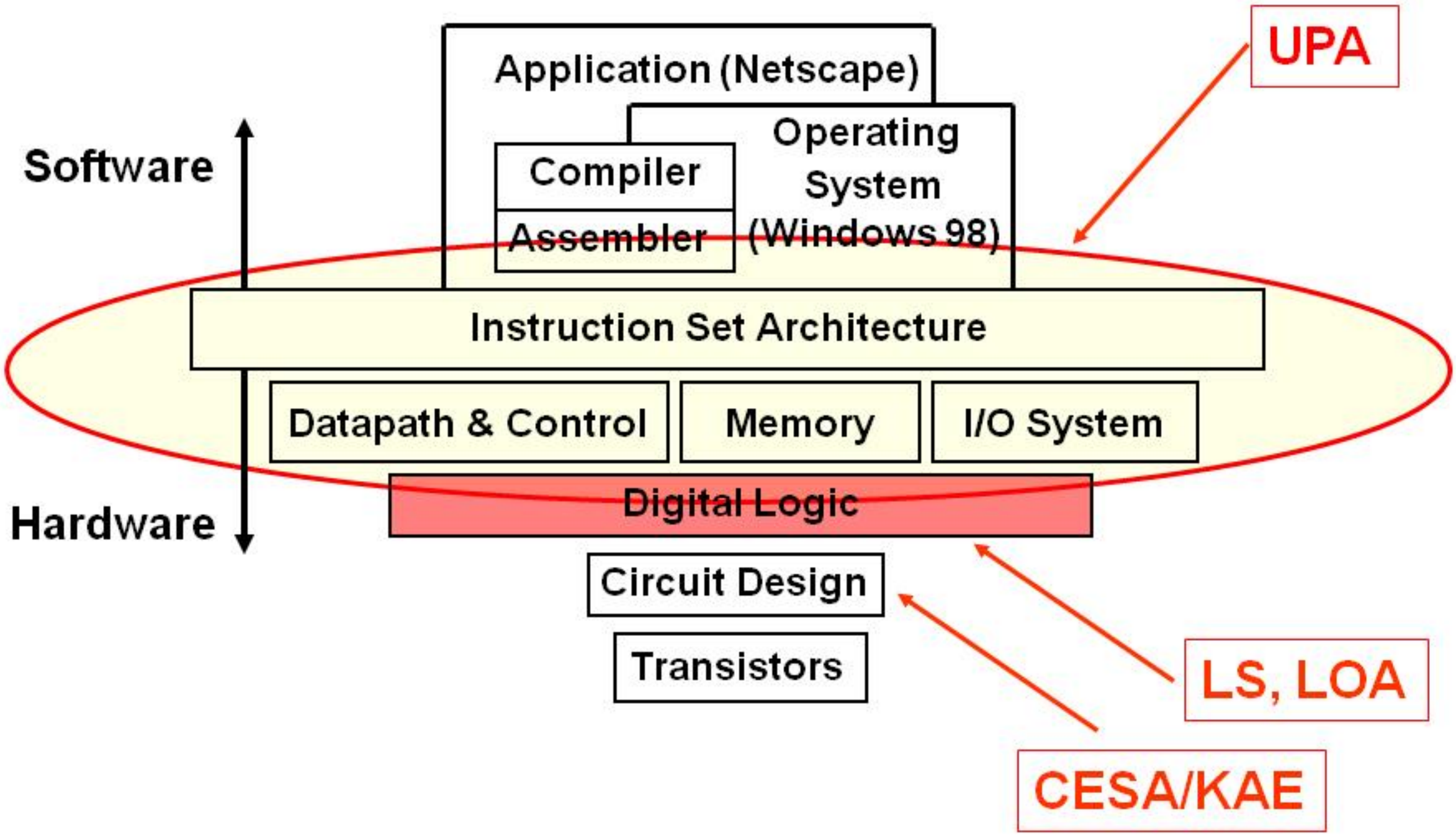
Řešení (?) Mooreova zákona

- **Kvantové počítače ☺**
  - Jiný přístup – všechny výsledky najednou
  - Jak nalézt "správný" výsledek?
  - Implementace: Optika? Křemík? ???
  - Zatím jen zprávy o úspěšných pokusech s kvantovými prvky
- **Hluboce experimentální technologie**
  - Využití spinu pro realizaci paměťového prvku
  - DNA Computing (Pattern Matching)

Nové téma – digitální logika

- Digitální logika – viz předmět LS, LOA
- Booleovské operace, Booleova algebra
- Tranzistory a digitální logika
- Základní hradla – *and*, *or*, *not*
  - Implementace tranzistoru
  - Pravdivostní tabulky
- Komplexní logické obvody
- Kombinační logické systémy
- Paměťové prvky
- Taktování

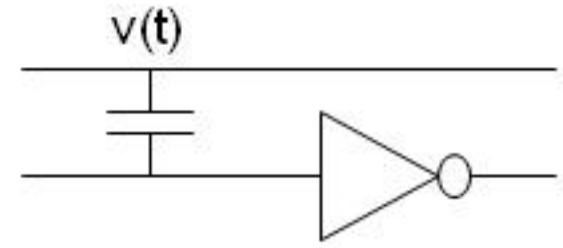
Digitální logika

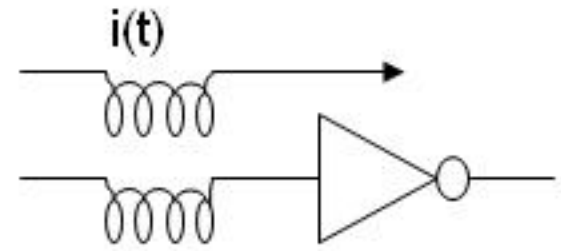


Spolehlivost a robustnost

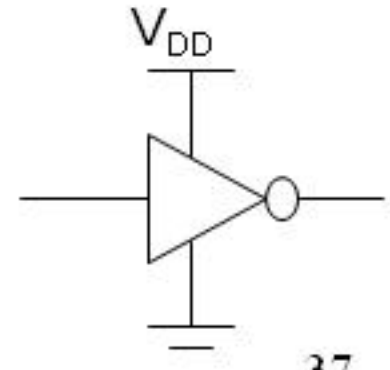
Šum v digitálních integrovaných obvodech

- 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



Příklad kapacitní vazby

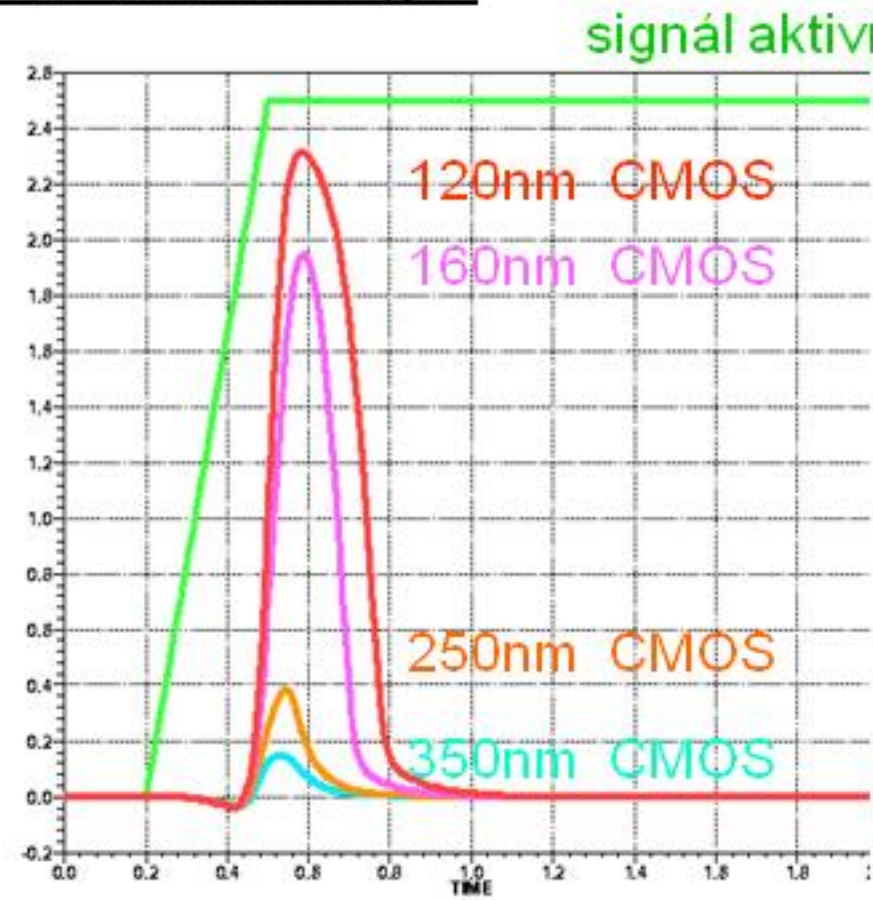
- Zákmity na signálovém vodiči o velikosti 80% napájecího napětí jsou běžné vlivem přeslechů mezi sousedními vodiči.

Přeslechy versus technologie

Černý vodič - v klidu

Červený vodič aktivní

Velikost špiček v závislosti na technologii



Dunlop, Lucent, 2000

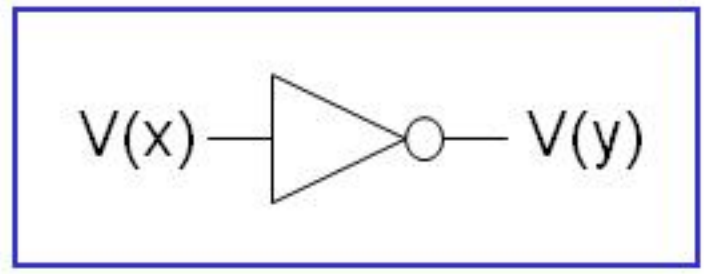
Statické chování hradla

- Statické parametry hradla – *statické chování* – říkají, jak je obvod odolný vzhledem k variacím ve výrobním procesu a jak je závislý na šumu.
- Digitální obvody pracují s Booleovskými proměnnými

$$x \in \{0,1\}$$

- Logická proměnná je asociována s *nominální napěťovou úrovní* pro každou logickou hodnotu

$$1 \Leftrightarrow V_{OH} \text{ a } 0 \Leftrightarrow V_{OL}$$

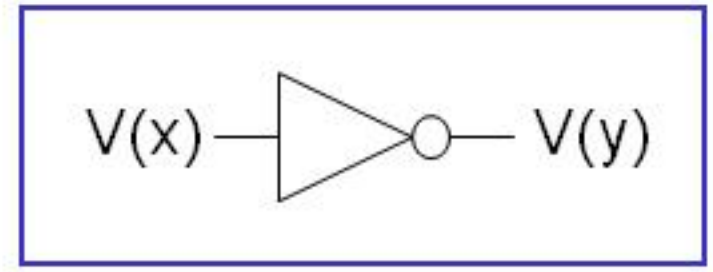


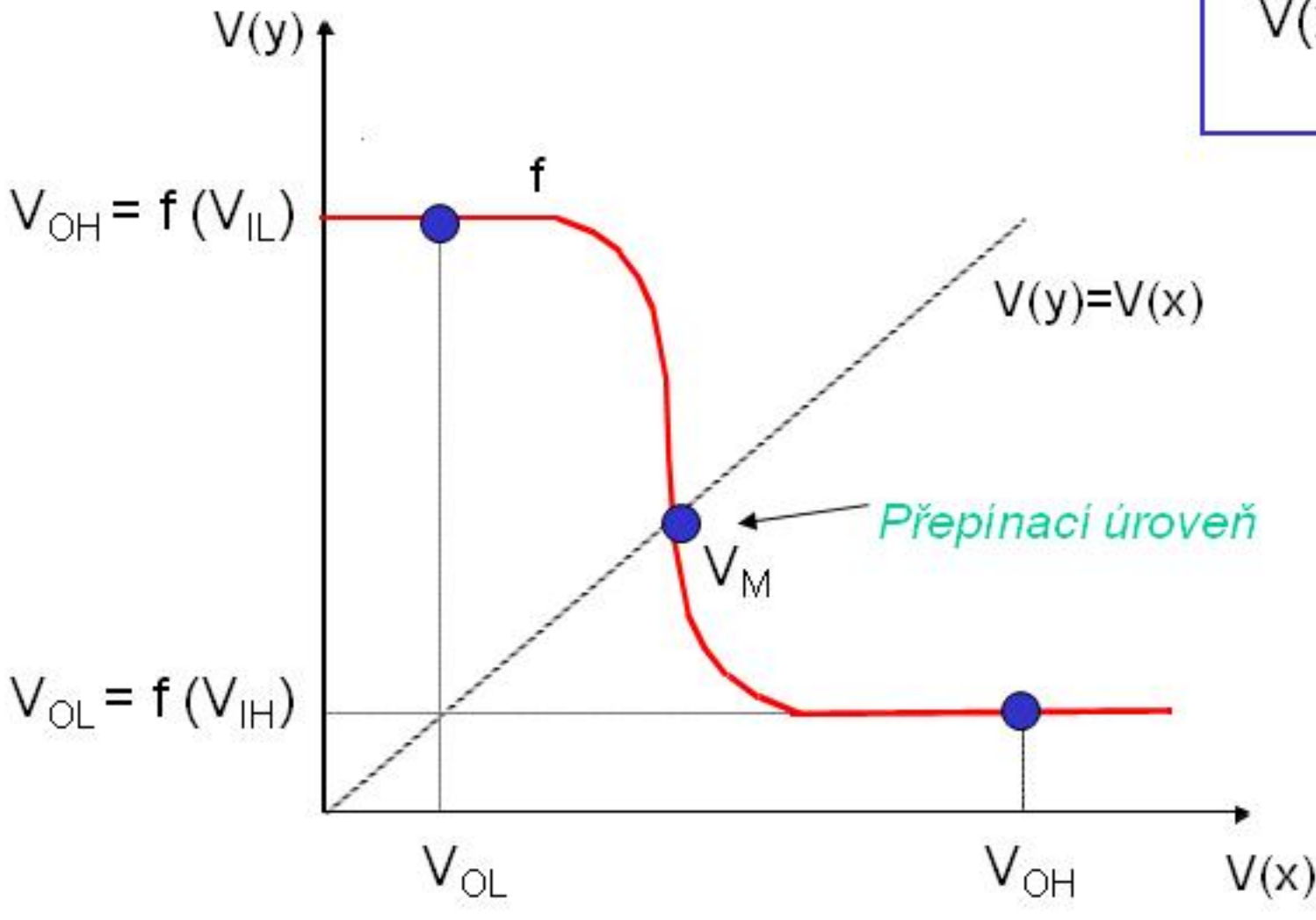
$$V_{OH} = !\,(V_{OL})$$
$$V_{OL} = !\,(V_{OH})$$

- Rozdíl mezi V_{OH} a V_{OL} je *logický* nebo *signálový zdvih* V_{sw}

Převodní charakteristika hradla

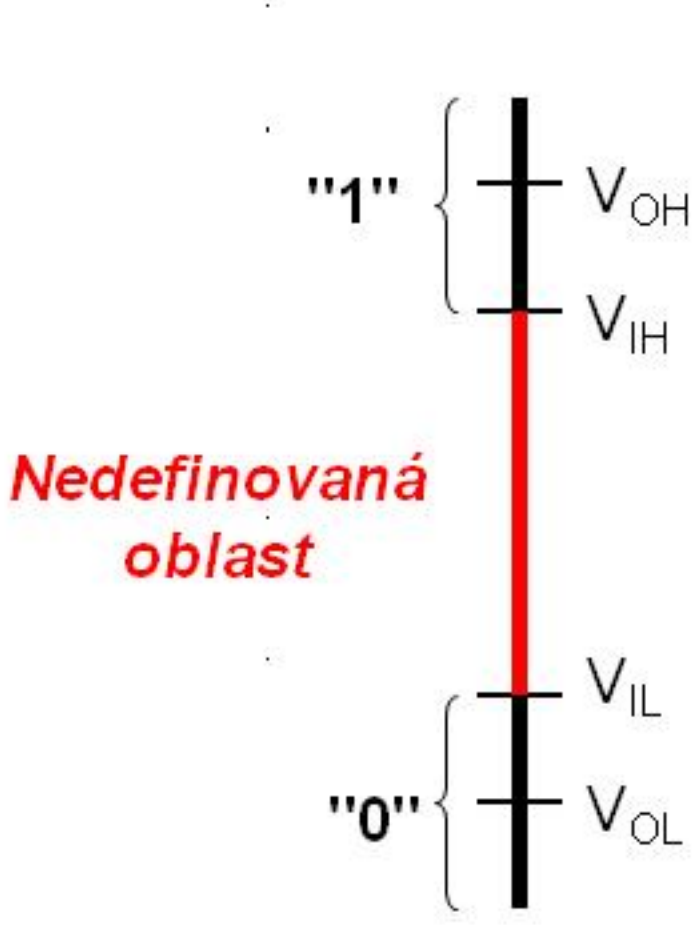
- Závislost výstupního napětí na vstupním napětí

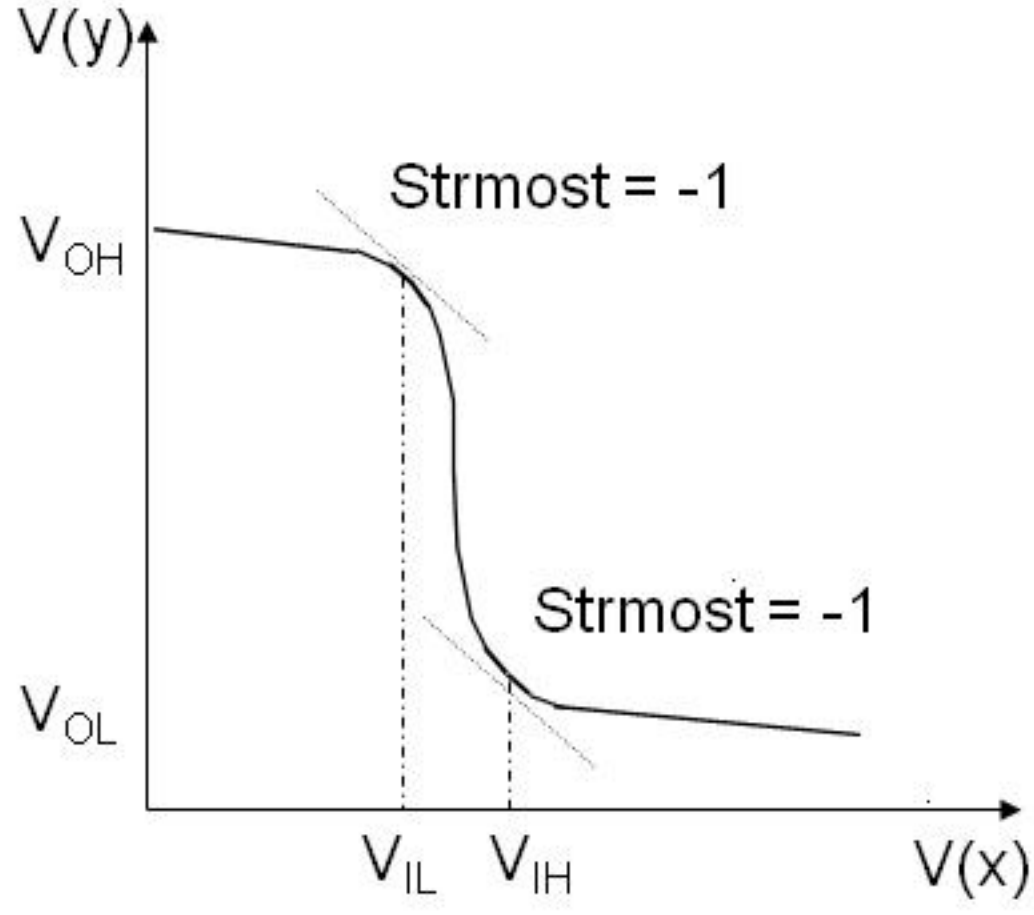




Mapování logických úrovní do napěťové oblasti

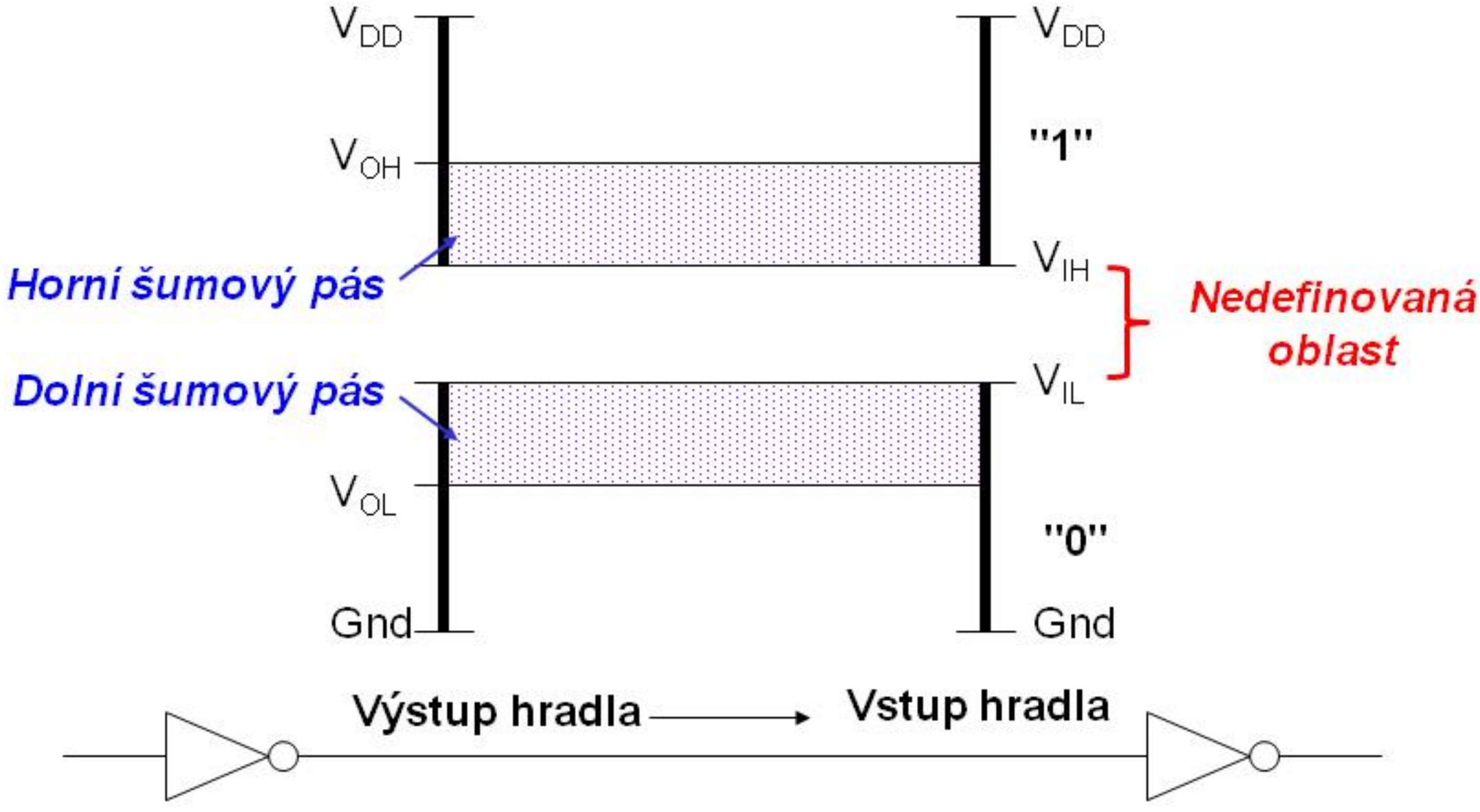
- Oblasti akceptovatelného vyššího a nižšího napětí jsou vymezeny hodnotami V_{IH} a V_{IL}, které reprezentují body na převodní charakteristice hradla se ziskem rovným -1.





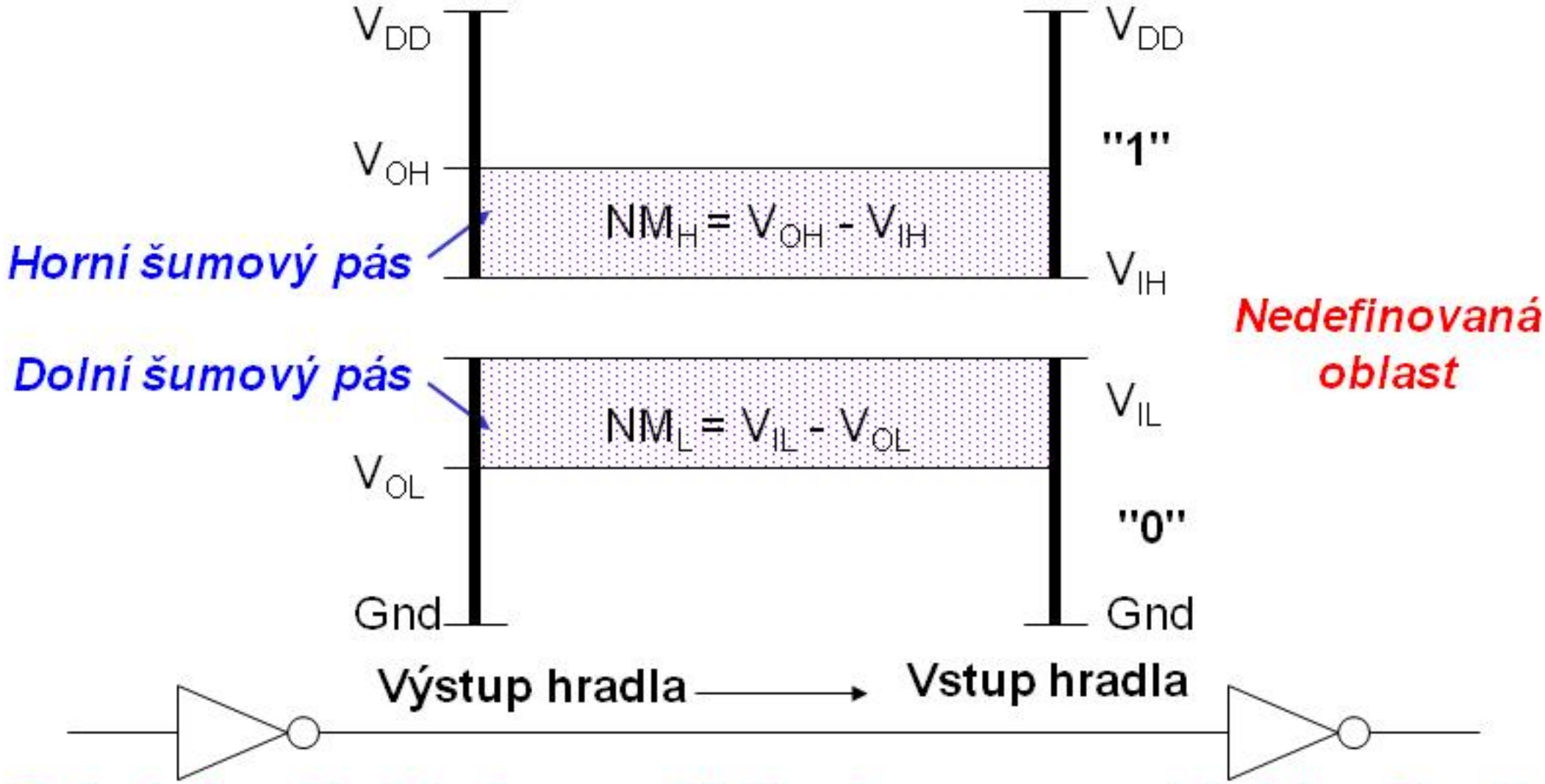
Odstup šumu

- „Robustní“ obvody vyžadují, aby intervaly pro “0” a “1” byly co možná nejširší



Odstup šumu

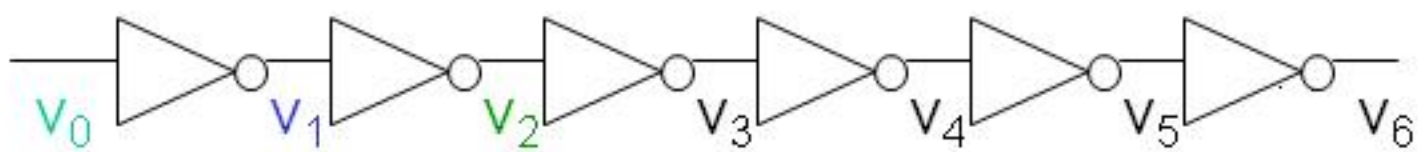
- „Robustní“ obvody vyžadují, aby intervaly pro “0” a “1” byly co možná nejširší

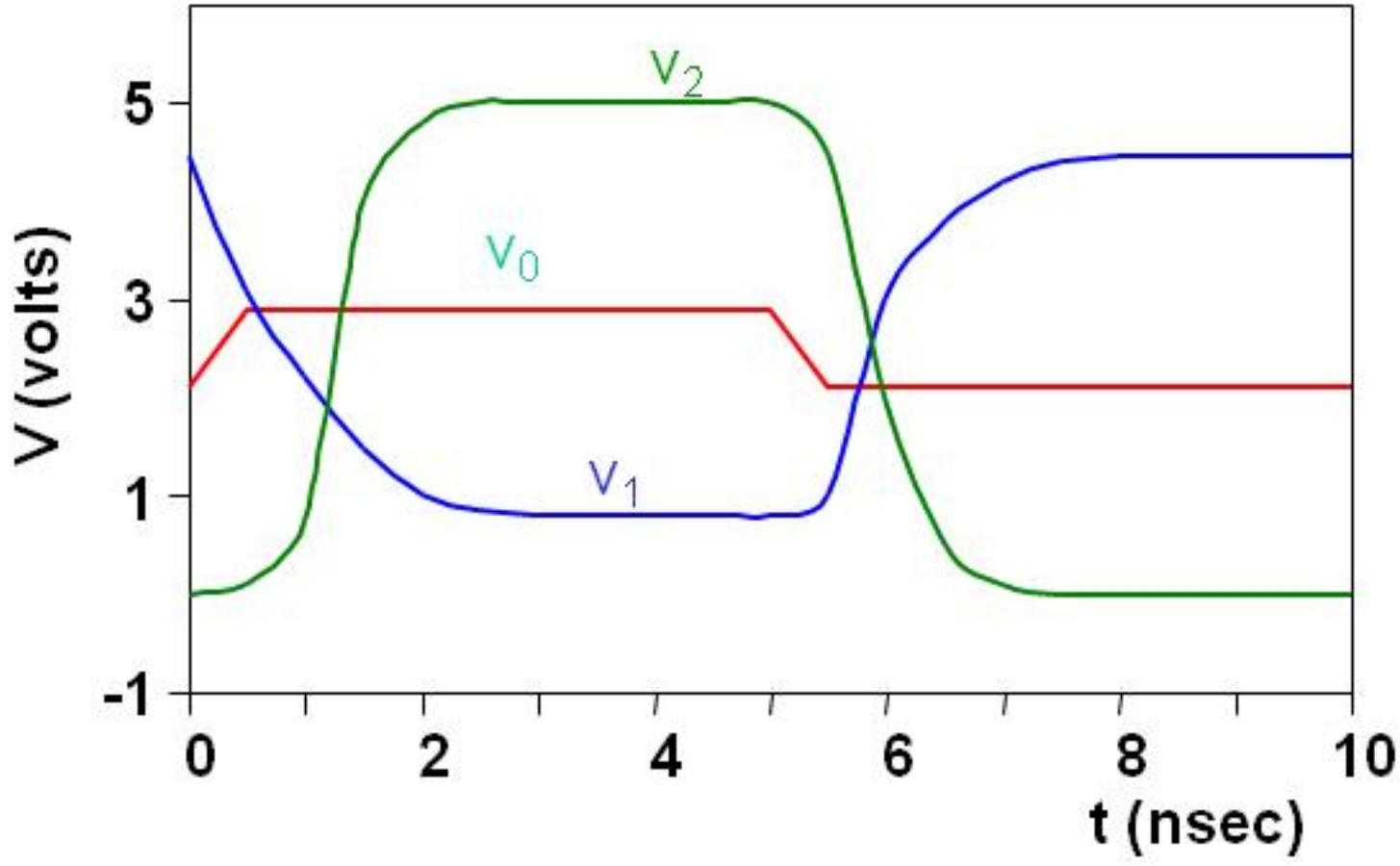


- Požadují se široké pásy napětí, které se reprezentuje jako „0“ nebo „1“, ale nestačí to …

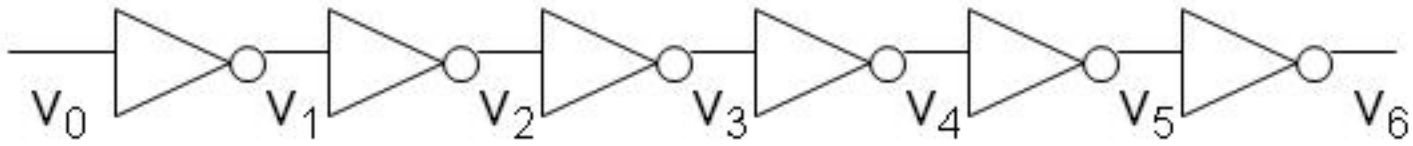
Vlastnost regenerace

- Hradlo s vlastností regenerace zajišťuje, že rušený signál konverguje zpět na nominální napěťové úrovně.

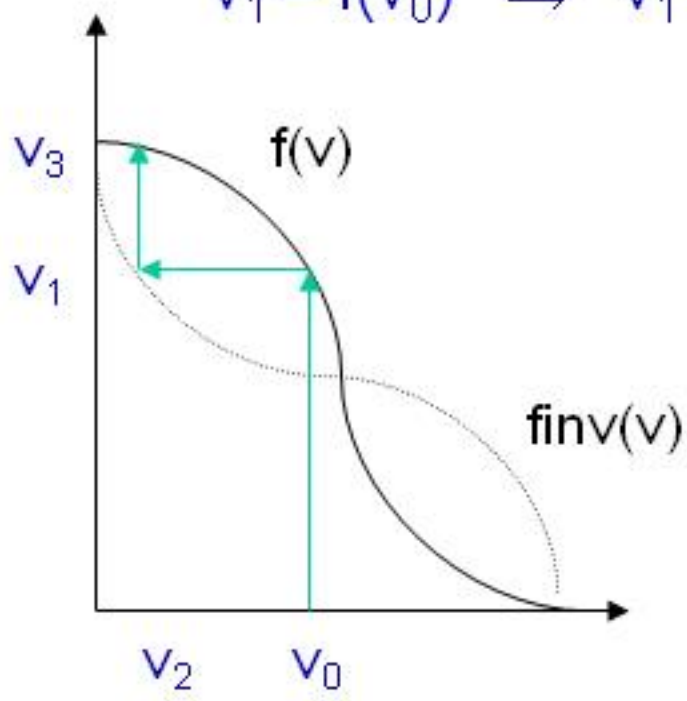




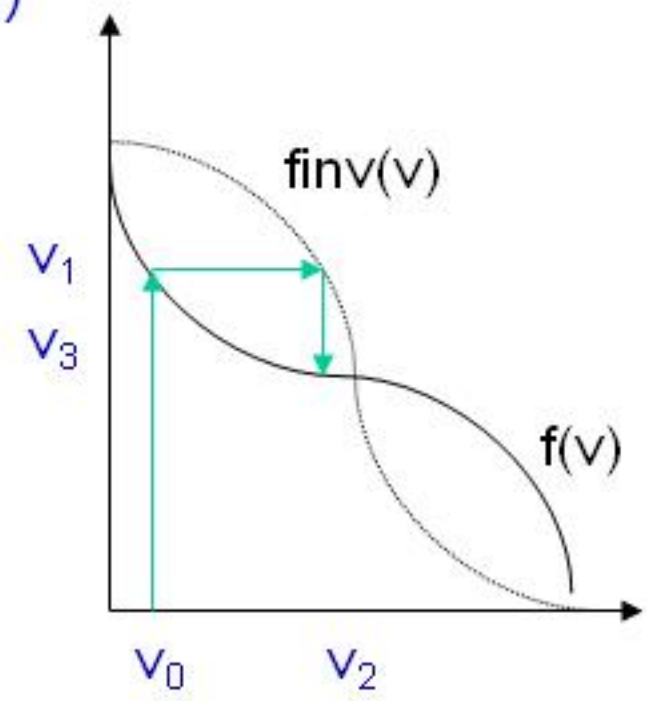
Podmínky regenerace



$v_1 = f(v_0) \Rightarrow v_1 = finv(v_2)$



Regenerativní hradlo



Neregenerativní hradlo

- Aby hradlo bylo regenerativní, musí mít převodní charakteristika zisk v přechodové oblasti větší než 1 (v absolutní hodnotě) a tato oblast musí být omezena dvěma zónami, kde zisk je menší než 1. Takové hradlo má pak dva stabilní operační body.

Šumová imunita

- „Šumové pásy“ vyjadřují schopnost obvodu „pohltit“ šumový signál, generovaný zdrojem šumu
  - zdroje šumu: šum zdroje, přeslechy, interference, ss posun (offset)
- Absolutní hranice pro šum jsou klamné
  - vodiče bez buzení jsou snáze rušeny než linky, buzené zdrojem o nízké impedanci (míníme napěťové rušení)

Šumová imunita vyjadřuje schopnost systému korektně přenášet a zpracovávat informace za přítomnosti šumu

- Pro správnou šumovou imunitu musí být signálový zdvih (rozdíl mezi V_{OH} a V_{OL}) a „šumové pásy“ dostatečně široké, aby překryly vliv šumu.

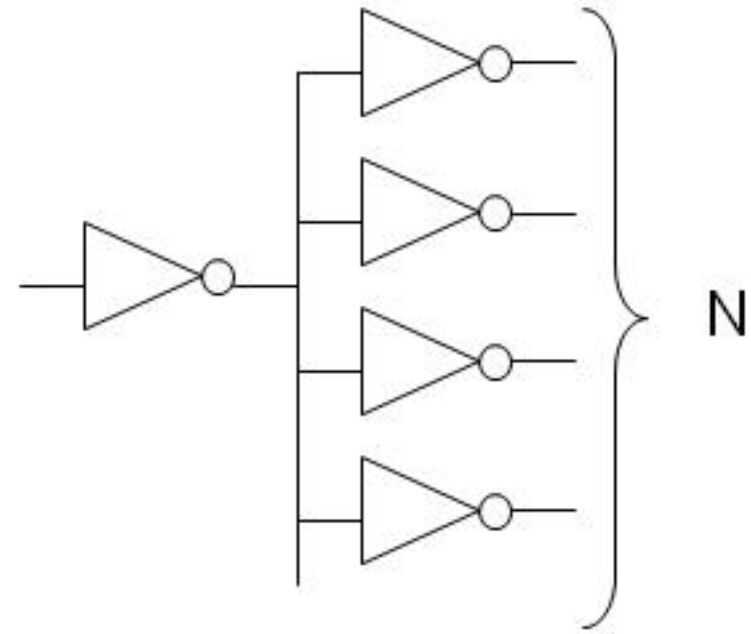
Směrovost

- 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)

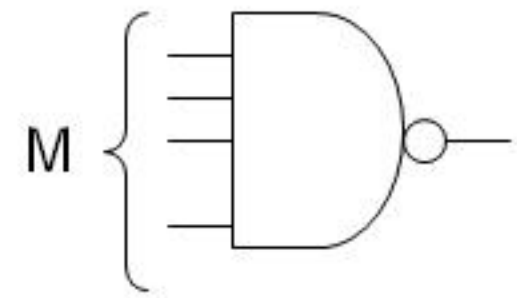
- Klíčové metriky: *výstupní impedance* budiče a *vstupní impedance* vstupu
  - ideálně, výstupní impedance budiče je nulová a
  - vstupní impedance vstupu je nekonečná

Logický zisk (Fan-In, Fan-Out)

- Fan-out – počet hradel připojitelných k výstupu hradla
  - hradla s velkým logickým ziskem jsou pomalejší

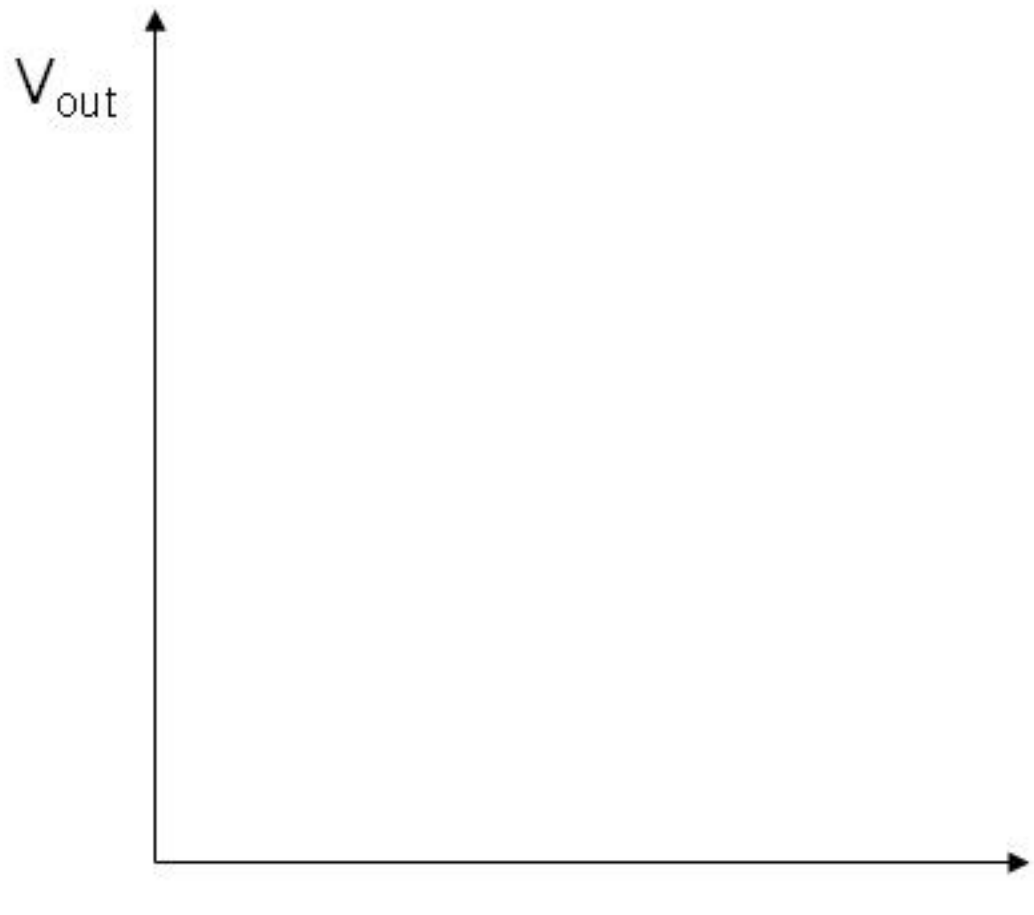


- Fan-in – počet vstupů hradla
  - hradla s velkým počtem vstupů jsou rozměrná a pomalejší



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



$R_i = \infty$

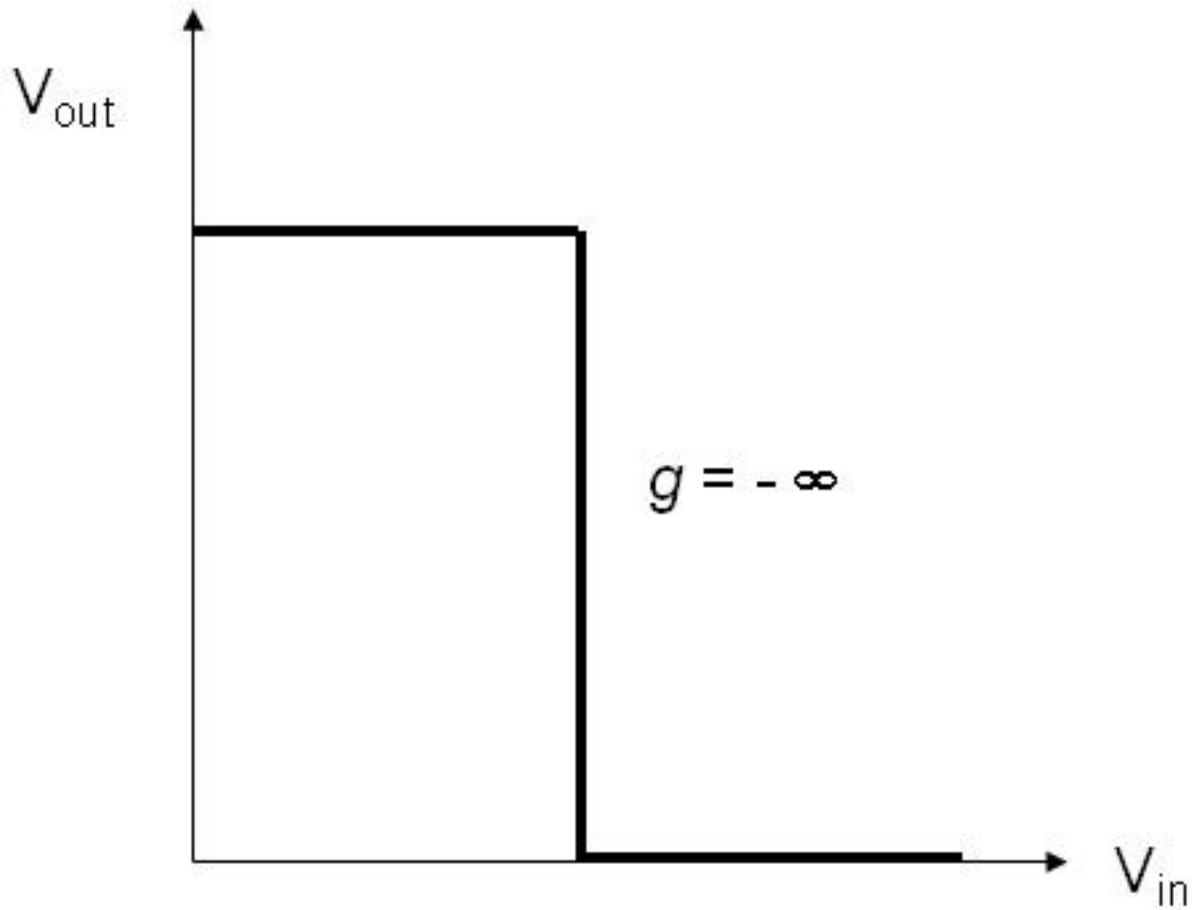
$R_o = 0$

Fanout = ∞

$NM_H = NM_L = VDD/2$

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



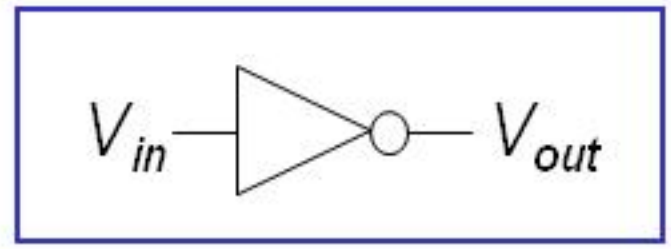
$R_i = \infty$

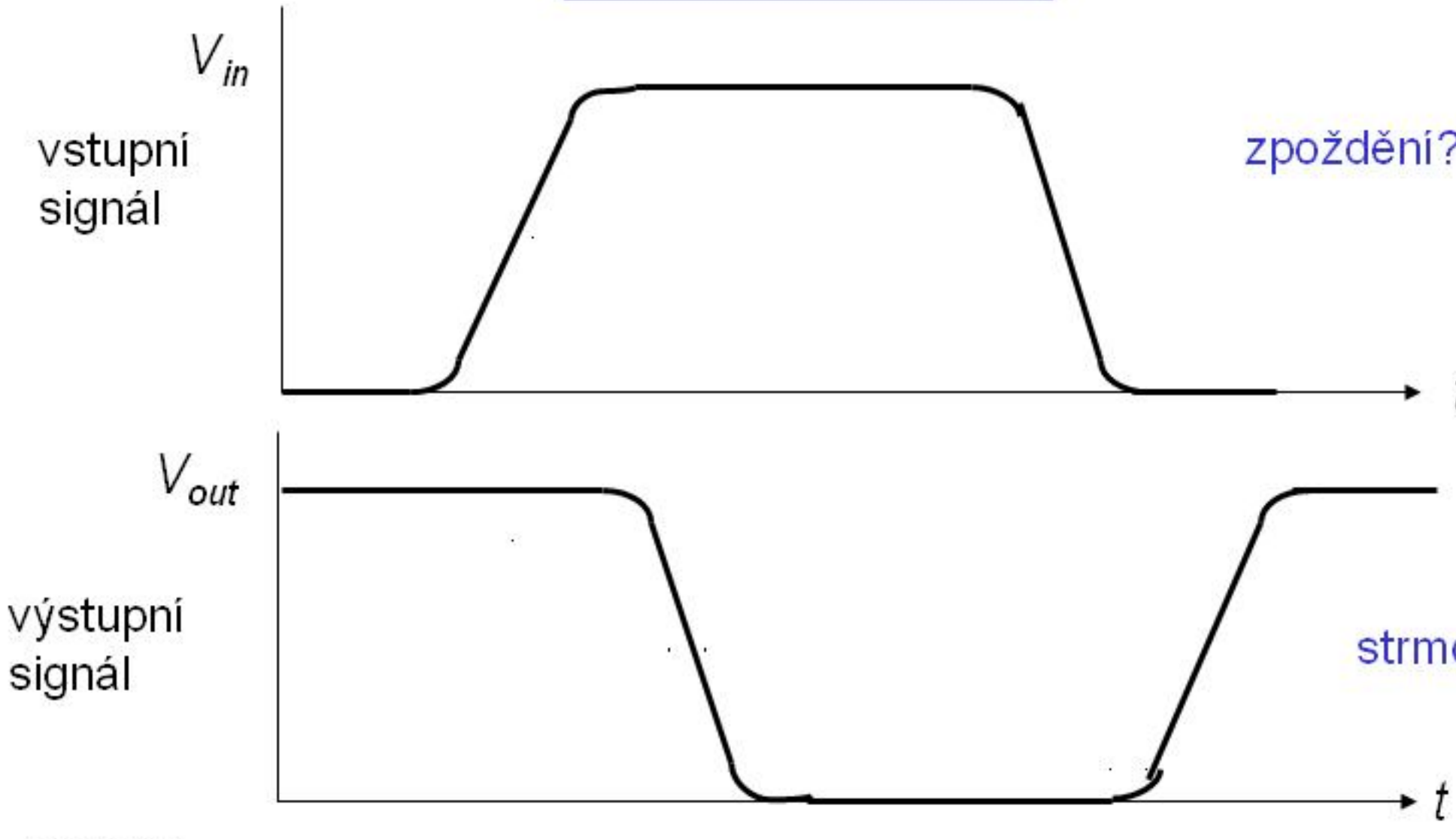
$R_o = 0$

Fanout = ∞

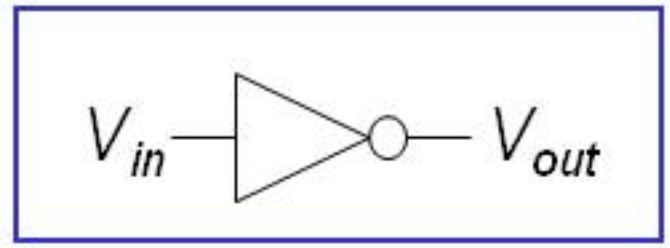
$NM_H = NM_L = VDD/2$

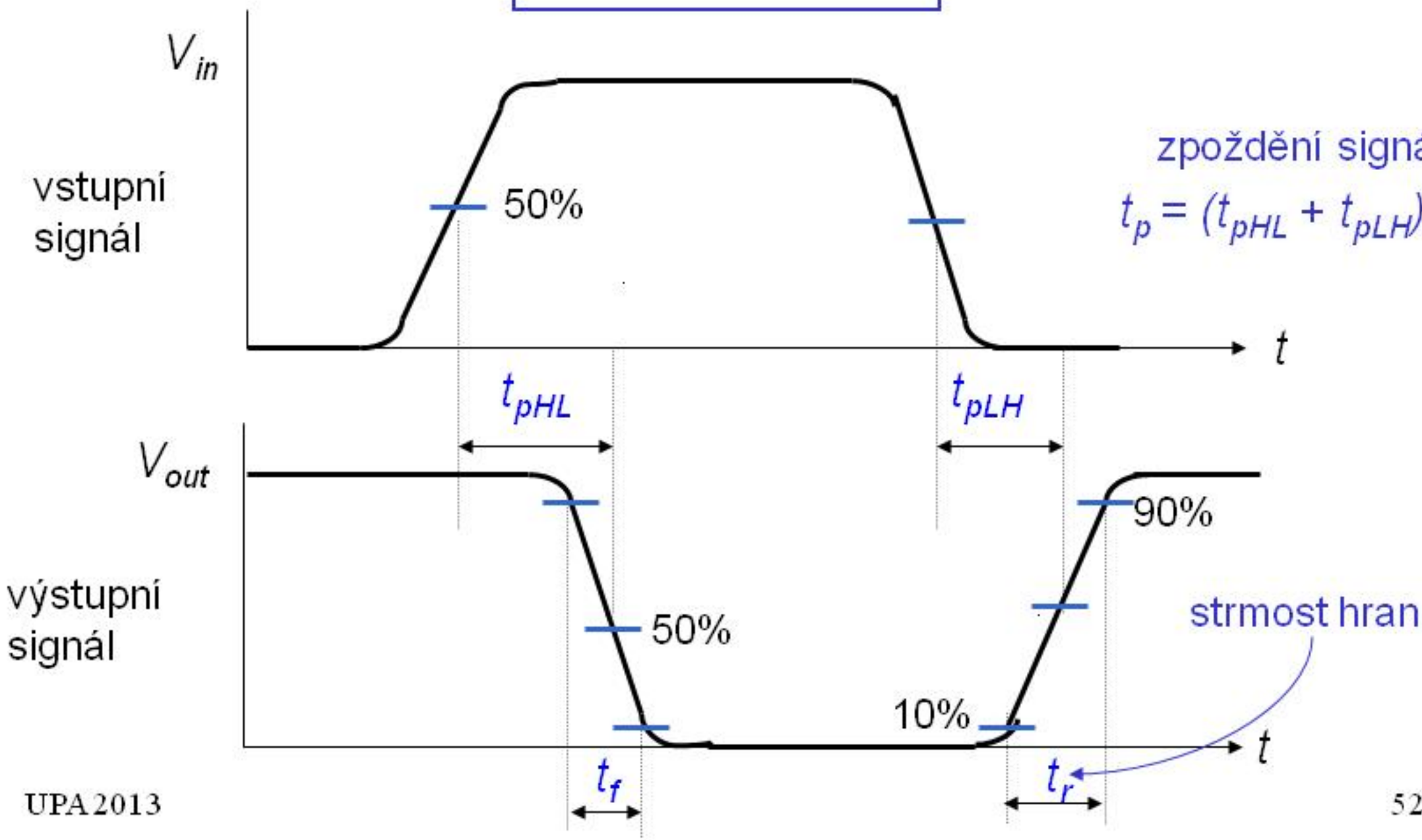
Definice zpoždění





Definice zpoždění



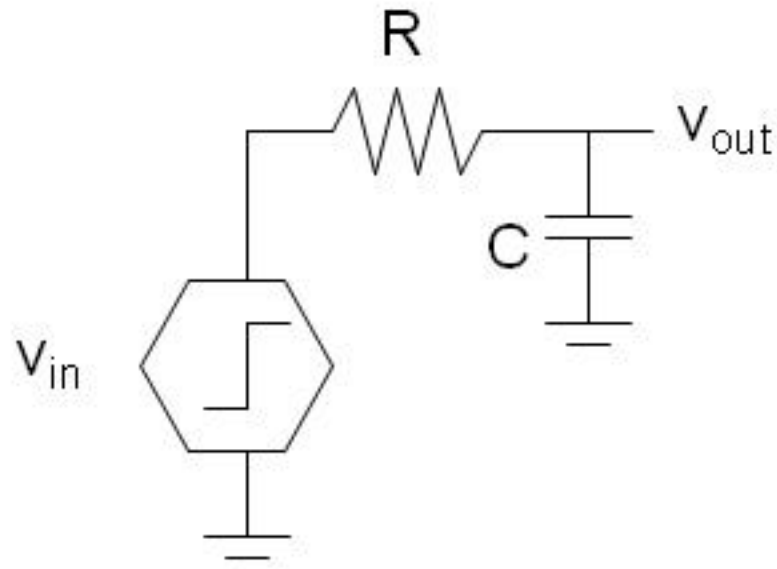


zpoždění signálu

$t_p = (t_{pHL} + t_{pLH})/2$

Model zpoždění

- Modelový obvod - RC člen 1. řádu



$$v_{out}(t) = (1 - e^{-t/\tau})V$$

kde $\tau = RC$

Doba potřebná pro dosažení 50% je rovna

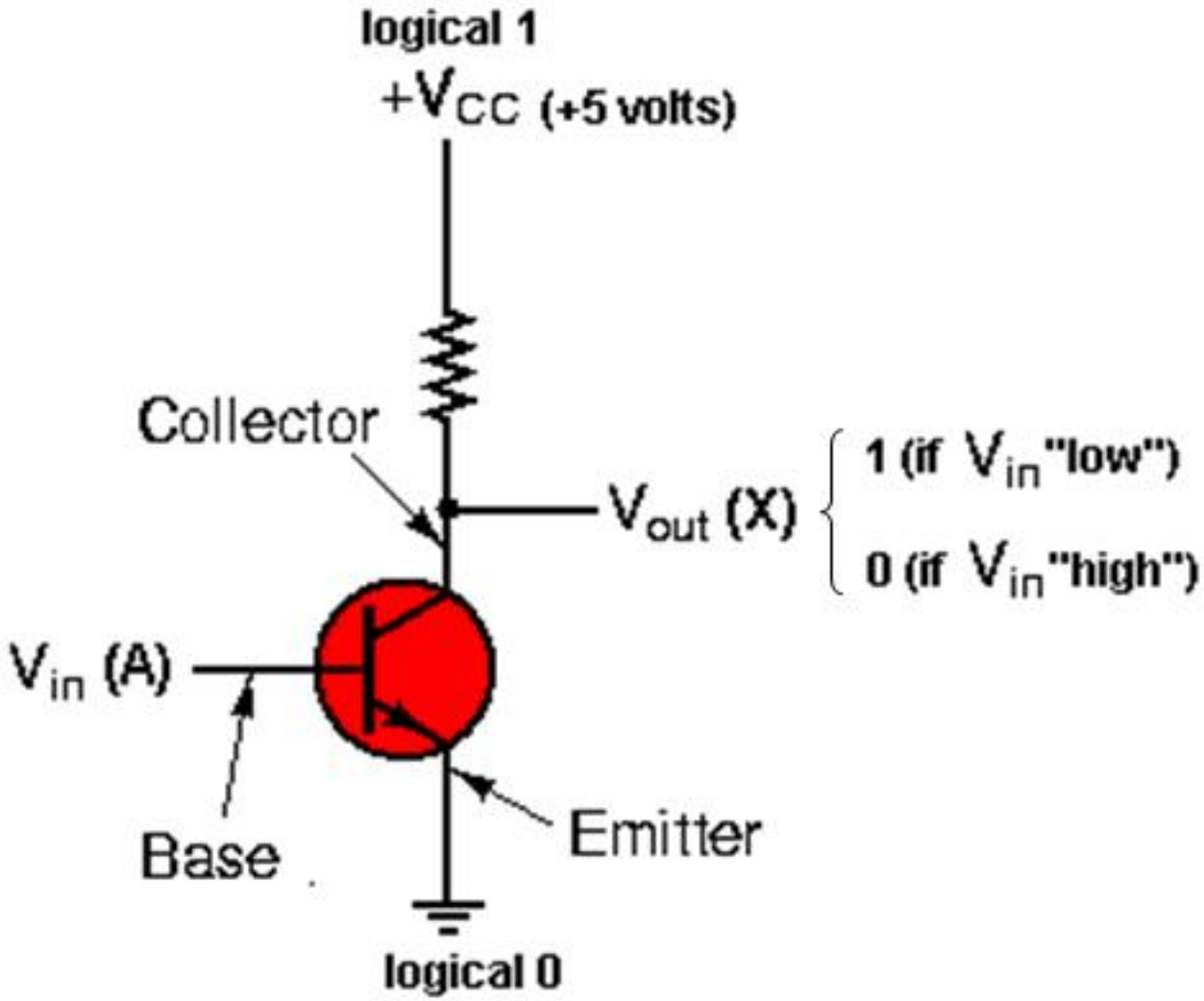
$$t = \ln(2)\,\tau = 0.69\,\tau$$

Doba potřebná pro dosažení 90% je rovna

$$t = \ln(9)\,\tau = 2.2\,\tau$$

- Odpovídá zpoždění hradla typu invertor

Tranzistory & Digitální logika



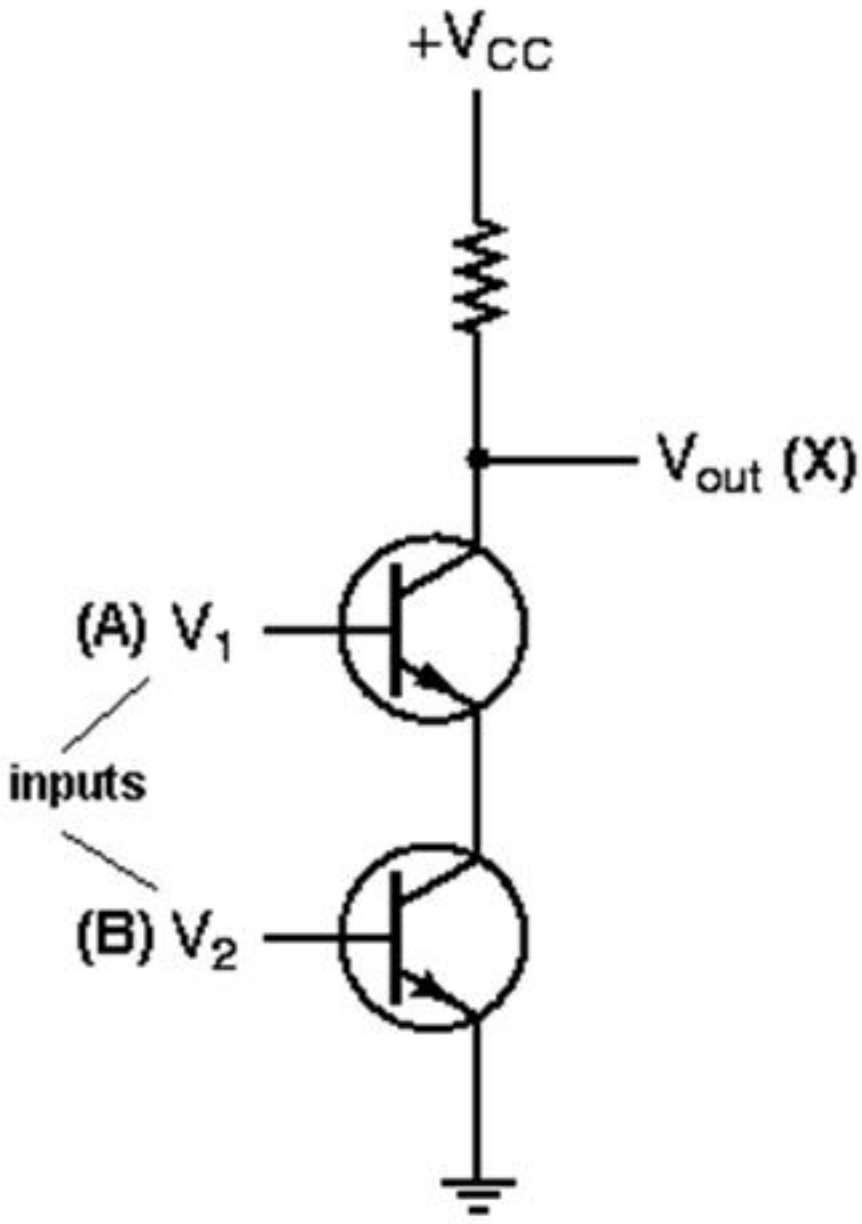
hradlo NOT
(Invertor)

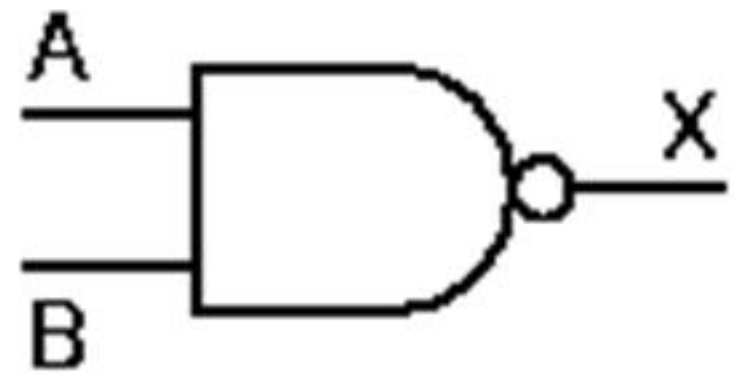
Symbol
hradla

A	X
0	1
1	0

Pravdivostní
tabulka
(funkční popis)

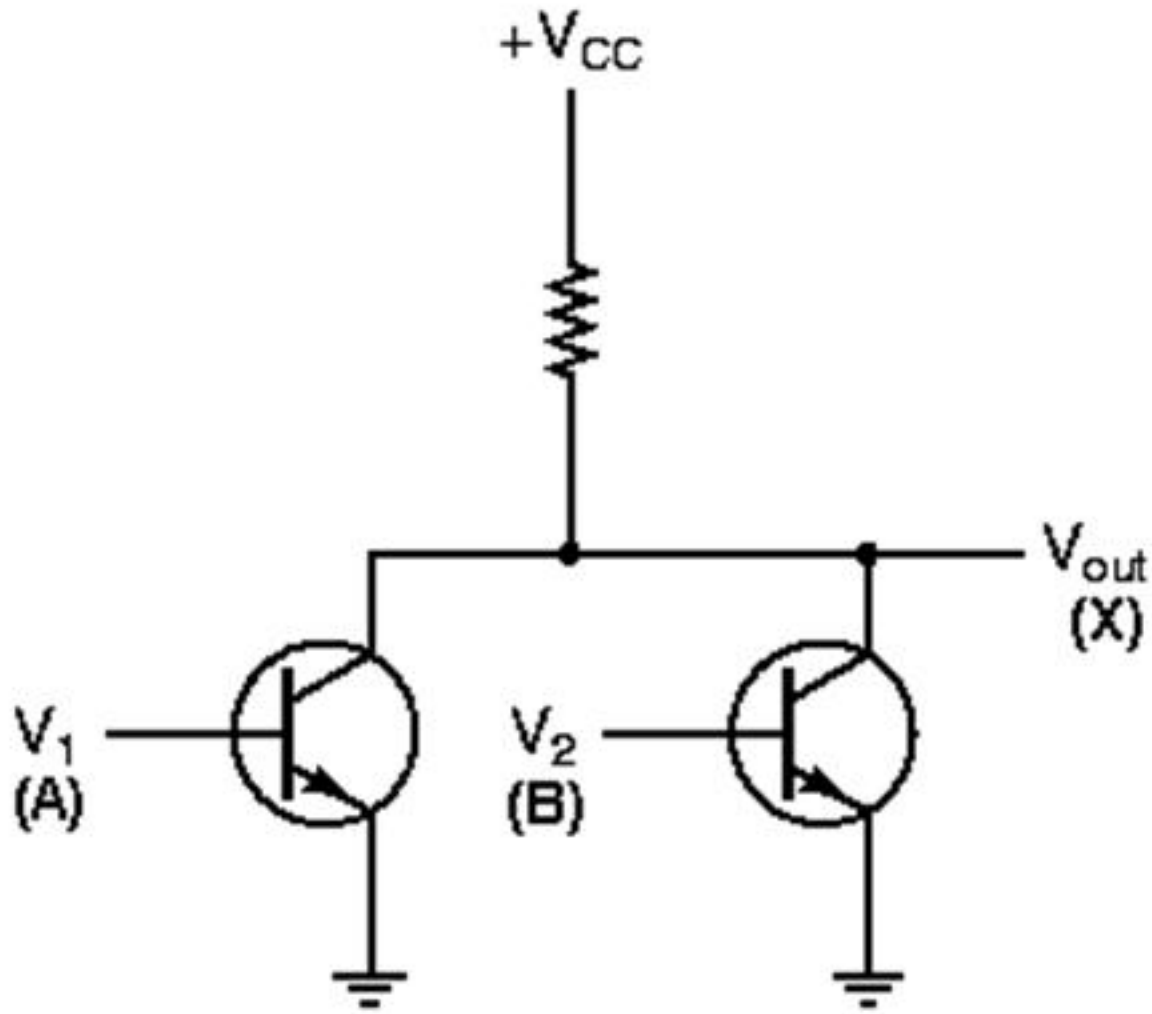
Hradlo NAND

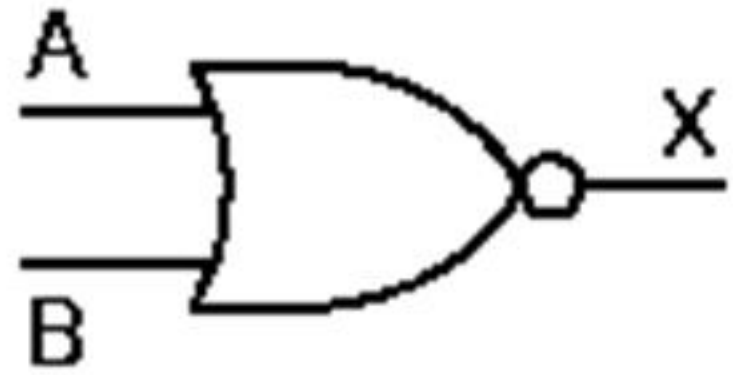




A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

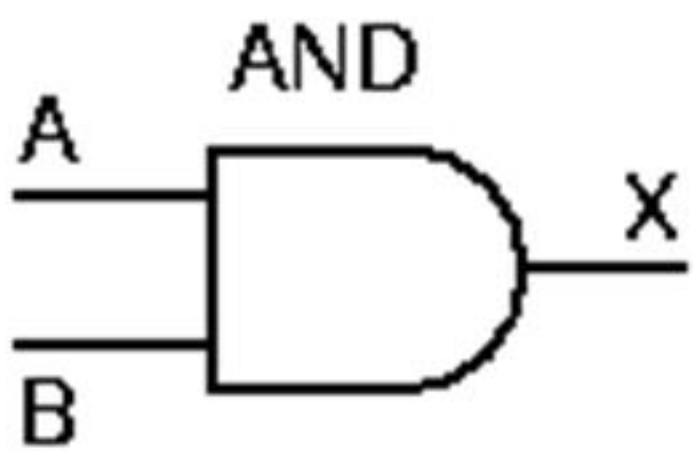
Hradlo NOR

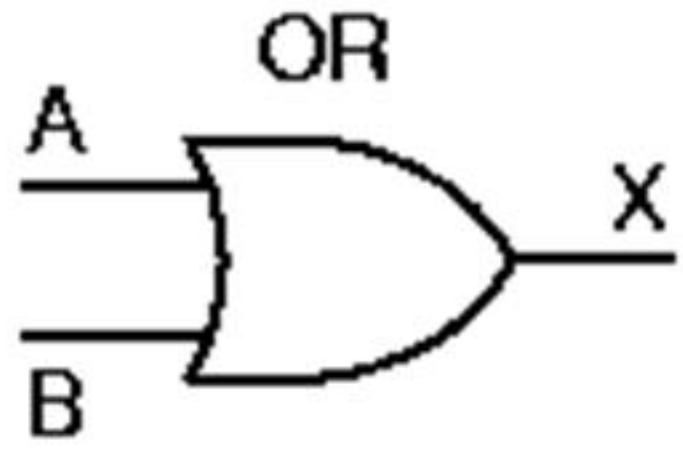




A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

Hradla AND & OR

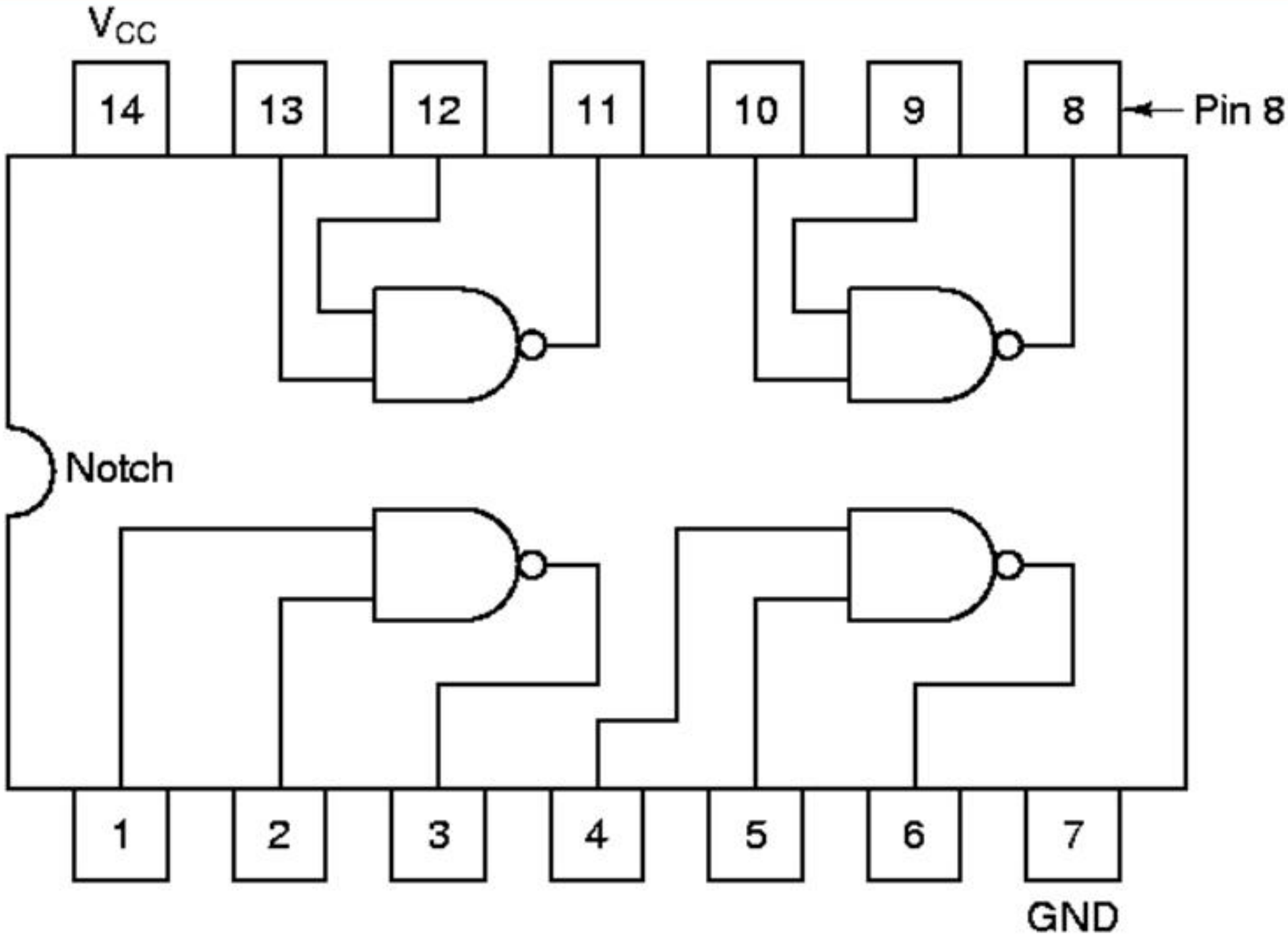




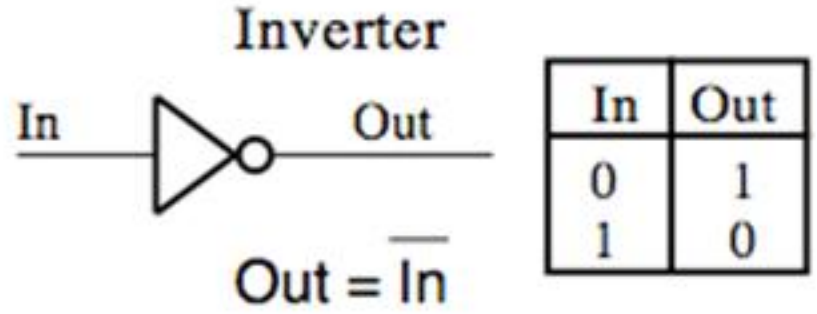
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

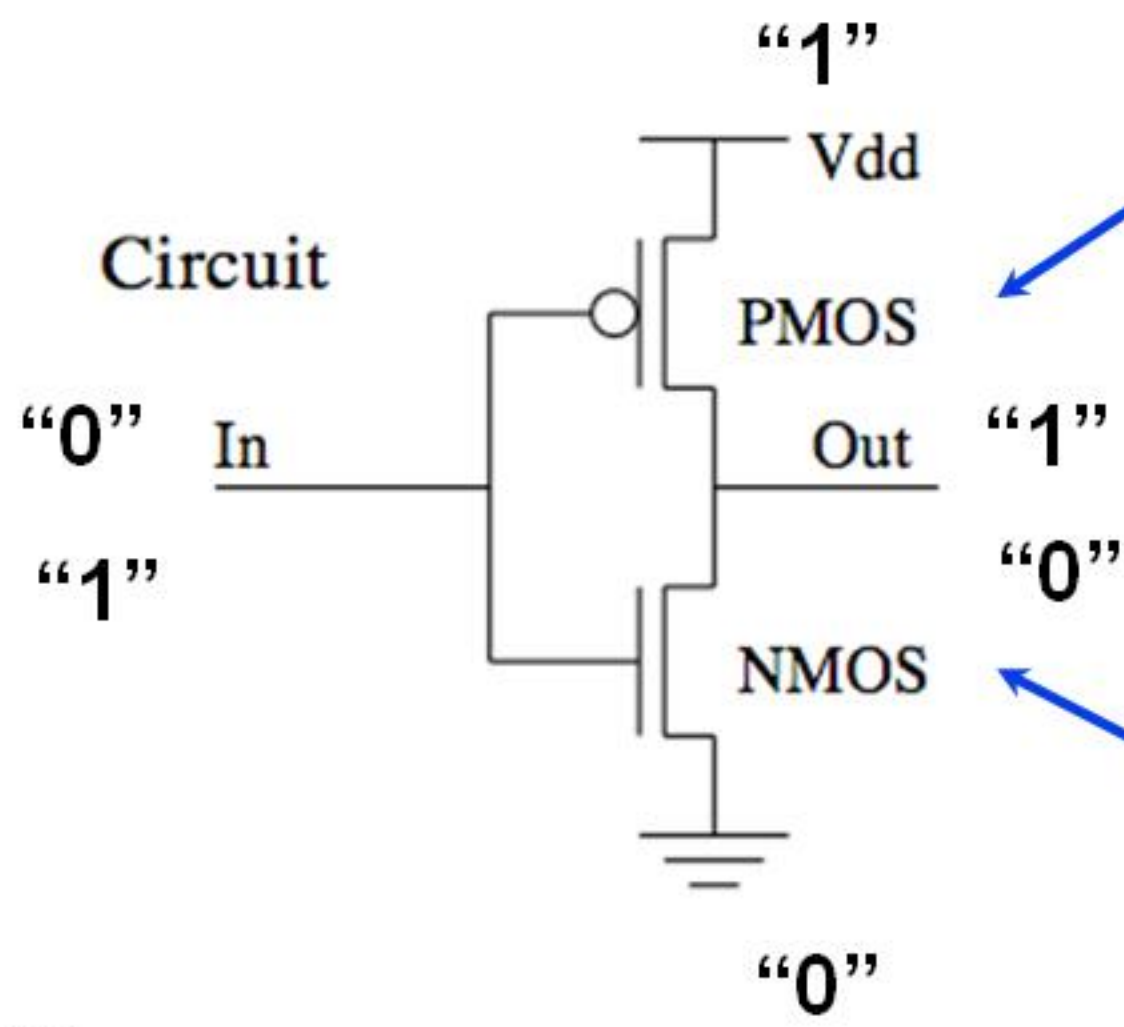
Integrované obvody



Invertory: Jednoduchý tranzistor. model



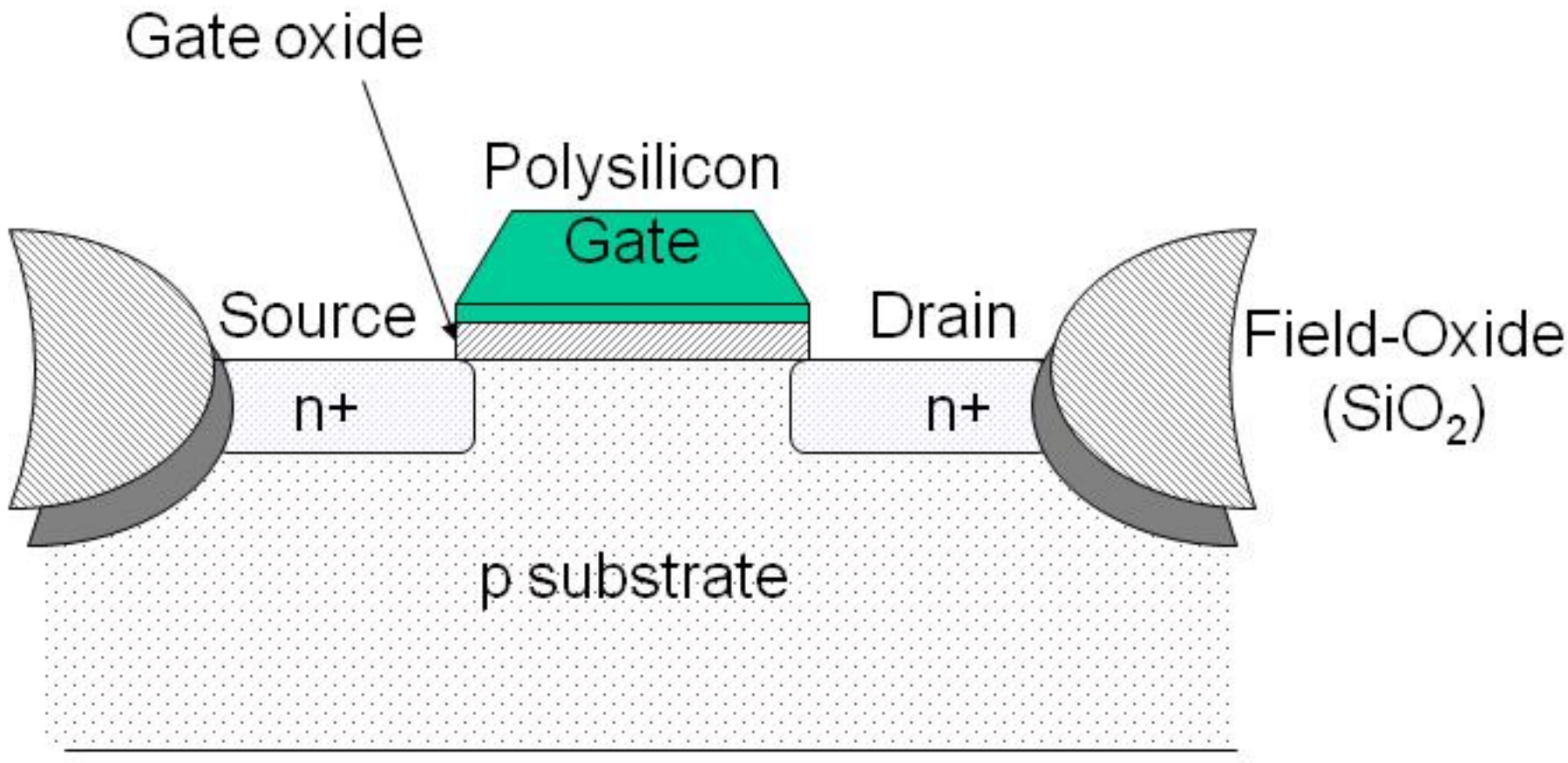
In	Out
0	1
1	0



pFET - spínač "On" je-li hradlo uzemněno.

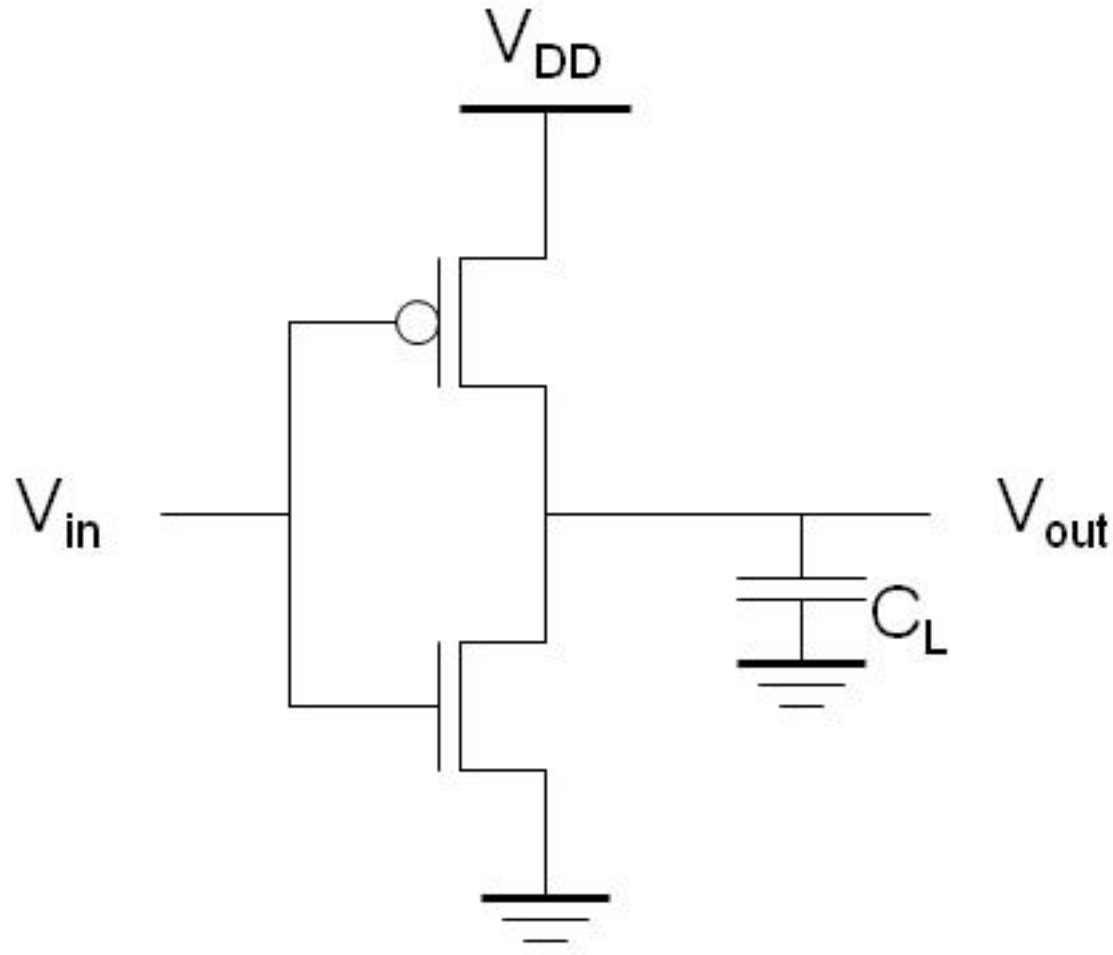
nFET - spínač "On" je-li hradlo na potenciálu Vdd.

MOS tranzistor



Řez NMOS tranzistorem

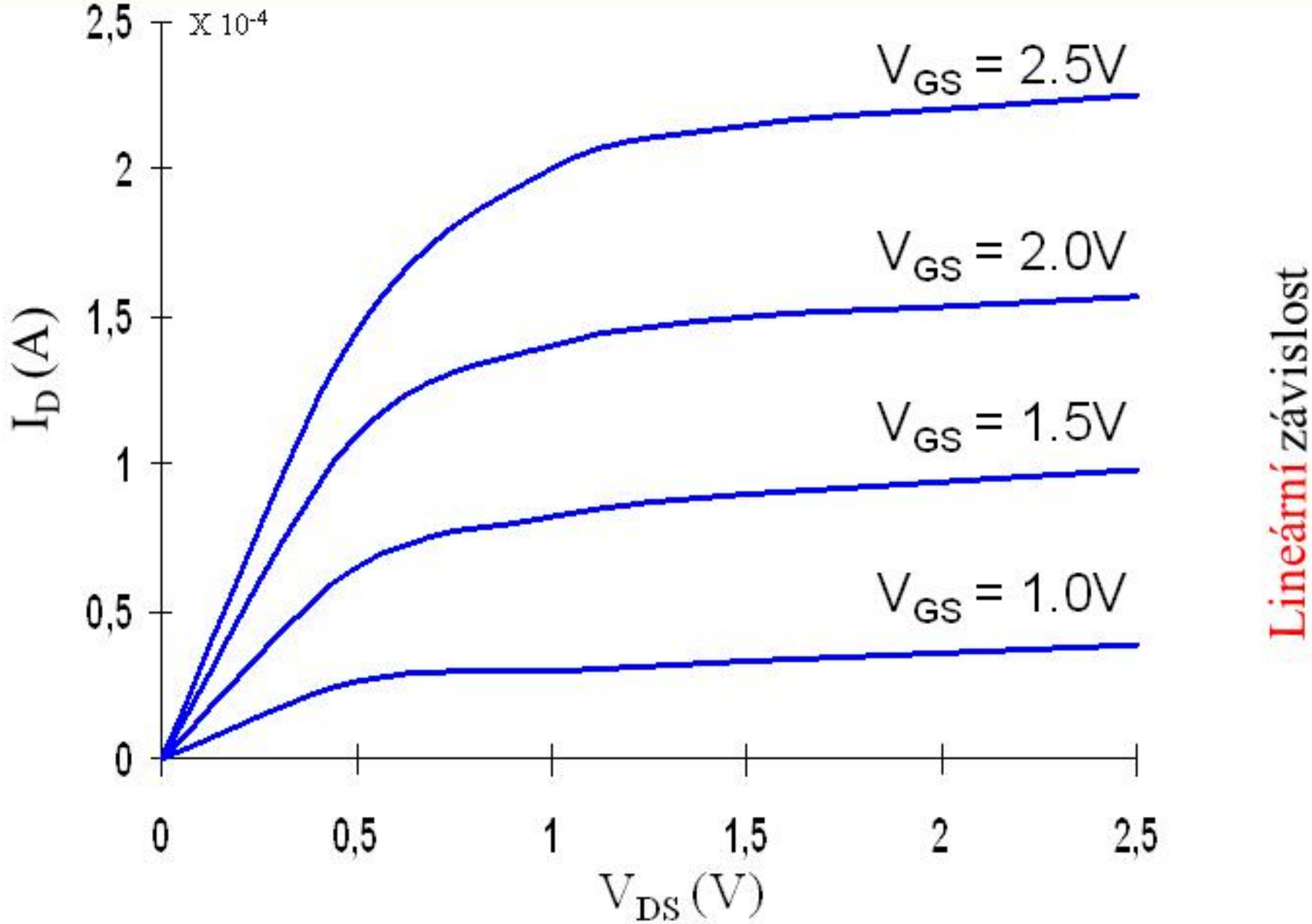
CMOS invertor



Vlastnosti CMOS tranzistoru

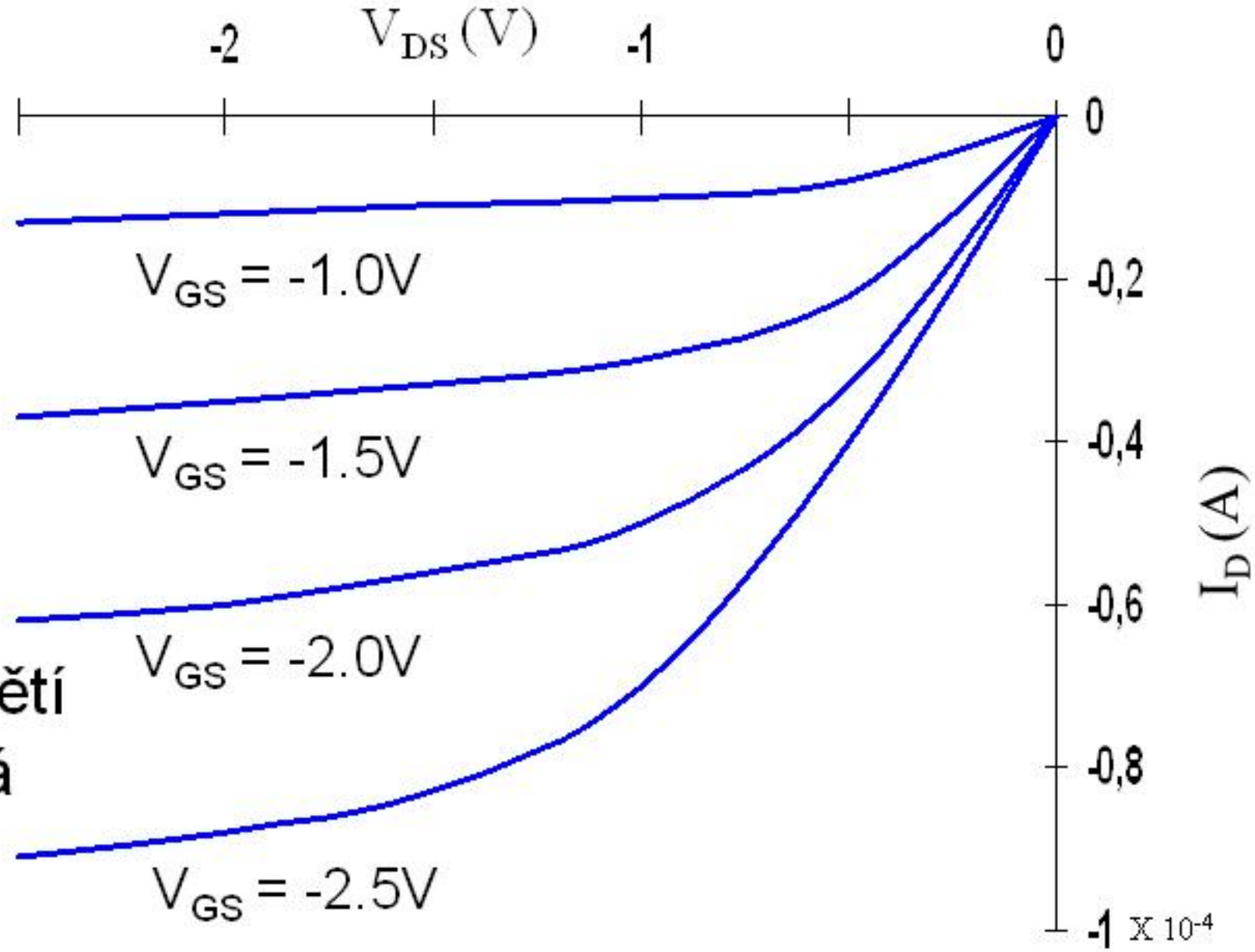
- Plný zdvih (rail-to-rail) ⟹ vysoké hranice šumu
  - Logické úrovně nezávisí na relativní velikosti prvku ⟹ tranzistory mohou mít minimální rozměry ⟹ ratioless
- Jedna z cest k V_{dd} nebo ke GND je otevřena ⟹ nízká výstupní impedance (v řádu kΩ) ⟹ velký „fan-out“ (i když to vede k degradaci výkonu)
- Extrémně vysoká vstupní impedance (gate MOS tranzistoru je téměř perfektní izolátor) ⟹ téměř nulový vstupní proud v ustáleném stavu
- Žádná přímá cesta mezi napájením a zemí v ustáleném stavu ⟹ nulová statická výkonová ztráta
- Zpoždění je funkcí kapacity zátěže a odporu tranzistorů

Charakteristika NMOS tranzistoru



NMOS tranzistor, 0.25um, L_d = 0.25um, W/L = 1.5, V_{DD} = 2.5V, V_T = 0.4V

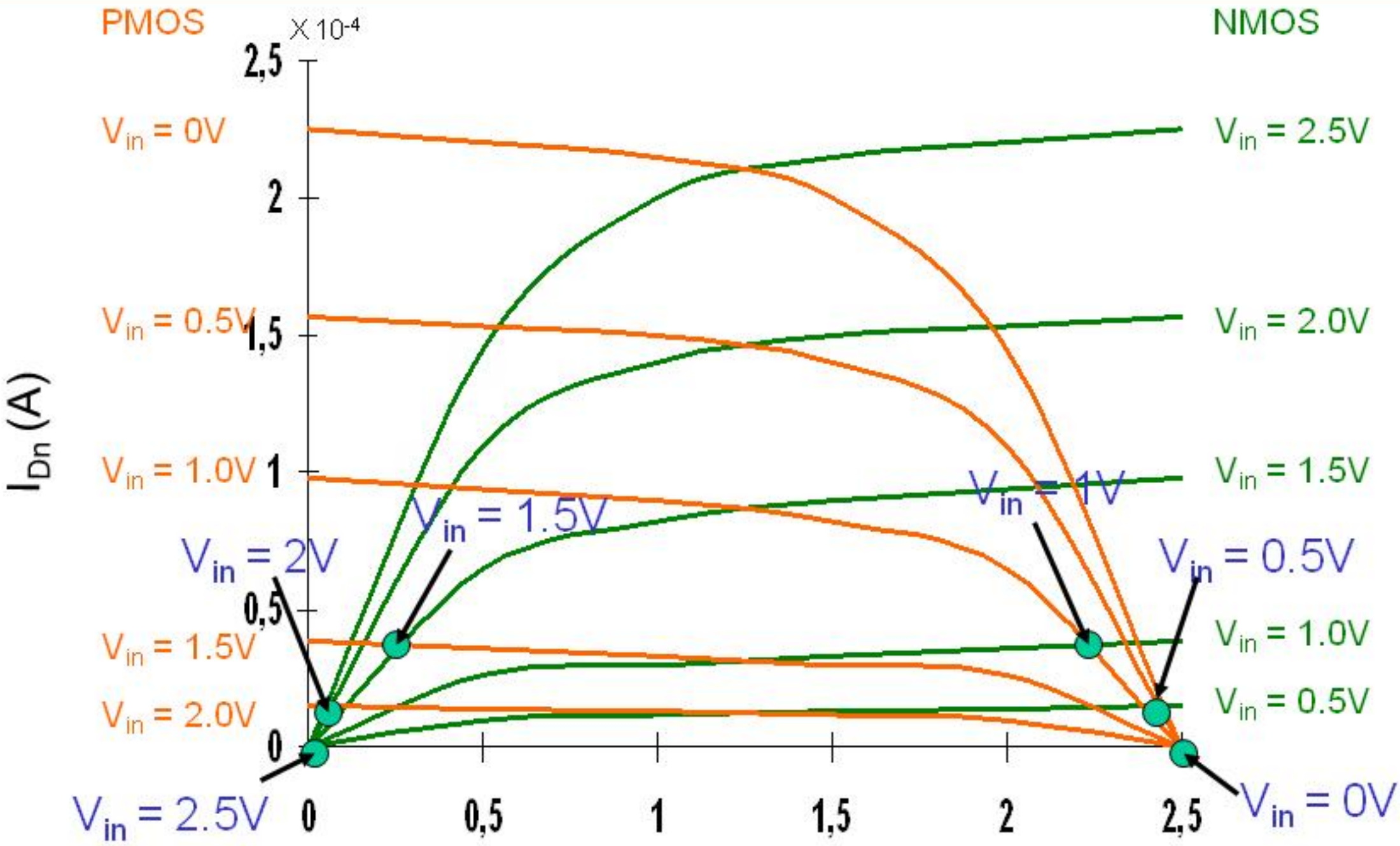
Charakteristika PMOS tranzistoru



Polarita všech napětí
a proudů je opačná

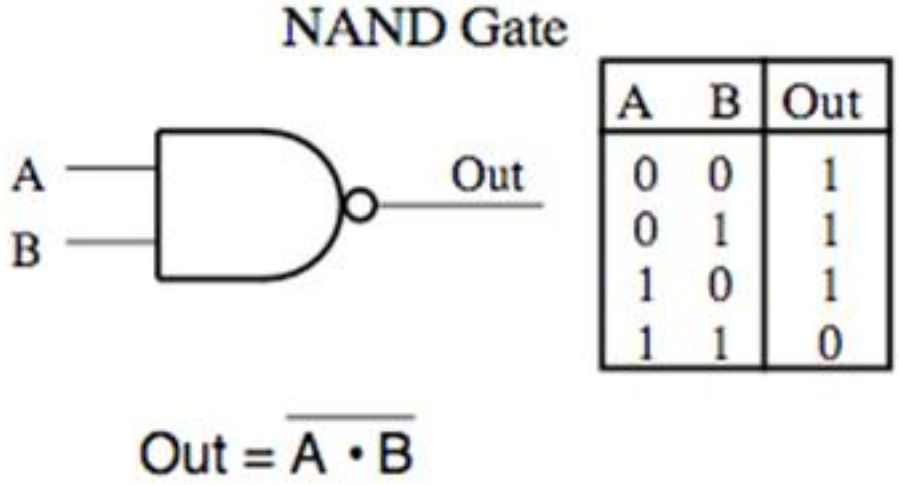
PMOS tranzistor, 0.25um, L_d = 0.25um, W/L = 1.5, V_{DD} = 2.5V, V_T = -0.4V

Zatěžovací charakteristiky invertoru

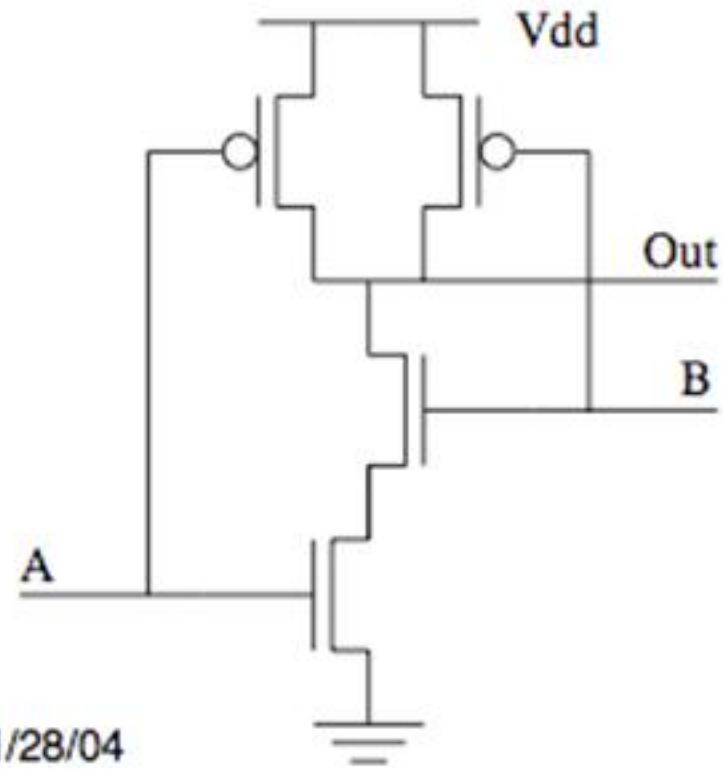


0.25um, W/L_n = 1.5, W/L_p = 4.5, V_{DD} = 2.5V, V_{Tn} = 0.4V, V_{Tp} = -0.4V

Hradla v technologii CMOS



Malý počet výkonných logických obvodů.



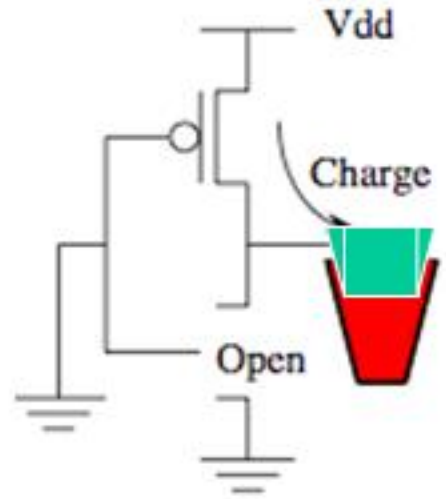
Lze postavit celý procesor pouze z hradel NAND ?

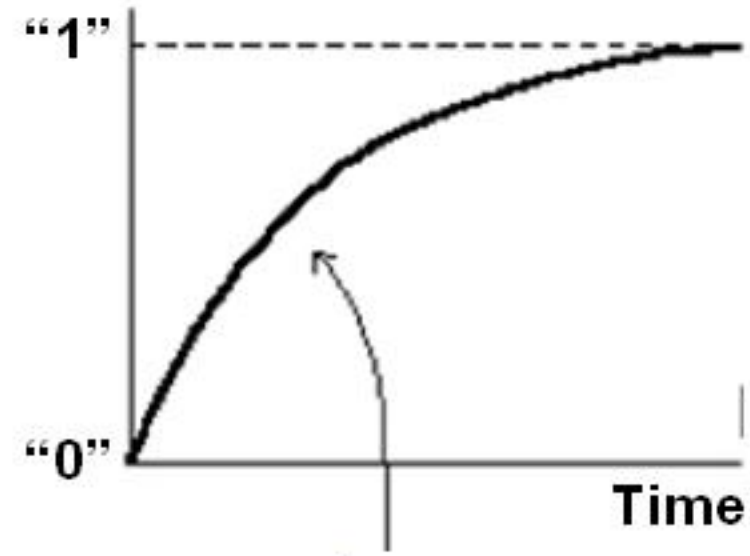
1/28/04 ©UCB

Tranzistory – analogie s „vodními vlnami“

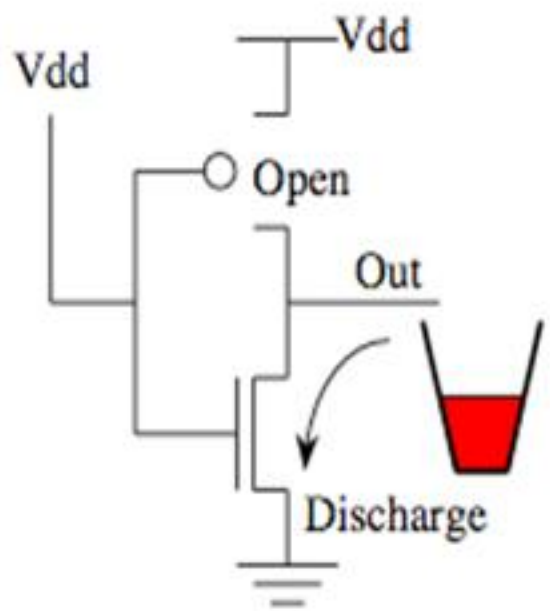
Elektrony jako molekuly vody, kondenzátor jako nádoba ...

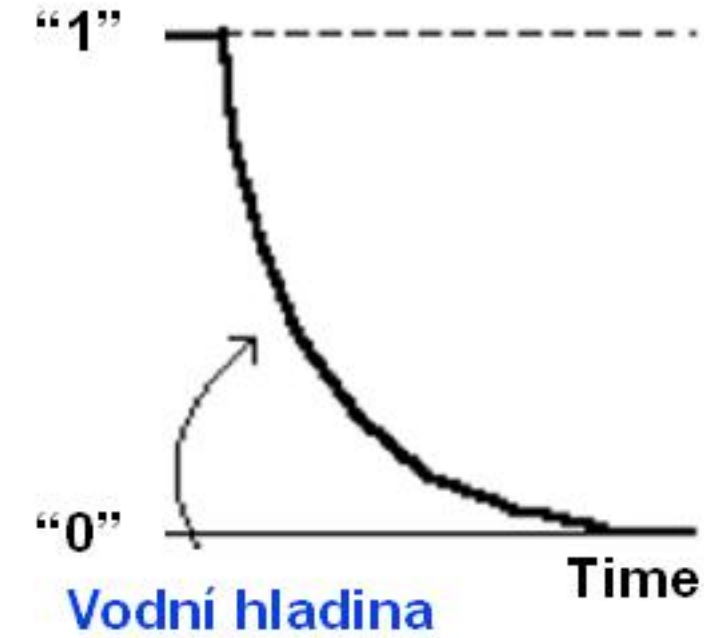
Otevřený p-FET plní kondenzátor nábojem





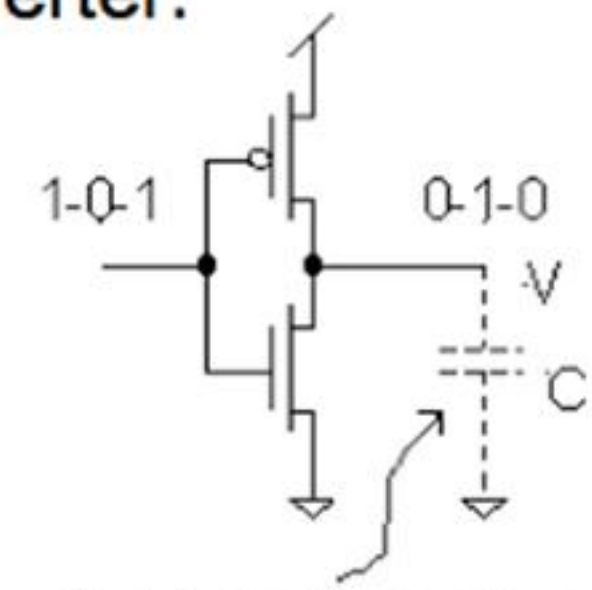
Otevřený n-FET vyprazdňuje nádobu

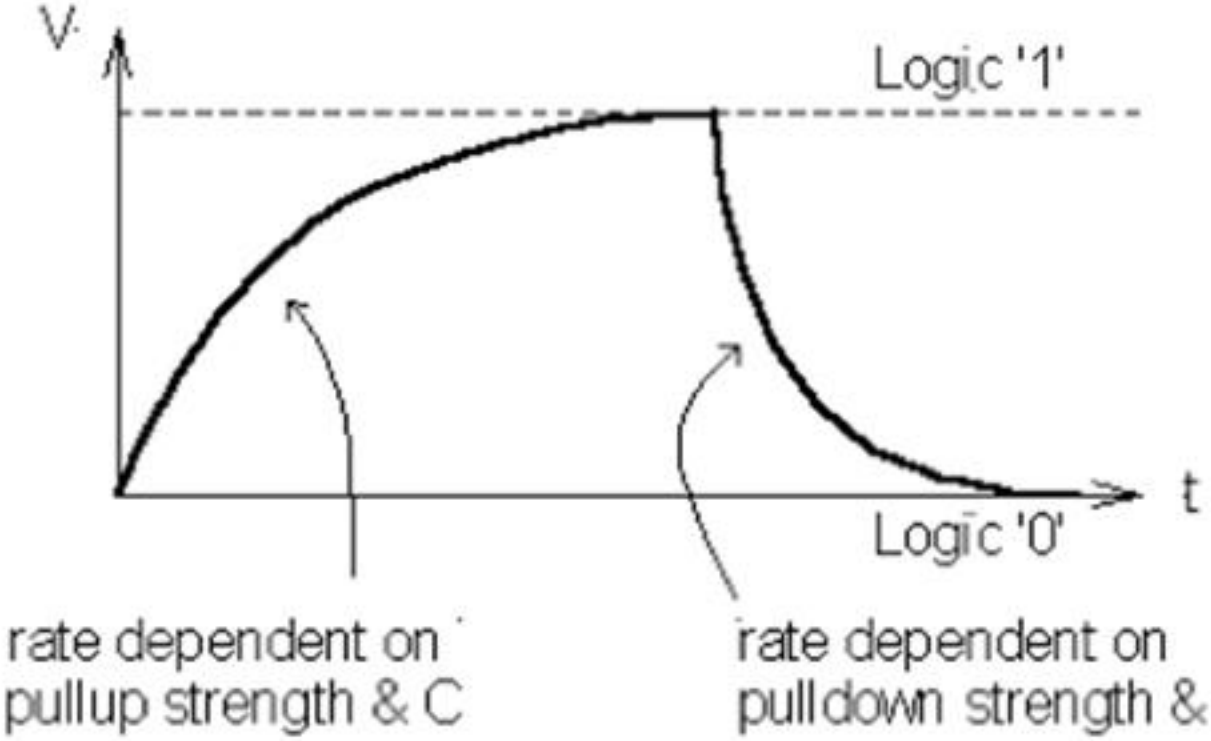




Logický zisk

Inverter:

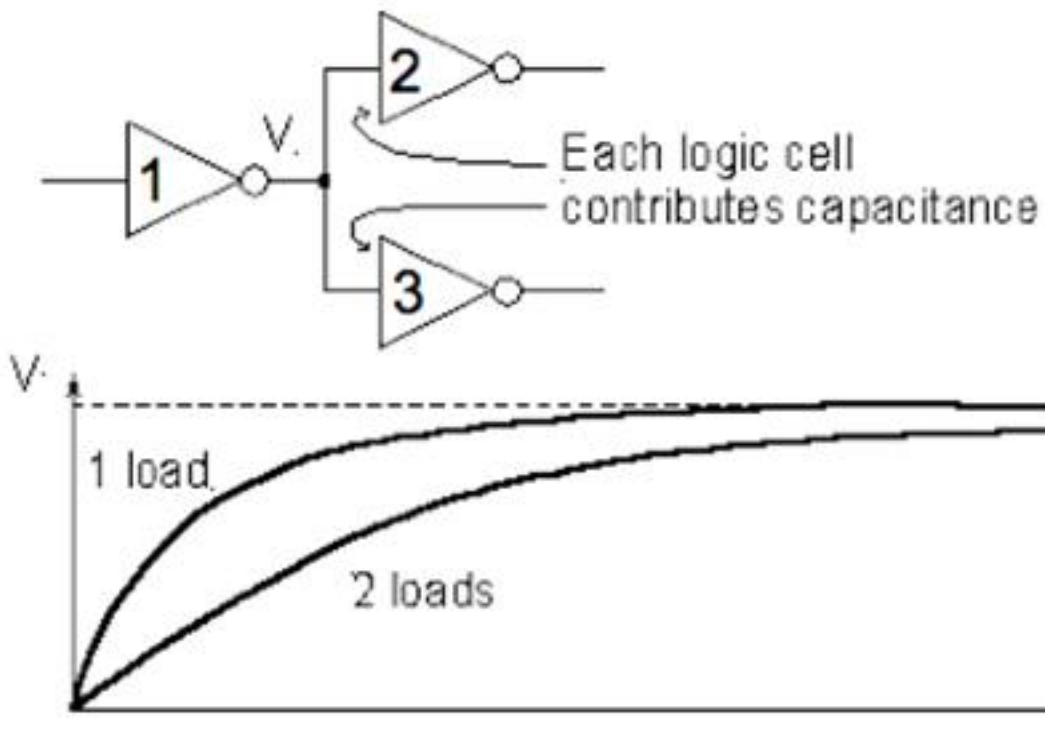




"Logický zisk": Počet vstupů hradel buzených výstupem hradla.

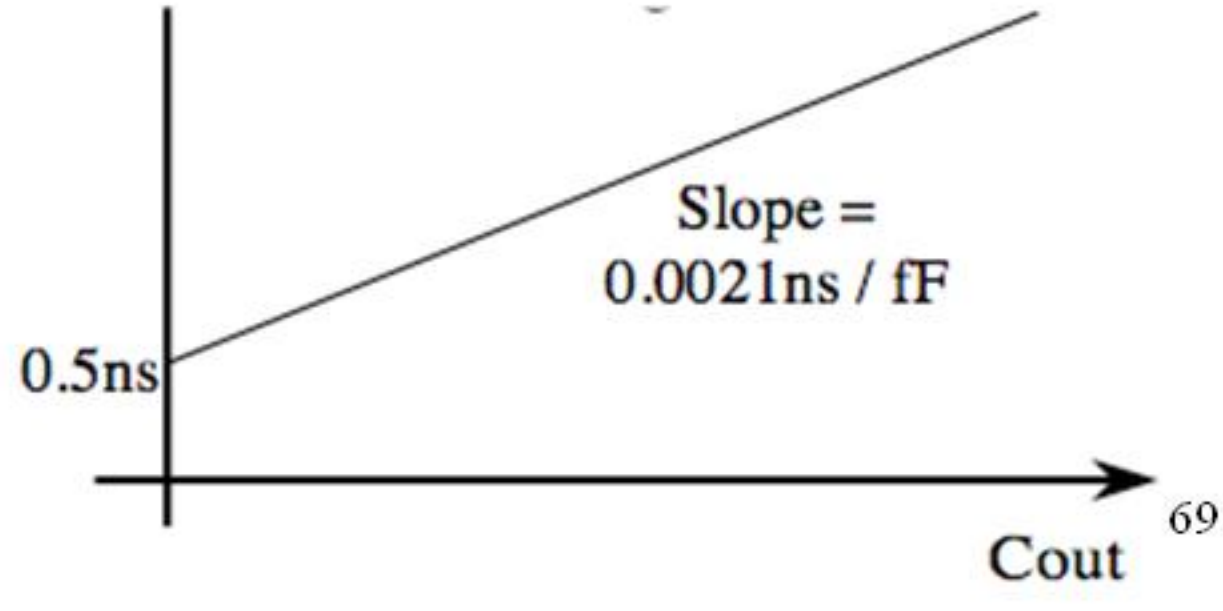
Buzení více hradel zpomaluje přechod signálu. Buzení vodičů zpomaluje přechod signálu.

Podrobnější náhled



Buzením více hradel roste zpoždění.

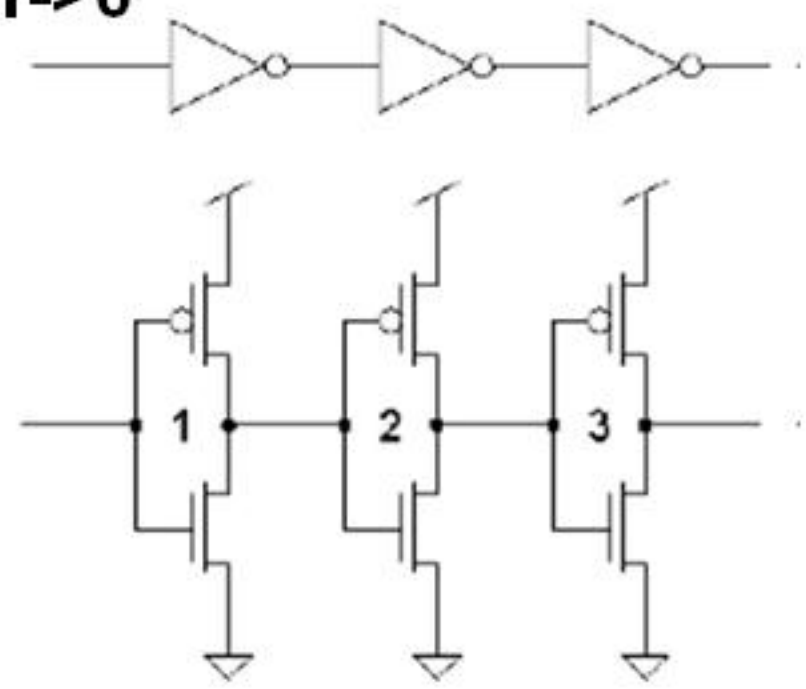
Lineární model dává dobré výsledky

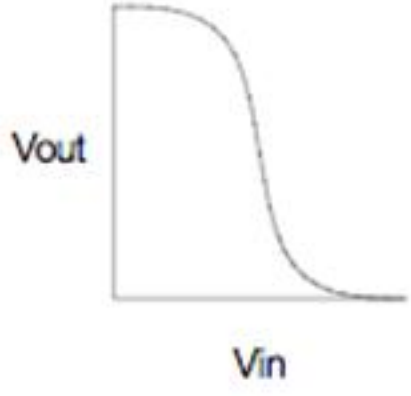


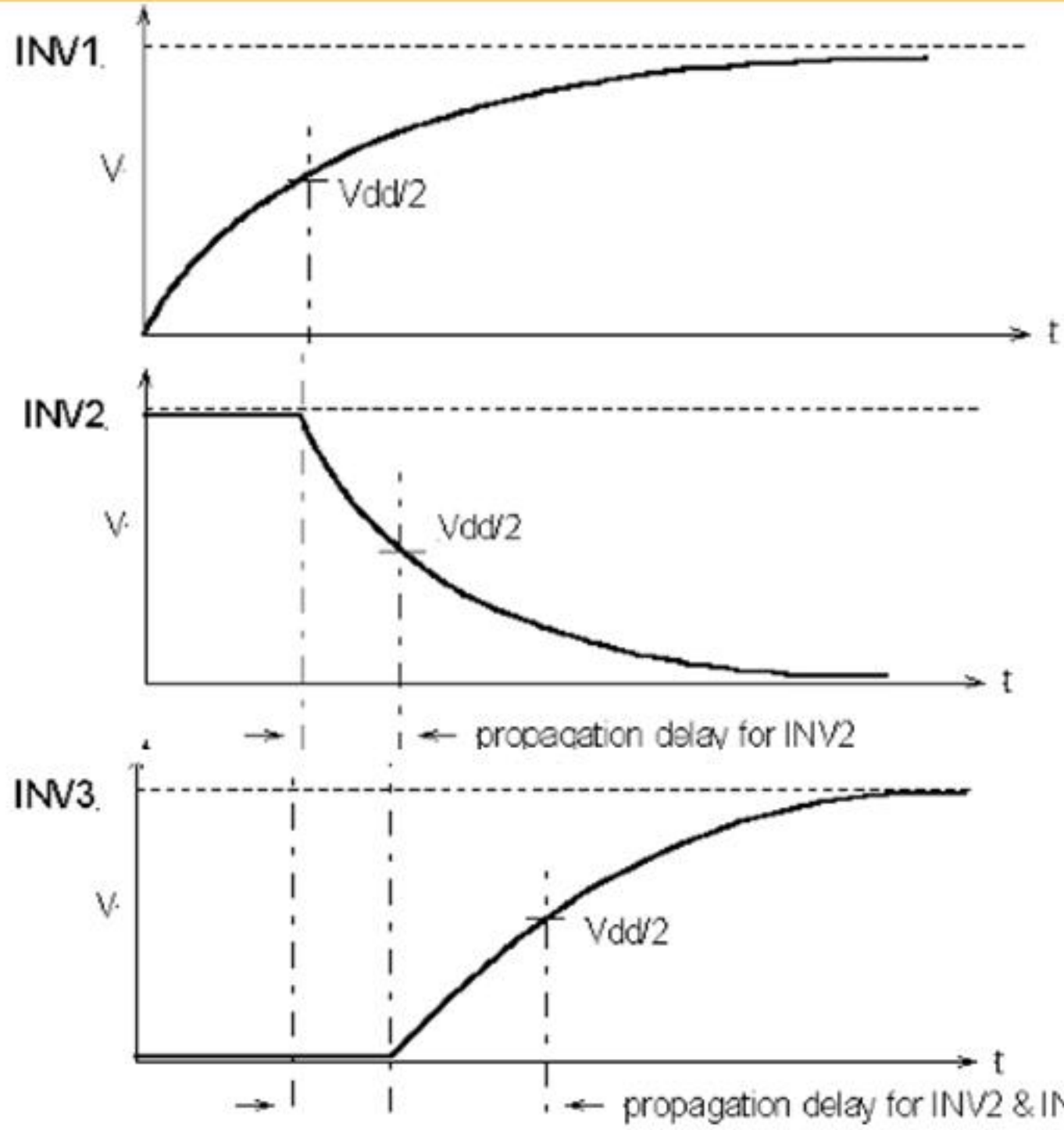
Grafy zpoždění signálu

- Cascaded gates:

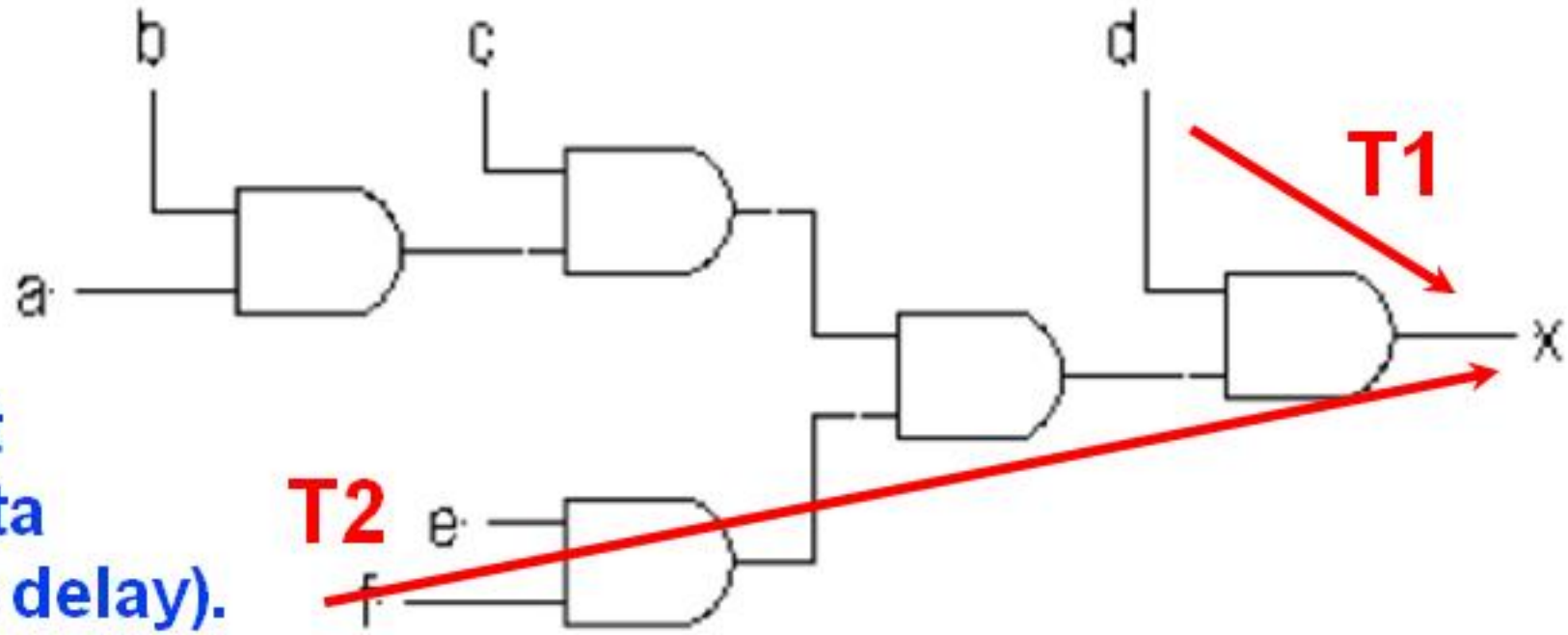
1->0







Intuice: Kritické cesty ...



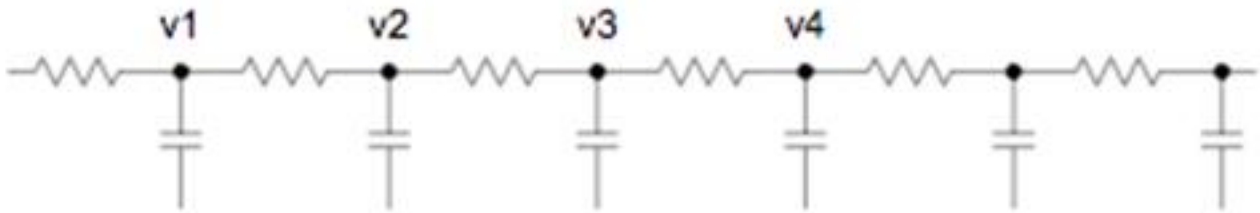
T2 může být kritická cesta (worst-case delay).

$$x = g(a, b, c, d, e, f)$$

Přechod d 0->1 přepne x 0->1, zpoždění je T1.
Přechod a 0->1 přepne x 0->1, zpoždění je T2.

Proč? Vodiče také vykazují zpoždění

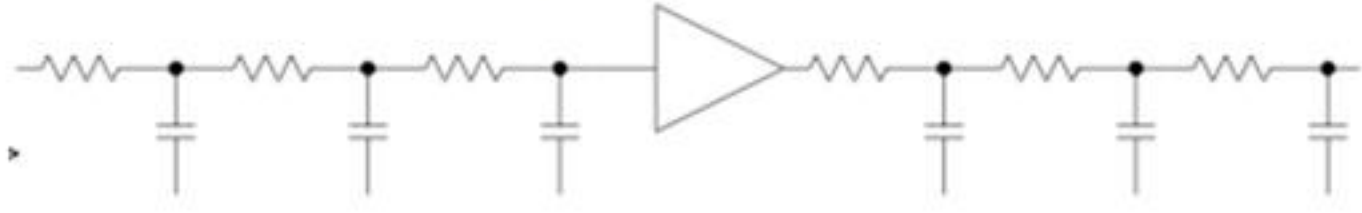
– Wires posses distributed resistance and capacitance



Vypadá neškodně, ale ...

– Time constant associated with distributed RC is proportional to the *square* of the length

– signals are typically "rebuffered" to reduce delay:



Operace Booleovy algebry

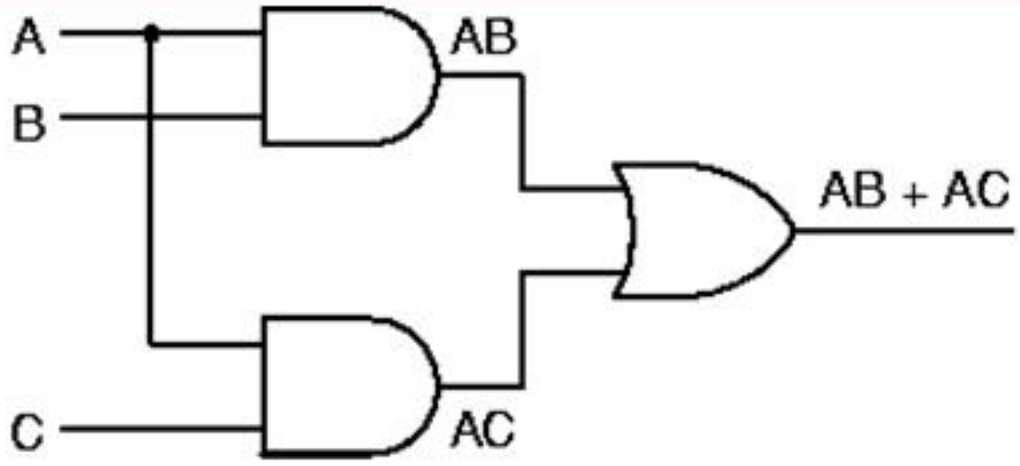
- 0 & 1: jediné hodnoty pro proměnné a funkce
  B = {0,1} se nazývají *Booleovská čísla*
- Funkce NOT : $f(A) = \begin{cases} 1 \text{ platí-li } A = 0 \\ 0 \text{ platí-li } A = 1 \end{cases}$
- Pravdivostní tabulky
  - Úplně definují Booleovskou funkci
  - n proměnných => 2^n řádek v pravdivostní tabulce => 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

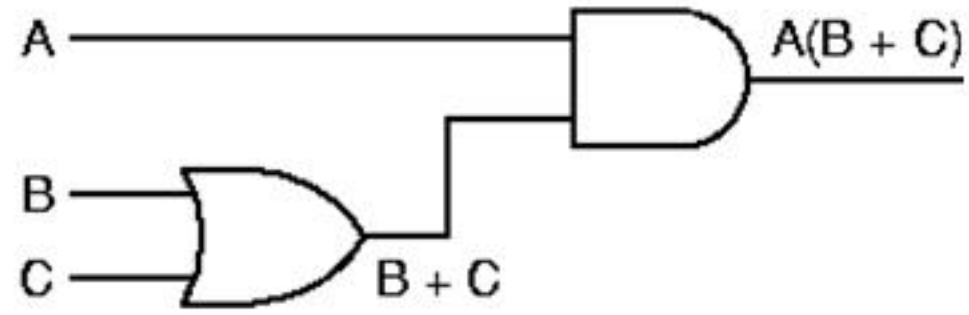
Booleova Algebra

- 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	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$

Ekvivalence obvodů





A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Kombinační logika

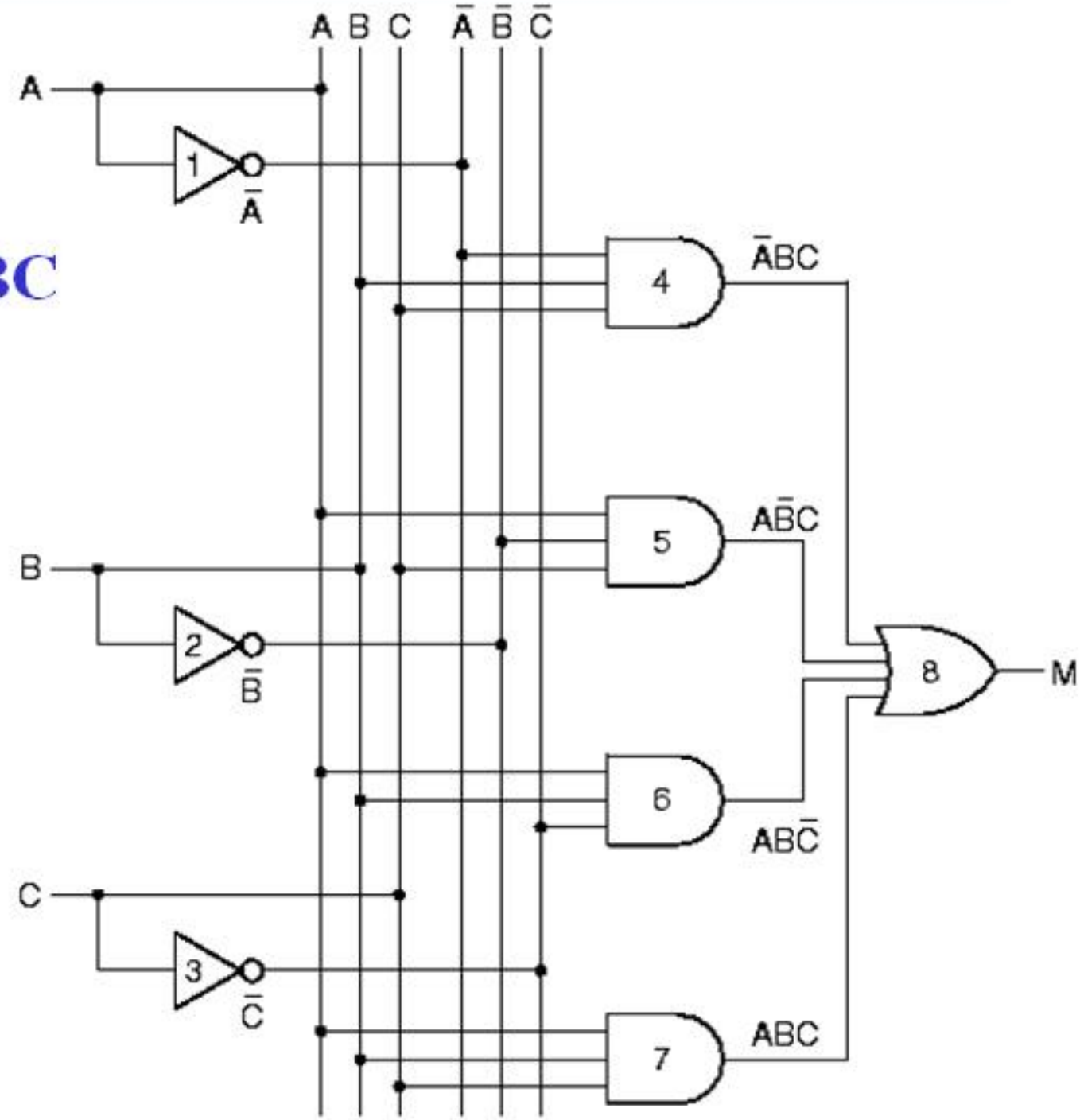
- Mnoho vstupů a mnoho výstupů
- Výstupy jsou jednoznačně určeny vstupy
- Absence paměťových prvků
- Neobsahuje zpětné vazby (skrytý paměťový prvek)
- Základní kombinační obvody
  - Multiplexery
  - Demultiplexery
  - Dekodéry
  - Komparátory (logické !)
  - Sčítačky

Majoritní funce

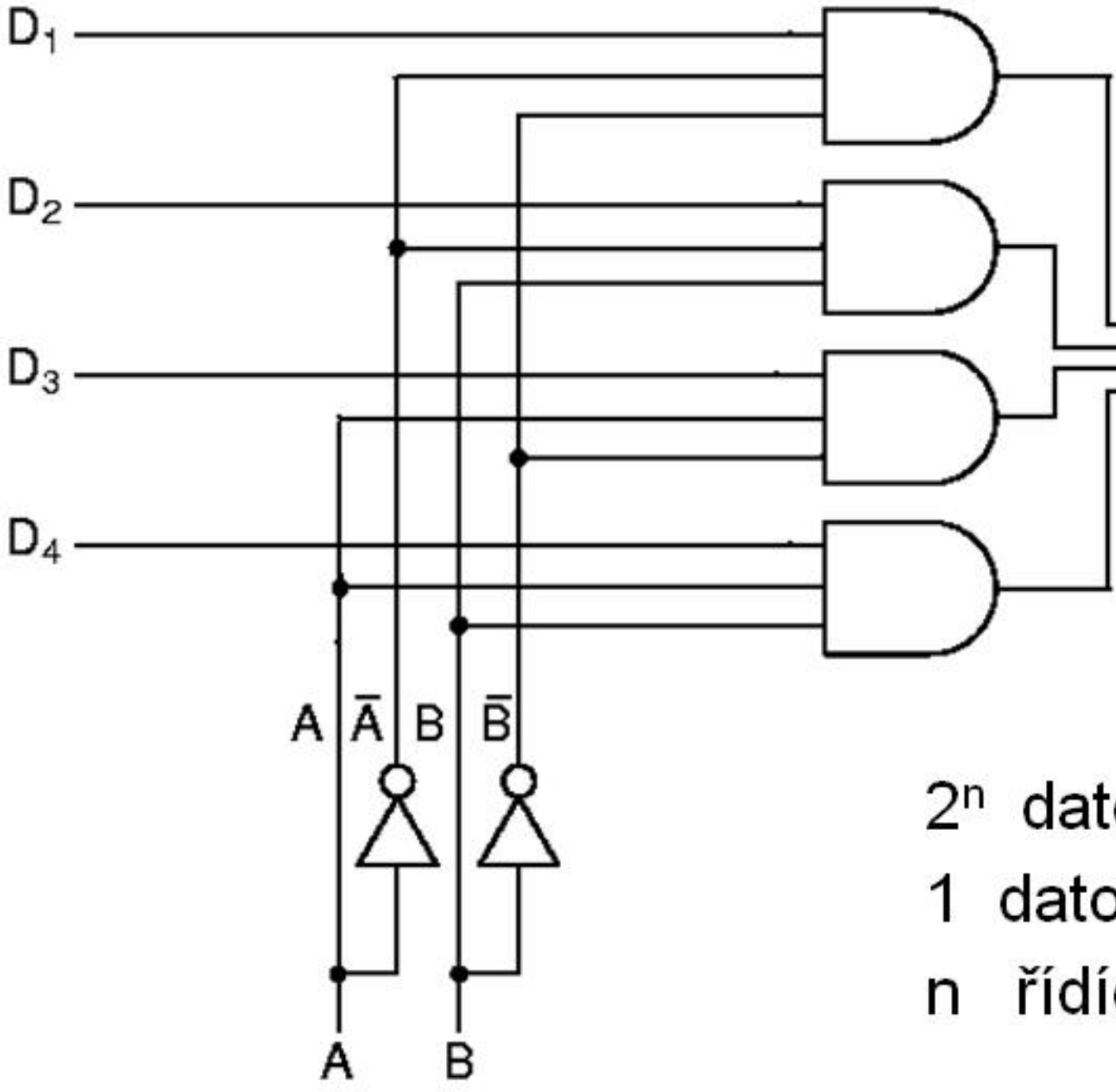
$$M = f(A, B, C)$$

$$\mathbf{M = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC}$$

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Multiplexer

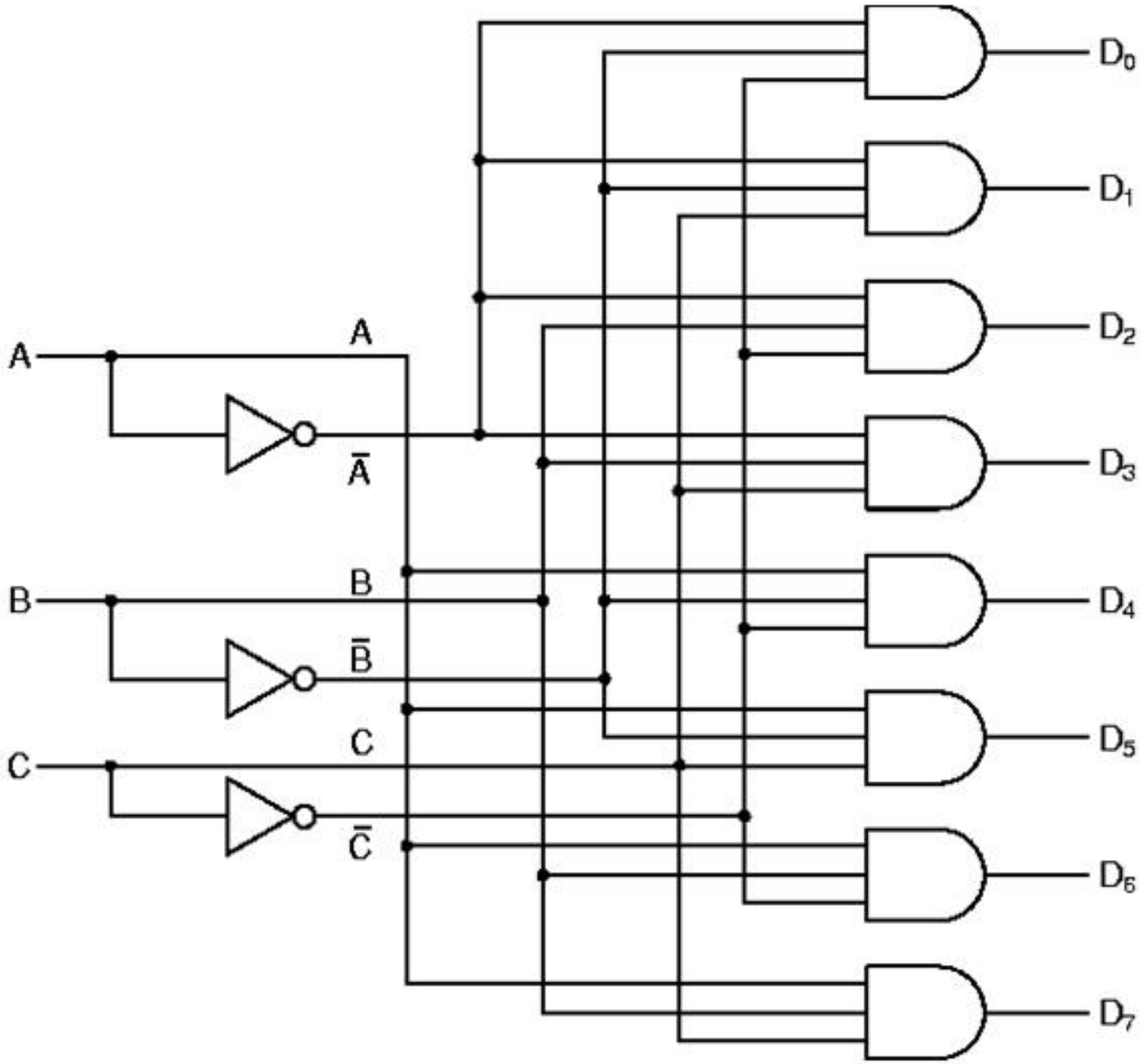


2^n datových vstupů

1 datový výstup

n řídících vstupů

Dekodér



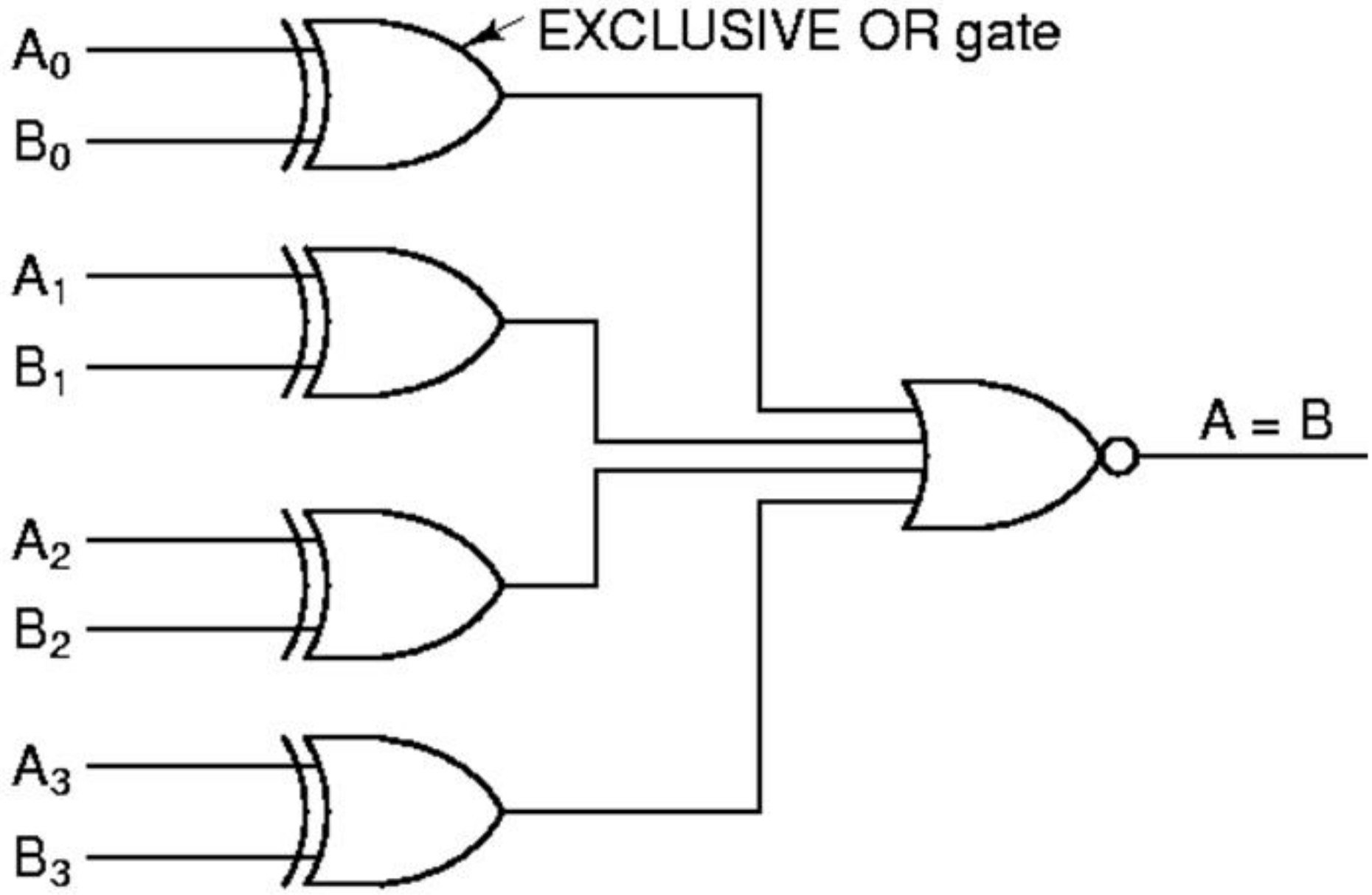
n datových vstupů

2^n datových výstupů

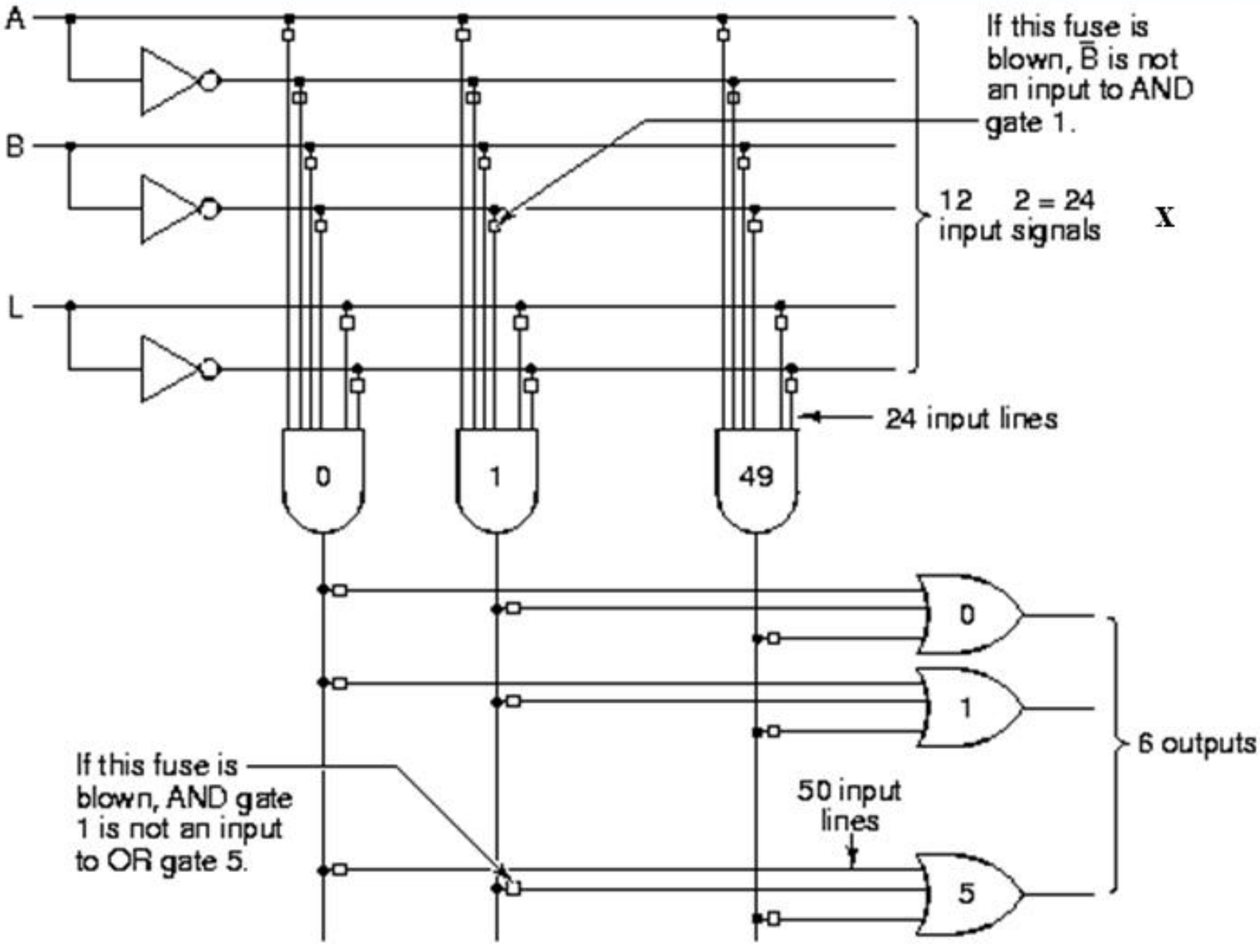
dekodér

„tři na jeden z osmi“

Komparátor (logický!!!)



Dvouúrovňová logika

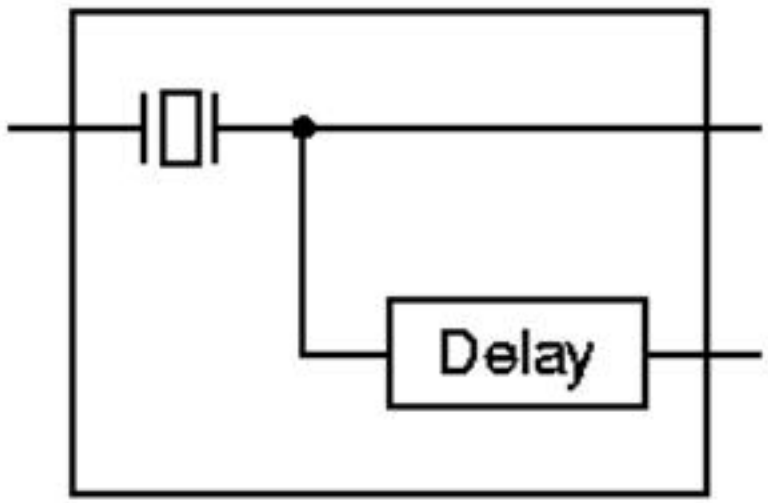


PLA

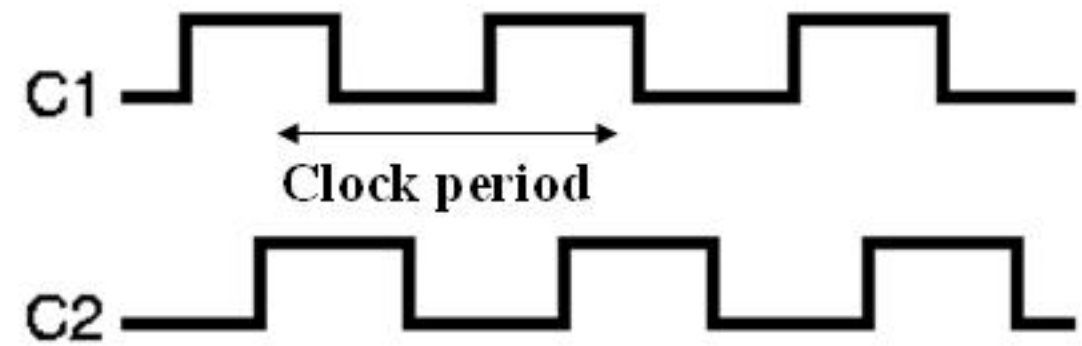
12 vstupů

6 výstupů

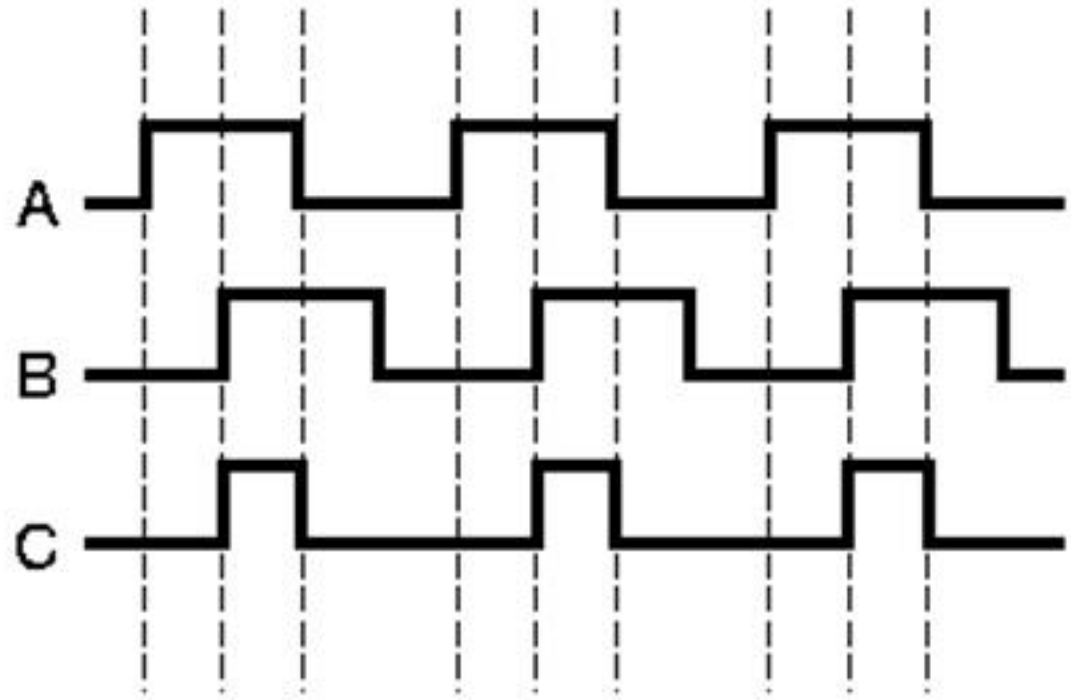
Hodiny



clock

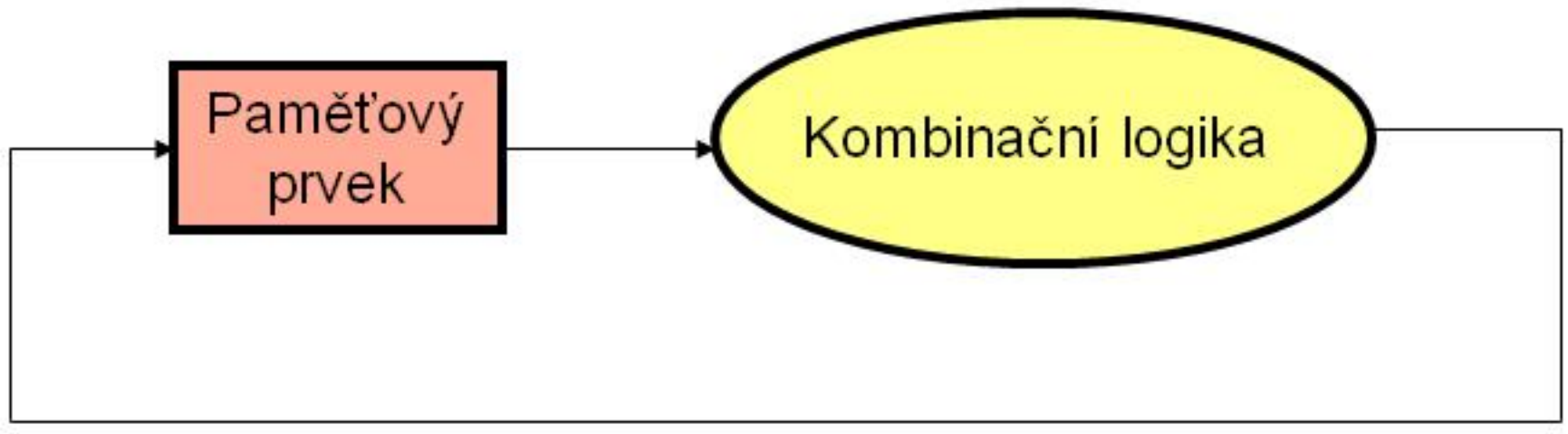


timing diagram

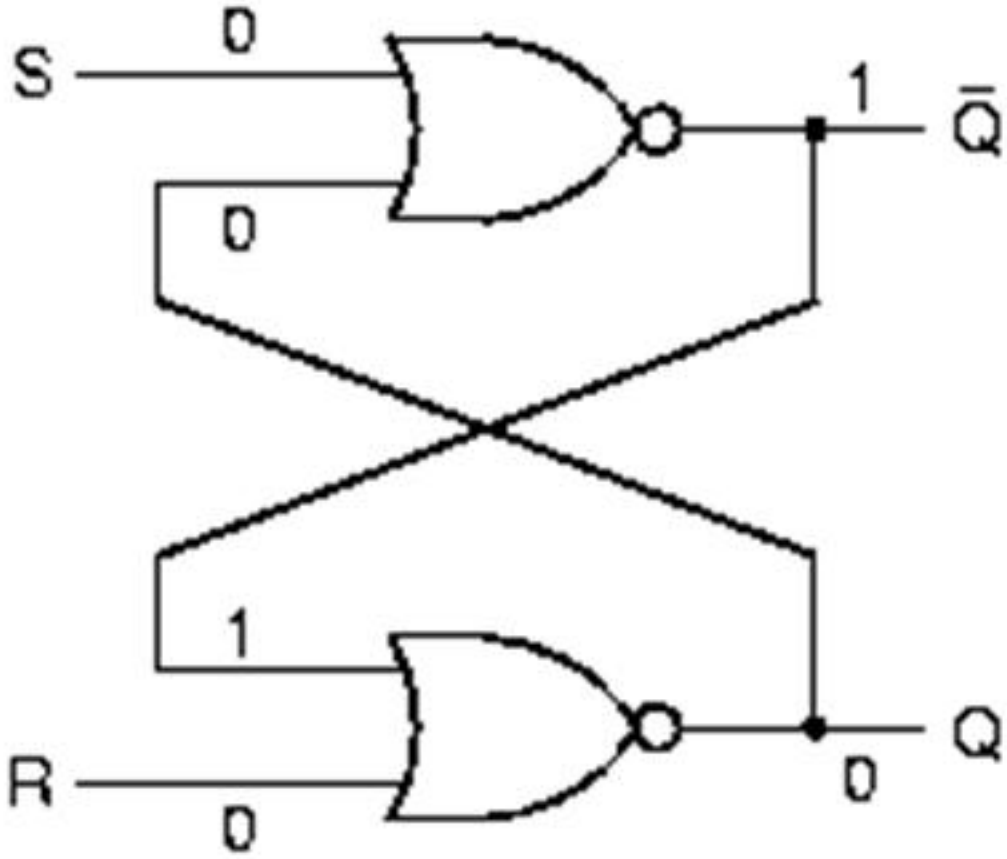


asymmetric clock

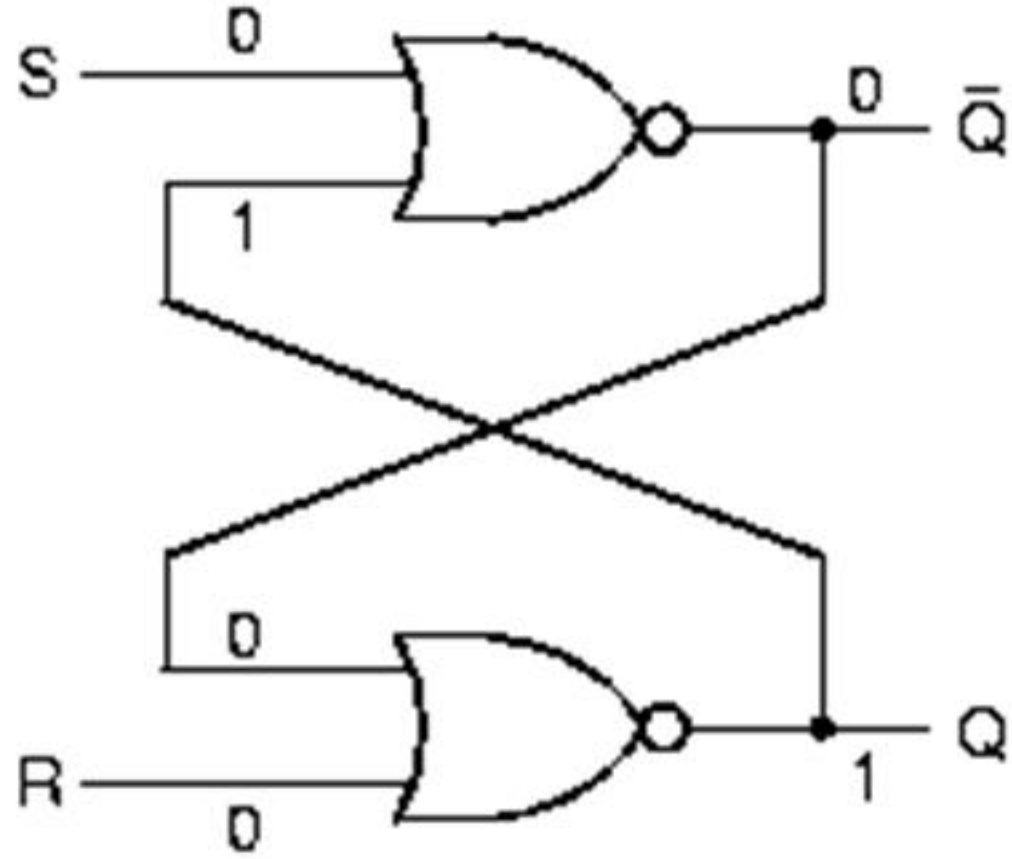
Taktování hranou signálu



NOR SR Latch



Stav 0

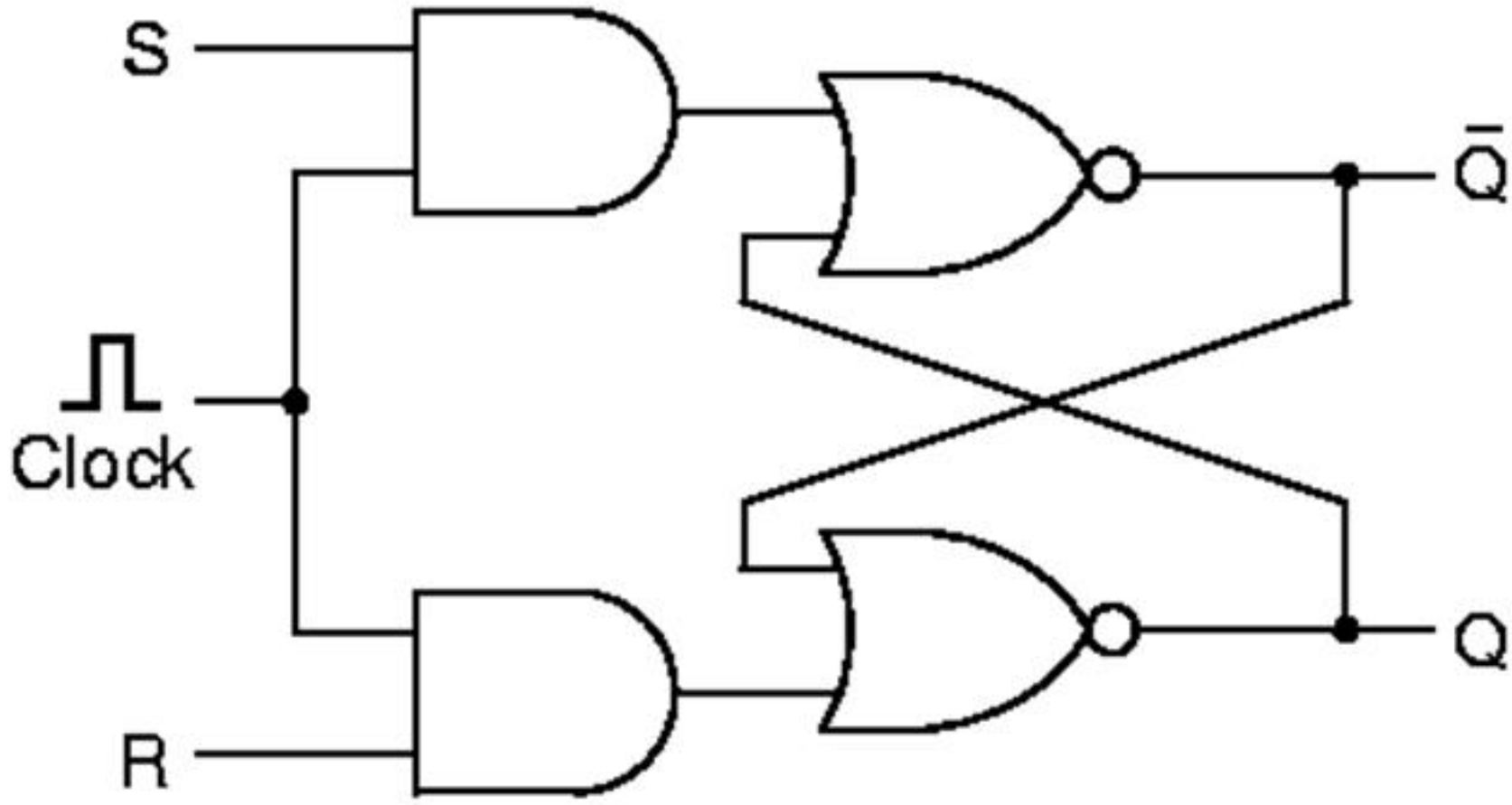


Stav 1

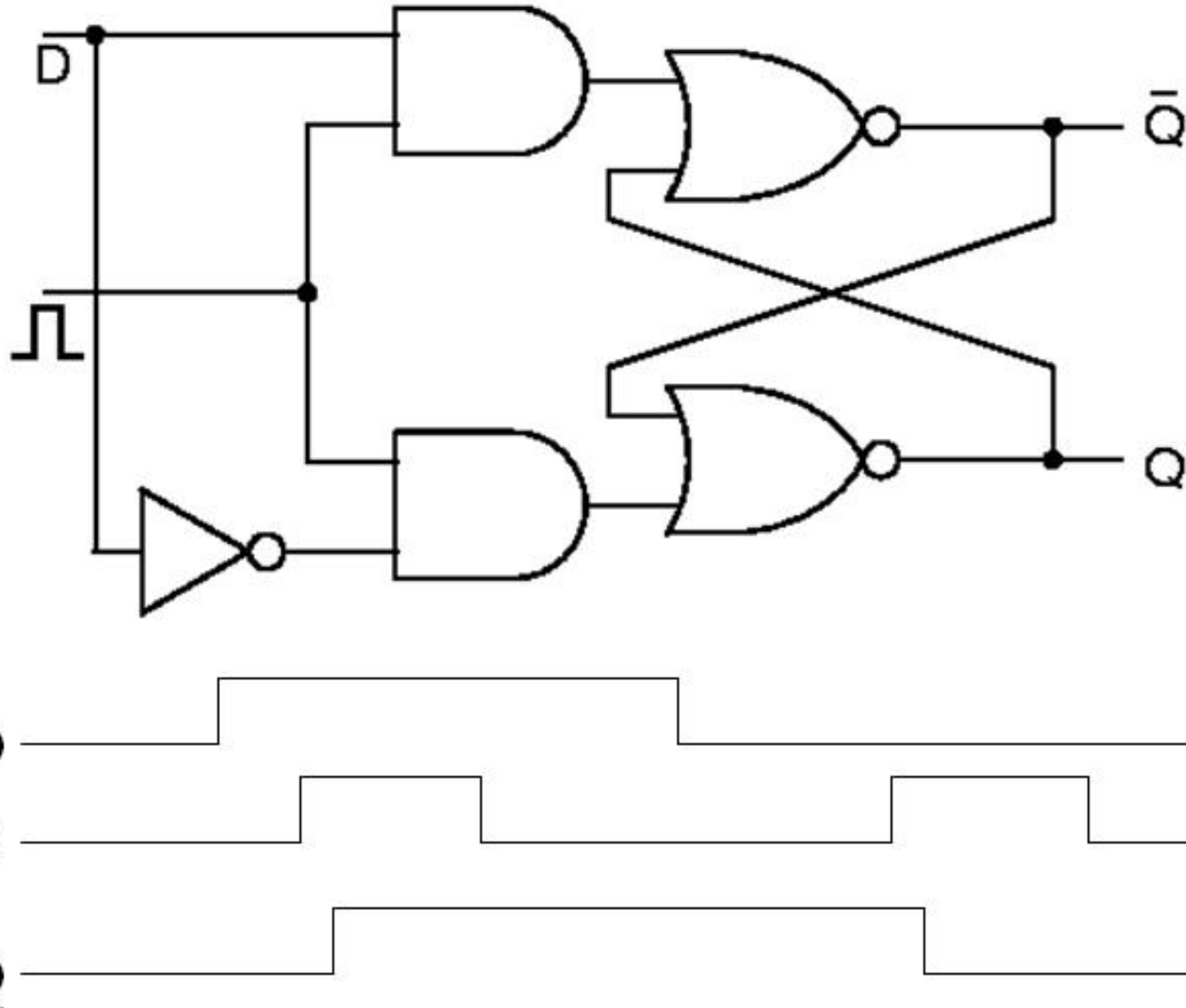
Vstupy { S - set, R - reset

Výstupy: Q and $\bar{Q}$

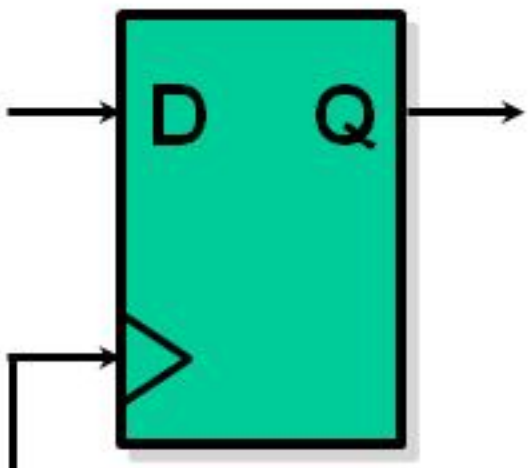
Taktovaný SR Latch



Taktovaný D Latch

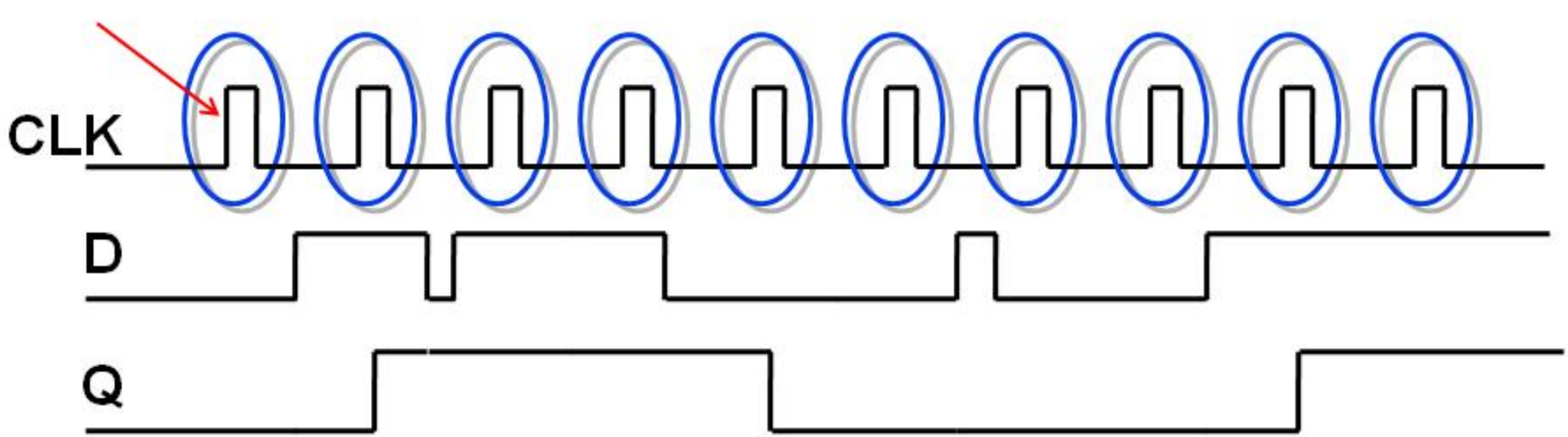


Hranový D-klopný obvod

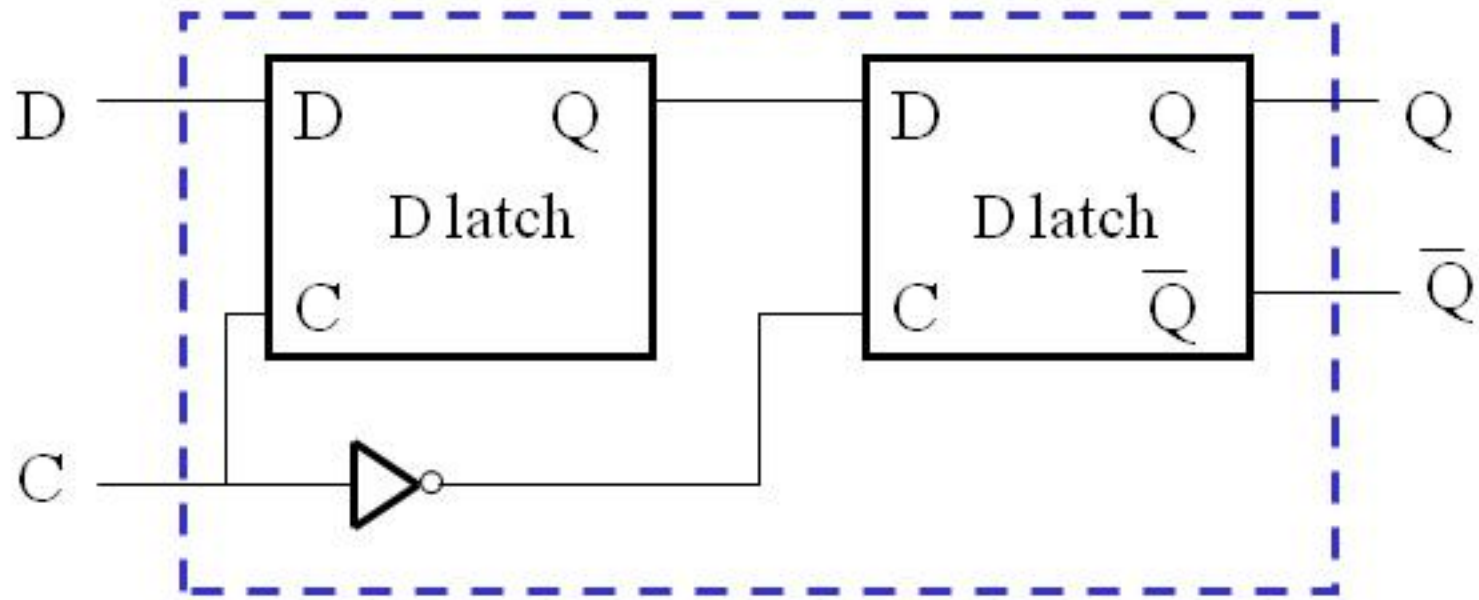


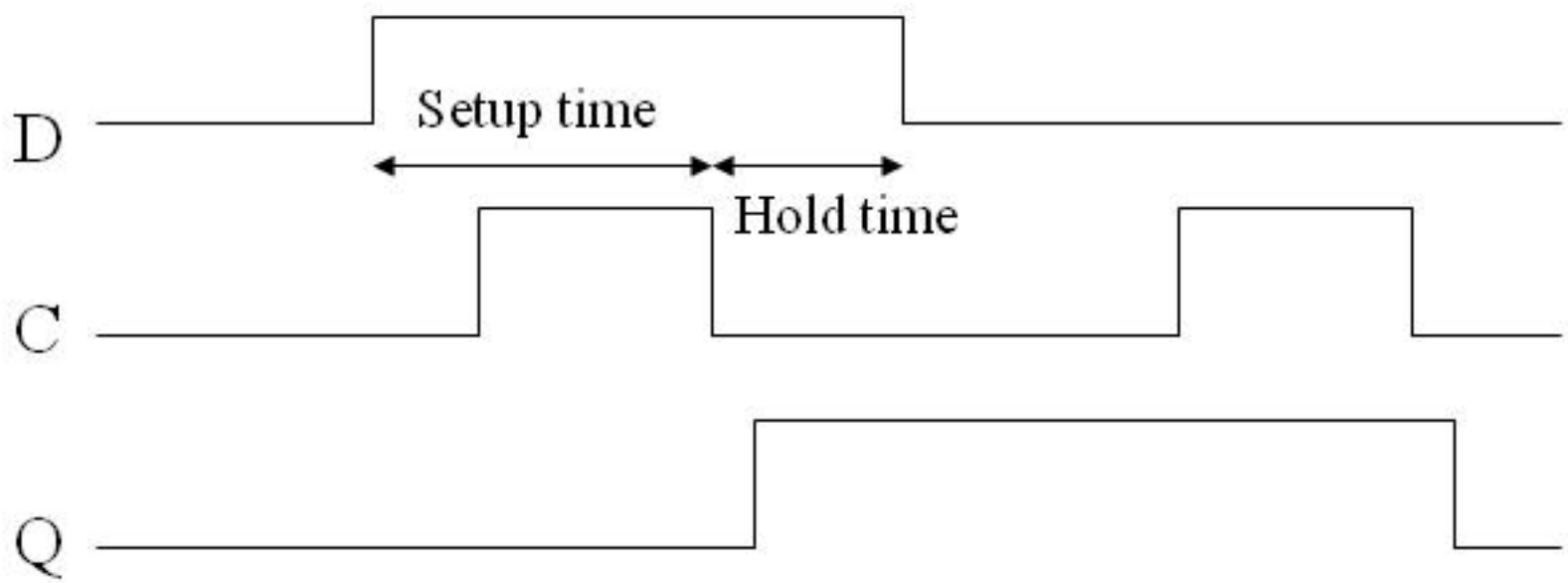
Hodnota na vstupu D je vzorkována na náběžné hraně hodin. pulzu.

Výstup Q dává navzorkovanou hodnotu po zbytek cyklu.



D flip-flop (D klopný obvod)





Závěr

- Vývoj technologie
- Organizace počítače intenzivně využívá pokroky technologie
- Digitální logika & Booleovská čísla
- Základní logická hradla a implementace
- Digitální logika – nejnižší úroveň, kterou se v kurzu budeme zabývat
- Koncepce *pravdivostních tabulek*

Závěr (pokr.)

Digitální logické obvody

- Postavené z prvků (AND, OR, NOT, ...)
- Kombinační (jednoduché nebo komplexní)
- Sekvenční
  - synchronní (taktované)
  - asynchronní

- Zopakujte si pravidla Booleovy algebry !!!

Měření výkonnosti počítačů, zkušební úlohy (benchmarks)

Přehled

- Vyhodnocení výkonu
- Omezení
- Metriky
- Rovnice pro výkon procesoru
- Reporty o vyhodnocení výkonu
- Amdahlův zákon

Přehled (pokr.)

- Zkušební úlohy
- Populární zkušební úlohy
  - Linpack
  - Intelský iCOMP
- SPEC
- Chyby a omyly

Co to je „výkon“?

- Uvažujme tři automobily. Porsche Boxster S, Honda Civic s hybridním pohonem a Ford E450 – velkoprostorový vůz
  - Boxster S má největší cestovní rychlost.
  - Honda Civic má největší dojezd.
  - E450 má největší kapacitu.

- Nyní budeme definovat výkon pomocí rychlosti.
  - Chcete-li dopravit jednoho cestujícího z jednoho místa do druhého, největší výkon má Boxster S.
  - Chcete-li dopravit 15 cestujících, největší výkon má E450.

- Co je to vlastně výkon?

Jak definovat výkon?

Letadlo	Kapacita [počet pasažerů]	Dolet [míle]	Cestovní rychlost [m.p.h.]	"Propustnost" pasažerů [počet pasažerů x m.p.h]
Boeing 777	370	4630	610	228,750
Boeing 747	470	4150	610	286,700
Concord	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

Dvě klíčové metriky výkonu

- Doba běhu úlohy
  - doba výpočtu, doba odezvy, spotřebovaný čas, latence
- Počet úloh za jednotku času
  - rychlost výpočtu, šířka pásma, propustnost

Letadlo	Washington D.C. Paris	Rychlost	Cestující	Propustnost (#cestujících x mph)
Boeing 747	6.5 hodiny	610mph	470	286,700
Concord	3 hodiny	1350mph	132	178,200

Latence versus propustnost

- Latence
  - "realný" čas potřebný pro vykonání úlohy
  - důležité, pokud se soustřeďujeme na jednotlivou úlohu
    - uživatel, který zpracovává jedinou aplikaci
    - kritická úloha ve vestavných systémech reálného času
- Propustnost (šířka pásma)
  - # úloh, vykonaných za jednotku času
  - metrika nezávislá na přesném počtu provedených úloh
  - důležité, jestliže se soustřeďujeme na běh mnoha úloh
    - manažera datového centra zajímá hlavně celkový objem práce, vykonané během dané doby

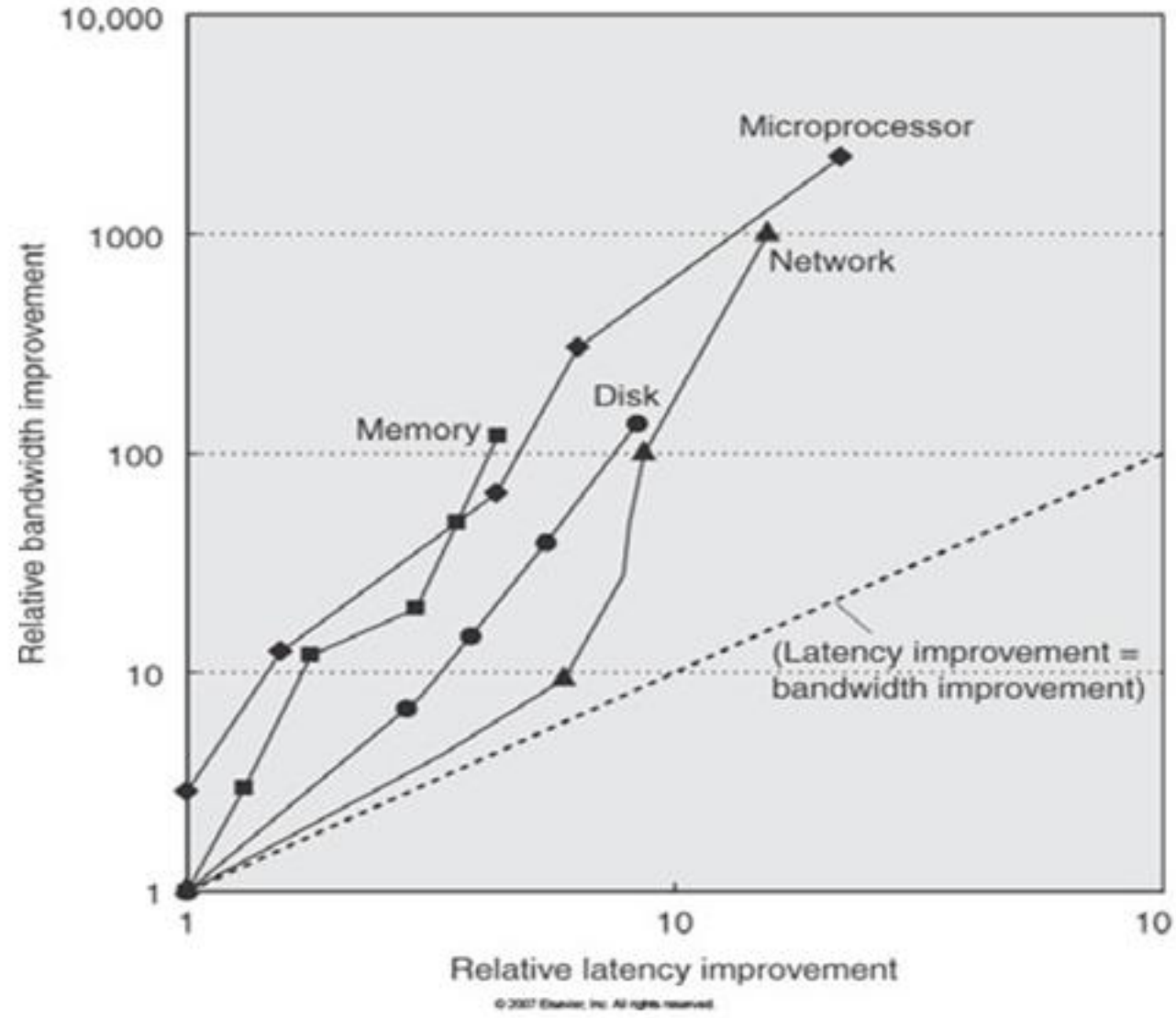
Latence zaostává za šířkou pásma

- **Šířka pásma „předběhla“ latenci ve všech hlavních počítačových technologiích**

„Existuje staré rčení:

- *Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed—you can't bribe God."*

[Anonymous]



Obrázek vyjadřuje poměrné zlepšování latence a propustnosti

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 „embedded“, přes stolní počítače, až po výkonné servery.

Doba výpočtu

- Nejjednodušší měření výkonu se zakládá na měření *doby výpočtu* (*execution time* nebo jinak *wall-clock time*).

- Ta se vztahuje na celkové množství času, potřebného k provedení úlohy včetně přístupů na disk, přístupů do paměti, režie operačního systému, I/O aktivit, doby aktivity CPU a dalších složek.

- Odstartujeme-li úlohu v 7:20 a úloha skončí v 7:26, je doba výpočtu 6 minut.

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.

Hodinové cykly

- Většina moderních počítačů používá hodinový signál s konstantní periodou. Perioda hodin je diskrétní interval, během něhož se v HW odehraje nějaký elementární krok.

- Frekvence hodin se udává v MHz nebo GHz.

- *Frekvence hodin* a *perioda hodin* jsou inverzní veličiny – jestliže systém používá hodiny s frekvencí 2.5 GHz, perioda hodin činí 400 ps.

- Nyní se můžeme pokusit vymezit pojem *výkon CPU*.

Koncepce hodnocení výkonnosti

- Hodnocení výkonu počítače je založeno na:
  - Propustnosti
  - Době výpočtu (*execution time*, *elapsed time*)
- Hodnocení výkonu komponent a vrstev
- Vyhodnocení výkonu procesoru
  - Založeno na **době výpočtu** *(execution time)* **programu:**

Doba od startu do ukončení. (nezapočítává se doba I/O a zpracování jiných programů)

$$\textbf{výkon}_X = 1/\ \textbf{doba výpočtu}_X$$

Relativní výkon: $n = \text{výkon}_X / \text{výkon}_Y$

$n > 1 \Rightarrow X$ *je n krát rychlejší než* Y

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

Příklad

- Důraz na propustnost a dobu odezvy:
  - Použití rychlejšího procesoru
    - *Snížení doby odezvy*
    - *Vyšší propustnost*
  - Zvýšení počtu procesorů v systému
    - Vyšší propustnost
    - Snížení doby odezvy (*pokud je nízká režie*)
      - U masivně paralelních procesorů (MPP)
      - U symetrických „multiprocesorových“ procesorů (SMP)
        - » Pouze když další procesory redukují čas čekání ve frontách

Měření výkonnosti

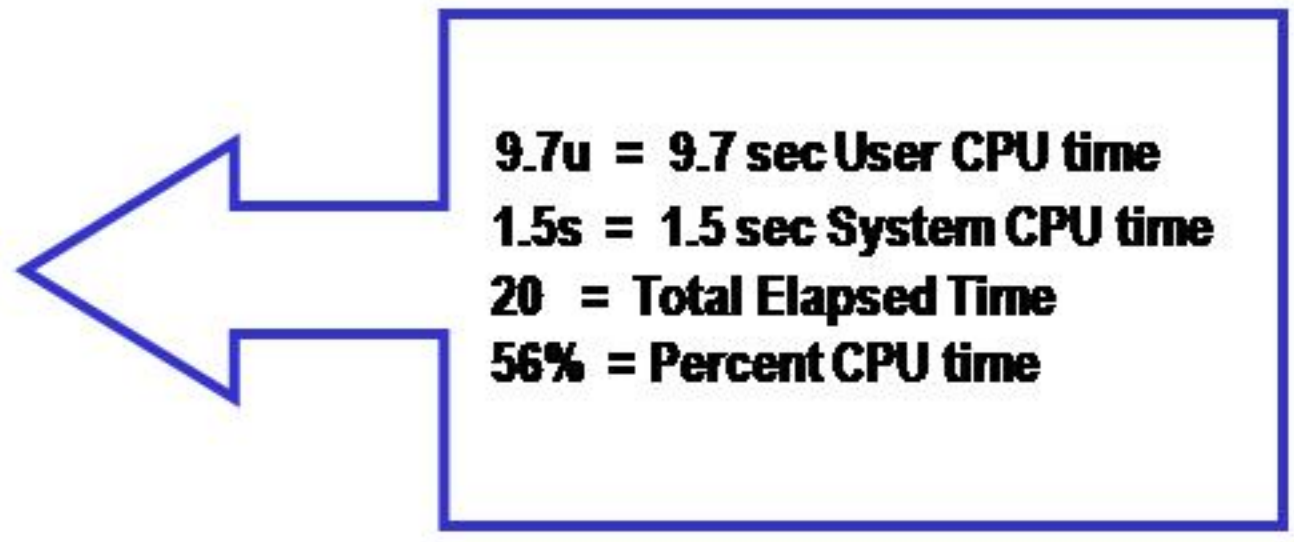
Složky doby výpočtu programu:

1. Čas práce CPU
   - čas CPU věnovaný uživatelskému programu
   - čas CPU spotřebovaný operačním systémem
2. Čas pro I/O operace
3. Čas věnovaný výpočtu jiných programů

Příkaz Unixu **time**

time cc prog.c

9.7u 1.5s 20 56%



Výpočet výkonu CPU

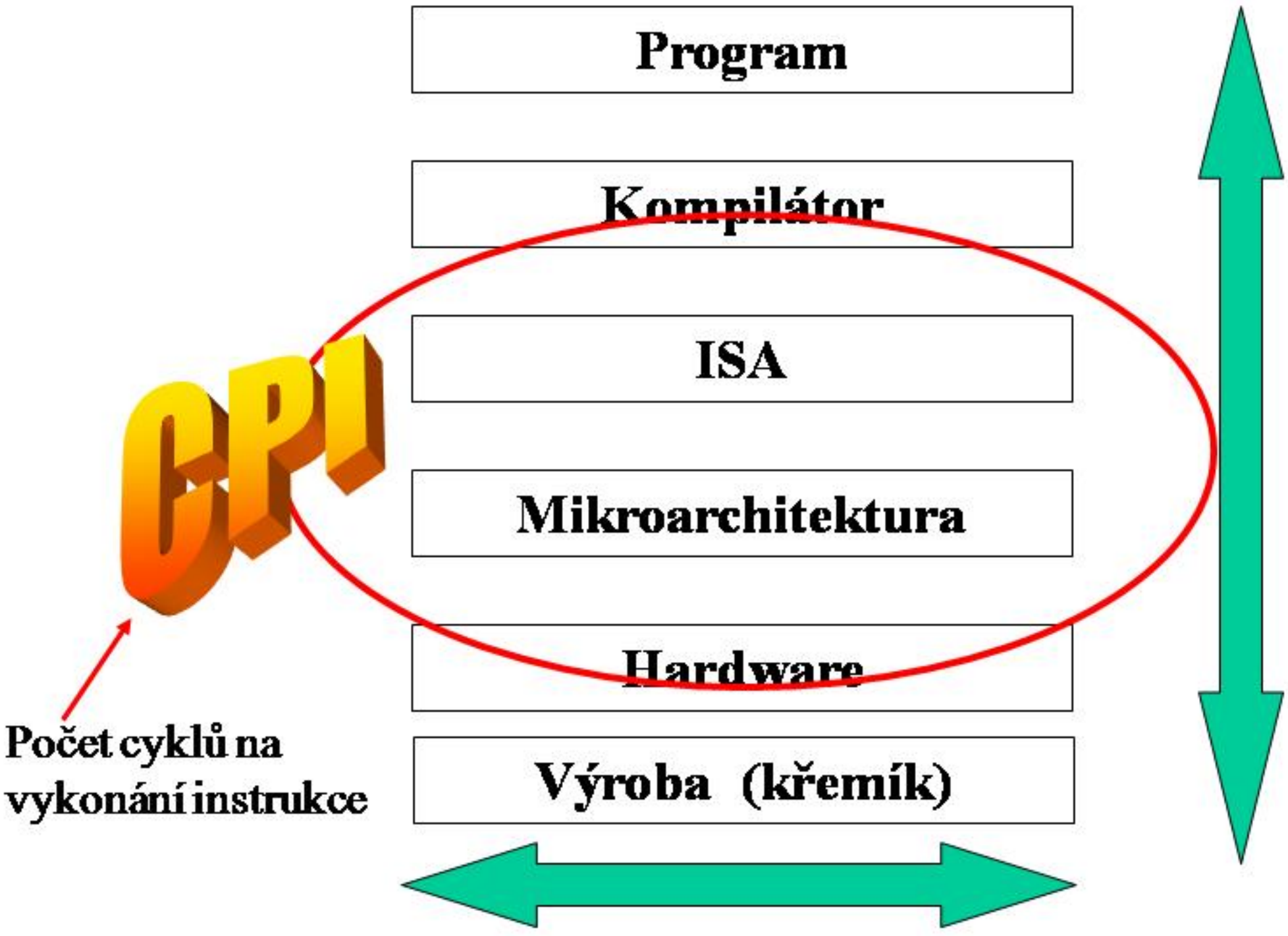
- **CPU time** = počet cyklů CPU * doba cyklu
  = počet cyklů CPU / frekvence hodin
  - počet cyklů CPU = IC * CPI
    - IC: počet instrukcí (počet instrukcí na program)
    - CPI: střední hodnota počtu cyklů na instrukci

- **CPU time = IC * CPI * doba cyklu**

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

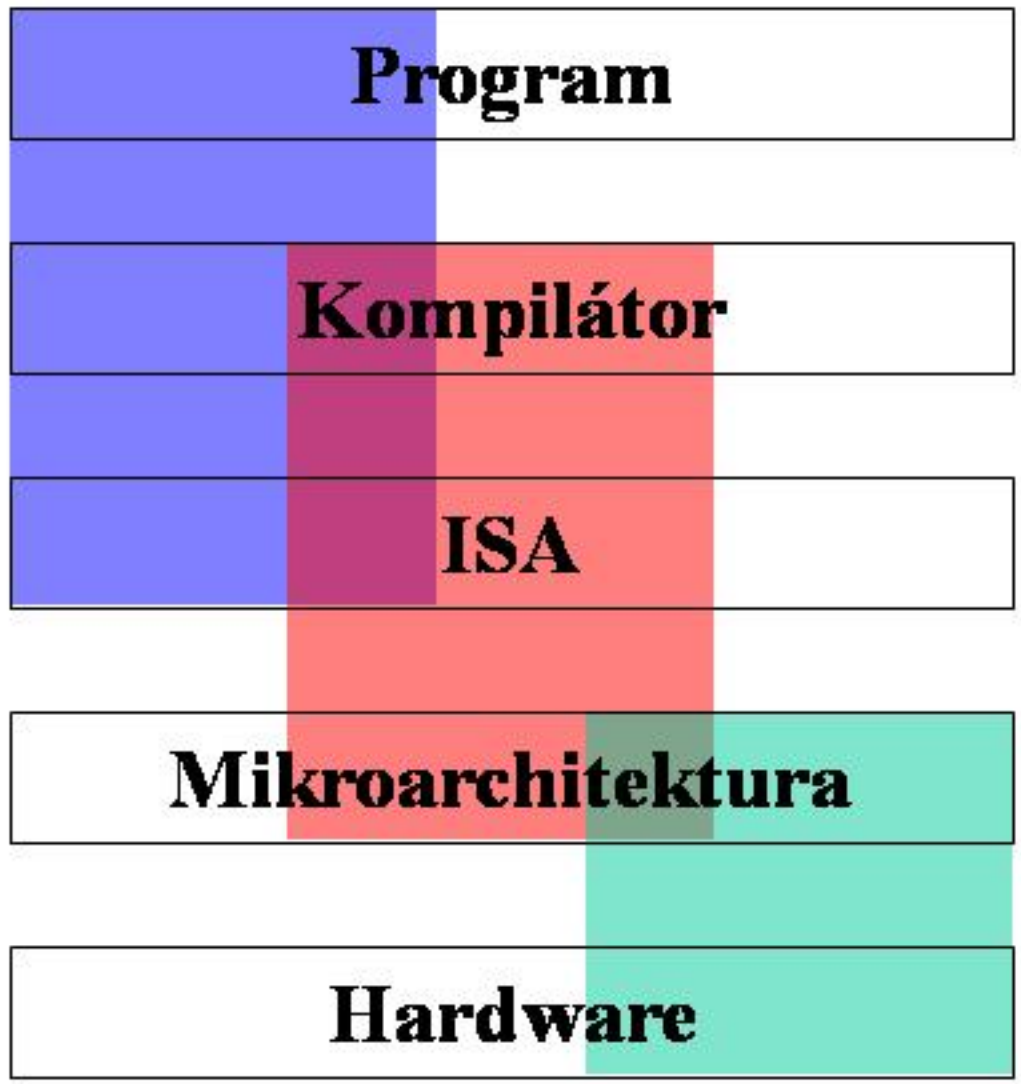
- **Cykly hodin CPU** = Σ_i (CPI_i * IC_i)
  - IC_i: počet instrukcí třídy *i*
  - CPI_i: cykly, které jsou třeba k provedení instrukcí třídy i

Co ovlivňuje CPI



Vliv jednotlivých složek na výkon

CPU time = IC * CPI * Cycle time



vzájemná závislost úrovní

Měření uvedených parametrů

- Dobu CPU lze získat provedením programu (moderní operační systémy tyto statistiky zaznamenávají).
- Hodinová frekvence je snadno dostupný parametr.
- Počet instrukcí se nezískává snadno. Tyto statistiky se získávají použitím simulátorů, emulátorů nebo přímo HW čítači v procesoru (HW ladicí prostředky). Protože počet instrukcí závisí na architektuře a nikoliv na vnitřní organizaci nižších úrovní, může být měřen nezávisle na implementaci (počet instrukcí je stejný jak pro Pentium, tak pro Pentium 4 - pro stejný program).

Měření uvedených parametrů (pokr.)

- Nejobtížněji se získává parametr CPI, který je závislý na konkrétní implementaci.

- CPI je ovlivňován všemi volbami při návrhu organizace počítače, včetně paměťového systému a dalších faktorů. Například určitá instrukce může být provedena během 27 period hodin na Pentiu, ale během 9 period hodin na Pentiu 4.

- CPI také závisí na typu instrukčního mixu (program s 80% 1- cyklových instrukcí bude mít nižší CPI než program s 80% 9 - cyklových instrukcí).

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 ?

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

(stávající doba výpočtu)

(cílová doba výpočtu)

Řešením pro X získáme **X = 4 GHz**

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

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.

Co určuje výkon CPU

CPU time = počet instrukcí x CPI x doba_cyklu

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

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

Příklad

Operace	Četnost	CPI_i	Četnost x CPI_i
ALU	50%	1	
Load	20%	5	
Store	10%	3	
Branch	20%	2	
			$\Sigma =$

- Jak se zrychlí počítač, zredukuje-li rychlejší cache dobu cyklu na dva takty?
- Jakého výsledku dosáhneme využitím predikce, která zkrátí cykl skoků?
- Co kdyby bylo možno vykonat dvě ALU instrukce současně?

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

- 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!)

$$MIPS = \frac{Počet\_instrukcí}{Doba\_výpočtu \times 10^6}$$

Hlavní problémy s měřítkem „MIPS“

- MIPS je jednoduché měřítko – rychlejší stroje mají vyšší ohodnocení MIPS.
- Existují tři hlavní problémy s MIPS jako měřítkem pro hodnocení výkonu ...
  - měřítko MIPS nezahrnuje výkonnost instrukcí. Nelze tedy porovnávat počítače s rozdílným instrukčním souborem.
  - měřítko MIPS se mění i u jednoho počítače (různé instrukce mají různý CPI).
  - měřítko MIPS se dokonce může měnit inverzně vzhledem k výkonu.

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.

Hodnocení měřítka „MIPS“

- Porovnávání jablek s pomeranči
- Nedostatek: 1 MIPS jednoho procesoru neznamená stejnou práci jako 1 MIPS jiného
  - *Podobné určení vítěze běžeckého závodu podle toho, kdo udělá méně kroků*
  - Některé procesory mají FP v software (asi 1FP = 100 INT)
  - Různé instrukce trvají různě dlouho
- Vhodné pouze pro porovnání 2 procesorů stejného výrobce se stejnou ISA a se stejným kompilátorem. (např. Intel iCOMP benchmark)

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.

Problémy s MFLOPS

- Metrika MFLOPS je založena na aktuálních operacích, nikoliv na instrukcích. Poskytuje tedy věrnější hodnocení než MIPS.
- Přesto existují vážné problémy …
  - ❖ Soubor FP operací se u různých strojů liší. Například Motorola 68882 má FP instrukce dělení, odmocniny, sinu a kosinu. Cray 2 je nemá. Motorola provádí operaci kosinus během jedné instrukce, kdežto Cray 2 jich musí udělat celou řadu.

Problémy s MFLOPS

- Hodnocení pomocí MFLOPS je ovlivněno nejen poměrem FP operací k non-FP operacím, ale také poměrem „rychlých“ FP operací ku „pomalým“ FP operacím. Například program, obsahující 100% FP dělení bude prováděn pomaleji než program, který obsahuje 100% FP sčítání. Byl by hodnocen výš.
- To lze kompenzovat zavedením vah jednotlivých typů instrukcí do výpočtu podle jejich složitosti. Takové měřítko se pak nazývá *normalizované 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!

Složky rovnice pro výkon CPU

- Doba cyklu hodin
  - Odhady časování a verifikace (kompletní návrh)
  - Cílová doba cyklu
- Počet instrukcí
  - Kompilátor
    - Simulátor instrukčního souboru
    - Monitorování (execution-based monitoring, *profiling*)
- CPI, je-li uplatněn princip pipeline, potom
  - CPI_i = Pipeline CPI_i + Memory CPI_i

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…

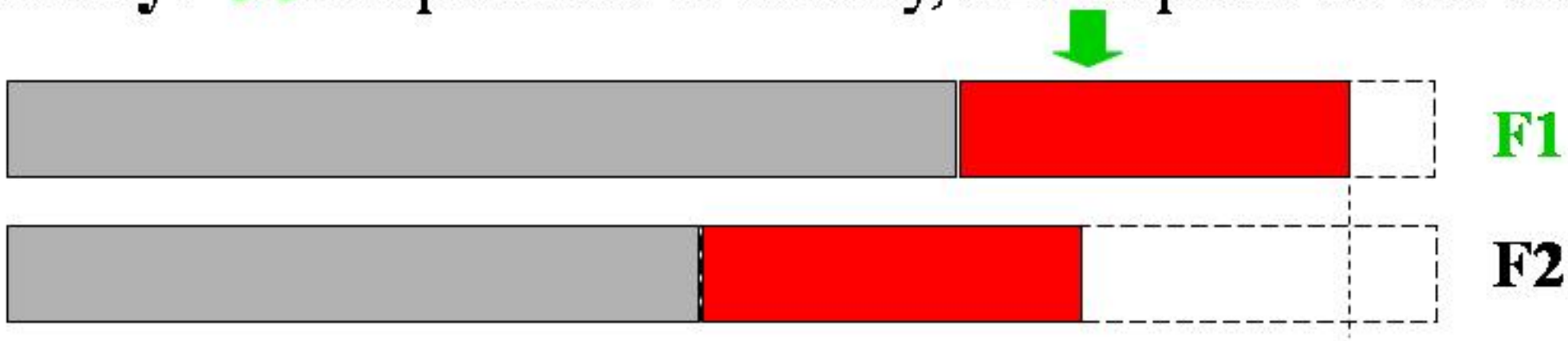
$$\text{Doba výpočtu po zlepšení} = \frac{\text{Doba výpočtu ovlivněná zlepšením}}{\text{Poměr zlepšení}} + \text{Doba výpočtu neovlivněná}$$

Amdahlův zákon

- Zákon o klesajícím zisku (návratnosti investic)

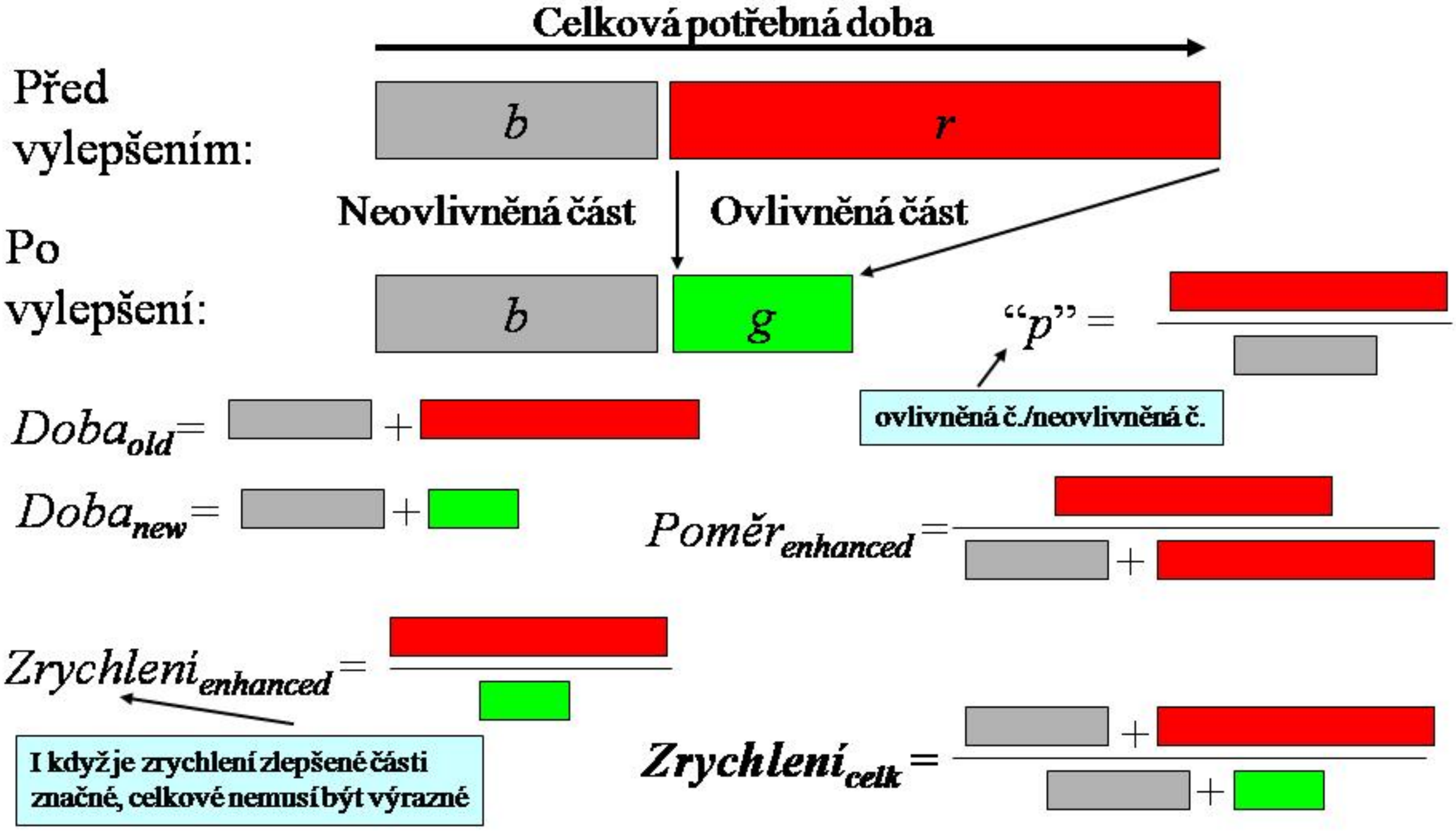
$$\text{Zrychlení} = \frac{\text{Doba výpočtu před zlepšením}}{\text{Doba výpočtu po zlepšení}}$$

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



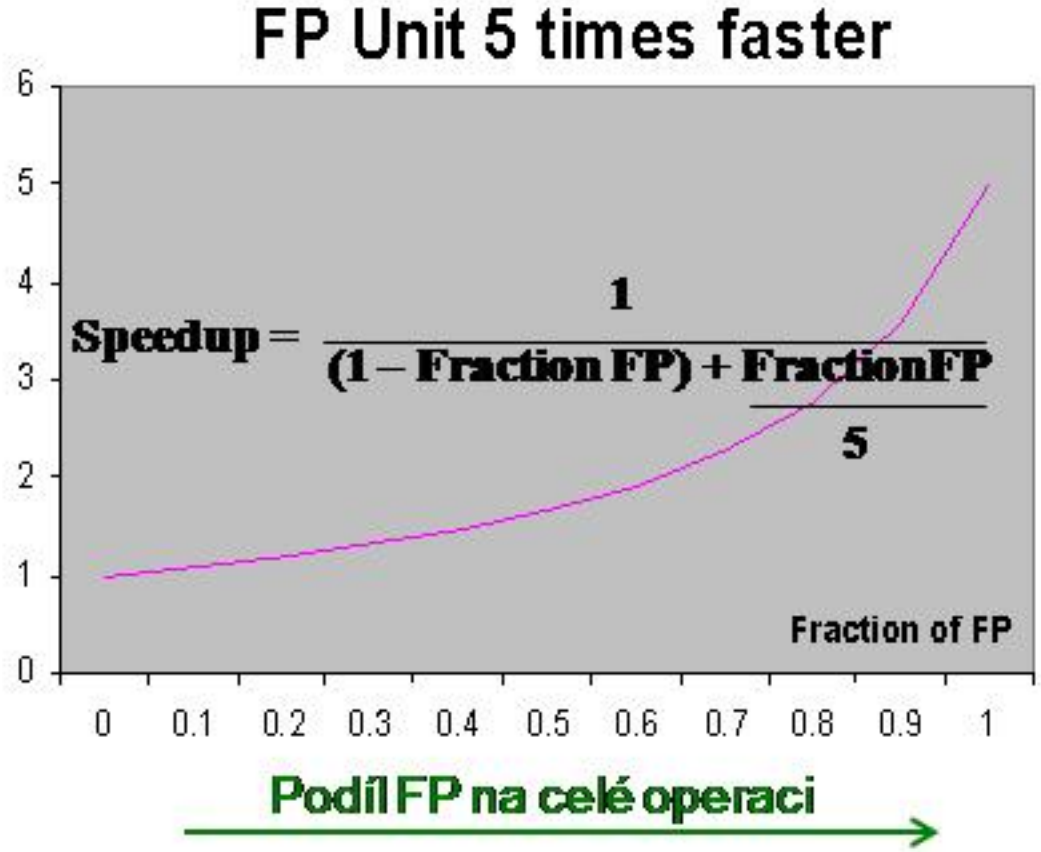
$$\text{Zrychlení} = \frac{1}{(1 - \text{zlepšená část}) + (\text{zlepšená část} / \text{koeficient zlepšení})}$$

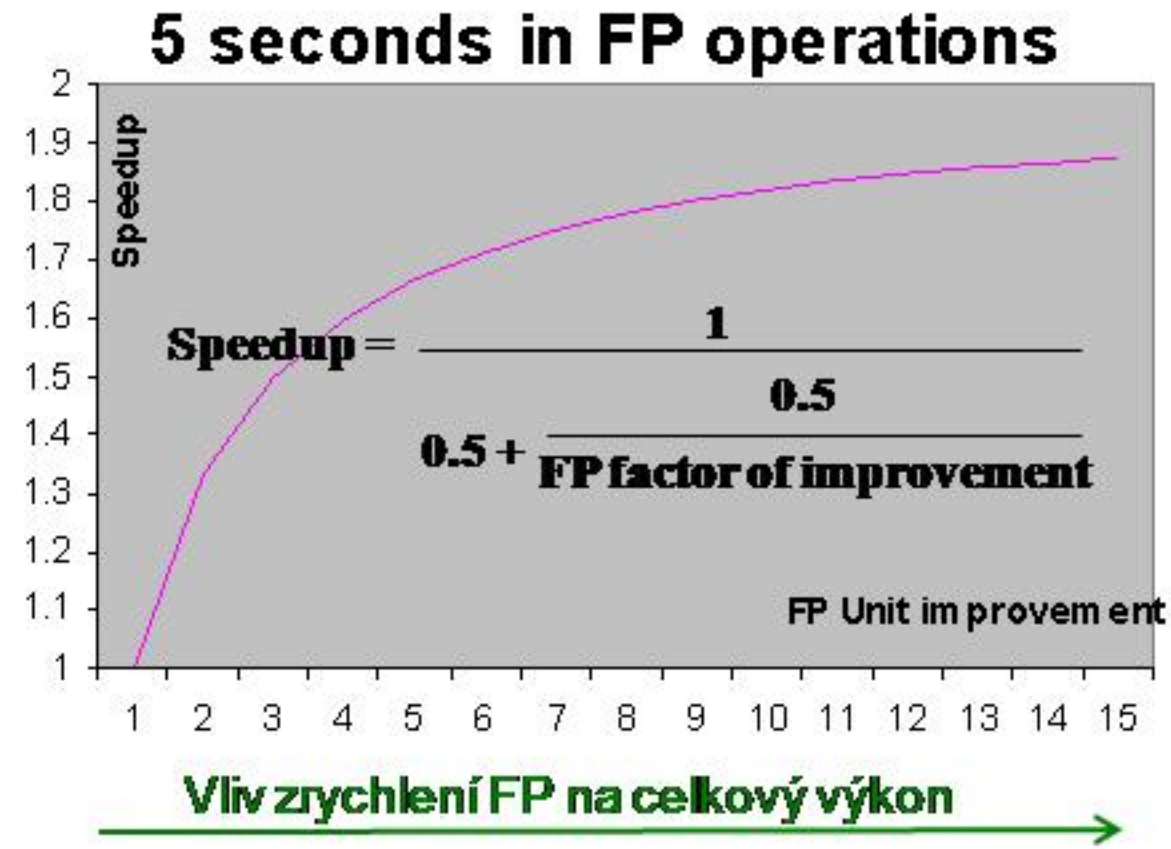
Grafické znázornění Amdahlova zákona



Příklad

- Doba zpracování programu je 10 sekund
- Jaké bude zrychlení, bude-li procesor doplněn rychlejší floating point jednotkou?





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, ….)

Zkušební úlohy - benchmarky

- *Benchmark* je program nebo soubor programů určených pro měření výkonu počítače.
- Čím lépe vystihuje benchmark typickou zátěž počítače, tím přesnější výsledky může poskytnout.
  - Kdyby většinu uživatelů tvořili inženýři, dobrým benchmarkem by byla sada typických inženýrských a vědeckotechnických aplikací.
  - Kdyby většina uživatelů byli vývojáři SW, potom dobrým benchmarkem by byly kompilátory a programy pro zpracování zdrojových textů.

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*: $GM = (\Pi\ exec\_time\_ratio_i)^{1/n}$

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

Podstatné!!!

Poznámka k harmonickému průměru

Harmonický průměr je podíl počtu prvků a součtu převrácených hodnot prvků. Používá se tam, kde má smysl sčítat převrácené hodnoty. Například máme vypočítat střední hodnotu v případě, že víme, že jeden zemědělec zorá pole za 5 dní a druhý za 2 dny. Jaký je jejich průměrný výkon. Můžeme uvažovat tak, že použijeme prostý aritmetický průměr, který má hodnotu 3.5. Ale tento údaj je chybný. Jak je to možné. Představte si, že mají zorat dvě pole. Každý začne na svém poli. Ke konci prvního dne má jeden zoraný jednu pětinu svého pole (celé pole zorá za pět dní, proto za jeden den zorá jednu pětinu) a druhý jednu polovinu. Druhý den má první zorané dvě pětiny, ale druhý už má zorané celé pole a může tedy pomoci prvnímu. Zbývají pouze 0.6 pole, a my víme, že oba dohromady zorají za jediný den 0.7 pole (jedna pětina a jedna polovina dávají jako výsledek právě 0.7). To tedy znamená, že zbytek musejí zorat za méně než jeden den, přesně je to 0.86 dne. Průměr nás však informuje, že by to mělo být ještě jeden a půl dne. Pokud bychom použili harmonický průměr, dostaneme přesně 2.86 dne, což souhlasí s naší úvahou. V tomto případě je tedy nutné použít právě harmonický průměr. (zdroj www.tiscali.cz)

Aritmetický a geometrický průměr

	A	B	C
P1	1	10	20
P2	1000	100	20

Doba výpočtu (v sekundách)
strojů: A, B a C
programů: P1 a P2

	Normalized to A			Normalized to B			Normalized to C		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
Program P1	1.0	10.0	20.0	0.1	1.0	2.0	0.05	0.5	1.0
Program P2	1.0	0.1	0.02	10.0	1.0	0.2	50.0	5.0	1.0
Arithmetic mean	1.0	5.05	10.01	5.05	1.0	1.1	25.03	2.75	1.0
Geometric mean	1.0	1.0	0.63	1.0	1.0	0.63	1.58	1.58	1.0
Total time	1.0	0.11	0.04	9.1	1.0	0.36	25.03	2.75	1.0

Shrnutí benchmarků

- Předpokládejme, že už jsou vybrány zkušební úlohy. Jak získáme výsledek?
- Například, jestliže dva benchmarky spuštěné na dvou počítačích dávají následující výsledky….

	Počítač A	Počítač B
Program 1 – doba výpočtu (v sekundách)	1	10
Program 2 – doba výpočtu (v sekundách)	1000	100
Celková doba výpočtu (v sekundách)	1001	110

Z měření vyplývá:

- A je 10 krát rychlejší než B pro program 1.
- B je 10 krát rychlejší než A pro program 2.

- Poměr výkonů počítačů A a B je nejasný. Lze vůbec některý prohlásit za rychlejší ?

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.

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.

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.

Cílené benchmarky

- Vývoji benchmarkových programů už bylo věnováno mnoho úsilí.
- Základním problémem při takovém vývoji („výzkumu“) je podvádění …
  - Kompilátory mohou obsahovat velmi účinné sekvence instrukcí pro specifické benchmarky (zadané ve zdrojovém kódu).
  - Procesory mohou být optimalizovány na provádění určitých instrukčních proudů, protože tyto jsou použity ve specifických benchmarcích.

Problémy cílených benchmarků

- Hlavním problémem takových benchmarků jsou „podvody“.
- Verze benchmarku SPEC z r. 1989, obsahovala program matrix300, který obsahoval operace násobení matic.
- 99% doby výpočtu tohoto benchmarku probíhalo v jediné řádce zdrojového textu.
- Důsledkem bylo, že mnoho výrobců kompilátorů se soustředilo na extrémní optimalizaci této jediné řádky. To zvýšilo rychlost provádění až na 700 násobek v porovnání se situací bez optimalizace. Takový benchmark ale dává velmi „matnou“ představu o celkovém výkonu systému.

„Z historie zkušebních úloh“

1. Vytvoříte benchmark zvaný *bmark* ☺
2. Ověříte jej na řadě počítačů
3. Publikujete výsledky na www.bmark.org
4. *bmark* and www.bmark.org se stanou populární ☺
   - Uživatelé začnou nakupovat podle výsledků *bmarku*
   - Stanete se populární i u výrobců
5. Výrobci analyzují *bmark* a upraví kompilátory, popř. i mikroarchitekturu tak, aby *bmark dával dobré výsledky*
6. Tímto krokem byl *bmark* znehodnocen ☹
7. Vytváříte *bmark* 2.0 ☺

Linpack Benchmark

- „Matka“ všech benchmarků
- Doba potřebná pro řešení systému lineárních rovnic

```
DO I = 1, N
  DY(I) = DY(I) + DA * DX(I)
END DO
```

- Metriky
  - R_{peak}: výkonová špička v Gflops
  - N_{max}: velikost matice dávající nejvíce Gflops
  - $N_{1/2}$: velikost matice dávající polovinu R_{max} Gflops
  - R_{max}: Gflops dosažené pro velikost matice N_{max}
- Použito http://www.top500.org

Intelský iCOMP Index 3.0

- Nová verze (3.0) zahrnuje:
  - Instrukční mix pro existující software.
  - Nárůst 3D, multimédia a internetový software.
- Benchmarky
  - 2 aplikace integer (20 % každá)
  - Výpočty geometrie v 3D a osvětlení (20 %)
  - FP inženýrské a finanční programy, hry (5 %)
  - Multimediální a internetové aplikační programy (25%)
  - Aplikační programy v jazyce JAVA (10 %)
- Vážený relativní výkon GM (geometrický průměr)
  - Jednotkový" procesor: Pentium II procesor na 350MHz

$$iCOMP\ Index\ baseline = \left(\frac{BM1}{Base\_BM1}\right)^{P_1} * \left(\frac{BM2}{Base\_BM2}\right)^{P_2} * \cdots * \left(\frac{BM6}{Base\_BM6}\right)^{P_6}$$

Benchmarky Př.3: SPEC2000

- System Performance Evaluation Corporation
- Je třeba provádět update/upgrade benchmarků
  - Delší doba běhu
  - Rozsáhlejší problémy
  - Různorodost aplikací
- Pravidla běhu a report
  - Základní a optimalizovaný
  - Geometrický průměr normalizovaných dob výpočtu
  - Referenční stroj: Sun Ultra5_10 (300-MHz SPARC, 256MB)
- CPU2000: poslední SPEC CPU benchmark (4. verze)
  - 12 integer a 14 floating point programů
- Metriky: doba odezvy a propustnost

Benchmarky SPEC CINT2000

1.	164.gzip	C	Compression
2.	175.vpr	C	FPGA Circuit Placement and Routing
3.	176.gcc	C	C Programming Language Compiler
4.	181.mcf	C	Combinatorial Optimization
5.	186.crafty	C	Game Playing: Chess
6.	197.parser	C	Word Processing
7.	252.eon	C++	Computer Visualization
8.	253.perlbmk	C	PERL Programming Language
9.	254.gap	C	Group Theory, Interpreter
10.	255.vortex	C	Object-oriented Database
11.	256.bzip2	C	Compression
12.	300.twolf	C	Place and Route Simulator

Benchmarky SPEC CFP2000

1.	168.wupwise	F77	Physics / Quantum Chromodynamics
2.	171.swim	F77	Shallow Water Modeling
3.	172.mgrid	F77	Multi-grid Solver: 3D Potential Field
4.	173.applu	F77	Parabolic / Elliptic Partial Differential Equations
5.	177.mesa	C	3-D Graphics Library
6.	178.galgel	F90	Computational Fluid Dynamics
7.	179.art	C	Image Recognition / Neural Networks
8.	183.equake	C	Seismic Wave Propagation Simulation
9.	187.facerec	F90	Image Processing: Face Recognition
10.	188.ammp	C	Computational Chemistry
11.	189.lucas	F90	Number Theory / Primality Testing
12.	191.fma3d	F90	Finite-element Crash Simulation
13.	200.sixtrack	F77	High Energy Nuclear Physics Accelerator Design
14.	301.apsi	F77	Meteorology: Pollutant Distribution

Metriky SPECINT2000

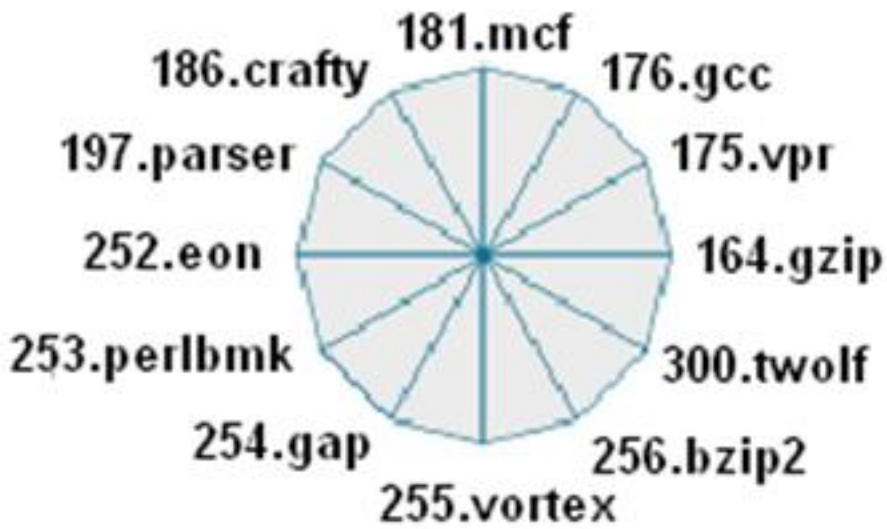
- **SPECint2000**: Geometrický průměr 12 normalizovaných poměrů (pro každý integer benchmark jeden), každý benchmark kompilován s "agresivní" optimalizací

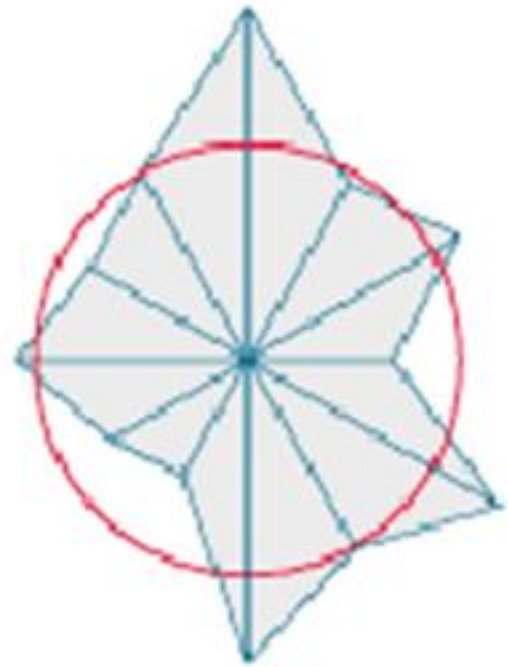
- **SPECint_base2000**: Geometrický průměr 12 normalizovaných poměrů, kompilováno s „konzervativní" optimalizací

- **SPECint_rate2000**: Geometrický průměr 12 normalizovaných poměrů pro **propustnost**, kompilováno s "agresivní" optimalizací

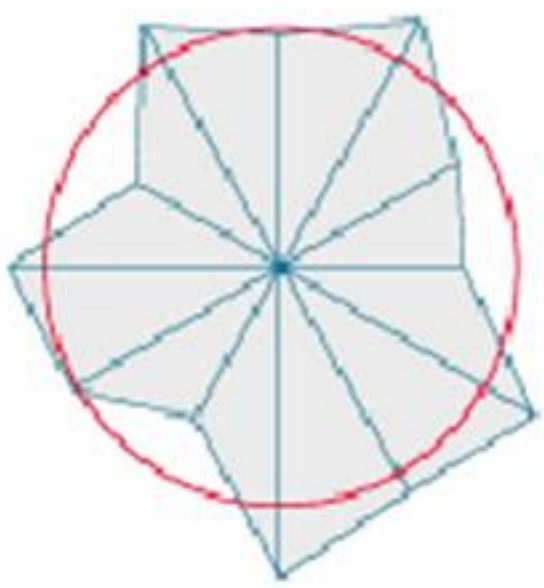
- **SPECint_rate_base2000**: Geometrický průměr 12 normalizovaných poměrů pro **propustnost**, kompilováno s „konzervativní" optimalizací

Výsledky SPECint_base2000

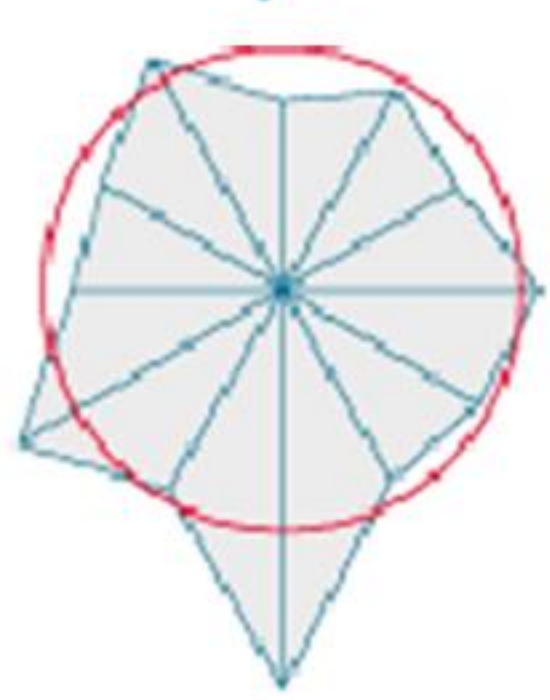




Mips/IRIX
R12000@ 400MHz

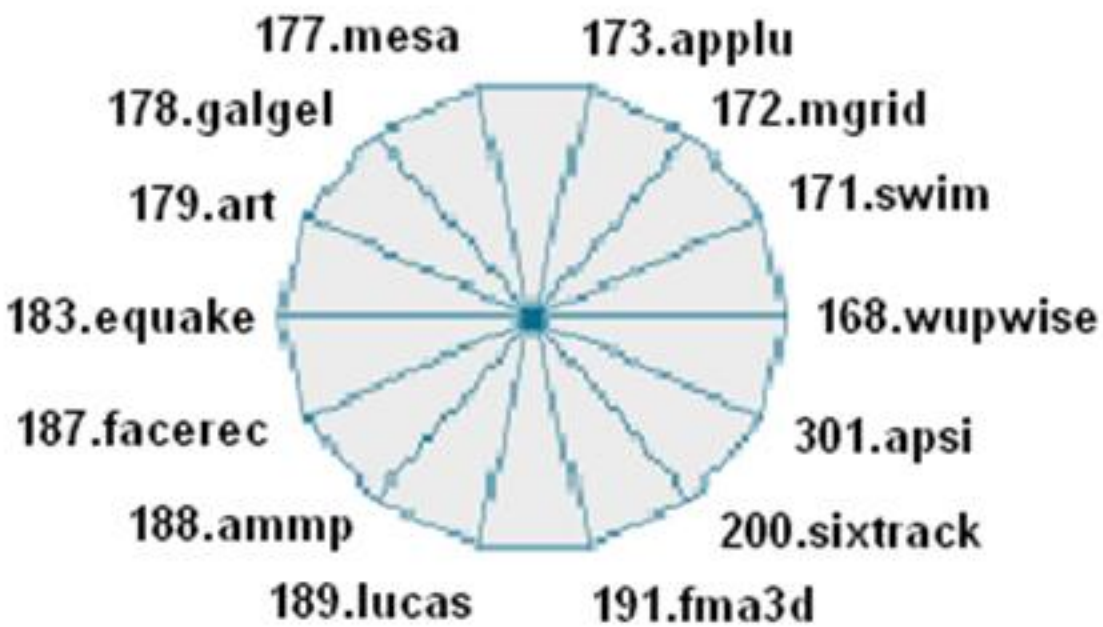


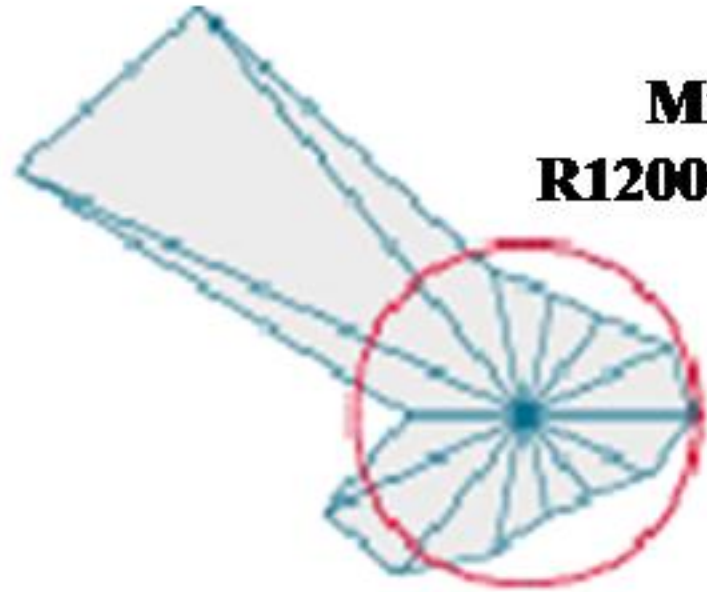
Alpha/Tru64
21264 @ 667 MHz



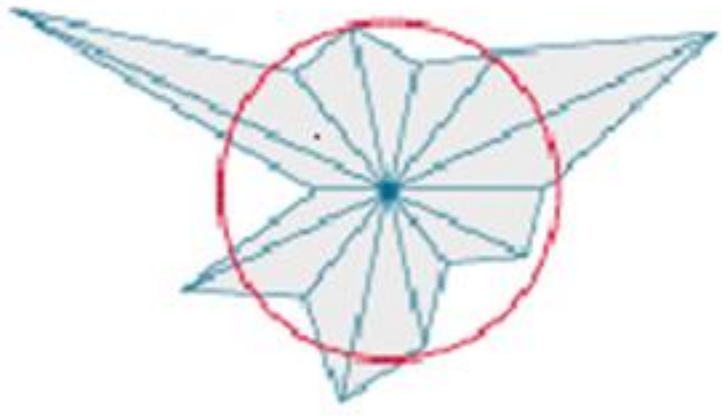
Intel/NT 4.0
PIII @ 733 MHz

Výsledky SPECfp_base2000





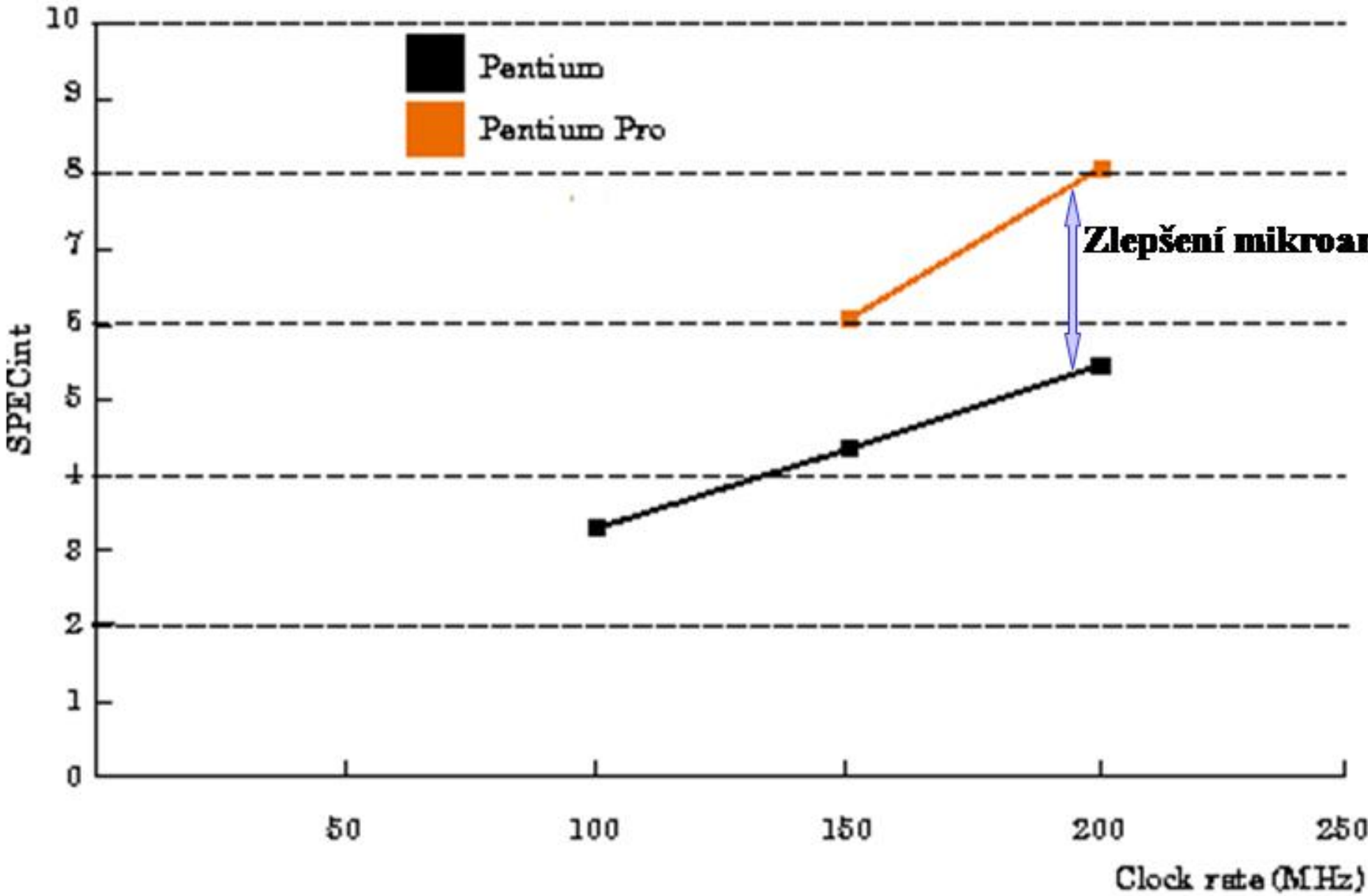
Mips/IRIX
R12000@ 400MHz



Alpha/Tru64
21264 @ 667 MHz

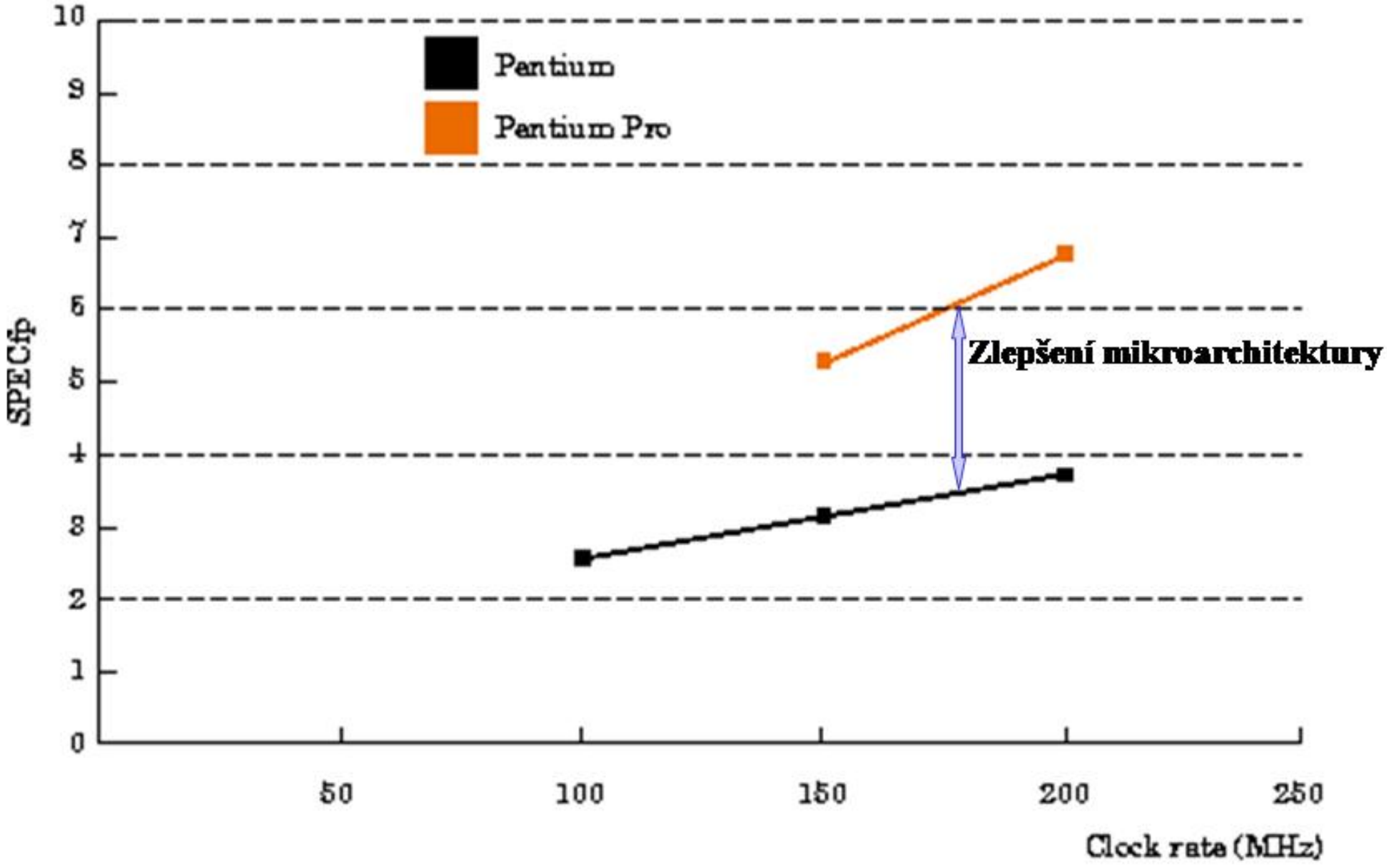
Intel/NT 4.0
PIII @ 733 MHz

Hodnocení SPECint95



CPU time = IC * CPI * clock cycle

Hodnocení SPECfp95



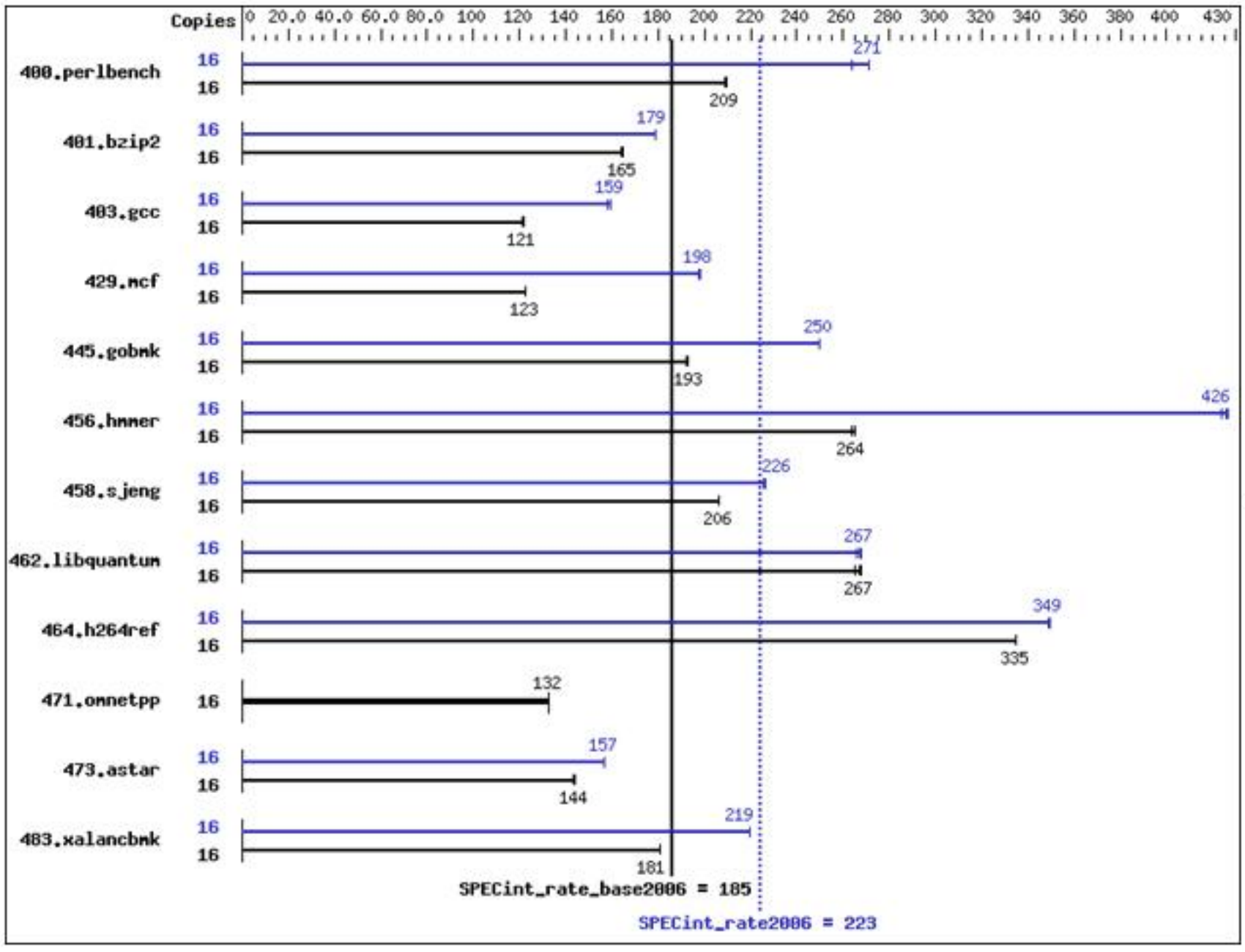
Benchmark SPECint2006

SPECint2006 test obsahuje 12 benchmarkových programů, navržených exkluzivně pro test výkonu systému v integer oblasti.

Benchmark	Language	Category
400.perlbench	C	Programming Language
401.bzip2	C	Compression
403.gcc	C	Compiler
429.mcf	C	Combinatorial Optimization
445.gobmk	C	Artificial Intelligence
456.hmmer	C	Search Gene Sequence
458.sjeng	C	Artificial Intelligence
462.libquantum	C	Physics / Quantum Computing
464.h264ref	C	Video Compression
471.omnetpp	C++	Discrete Event Simulation
473.astar	C++	Path-finding Algorithms
483.xalancbmk	C++	XML Processing

SPEC® CINT2006 Result

Dell Inc., PowerEdge M905 (AMD Opteron 8378, 2.40 GHz)



Hardware
CPU Name: AMD Opteron 8378
CPU Characteristics:
CPU MHz: 2400
FPU: Integrated
CPU(s) enabled: 16 cores, 4 chips, 4 cores/chip
CPU(s) orderable: 4 chips
Primary Cache: 64 KB I + 64 KB D on chip per core
Secondary Cache: 512 KB I+D on chip per core
L3 Cache: 6 MB I+D on chip per chip
Other Cache: None
Memory: 64 GB (16 x 4 GB DDR2-800)
Disk Subsystem: 1 x 73 GB 10000 RPM SAS
Other Hardware: None

Software
Operating System: SUSE Linux Enterprise Server 10 (x86_64) SP2, Kernel 2.6.16.60-0.21-smp
Compiler: PGI Server Complete Version 7.2
PathScale Compiler Suite Version 3.2
Auto Parallel: No
File System: ReiserFS
System State: Run level 3 (multi-user)
Base Pointers: 32/64-bit
Peak Pointers: 32/64-bit
Other Software: binutils 2.18
32-bit and 64-bit libhugetlbfs libraries
SmartHeap 8.1 32-bit Library for Linux

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!!]

Závěr

- Přesné měření výkonu počítače je vlastně obtížné !
- Benchmarky musí respektovat předpokládanou zátěž. Jedině tak jsou použitelné.
- 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.
  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.
- 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!

Instruction Set Architectures

Formáty instrukcí MIPS

Reprezentace instrukcí

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)

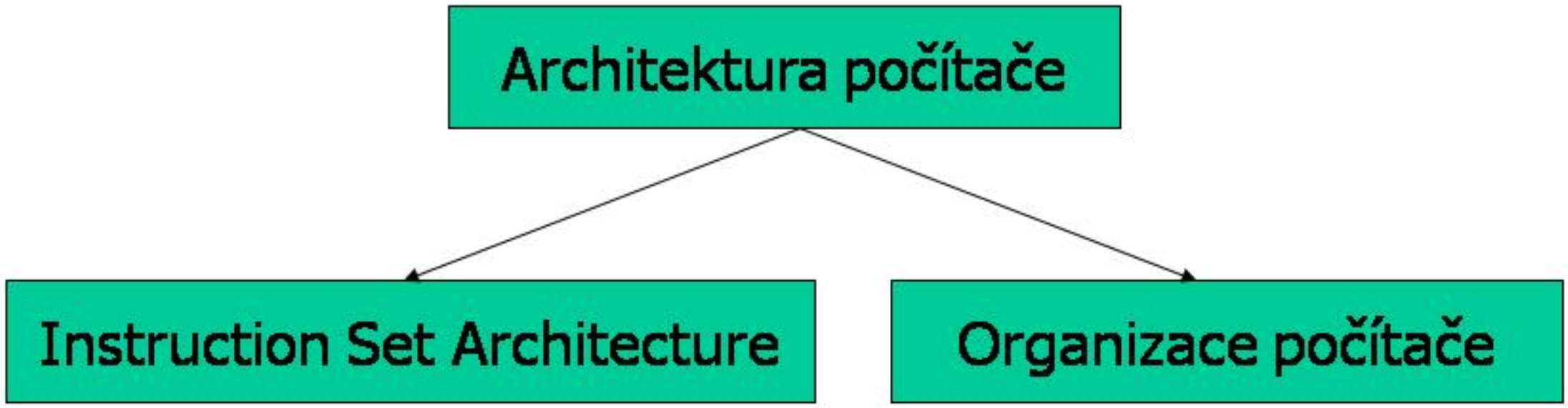
- Co je "architektura počítače?"
  - "… architekturou rozumíme strukturu modulů, jak jsou organizovány v počítačovém systému …" (H. Stone, 1987)
  - "Architektura počítače je interface mezi strojem a softwarem" (Andris Padges, architekt IBM 360/370)

Definice

- Co je "architektura počítače?"
  - Amdahl, Blaauw, Brooks (IBM 1964):

    "... the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine".

Dvě hlavní oblasti



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

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

- Šest hlavních typů instrukcí
  - Čtení/zápis (Load/Store)
  - Aritmetické
  - Skoky a větvení
  - Floating Point operace
  - Správa paměti (Memory Management)
  - Speciální

Registry (zde 35)

R0 - R31
PC
HI
LO

- Tři formáty instrukcí – všechny o šíři 32 bitů ...

opcode	rs	rt	rd	sa	funct

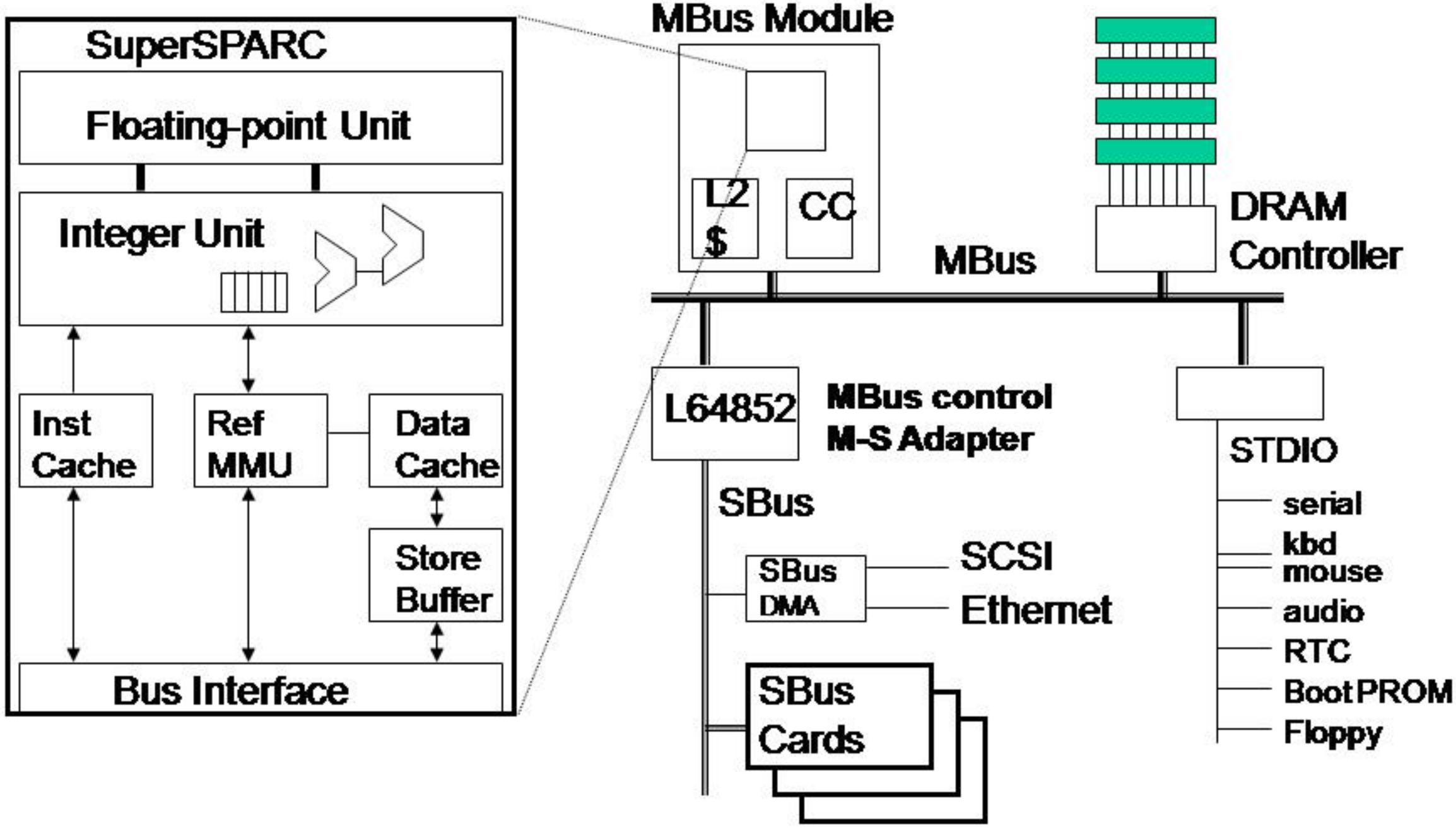
opcode	rs	rt	immediate value

opcode	jump or branch target

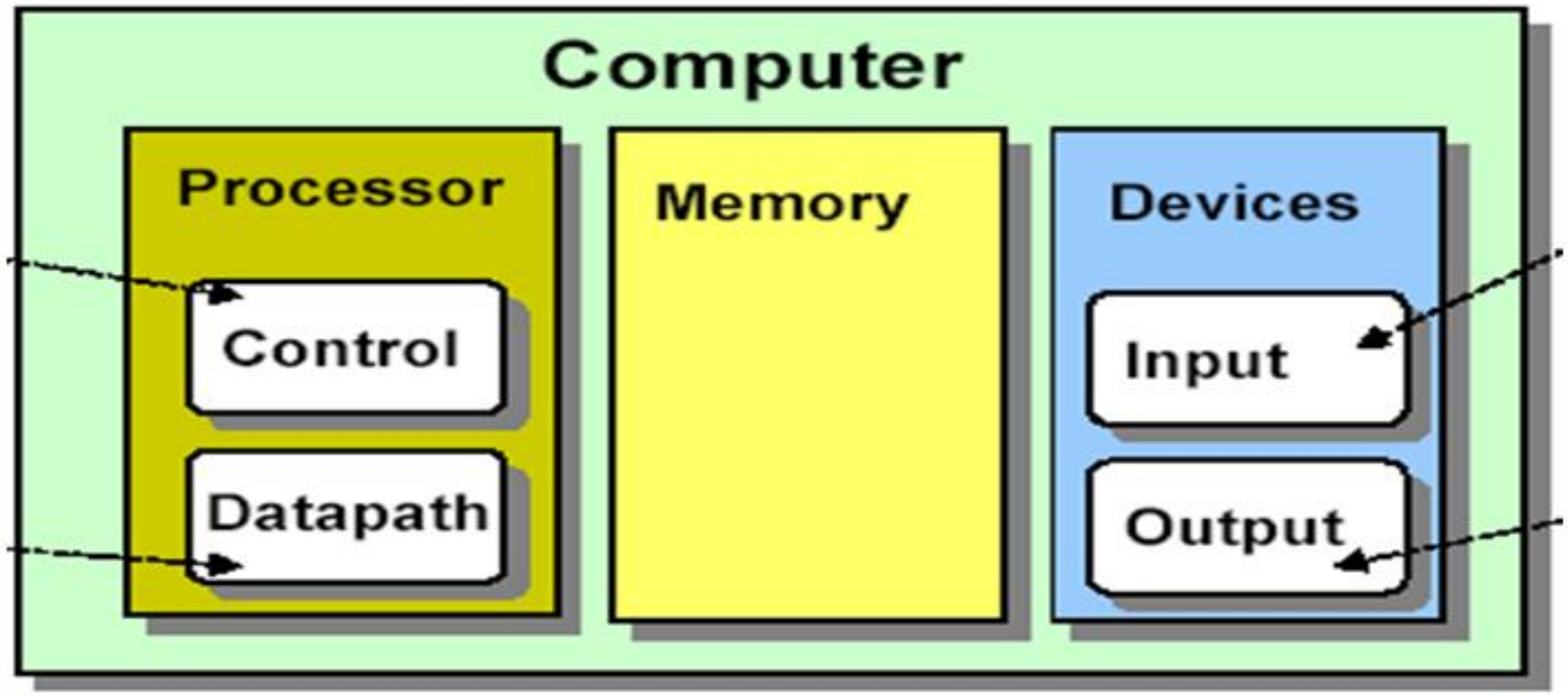
Organizace počítače

- Organizace počítače 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)

Příklad: TI SuperSPARC



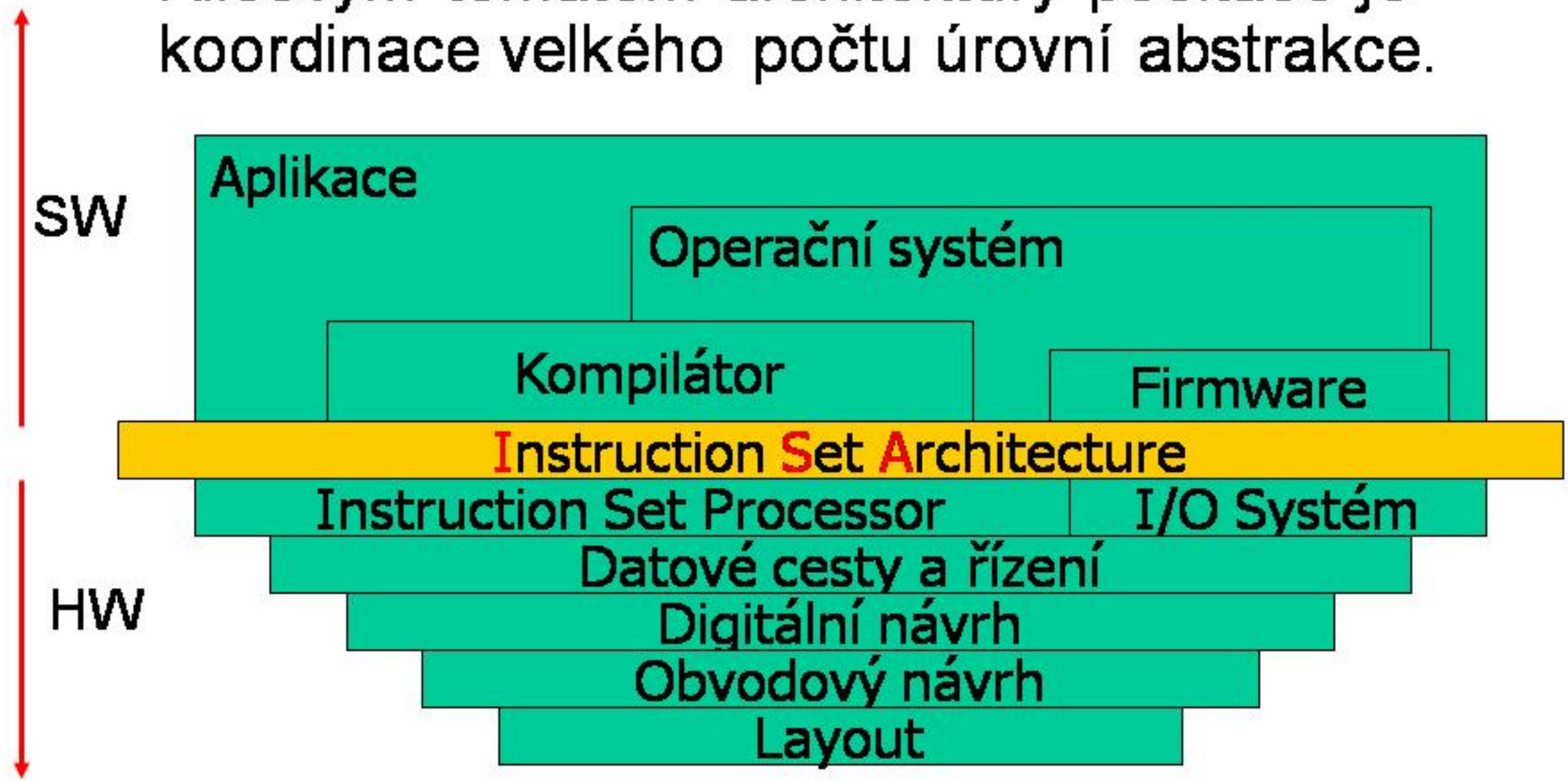
Organizace počítače - obecně



- Každá část každého počítače může být zařazena do jedné z pěti skupin.
- Návrh každého systému je podřízen požadavkům na cenu, velikost a schopnosti každé komponenty. Typický návrh uvažuje: procesor 25% celkových nákladů, paměť 25% celkových nákladů a 50% celkových nákladů ostatní zařízení.

Klíčové téma: abstrakce

- Klíčovým tématem architektury počítače je koordinace velkého počtu úrovní abstrakce.



Jak se budeme architekturou zabývat?

- Architekturou se budeme zabývat od základů, směrem zdola nahoru.
- Budeme se zabývat návrhem hardware, včetně návrhu některých komponent (použití HDL).
- Cílem je studie architektury, jako směs teorie a také praktických příkladů. Budeme se zabývat návrhem, simulací a verifikací.

Zvolený přístup

- Klasický návrh mikroprocesoru, který je použit k ilustraci je MIPS (navržen autory učebnice - D. A. Patterson and J. L.Hennessy: Computer Organization and Design: The Hardware Software Interface, 4rd Edition, 2009).
- Vedle této základní architektury se také částečně zmíníme o dalších typech architektur jako např. PowerPC a Intel IA-32/x86.
- MIPS je relativně stará architektura – je jednoduchá a vyhýbá se mnoha složitostem, které se v současných systémech používají (což je velmi vhodné pro výuku).
- Tento přístup dovolí přiblížit různé přístupy k návrhu a řešení problémů.

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

Programovací jazyky

Jazyky vyšší úrovně

```
temp   = v[k];
v[k]   = v[k+1];
v[k+1] = temp;
```

```
TEMP = V(K)
V(K) = V(K+1)
V(K+1) = TEMP
```

Kompilátor C/Java

Kompilátor Fortranu

Asembler

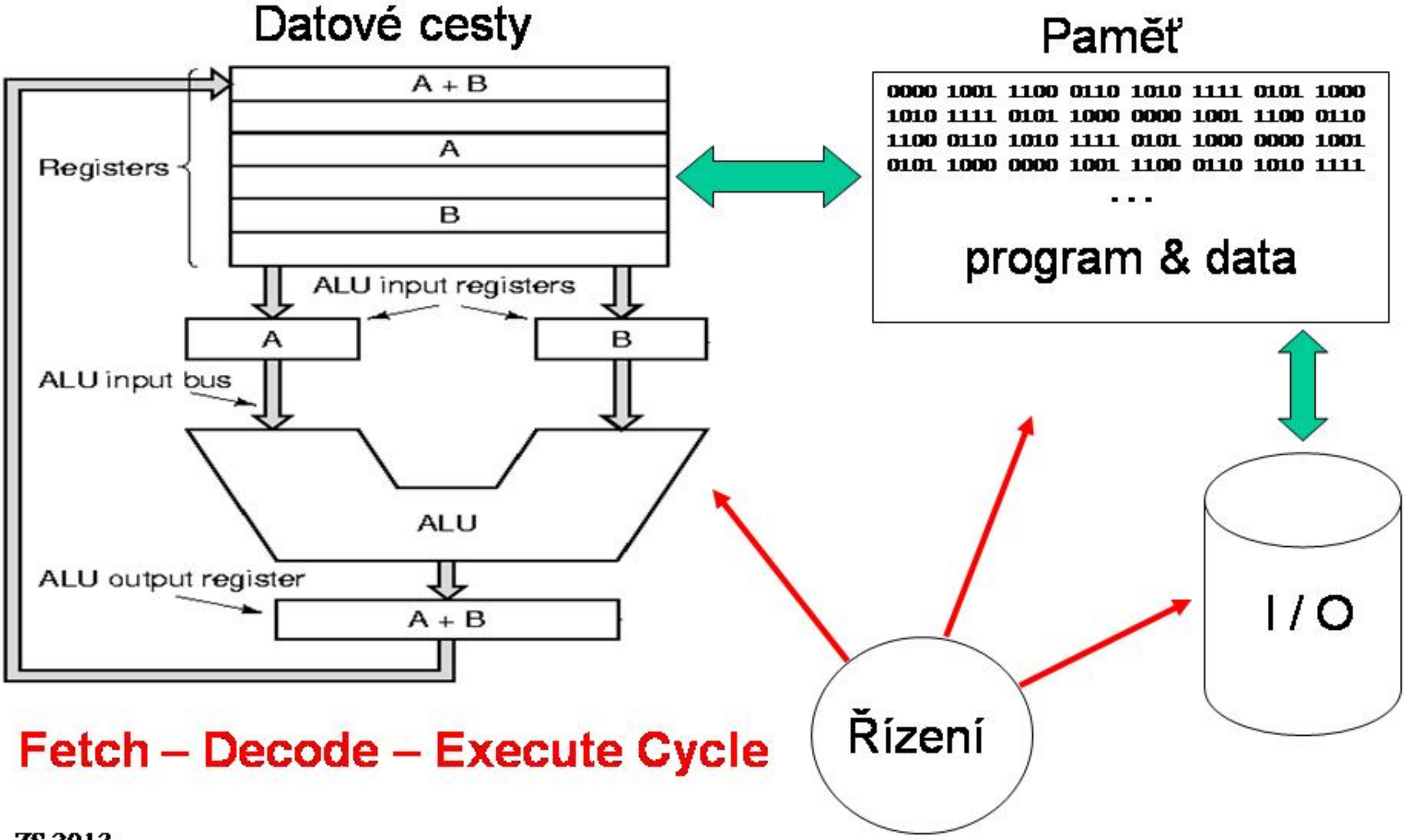
```
lw  $to,   0($2)
lw  $t1,   4($2)
sw  $t1,   0($2)
sw  $t0,   4($2)
```

Assembler MIPS

Strojní jazyk

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Výpočet programu



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)

Instrukce MIPS

- Jednoduché instrukce
- **Zásada_1: Jednoduchost podporuje regularitu**

```
add   a,   b,   c        # a = b + c;
```

Právě 3 operandy (registry)

poznámka

```
# a = b + c + d + e;
add  a,  b,  c
add  a,  a,  d
add  a,  a,  e
```

- Tyto instrukce jsou **symbolickou** reprezentací toho, čemu MIPS „rozumí“

Příklad

Kompilace přiřazení v C do MIPS

f = (g + h) – (i + j);

```
Add  t0,  g,  h      # dočasná proměnná t0 obsahuje g+h
Add  t1,  i,  j      # dočasná proměnná t1 obsahuje i+j
Sub  f,   t0, t1     # do f je uložen výsledek
```

Operandy

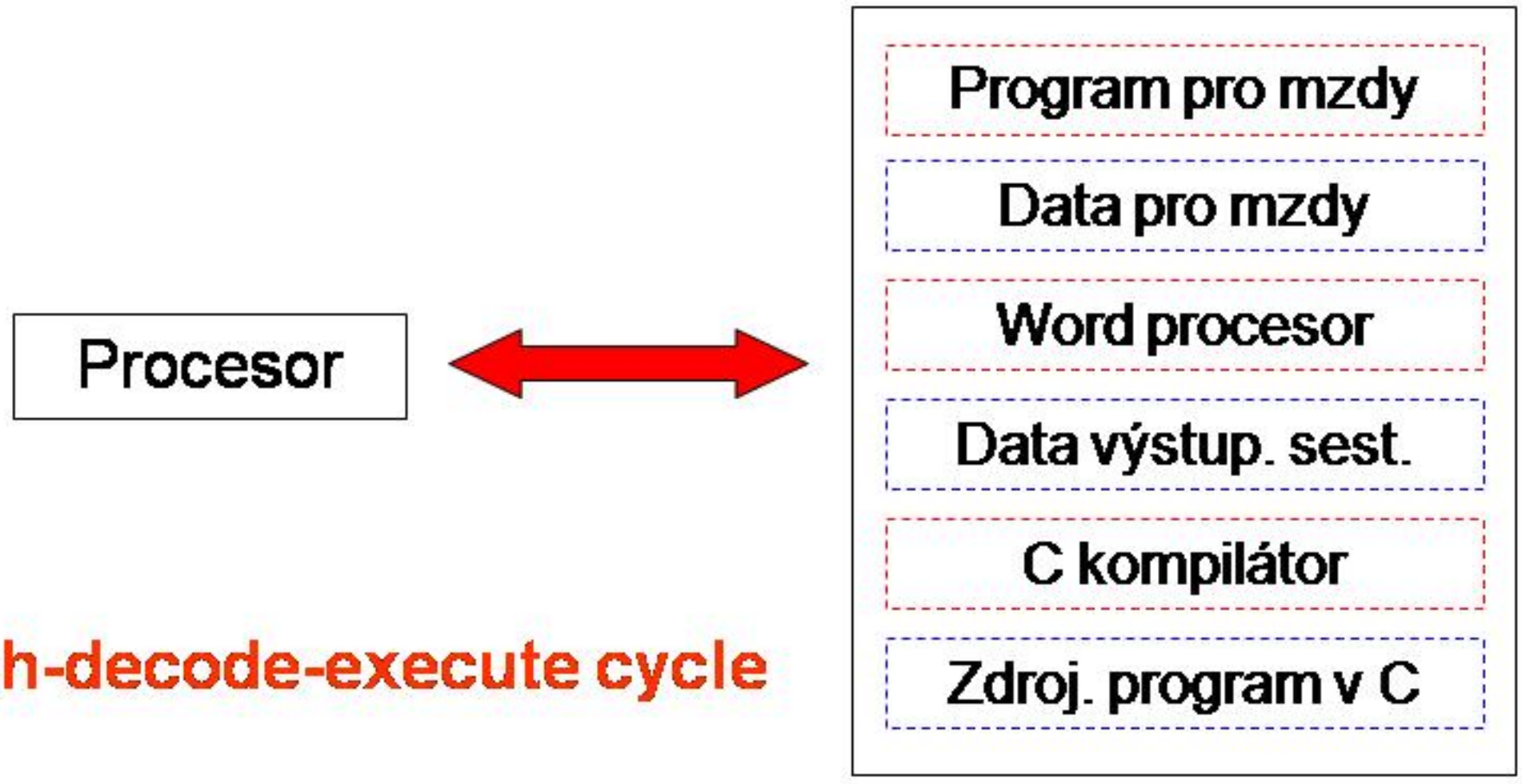
- Operand nemůže být libovolná proměnná (jako v C)
  - princip KISS – odstraňuje nedostatky CISC
- Registry (Keep It Small and Simple)
  - Omezený počet (32 32-bitových registrů u MIPS)
    - **Zásada_2: Menší je rychlejší**
  - Pojmenování: čísla nebo *jména*
    - **$8 - $15 => $t0 - $t7 (vztahuje 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

Koncepce „programu v paměti“

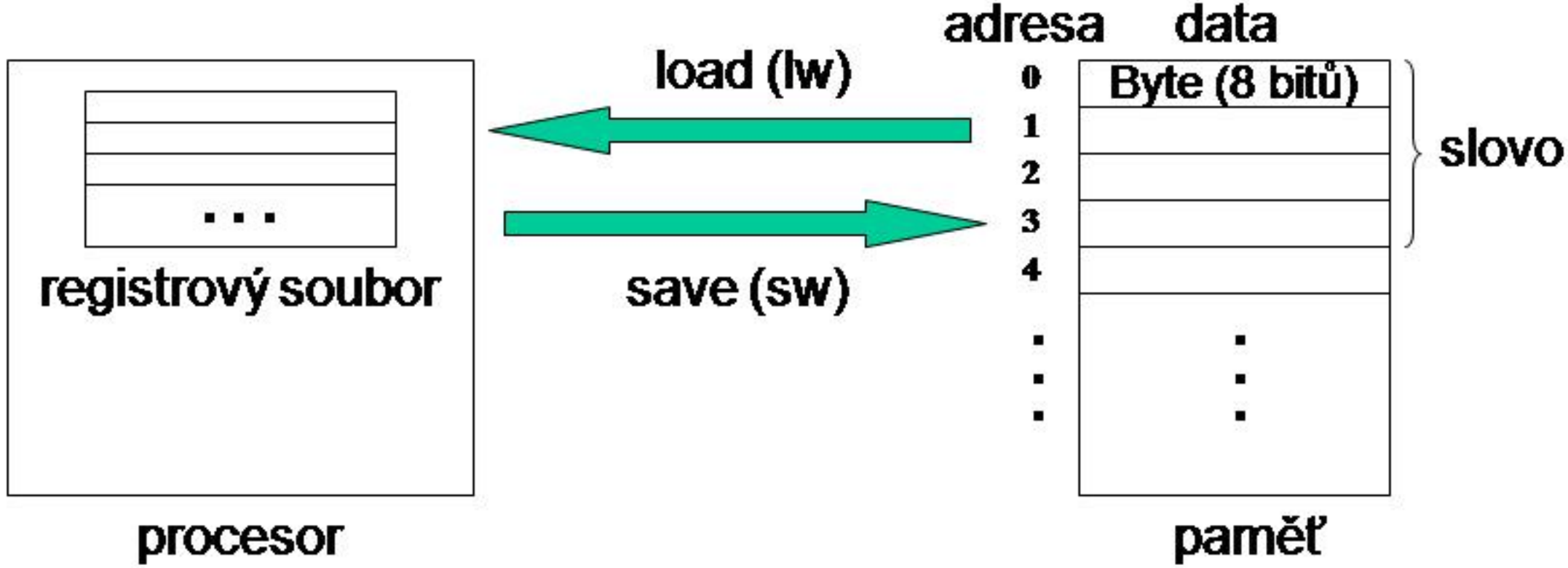
- Instrukce reprezentují 32-bitová čísla
- Programy jsou uloženy v paměti
  - Lze je číst i zapisovat stejně jako čísla



Fetch-decode-execute cycle

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



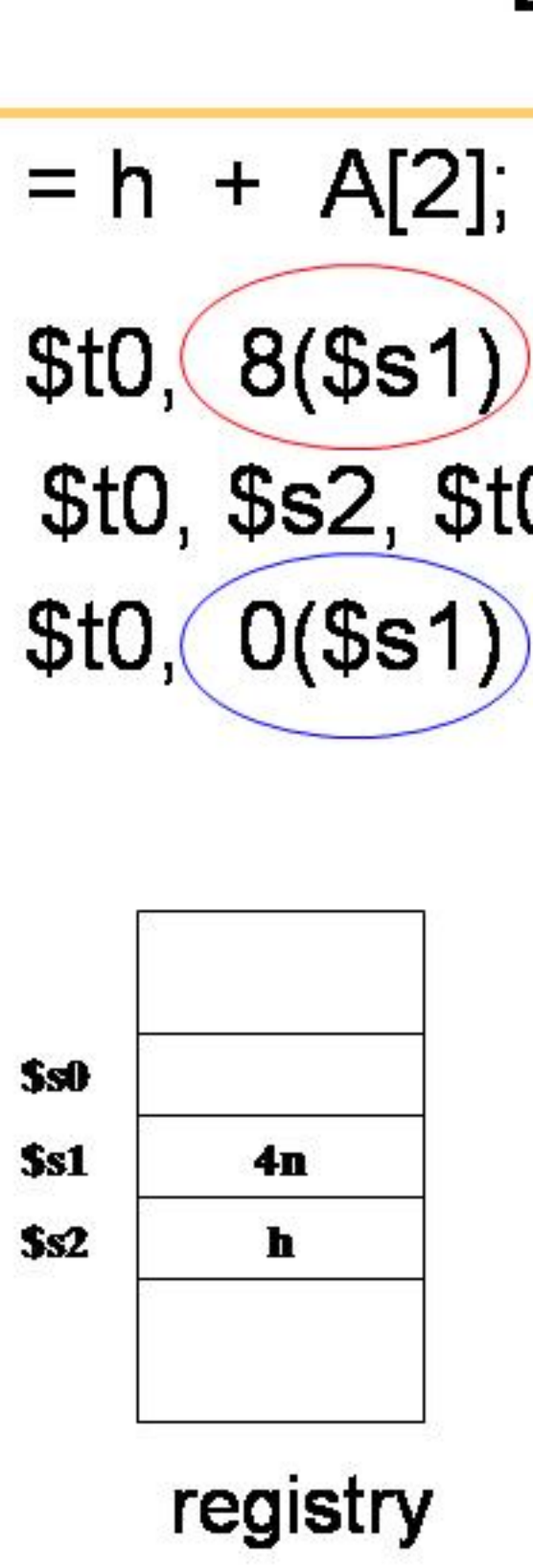
Paměťový model

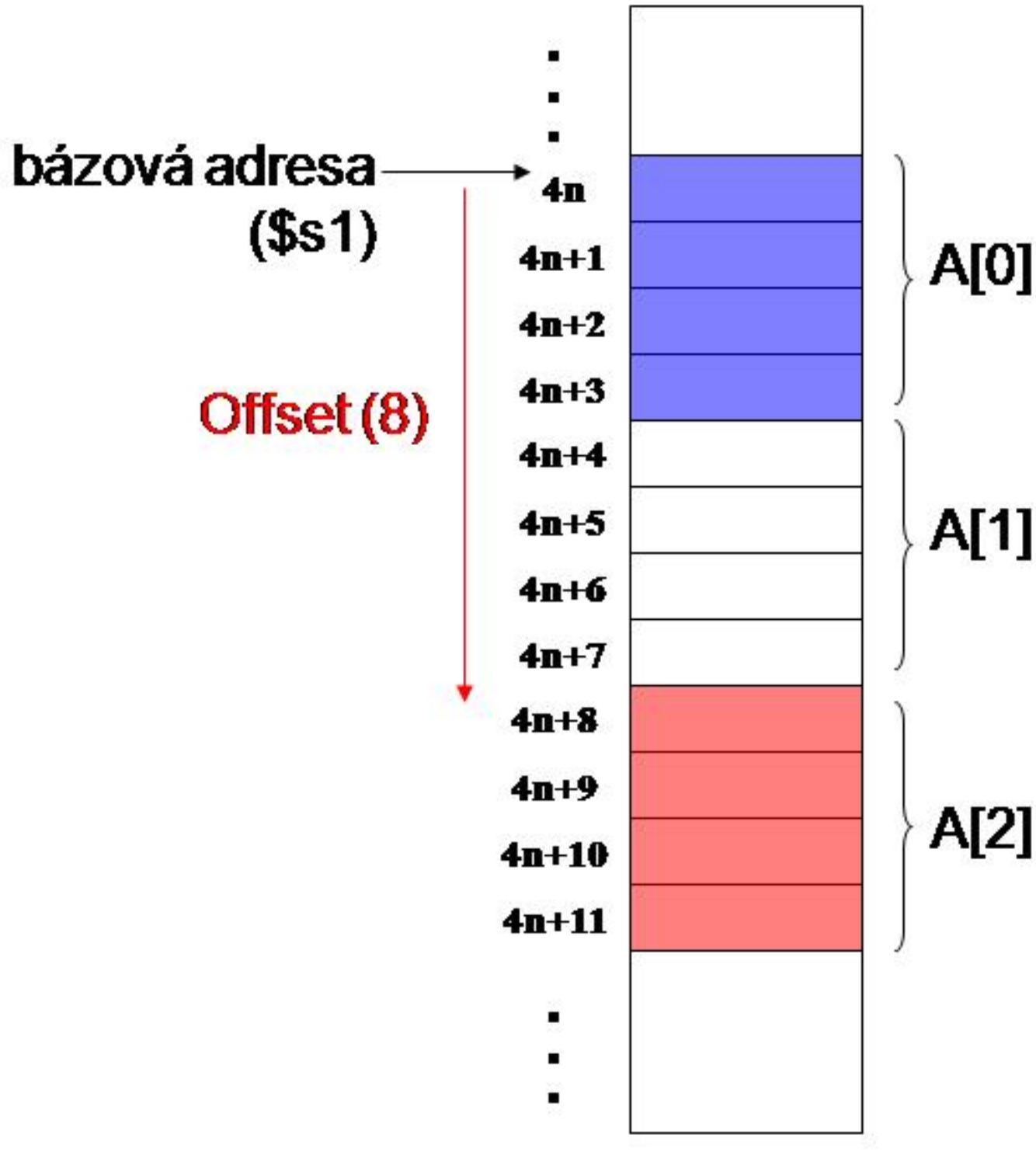
- Paměť je adresovatelná po bytech (1 byte = 8 bitů)
- Load/Store - jediný přístup k datům v paměti
- Jednotka pro přenos: *word* (4 byty)
  - M[0], M[4], M[4n], …., M[4,294,967,292]
- Slova musí být **zarovnána** !!!
  - **Slovo začíná na adresách 0, 4, … 4n**
- Adresa je dlouhá 32 bitů
  - 2^{32} bytů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

Load / Store

```
A[0] = h  +  A[2];

lw   $t0, 8($s1)
add  $t0, $s2, $t0
sw   $t0, 0($s1)
```

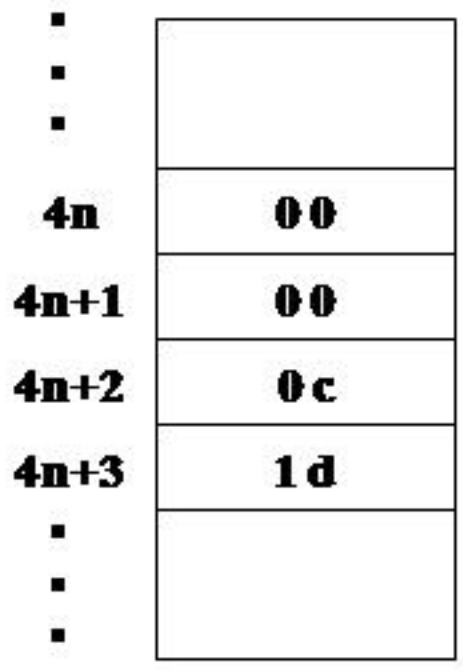




„Big“ a „Little“ Endian

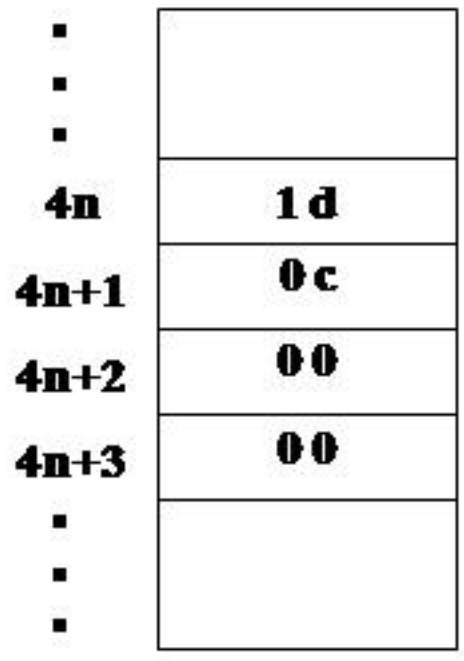
$$(3101)_{10} = 12 * 16^2 + 1 * 16^1 + 13 * 16^0$$

$$= (00\ \ 00\ 0c\ 1d)_{16}$$



big endian

(MIPS R3000)



little endian

(DEC, Intel)

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

Strojový jazyk

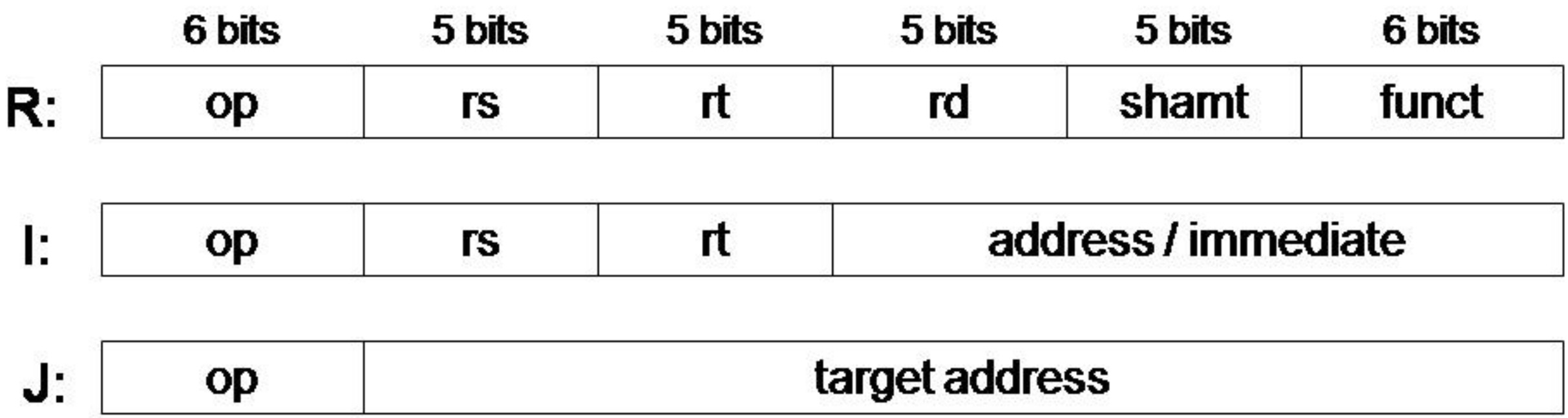
- Všechny instrukce mají stejnou délku (32 bitů)
- Zásada_3: Dobrý návrh vyžaduje dobré kompromisy
  - Stejná délka nebo stejný formát
- Tři různé formáty instrukcí
  - R: formát aritmetických instrukcí
  - I: formát pro přesuny dat, větvení, immediate
  - J: formát skokové instrukce
- add $t0, $s1,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

```
10101101001010000000010010110000
00000010010010000100000000100000
10001101001010000000010010110000
```

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

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

Formáty instrukcí



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.“)

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

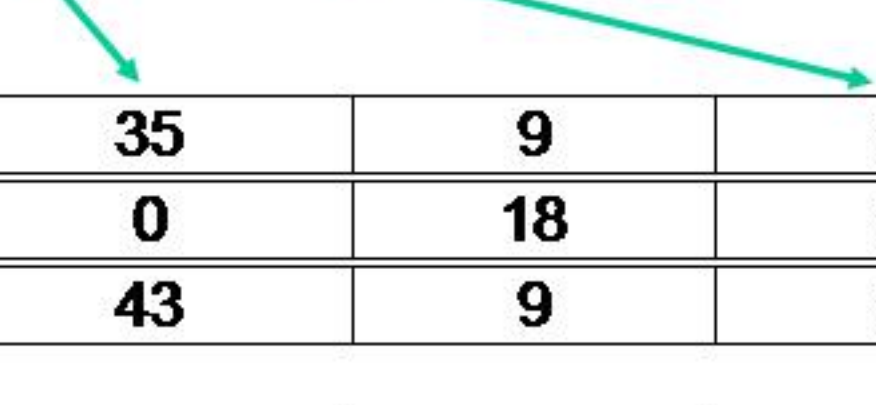
Příklad

```
A[300] = h + A[300];      /*  $t1 <= base of array A;  $s2 <= h  */
```

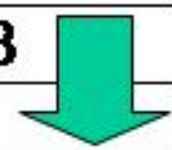
Kompilátor

```
lw      $t0, 1200($t1)      # dočasný registr $t0 naplněn A[300]
add     $t0,  $s2,  $t0      # dočasný registr $t0 naplní se h +A[300]
sw      $t0, 1200($t1)      # uloží h + A[300] zpět do A[300]
```

Assembler



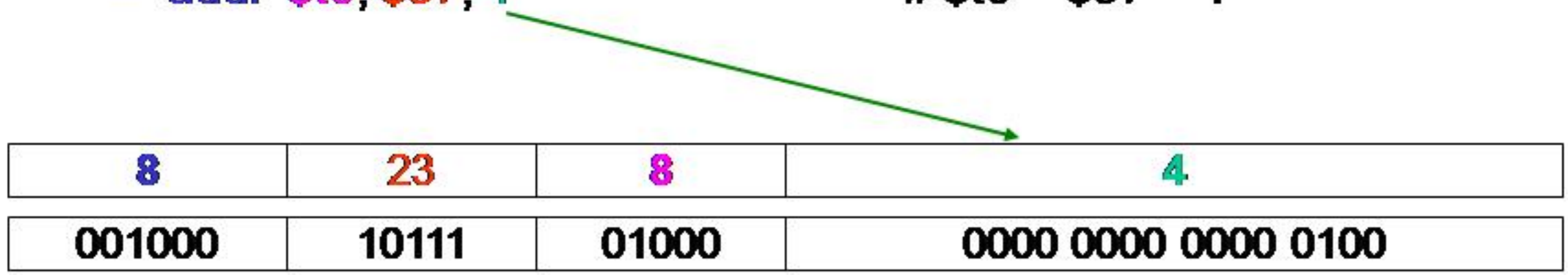
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		



100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

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



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

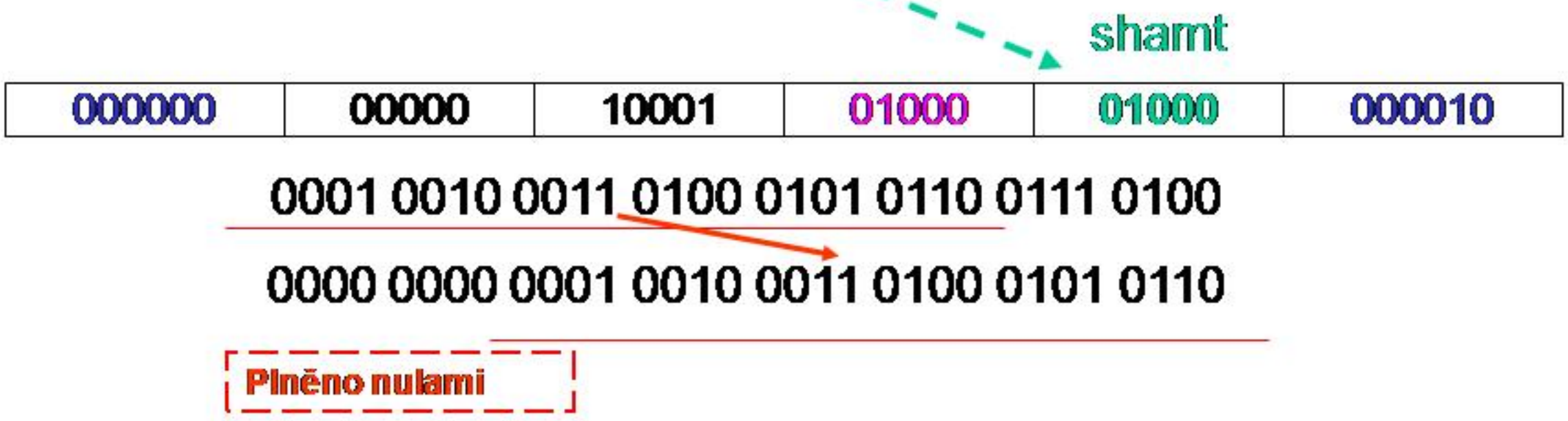
Logické instrukce

- 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

- 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)



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 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

Nové – instrukce větvení programu

- Podmíněné skoky
  - If-then
  - If-then-else
- Smyčky
  - While
  - Do while
  - For
- Nerovnosti
- Přepínače

Podmíněné skoky

- Instrukce rozhodování
- Skok při rovnosti
  - **beq register1, register2, *destination_address***
- Skok při nerovnosti
  - **bne register1, register2, *destination_address***
- Příklad: beq $s3, $s4, 20

6 bitů	5 bitů	5 bitů	16 bitů
4	19	20	5
000100	10011	10100	0000 0000 0000 0101

Návěští

- Není třeba vypočítávat adresu cíle skoku

```
        if (i = = j) go to L1;
        f = g + h;
L1:     f = f – i;
```

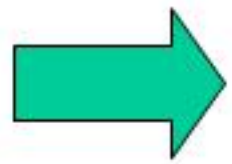
```
f => $s0
g => $s1
h => $s2
i => $s3
j => $s4
```

```
     (4000) beq $s3, $s4, L1      # if i equals j go to L1
     (4004) add $s0, $s1, $s2     # f = g + h
L1:  (4008) sub $s0, $s0, $s3     # f = f - i
```

L1 odpovídá adrese instrukce odčítání

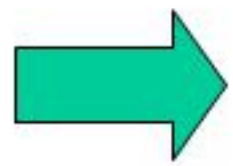
Příkaz if

```
if (condition)
    clause1;
else
    clause2;
```



```
if (condition) goto L1;
      clause2;
      goto L2;
L1: clause1;
L2:
```

```
if (i = = j)
    f = g + h;
else
    f = g - h;
```



```
          beq $3, $4, True
          sub $0, $s1, $s2
          j False
True:   add $s0, $s1, $s2
False:
```

Smyčky

```
Loop: g = g + A[i];
      i = i + j;
      if (i != h) goto Loop;
```

```
g: $s1
h: $s2
i:  $s3
j:  $s4
Base of A: $s5
```

```
Loop: add $t1, $s3 $s3      #  $t1 = 2 * i
      add $t1, $t1, $t1     #  $t1 = 4 * I
      add $t1, $t1, $5      #  $t1=address of A[i]
      lw   $t0, 0($t1)      #  $t0 = A[i]
      add $s1, $s1, $t0     #  g = g + A[i]
      add $s3, $s3, $s4     #  i = i + j
      bne $s3, $s2, Loop    # go to Loop if i != h
```

Základní blok

Smyčka while

```
while (save[i] = = k)
    i = i + j;
```

```
# i: $s3; j: $s4; k: $s5; base of save: $s6

Loop: add $t1, $s3, $s3     # $t1 = 2 * i
      add $t1, $t1, $t1     # $t1 = 4 * i
      add $t1, $t1, $s6     # $t1 = address of save[i]
      lw  $t0, 0($t1)       # $t0 = save[i]
      bne $t0, $s5, Exit    # go to Exit if save[i] != k
      add $s3, $s3, $s4     # i = i +j
      j   Loop              # go to Loop
Exit:
```

Počet provedených instrukcí if save[i + m * j] does not equal k pro m = 10 a does equal k for $0 \leq m \leq 9$ je roven 10 * 7 + 5 = 75

Optimalizace

6 Instrukcí

```
        add $t1, $s3, $s3      # Temp reg $t1 = 2 * i
        add $t1, $t1, $t1      # Temp reg $t1 = 4 * i
        add $t1, $t1, $s6      # $t1 = address of save[i]
        lw $t0, 0($t1)         # Temp reg $t0 = save[i]
        bne $t0, $s5, Exit     # go to Exit if save[i] ≠ k
Loop:   add $s3, $s3, $s4      # i = i + j
        add $t1, $s3, $s3      # Temp reg $t1 = 2 * i
        add $t1, $t1, $t1      # Temp reg $t1 = 4 * i
        add $t1, $t1, $s6      # $t1 = address of save[i]
        lw $t0, 0($t1)         # Temp reg $t0 = save[i]
        beq $t0, $s5, Loop     # go to Loop if save[i] = k
Exit:
```

Počet instrukcí provedených touto novou formou smyčky je roven $5 + 10 * 6 = 65$ ➔ Úspora = 1.15 = 75/65. Jestliže 4 * i je vypočítáno před smyčkou, lze dosáhnout další úspory.

Smyčka Do-While

```
do {
   g = g + A[i];
   i = i + j;
} while (i != h);
```

```
L1:  g = g + A[i];
     i = i + j;
     if (i != h) goto L1
```

```
g: $s1
h: $s2
i:  $s3
j:  $s4
Base of A: $s5
```

```
L1:  sll  $t1, $s3, 2      # $t1 = 4*i
     add $t1, $t1, $s5     #  $t1 = addr of A
     lw   $t1, 0($t1)      #  $t1 = A[i]
     add $s1, $s1, $t1     #  g = g + A[i]
     add $s3, $s3, $s4     #  i  = i + j
     bne $s3, $s2, L1      #  go to L1 if i != h
```

- Podmíněný skok je klíčová instrukce rozhodování

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
h: $s1
```

```
slt     $t0, $s0, $s1
bne     $t0, $0, Less
```

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

```
slti    $t0, $s0, 1     # $t0 = 1 if g < 1
beq     $t0, $0, Loop   # goto Loop if g >= 1
```

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

Relativní podmínky

- == != < <= > >=
  - MIPS přímo nepodporuje poslední čtyři
  - Kompilátory používají **slt, beq, bne, $zero, and $at**
- Pseudoinstrukce
  - blt $t1, $t2, L # if ($t1 < $t2) go to L

```
slt  $at, $t1, $t2
bne $at, $zero, L
```

  - ble $t1, $t2, L # if ($t1 <= $t2) go to L

```
slt  $at, $t2, $t1
beq $at, $zero, L
```

  - bgt $t1, $t2, L # if ($t1 > $t2) go to L

```
slt  $at, $t2, $t1
bne $at, $zero, L
```

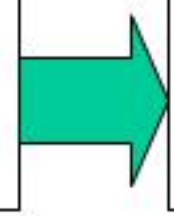
  - bge $t1, $t2, L # if ($t1 >= $t2) go to L

```
slt  $at, $t1, $t2
beq $at, $zero, L
```

Přepínač jazyka C

```
switch (k) {
  case 0: f = i + j; break;
  case 1: f = g + h; break;
  case 2: f = g - h; break;
  case 3: f = i - j; break;
}
```

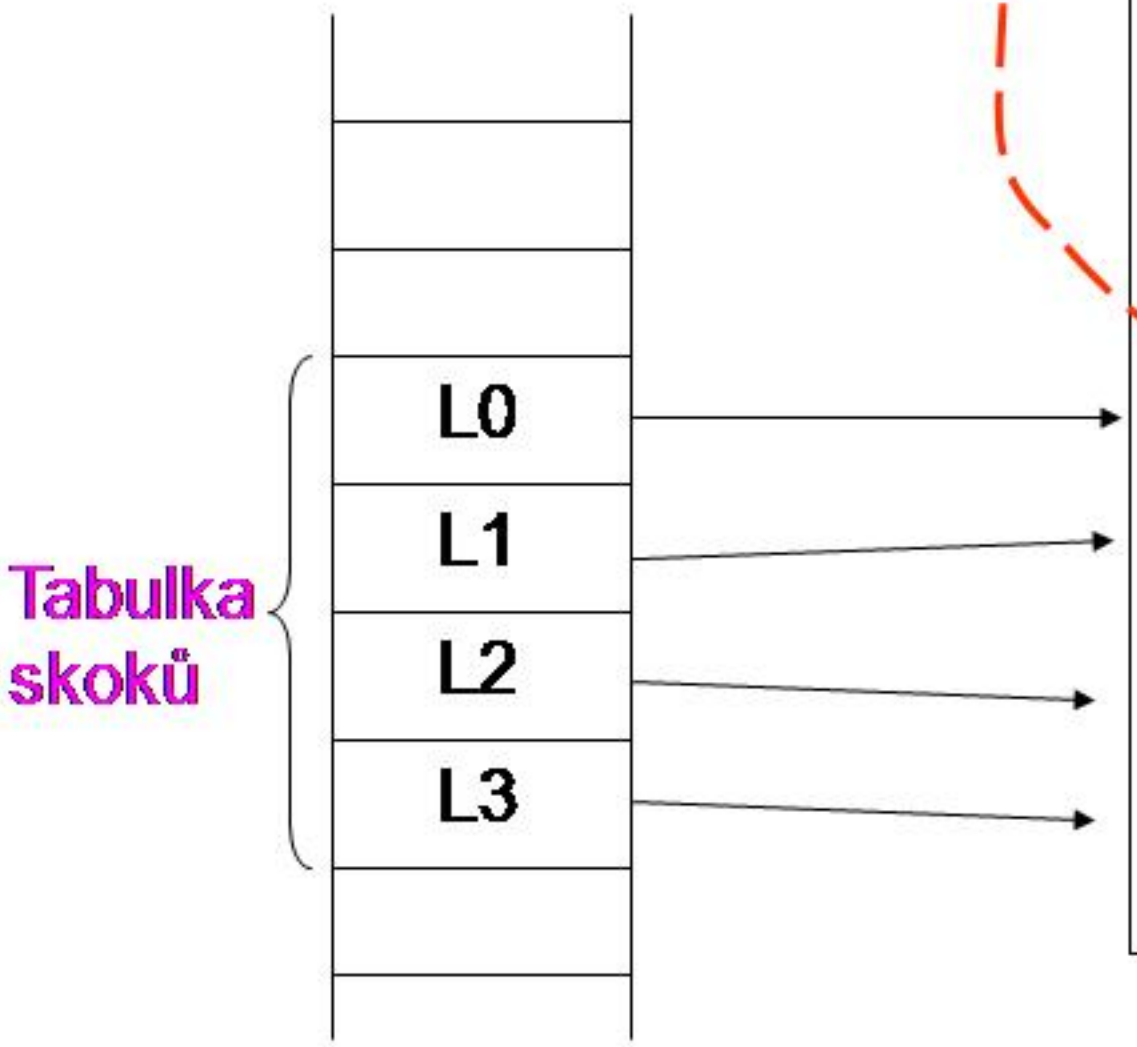
```
if (k==0) f = i + j;
  else if (k==1) f = g + h;
    else if (k==2) f = g - h;
      else if (k==3) f = i - j;
```



```
#  f: $s0; g: $s1; h: $s2; i: $s3; j: $s4; k:$s5
     bne   $s5, $0, L1       # branch k != 0
     add   $s0, $s3, $s4     # f = i + j
     j     Exit              # end of case
L1:  addi  $t0, $s5, -1      # $t0 = k - 1
     bne   $t0, $0, L2       # branch k != 1
     add   $s0, $s1, $s2     # f = g + h
     j     Exit              # end of case
L2:  addi  $t0, $s5, -2      # $t0 = k - 2
     bne   $t0, $0, L3       # branch k != 2
     sub   $s0, $s1, $s2     # f = g - h
     j     Exit              # end of case
L3:  addi  $t0, $s5, -3      # $t0 = k - 3
     bne   $t0, $0, Exit     # branch k != 3
     sub   $s0, $s3, $s4     # f = i - j
Exit:
```

Tabulky skoků

- Jump register instruction
  - jr <register>
  - nepodmíněný skok na adresu obsaženou v registru




```
#  f: $s0; g: $s1; h: $s2; i: $s3; j: $s4; k:$s5
#  $t2 = 4; $t4 = base address of JT
      slt    $t3, $s5, $zero     # test k < 0
      bne    $t3, $zero, Exit    # if so, exit
      slt    $t3, $s5, $t2       # test k < 4
      beq    $t3, $zero, Exit    # if so, exit
      add    $t1, $s5, $5        # $t1 = 2*k
      add    $t1, $t1, $t1       # $t1 = 4*k
      add    $t1, $t1, $t4       # $t1 = &JT[k]
      lw     $t0, 0($t1)         # $t0 = JT[k]
      jr     $t0                 # jump register
L0:   add    $s0, $s3, $s4       # k == 0
      j      Exit                # break
L1:   add    $s0, $1, $s2        # k == 1
      j      Exit                # break
L2:   sub    $s0, $s1, $s2       # k == 2
      j      Exit                # break
L3:   sub    $s0, $s3, $s4       # k == 3
Exit:
```

Závěr

- ISA se navrhuje tak, aby dlouho přežila trendy technologie
- *V jednoduchosti je síla* (*Simpler is better* - KISS)
  - Jednoduché instrukce (omezují nedostatky CISC)
- *Malé je rychlejší*
  - Malý počet registrů nahrazuje proměnné v C
- Paměťový model
  - Adresace po bytech, zarovnání, lineární pole
- Instrukční formát MIPS – 32 bitů

Závěr (pokrač.)

- Assembler: výsledek = prvý operand (vlevo)
- Strojový jazyk: výsledek = poslední operand
- Tři formáty instrukcí MIPS : R (aritmetické)
  I (immediate)
  J (skoky)
- Rozhodovací instrukce – používají *jump* (goto)
- Zlepšení výkonu – „loop unrolling“

rozvinutí smyček

Úvod do organizace počítače

Podpora procedur
& reprezentace čísel

Datové typy a adresování
Pointery & pole
Programy pro MIPS

Přehled

- Mapa paměti
- Funkce jazyka C
- Instrukční podpora procedur (MIPS)
- Stack
- Procedury - konvence
- Manuální kompilace
- Závěr

Přehled (pokrač.)

- Datové typy
  - Aplikace / požadavky HLL (High Level Languages)
  - Podpora HW (data a instrukce)
- Datové typy procesoru MIPS
  - Podpora pro operace s byty a řetězci
  - Adresní módy (adresní režimy)
    - Data
    - Instrukce
  - Velké konstanty a dlouhé adresy
  - Kód SPIM (freeware simulátor)

Přehled (pokrač.)

- Pointery (adresy) a hodnoty
- Předávání argumentů
- „Doba života“ paměti (obsahu!) a dosah
- Aritmetika pointerů
- Pointery a pole
- Pointery u procesoru MIPS

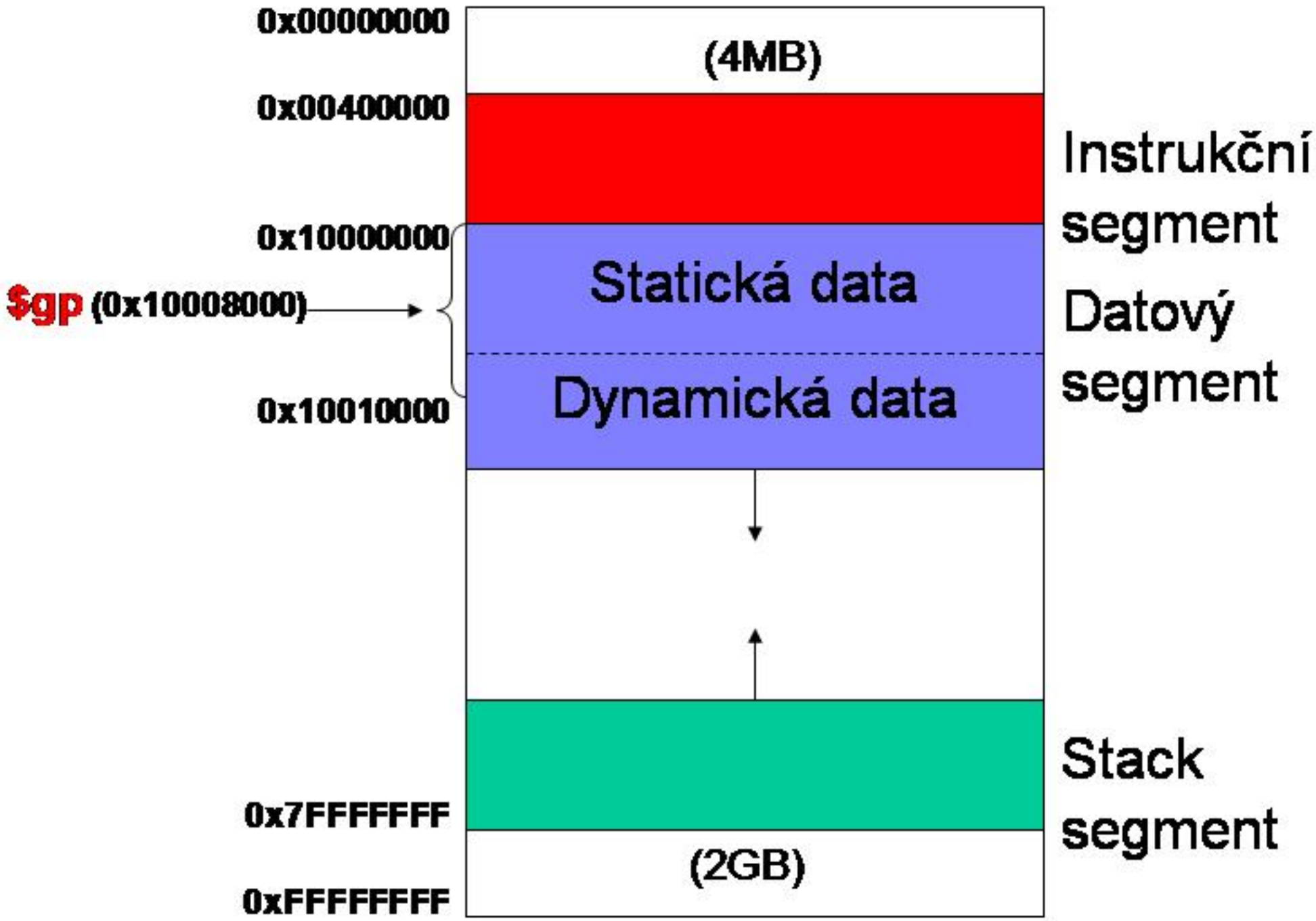
Přehled

- 3 formáty instrukcí MIPS v binárním tvaru:
  - Operační kód (op) určuje formát

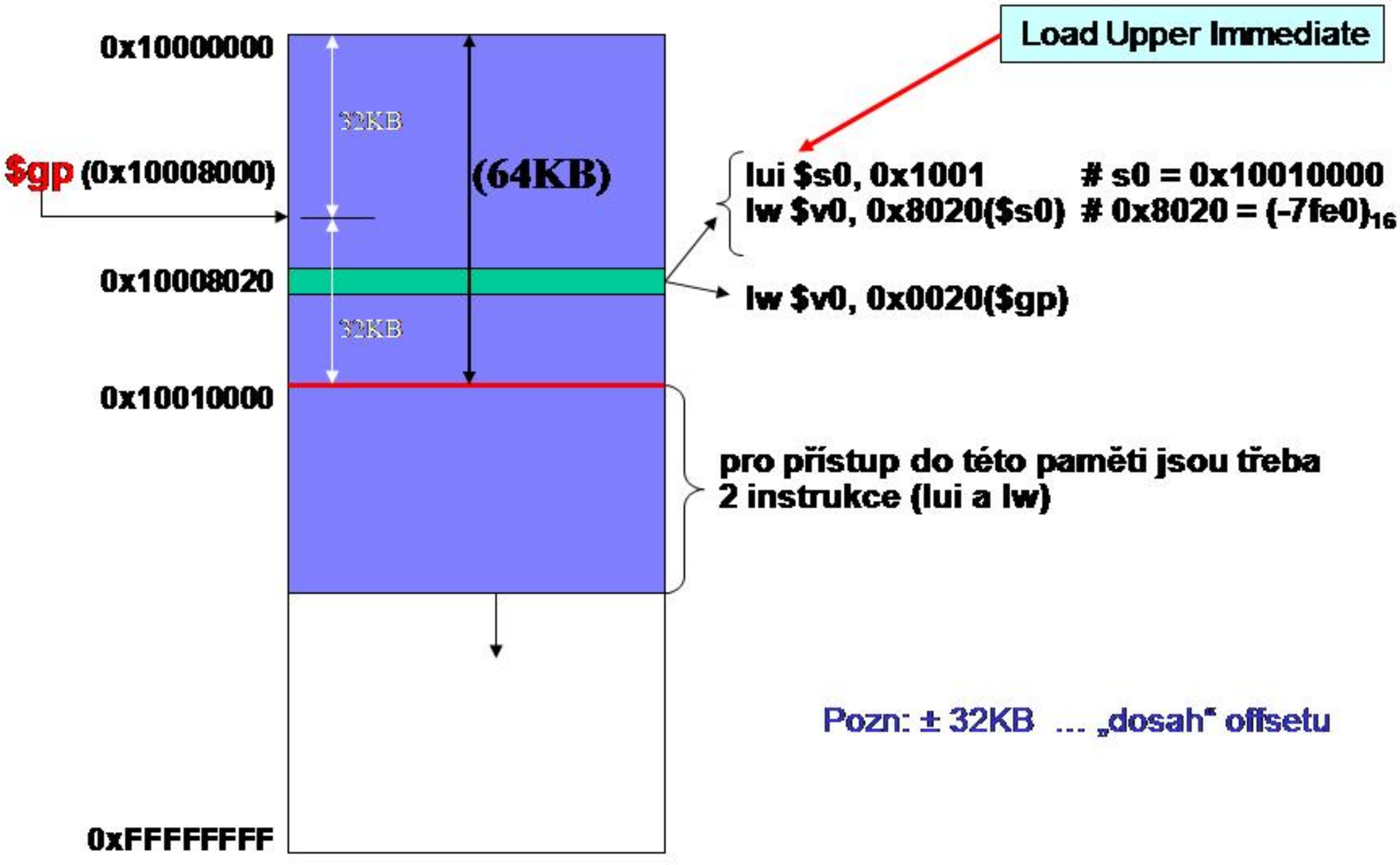
	6 bitů	5 bitů	5 bitů	5 bitů	5 bitů	6 bitů
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	immediate		
J	op	destination address				

- Operandy
  - Registry: $0 až $31 mapovány: `$zero, $at, $v_, $a_, $s_, $t_, $gp, $sp, $fp, $ra`
  - Paměť: `Mem[0], Mem[4], ... , Mem[4294967292]`
    - Index je "adresa" (Array index => Memory index)
- Koncepce programu uloženého v paměti (instrukce jsou čísla!)

Mapa paměti


0x00000000
(4MB)
0x00400000
Instrukční segment
0x10000000
$gp (0x10008000)
Statická data
Datový segment
Dynamická data
0x10010000
0x7FFFFFFF
Stack segment
(2GB)
0xFFFFFFFF

Datový segment



Funkce / procedury jazyka C

```
main() {
    int i, j, k;

    …

    i = mult(j,k);

    …

}
```

```
int mult (int mcand, int mlier) {
    int product;

    product = 0;
    while (mlier > 0)  {
    product = product + mcand;
    mlier = mlier -1; }
    return product;

}
```

Jakou informaci musí kompilátor sledovat?

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

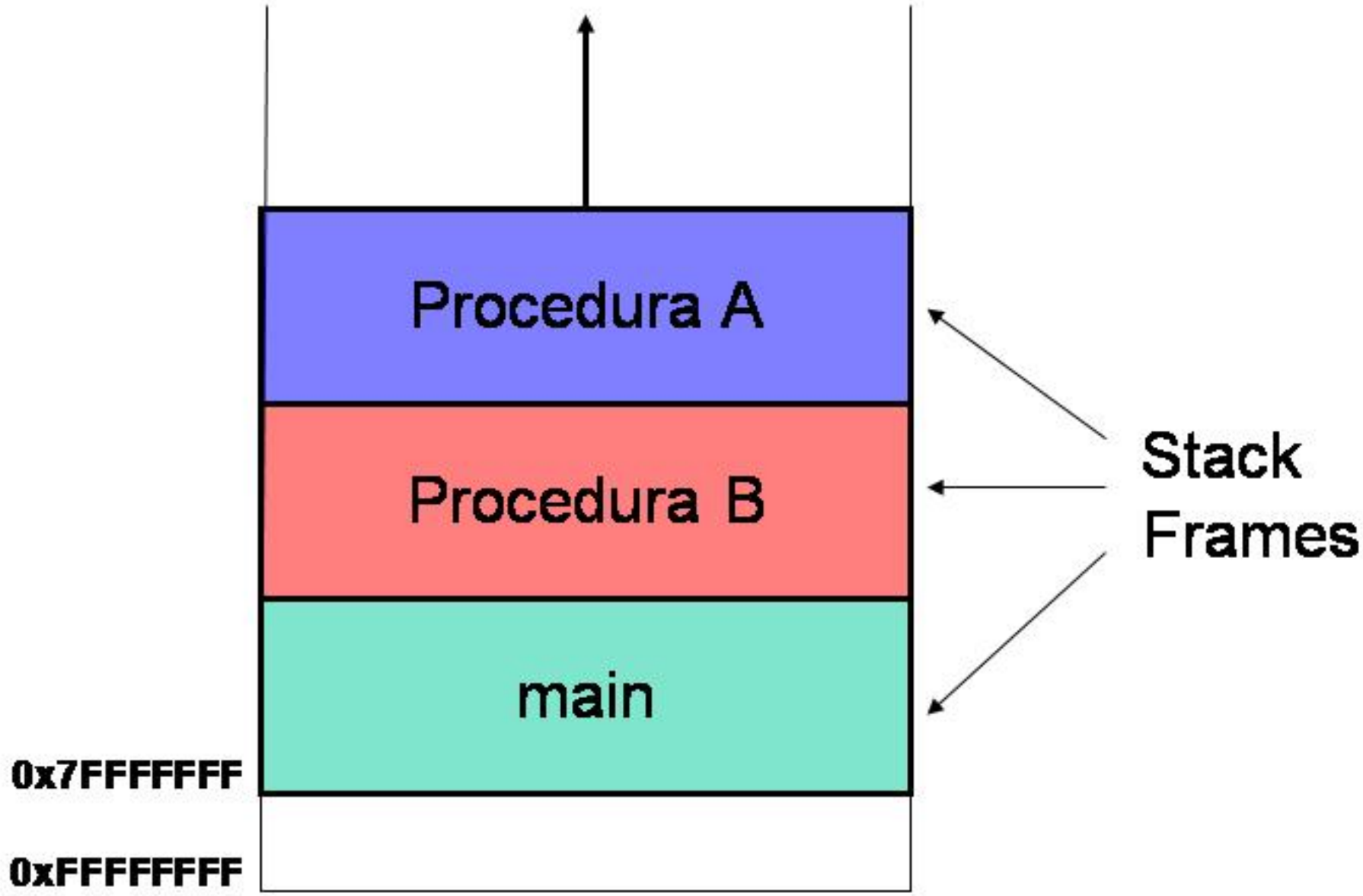
- Dynamická povaha procedur
  - Rámce volání procedur (frames)
    - Argumenty, ukládání registrů, lokální proměnné

Konvence volání procedur

- **Softwarová** pravidla používání registrů

Jméno	Číslo registru	Použití	Rezervován pro call
$zero	0	the constant value 0	n.a.
$at	1	reserved for the assembler	n.a.
$v0-$v1	2-3	expr. evaluation and function result	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

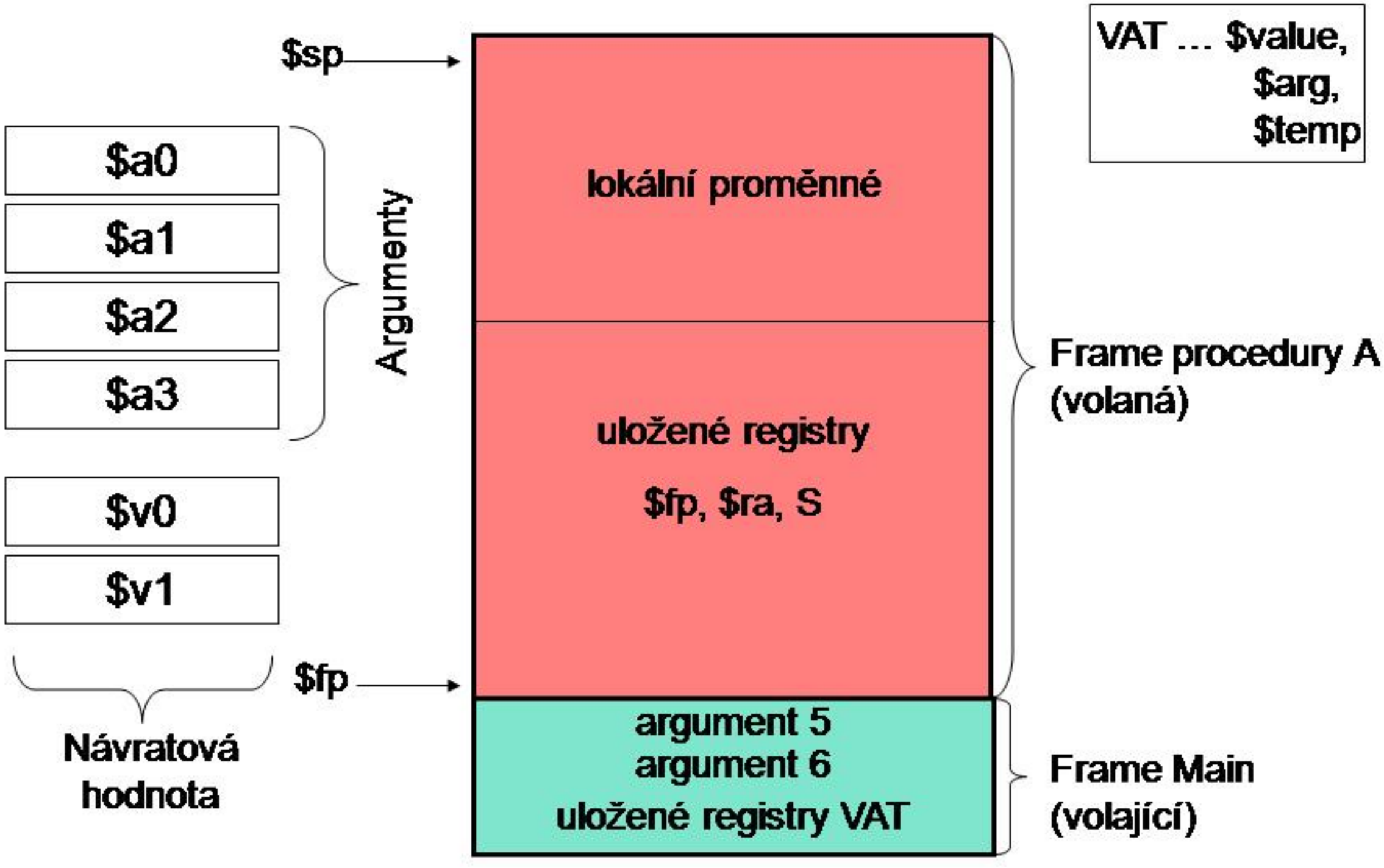
Stack


Procedura A
Procedura B
main
Stack
Frames
0x7FFFFFFF
0xFFFFFFFF

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)



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

Podpora procedur

```
main( ) {
      ...
      s = sum (a, b);
      ...
}
```

```
int sum(int x, int y) {
     return x + y;
}
```

address

```
1000 add  $a0,$s0,$zero  # $a0 = x
1004 add  $a1,$s1,$zero  # $a1 = y
1008 addi $ra,$zero,1016 # $ra=1016
1012 j    sum            # jump to sum
1016 ...
2000 sum: add $v0,$a0,$a1
2004 jr   $ra              # jump to 1016
```

Jak by bylo možno řešit volání procedury bez podpory

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 návěstí**

```
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**

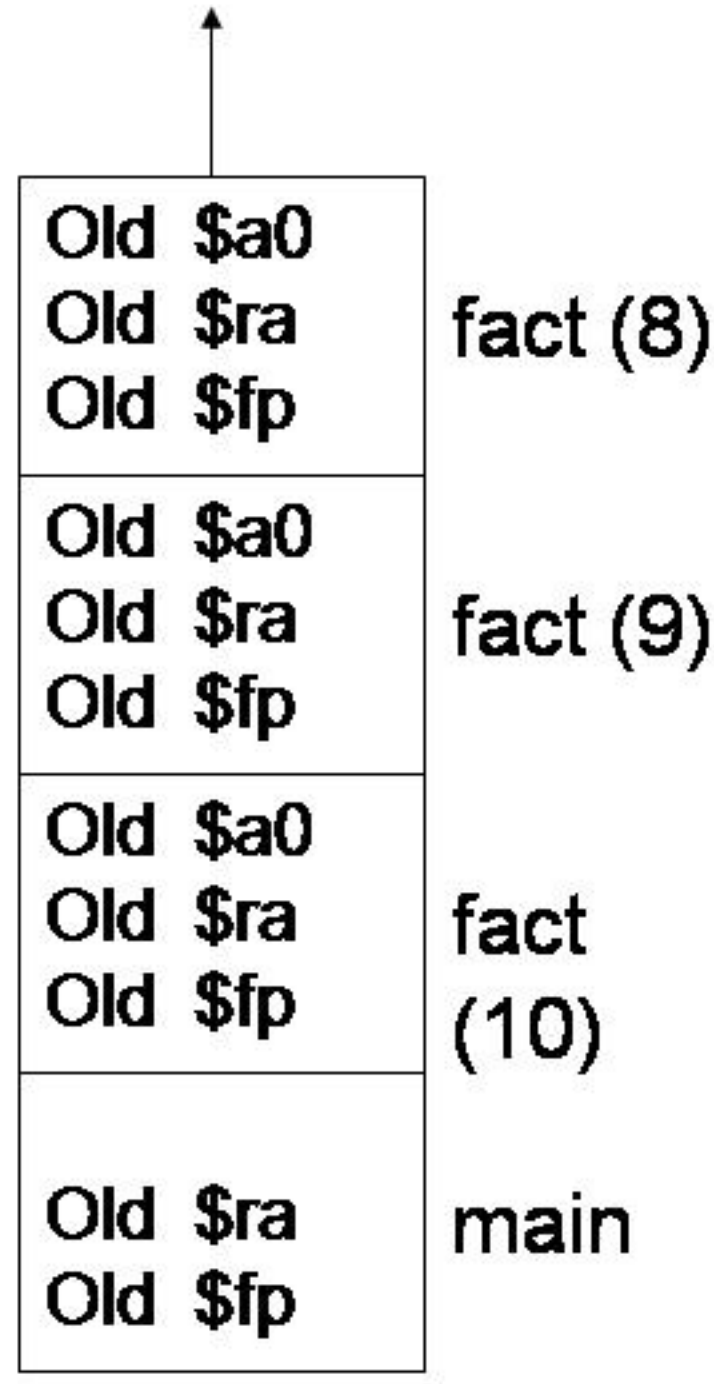
Vnořené procedury

```
      int sumSquare(int x, int y) {
         return mult(x,x) + y;
      }
sumSquare:
         subi $sp,$sp,12      # space on stack
         sw $ra,$ 8($sp)      # save ret addr
         sw $a0,$ 0($sp)      # save x
         sw $a1,$ 4($sp)      # save y
         addi $a1,$a0,$zero   # mult(x,x)
         jal mult             # call mult
         lw $ra,$ 8($sp)      # get ret addr
         lw $a0,$ 0($sp)      # restore x
         lw $a1,$ 4($sp)      # restore y
         add $vo,$v0,$a1      # mult()+y
         addi $sp,$sp,12      # free stack space
         jr $ra
```

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
```

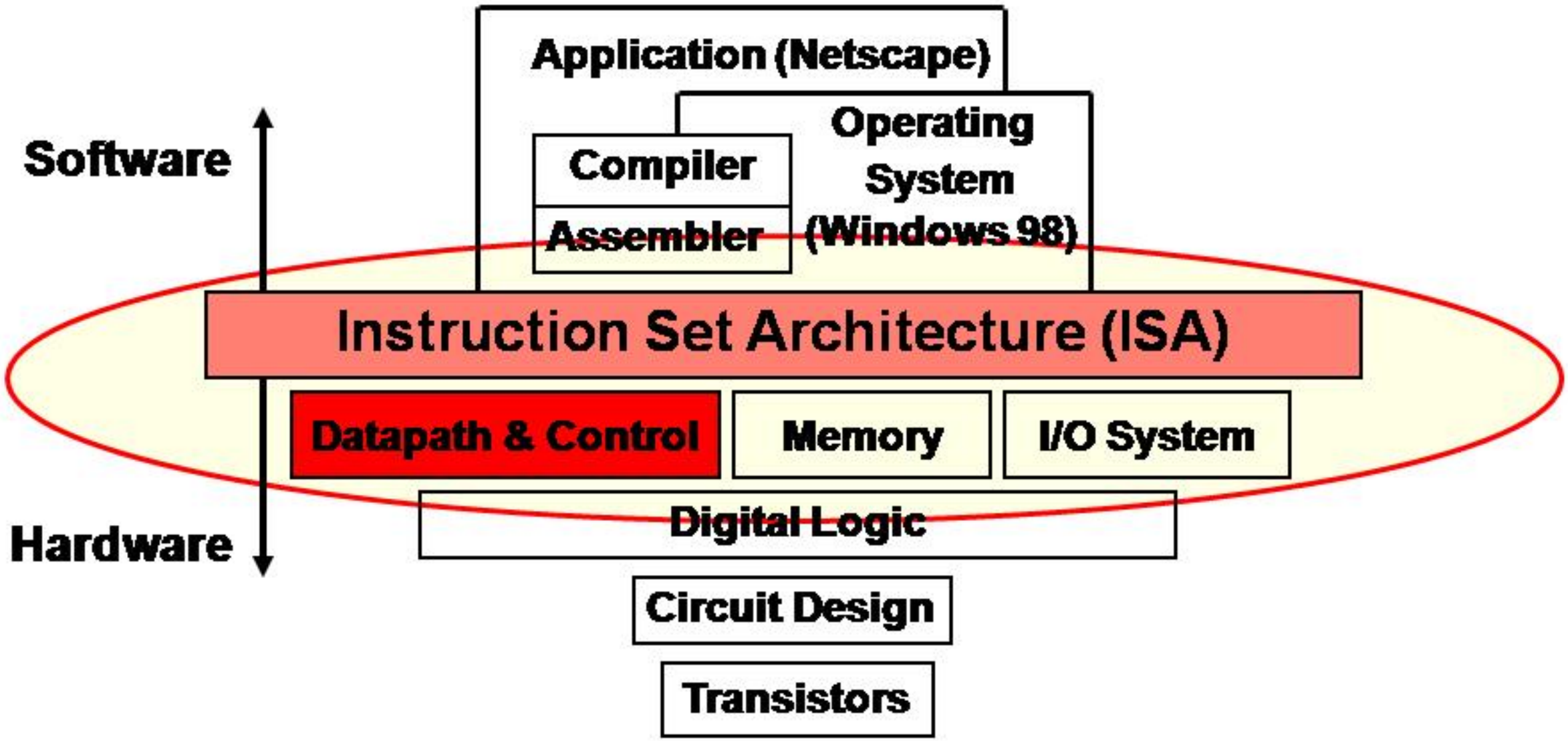
```
fact:   subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sw      $fp, 16($sp)
        addiu   $fp, $sp, 28
        sw      $a0, 0($fp)
        lw      $v0, 0($fp)
        bgtz    $v0, L2
        li      $v0, 1
        j       L1
L2:     lw      $v1, 0($fp)
        subu    $v0, $v1, 1
  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

Souhrn

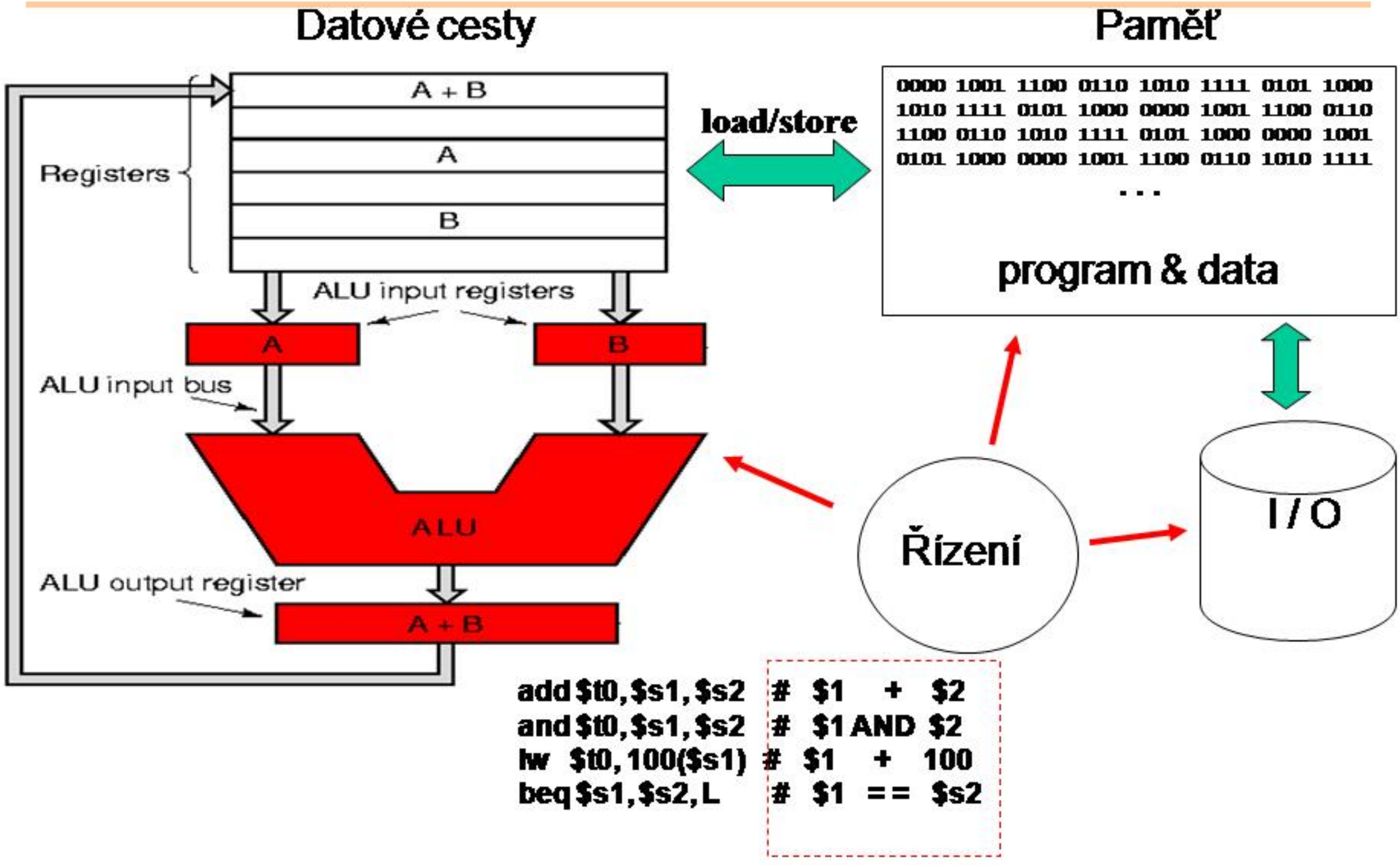
- Konvence volající / volaný
  - Práva a úkoly
  - Volaný používá volně registry VAT
  - Volající používá registry S bez obav z přepsání
- Podpora instrukcí: jal *label* a jr $ra

  volání návrat
- Stack lze použít kdykoliv je třeba něco uložit.
  Nezapomeňte jej opustit ve stejném stavu !!!
- Konvence použití registrů
  - Účel a limity použití
  - Dodržujte pravidla i v případě, že celý program píšete vy !

Nová látka – Číselné systémy



Arithmeticko-logická jednotka




```
add $t0,$s1,$s2   #  $1   +  $2
and $t0,$s1,$s2   #  $1 AND $2
lw  $t0,100($s1)  #  $1   +  100
beq $s1,$s2,L     #  $1  ==  $s2
```

Počítačová aritmetika

- Počítačová aritmetika vs. matematická teorie
  - Čísla s limitovanou přesností
    - **Množina není uzavřená vzhledem k operacím +, -, *, /**
      ==>
      - **Přetečení**
      - **Podtečení**
      - **Nedefinováno**

    - ***Zákony „obyčejné" algebry vždy v počítačích neplatí (zaokrouhlovací procesy !!!)***
      - **a + (b - c) = (a + b) - c**
      - **a * (b – c) = a * b – a * c**
  - Binární čísla
    - **Základ: 2**
    - **Číslice: 0 a 1**

$$(1101\ 1010\ 1101)_2$$
$$(\ B\ \ A\ \ D\ )_{16}$$

$$\text{Dekadická čísla} = \sum_{i=0}^{k-1} d_i.10^i \qquad (d_{k-1} \ldots d_2 d_1 d_0) \qquad d_i \in \{0,1,\ldots 9\}$$

Reprezentace dat

Bity mohou reprezentovat cokoliv:

- Znaky
  - 26 písmen => 5 bitů
  - Velká/malá + diakritika => 7 bitů (z 8)
  - „Zbytek“ světových jazyků => 16 bitů (Unicode)
- Čísla bez znaménka (0, 1, …, 2^{n-1})
- Logické hodnoty
  - 0 -> False, 1 => True
- Barvy
- Polohu / adresy / příkazy
- n-bitů může reprezentovat pouze 2^n různých objektů

Záporná čísla

- Dosud čísla bez znaménka – *Jak je to se znaménkem?*
- Naivní řešení: bit na levém okraji slova definujeme jako znaménko
  - 0 značí +, 1 značí - => *znaménkový bit*
  - Ostatní bity tvoří numerickou hodnotu čísla
  - Této reprezentaci se říká znaménko a amplituda
- MIPS používá 32-bitová čísla integer (16-bitů immediate/displacement)

  +1 se zobrazí:

  0000 0000 0000 0000 0000 0000 0000 0001

  - 1 se zobrazí :

  1000 0000 0000 0000 0000 0000 0000 0001

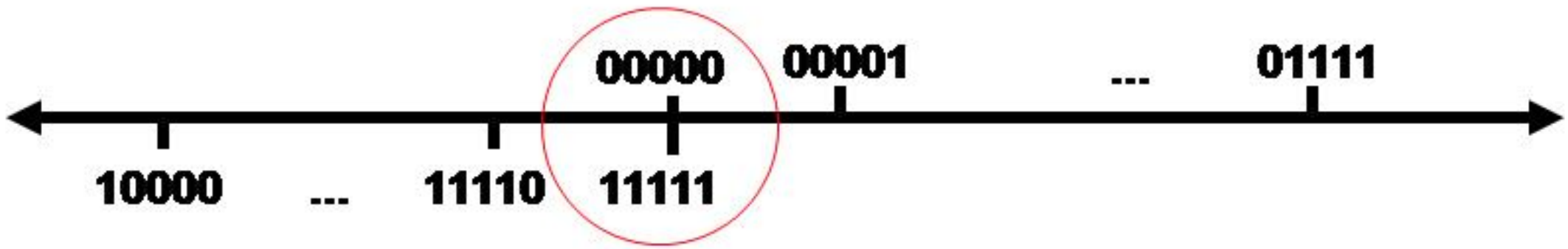
Problémy se znaménkem a amplitudou

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)

Vyzkoušejme: Jednotkový doplněk

- Získání záporného čísla ==> inverze bitů
- Příklad: $7_{10} = 00111_2$ $-7_{10} = 11000_2$
- Kladná čísla začínají 0, záporná čísla mají na počátku (vlevo) 1.



- Stále existují dvě nuly (operace.. ☹)
  - 0x00000000 = +0
  - 0xFFFFFFFF = - 0
- Aritmetika není složitá ☺

Dvojkový doplněk

- Kladná čísla začínají 0
- Záporná čísla ==> inverze kladného + jedna
- Příklad

$1_{10} = 00000001_2$

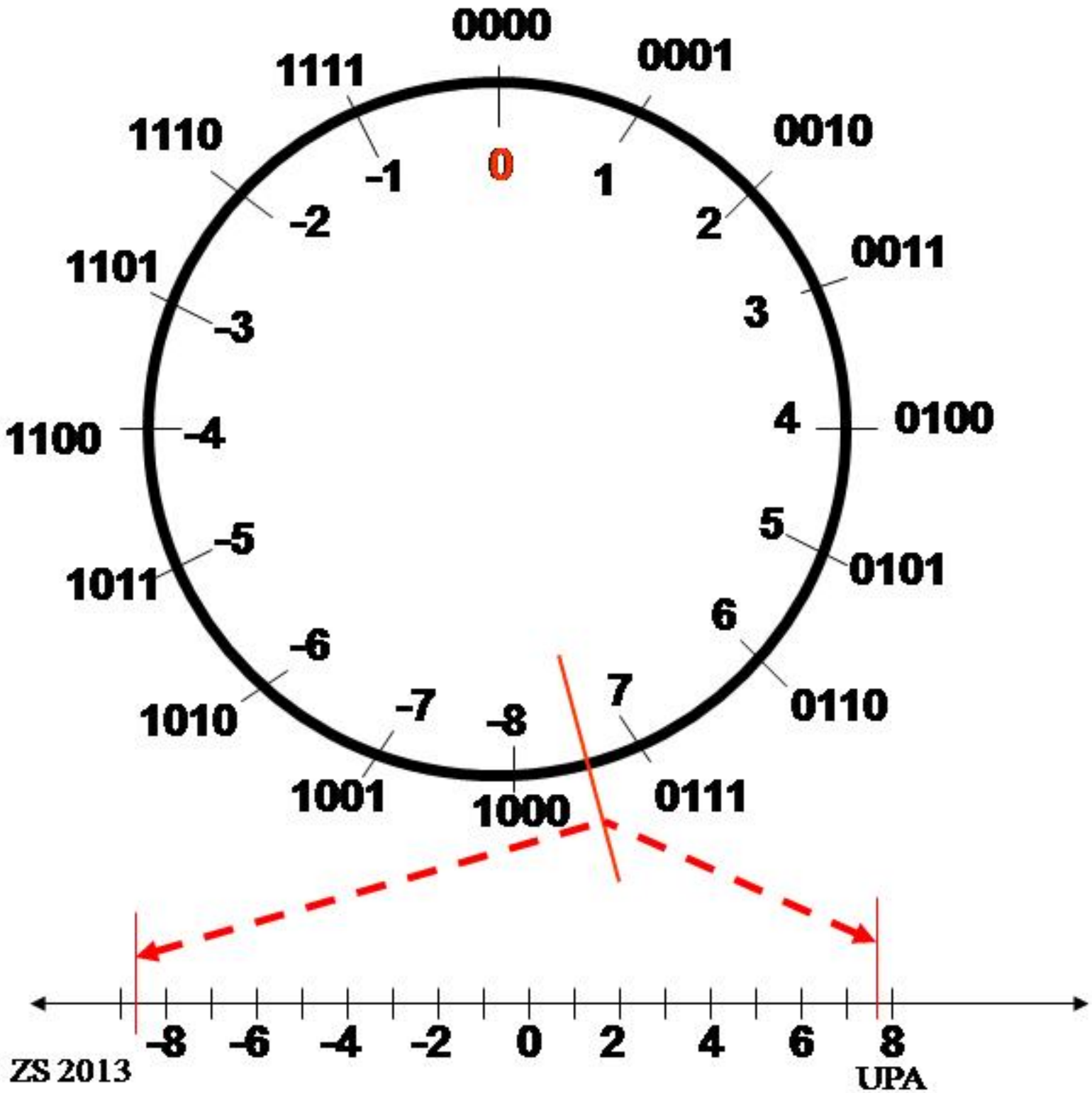
$-1_{10} = 11111110_2 + 1_2 = 11111111_2$

$7_{10} = 00000111_2$

$-7_{10} = 11111000_2 + 1_2 = 11111001_2$

- Kladná čísla mají nekonečně mnoho úvodních 0
- Záporná čísla také mají nekonečně úvodních 1

Grafické znázornění čísel dvojkového doplňku



- 2^{n-1} nezáporných
- 2^{n-1} záporných
- **jediná nula**
- $2^{n-1}-1$ kladných
- porovnání
- přetečení

Příklady: Dvojkový doplněk

$0000 \ldots 0000\ 0000\ 0000\ 0000_2 =$	0_{10}
$0000 \ldots 0000\ 0000\ 0000\ 0001_2 =$	1_{10}
$0000 \ldots 0000\ 0000\ 0000\ 0010_2 =$	2_{10}
. . .	
$0111 \ldots 1111\ 1111\ 1111\ 1101_2 =$	$2{,}147{,}483{,}645_{10}$
$0111 \ldots 1111\ 1111\ 1111\ 1110_2 =$	$2{,}147{,}483{,}646_{10}$
$0111 \ldots 1111\ 1111\ 1111\ 1111_2 =$	$2{,}147{,}483{,}647_{10}$
$1000 \ldots 0000\ 0000\ 0000\ 0000_2 =$	$-2{,}147{,}483{,}648_{10}$
$1000 \ldots 0000\ 0000\ 0000\ 0001_2 =$	$-2{,}147{,}483{,}647_{10}$
$1000 \ldots 0000\ 0000\ 0000\ 0010_2 =$	$-2{,}147{,}483{,}646_{10}$
. . .	
$1111 \ldots 1111\ 1111\ 1111\ 1101_2 =$	-3_{10}
$1111 \ldots 1111\ 1111\ 1111\ 1110_2 =$	-2_{10}
$1111 \ldots 1111\ 1111\ 1111\ 1111_2 =$	-1_{10}

Dvojkový doplněk

- Může reprezentovat kladná i záporná čísla – znaménkový bit (MSB). Zápis:

  $d_{31} \times (-2^{31}) + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$

- Příklad

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$

  $= 1 \times (-2^{31}) + 1 \times 2^{30} + 1 \times 2^{29} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

  $= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^2 + 0 + 0$

  $= -2{,}147{,}483{,}648_{10} + 2{,}147{,}483{,}644_{10}$

  $= -4_{10}$

- Pozn.! Musí být známa délka zobrazení =>poloha MSB

  => MIPS používá 32 bitů, takže MSB je d_{31}

Dvojkový komplement - algoritmus

- Invertovat (každou 0 na 1 a každou 1 na 0),
  potom přičíst 1 k výsledku
  - Součet čísla a jeho 1 doplňku musí být $111\ldots111_2 = -1_{10}$
  - Nechť x' značí invertovanou representaci x
  - Potom $x + x' = -1 \Rightarrow x + x' + 1 = 0 \Rightarrow x' + 1 = -x$

- Příklad:

  x = -4 : $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$

  x' : $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_2$

  x' + 1: $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100_2$

  invert: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_2$

  add 1 : $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$

Porovnání se zn. a bez zn.

- $X = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
- $Y = 0011\ 1011\ 1001\ 1010\ 1000\ 1010\ 0000\ 0000_2$

Nejednoznačnost:

- Je X > Y?
  - Bez znaménka: ANO
  - Se znaménkem: NE
- Konverze na dekadický tvar (pro kontrolu)
  - Porovnání se znaménkem:
    $-4_{10} < 1{,}000{,}000{,}000_{10}$?
  - Porovnání bez znaménka:
    $-4{,}294{,}967{,}292_{10} < 1{,}000{,}000{,}000_{10}$

Dvojkový doplněk – rozšíření znaménka

- *Problém:* Konvertovat číslo na větší počet bitů

- *Řešení:* MSB rozšířit na nové pozice vlevo
  - –2 doplněk - kladné číslo má nekonečně 0 vlevo
  - –2 doplněk zápomé číslo má nekonečně 1 vlevo
  - –Bitová reprezentace maskuje tyto bity; rozšířením znaménka se jich část obnoví
  - –16-bitů -4_{10} konvertovat na délku 32-bitů:

Definice kódů pro zobrazení záporných čísel

- Přímý kód:

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

- 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

- 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

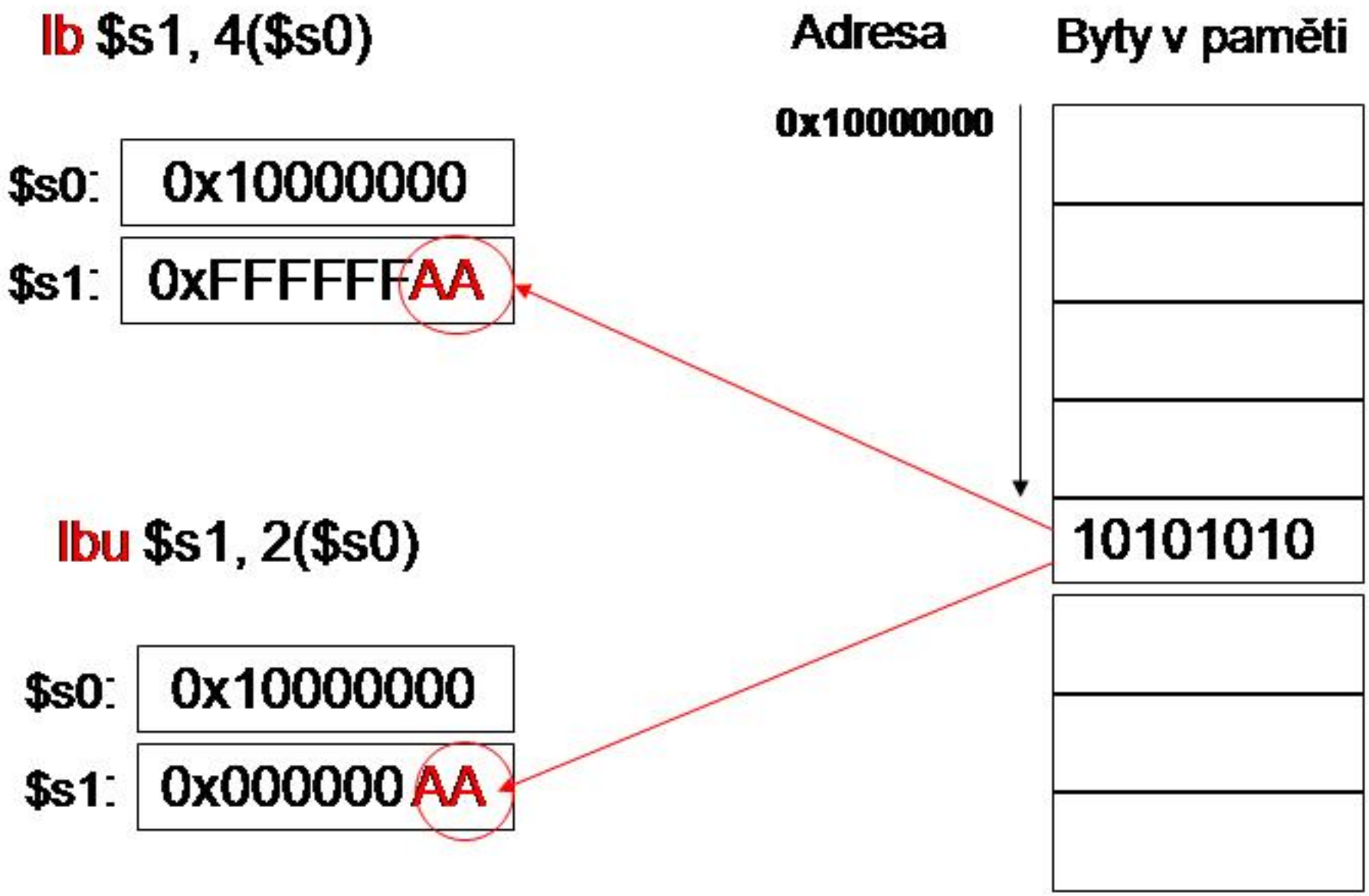
Datové typy MIPS (1/2)

- 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)

Instrukce pro bytové operace



U instrukce lb dochází k rozšíření znaménka

Manipulace s řetězci

```
Void strcpy (char[], char y[]) {
  int i;
  i = 0;
  while ((x[i]=y[i]) != 0)
     i = i + 1;
}
```

Konvence C :

Nulový byte (00000000)
reprezentuje konec řetězce

```
strcpy:
      subi $sp, $sp, 4
      sw   $s0, 0($sp)
      add  $s0, $zero, $zero
L1:   add $t1, $a1, $s0
      lb    $t2, 0($t1)
      add  $t3, $a0, $s0
      sb    $t2, 0($t3)
      beq  $t2, $zero, L2
      addi $s0, $s0, 1
      j      L1
L2:   lw   $s0, 0($sp)
      addi $sp, $sp, 4
      jr     $ra
```

Důležitost komentářů pro MIPS!

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
```

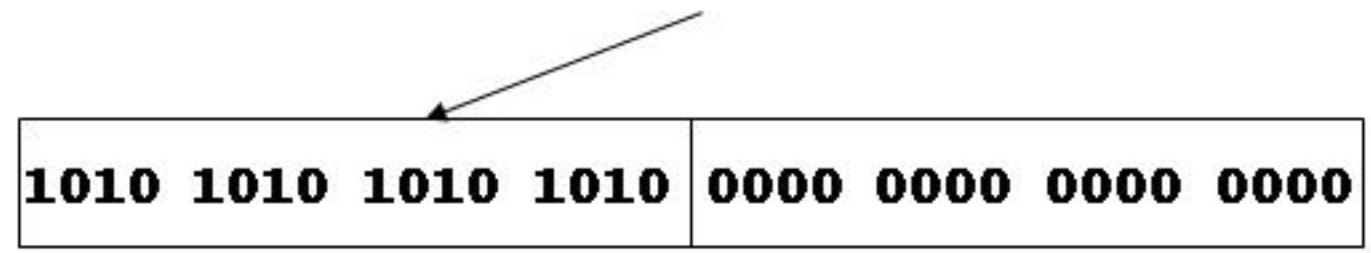
8	29	29	4

101011	10011	01000	0000 0000 0011 0100

Velké konstanty

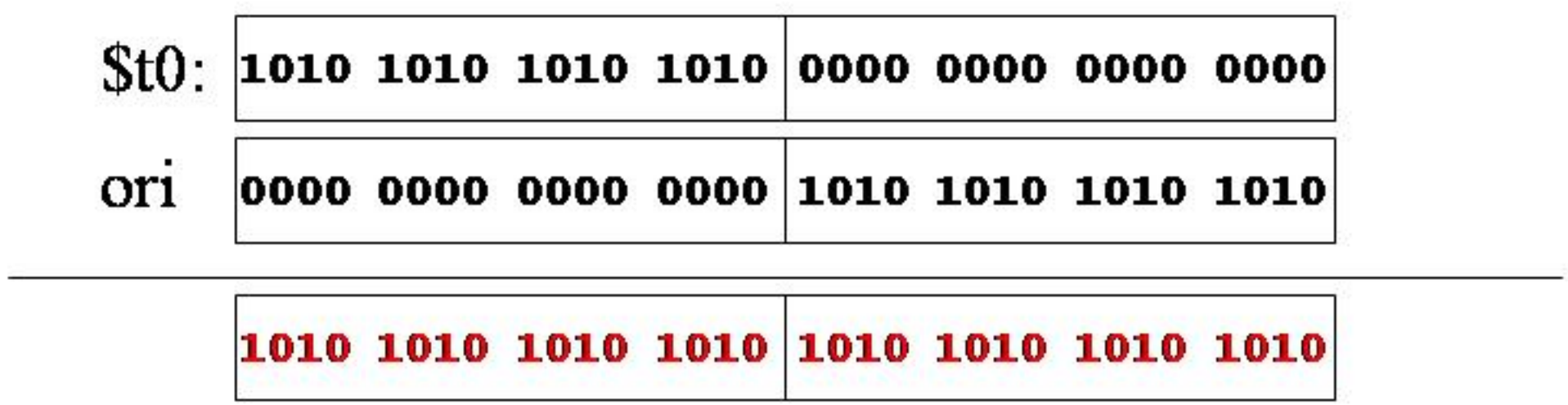
- Naplnění 32 bitového registru konstantou:
  1. Naplnění (16) vyšších bitů

```
lui $t0, 1010101010101010
```



  2. Pak se musí nižší bity přesunout doprava, t. zn.

```
ori $t0, $t0, 1010101010101010
```



Adresní režimy

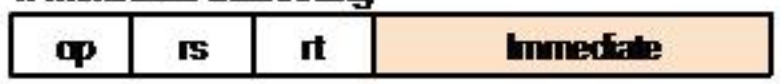
- Adresy pro *data* a *instrukce*
- Data (operandy a výsledky)
  - Registry
  - Místa v paměti
  - Konstanty
- Úsporné kódování adres (prostor: 32 bitů)
  - Registry (32) => použito 5 bitů pro zakódování adresy
  - *Destruktivní* instrukce: reg2 = reg2 + reg1
  - Akumulátor
  - Stack
- ***Ortogonalita***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

Adresní režimy dat

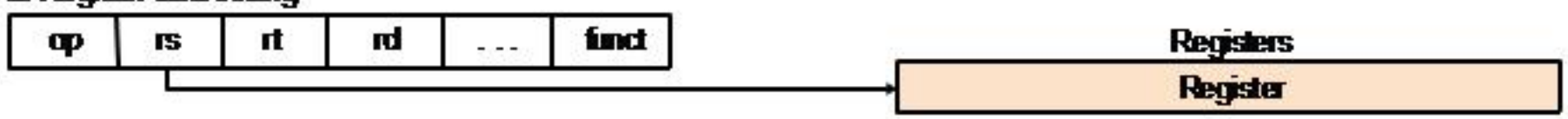
- Registrové adresování
  - Nejobvyklejší způsob (nejrychlejší a nejkratší)
  - add $3, $2, $1
- Bázované adresování
  - Operand je v paměti na místě udaném offsetem
  - lw $t0, 20 ($t1)
- „Immediate“ operandy
  - Operand je malá konstanta uvnitř instrukce
  - addi $t0, $t1, 4 (16-bit integer se znaménkem)

Adresní módy

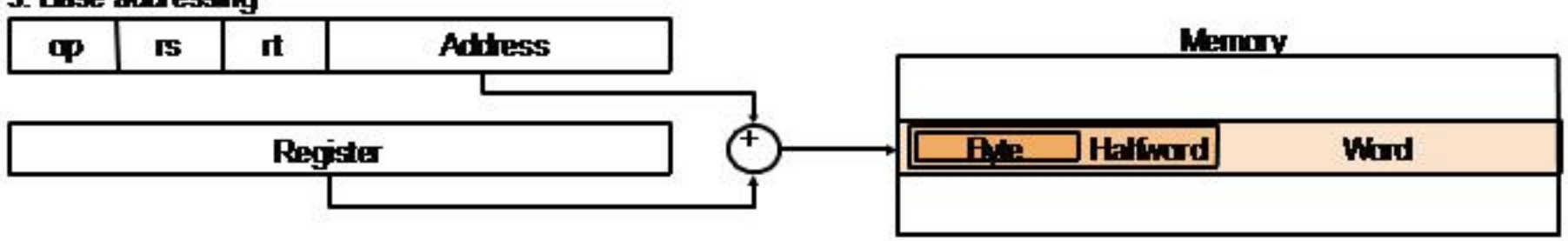
1. Immediate addressing



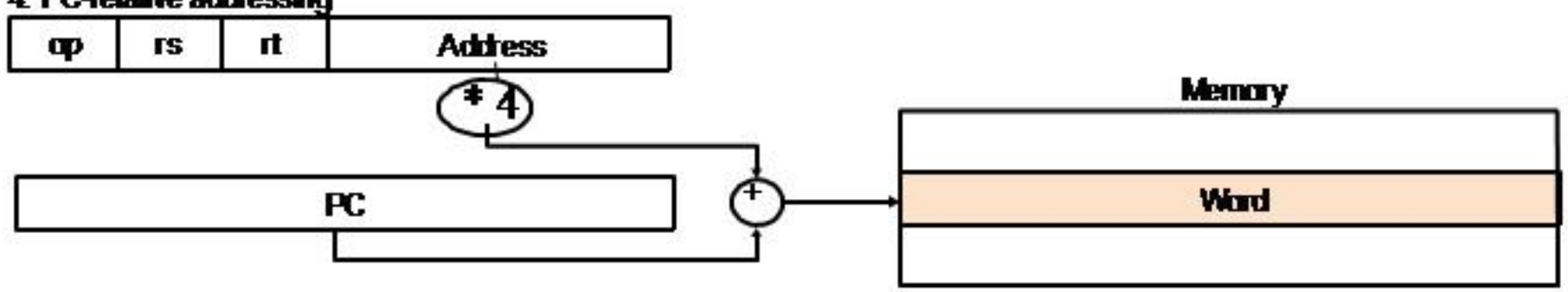
2. Register addressing



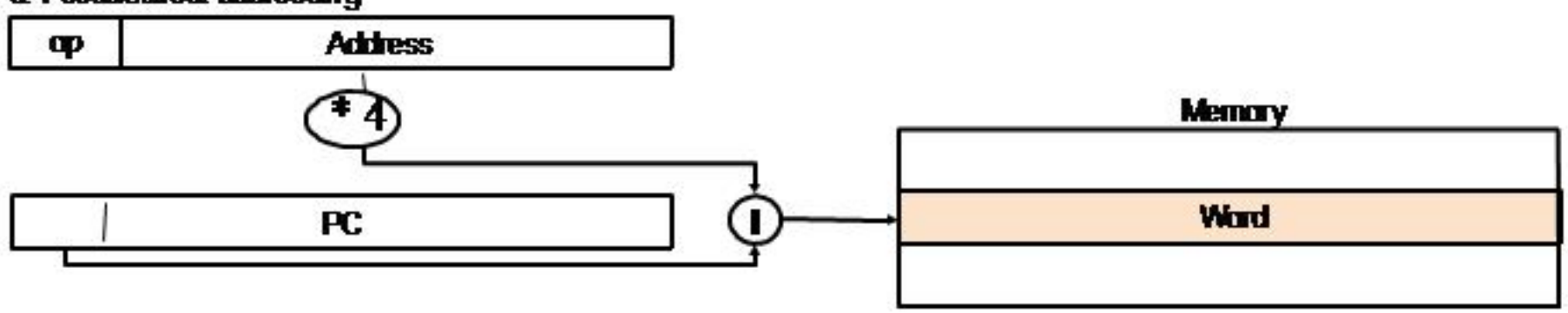
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



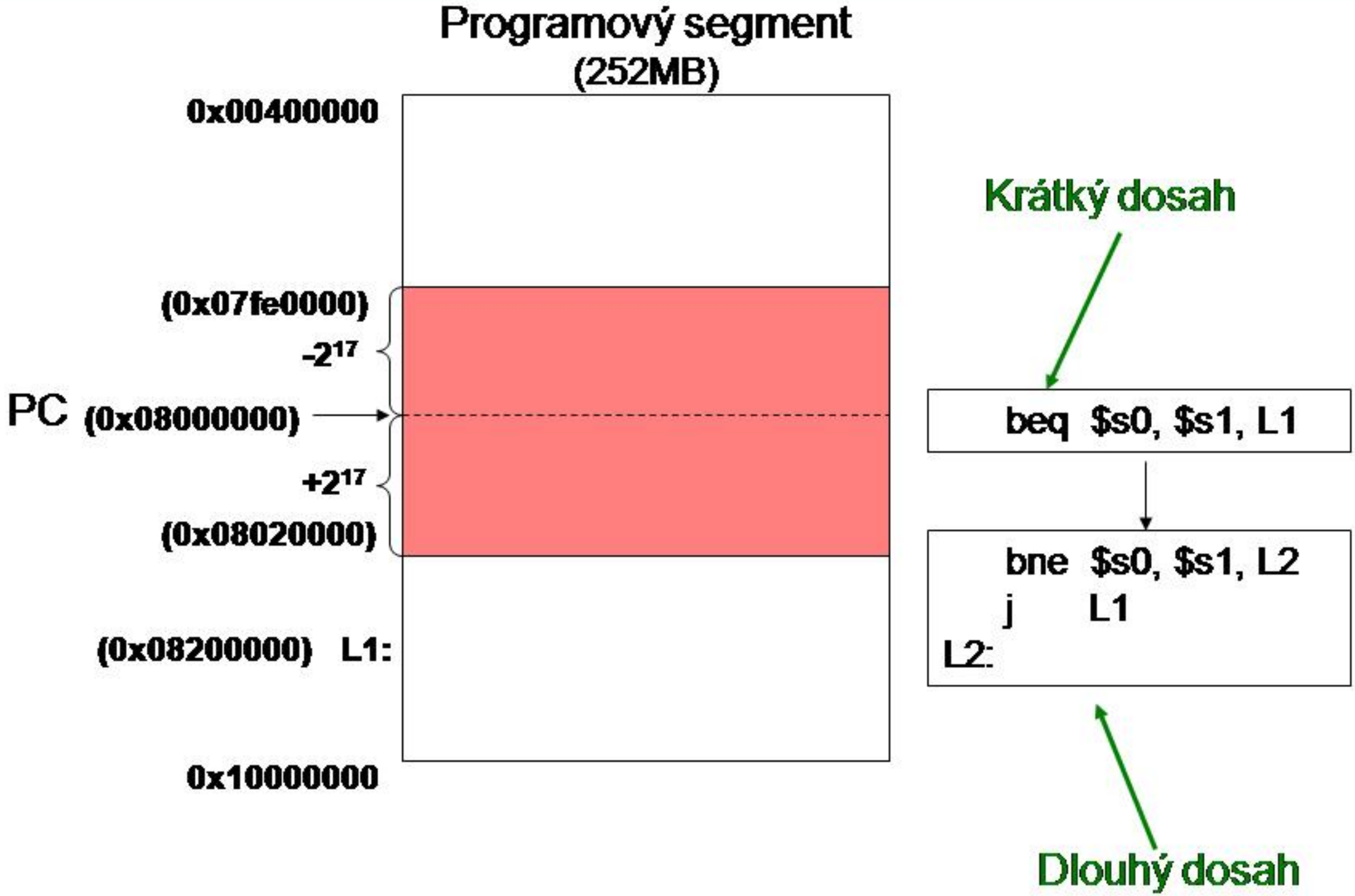
Adresní módy instrukcí

- 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

Kód SPIM

PC	MIPS	strojní kód	Pseudo MIPS
main [0x00400020]	add $9, $10, $11	(0x014b4820)	main: add $t1, $t2, $t3
[0x00400024]	j 0x00400048 [exit]	(0x08100012)	j exit
[0x00400028]	addi $9, $10, -50	(0x2149ffce)	addi $t1, $t2, -50
[0x0040002c]	lw $8, 5($9)	(0x8d280005)	lw $t0, 5($t1)
[0x00400030]	lw $8, -5($9)	(0x8d28fffb)	lw $t0, -5($t1)
[0x00400034]	bne $8, $9, 20 [exit-PC]	(0x15090005)	bne $t0, $t1, exit
[0x00400038]	addi $9, $10, 50	(0x21490032)	addi $t1, $t2, 50
[0x0040003c]	bne $8, $9, -28 [main-PC]	(0x1509fff9)	bne $t0, $t1, main
[0x00400040]	lb $8, -5($9)	(0x8128fffb)	lb $t0, -5($t1)
[0x00400044]	j 0x00400020 [main]	(0x08100008)	j main
[0x00400048] exit	add $9, $10, $11	(0x014b4820)	exit: add $t1, $t2, $t3

Dlouhé cílové adresy



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 (???)

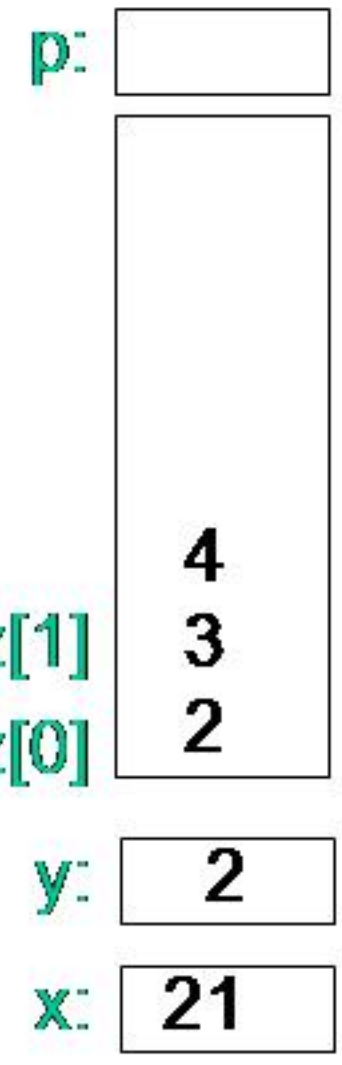
Operátory pro pointery v C

- Předpokládejme, že c má hodnotu 100 a leží v paměti na adrese 0x10000000
- Unární operátor & dává adresu:
  p = &c; obsahem p bude adresa c
  - p "ukazuje na" c (p == 0x10000000)
- Unární operátor * dává hodnotu, na kterou pointer ukazuje
  - if p = &c => * p == 100 („Dereferencování" pointeru)
- Deferencing ⟹ přenos dat v assembleru
  - ... = ... *p ...; ⟹ load
    (čtení hodnoty z místa kam ukazuje p)
  - *p = ...; ⟹ store
    (uložení hodnoty do místa kam ukazuje p)

Aritmetika pointerů

```
int x = 1,  y = 2;      /* x a y jsou proměnné typu integer */
int z[10];              /* pole 10 int, z ukazuje na počátek */
int *p;                 /* p je pointer na int */

x = 21;                 /* přiřadí x novou hodnotu 21 */
z[0] = 2; z[1] = 3      /* přiřadí 2 prvému, 3 dalšímu prvku pole */
p = &z[0];              /* p ukazuje na první prvek z */
p = z;                  /* totéž jako;  p[ i ] == z[ i ]*/
p = p+1;                /* nyní pointer ukazuje na další prvek, z[1] */
p++;                    /* a opět na další, tentokrát na z[2] */
*p = 4;                 /* přiřadí tam 4, z[2] == 4*/
p = 3;                  /* špatně! Je to absolutní adresa !!! */
p = &x;                 /* p ukazuje na x, *p == 21  */
z = &y                  nepřípustné !!!!!   jméno pole není proměnná
```



Pointery a assembler

c je `int`, má hodnotu 100, v paměti na adrese 0x10000000,
p je v `$a0`, x je v `$s0`

```
1. p = &c;     /* p gets 0x10000000*/
    lui $a0,0x1000 # p = 0x10000000
2.  x = *p;   /* x gets 100 */
    lw  $s0, 0($a0) # dereferencing p
3. *p = 200; /* c gets 200 */
    addi $t0,$0,200
    sw   $t0, 0($a0) # dereferencing p
```

Příklad

```
int strlen(char *s) {
  char *p = s;          /* p points to chars */
  while (*p != '\0')
    p++;                /* points to next char */
  return p - s;         /* end - start */
}
```

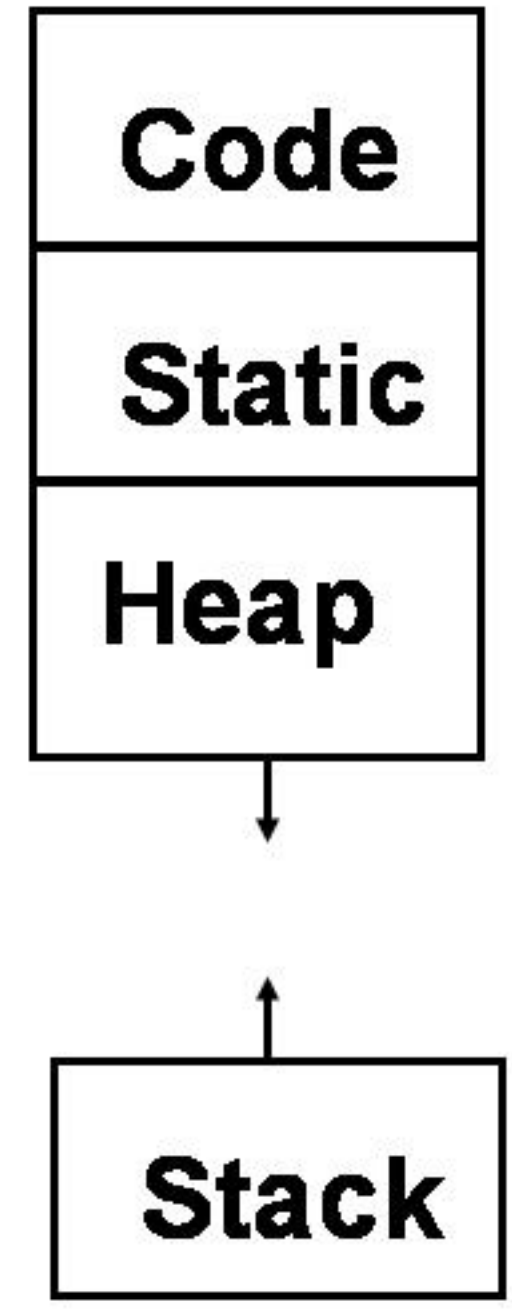
```
      mov   $t0,$a0
      lbu   $t1,0($t0)  /* derefence p */
      beq   $t1,$zero, Exit
Loop:addi   $t0,$t0,1   /* p++ */
      lbu   $t1,0($t0)  /* derefence p */
      bne   $t1,$zero, Loop
Exit:sub    $v0,$t0,$a0
      jr    $ra
```

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

Pamět, její časový rámec a dosah

- Automaticky (přidělen stack)
  - Typicky lokální proměnné uvnitř funkce
  - Vytvořeny po volání call, uvolněny po return
  - Platnost uvnitř funkce
- Přidělen heap
  - Vytvořen pomocí malloc, uvolněn pomocí free
  - Přístup pomocí pointerů
- Externí / statická
  - Existuje pro celý program



Pole, pointery a funkce

- 4 verze funkcí, které sčítají dvě pole a ukládají součty do třetího pole (*sumarray*)
  1. Třetí pole je předáváno funkci adresou v parametrech
  2. Použití lokálního pole (ve stacku) pro výsledek a předání pointeru na toto pole
  3. Třetí pole je alokováno v heapu
  4. Třetí pole je deklarováno jako statické
- Smyslem příkladu je ukázat interakci příkazů jazyka C, pointerů a přidělovacích mechanizmů paměti

Verze 1

```
int x[100], y[100], z[100];
sumarray(x, y, z);
```

- Volání v C je interpretováno:

```
sumarray(&x[0], &y[0], &z[0]);
```

- Skutečné předání pointerů na pole

```
addi $a0,$gp,0    # x[0] starts at $gp
addi $a1,$gp,400  # y[0] above x[100]
addi $a2,$gp,800  # z[0] above y[100]
jal  sumarray
```

Verze 1: přeložený kód

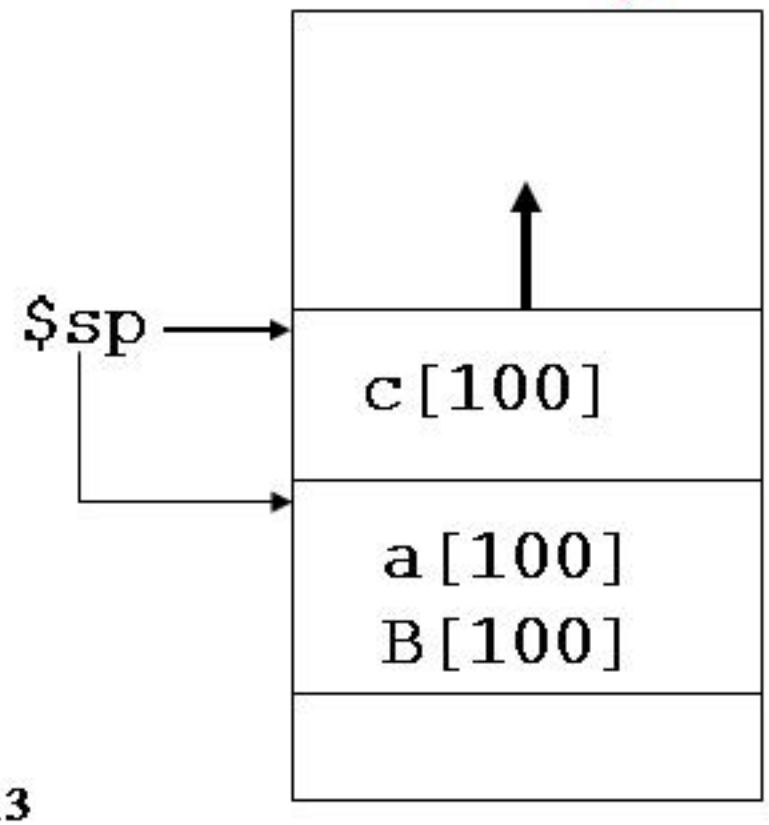
```
void sumarray(int a[], int b[], int c[]) {
    int i;
    for(i = 0; i < 100; i = i + 1)
            c[i] = a[i] + b[i];
}
```

```
        addi    $t0,$a0,400     # beyond end of a[]
Loop:   beq     $a0,$t0,Exit
        lw      $t1, 0($a0)     # $t1=a[i]
        lw      $t2, 0($a1)     # $t2=b[i]
        add     $t1,$t1,$t2     # $t1=a[i] + b[i]
        sw      $t1, 0($a2)     # c[i]=a[i] + b[i]
        addi    $a0,$a0,4       # $a0++
        addi    $a1,$a1,4       # $a1++
        addi    $a2,$a2,4       # $a2++
        j       Loop
Exit:   jr      $ra
```

Verze 2

```
int *sumarray(int a[], int b[]) {
      int i, c[100];
      for(i=0; i<100; i=i+1)
         c[i] = a[i] + b[i];
      return c;
}
```

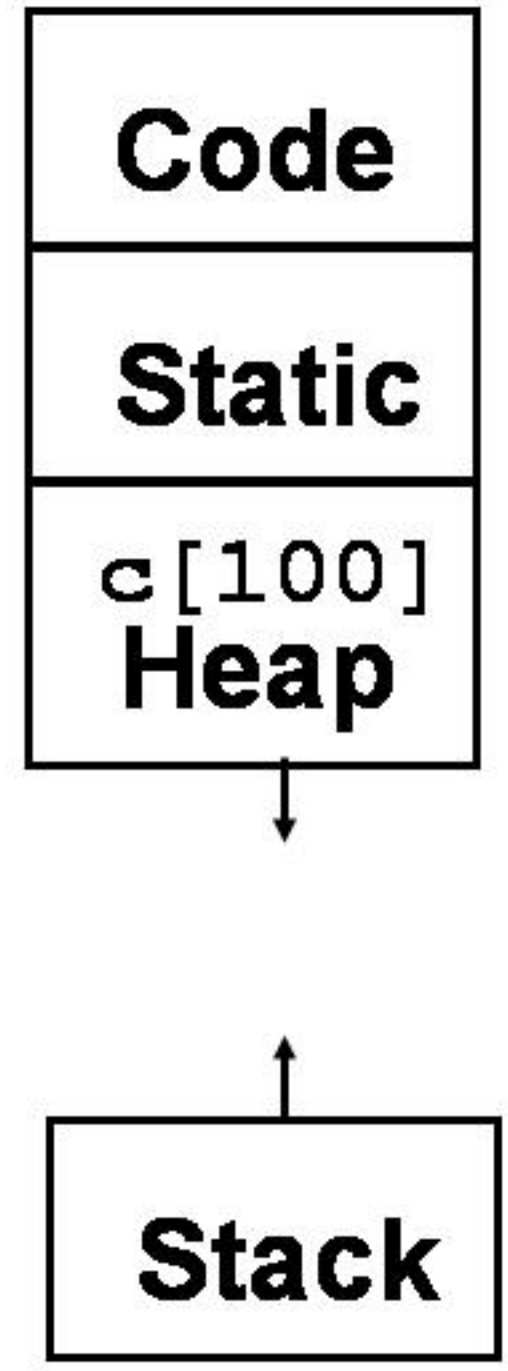
Dostaneme správný výsledek ??




```
      addi  $t0,$a0,400  # beyond end of a[]
      addi  $sp,$sp,-400 # space for c
      addi  $t3,$sp,0    # ptr for c
      addi  $v0,$t3,0    # $v0 = &c[0]
Loop: beq   $a0,$t0,Exit
      lw    $t1, 0($a0)  # $t1=a[i]
      lw    $t2, 0($a1)  # $t2=b[i]
      add   $t1,$t1,$t2  # $t1=a[i] + b[i]
      sw    $t1, 0($t3)  # c[i]=a[i] + b[i]
      addi  $a0,$a0,4    # $a0++
      addi  $a1,$a1,4    # $a1++
      addi  $t3,$t3,4    # $t3++
      j     Loop
Exit: addi  $sp,$sp, 400 # pop stack
      jr    $ra
```

Verze 3

```
int * sumarray(int a[], int b[])
   {
        int i;
        int *c;
        c = (int *) malloc(100*sizeof(int));
        for(i=0; i<100; i=i+1)
           c[i] = a[i] + b[i];
        return c;
   }
```



- Dokud se neuvolní, nelze znova použít
  - Mohou vznikat „díry“ v paměti
  - Java, … mají na uvolňování prostoru garbage kolektory

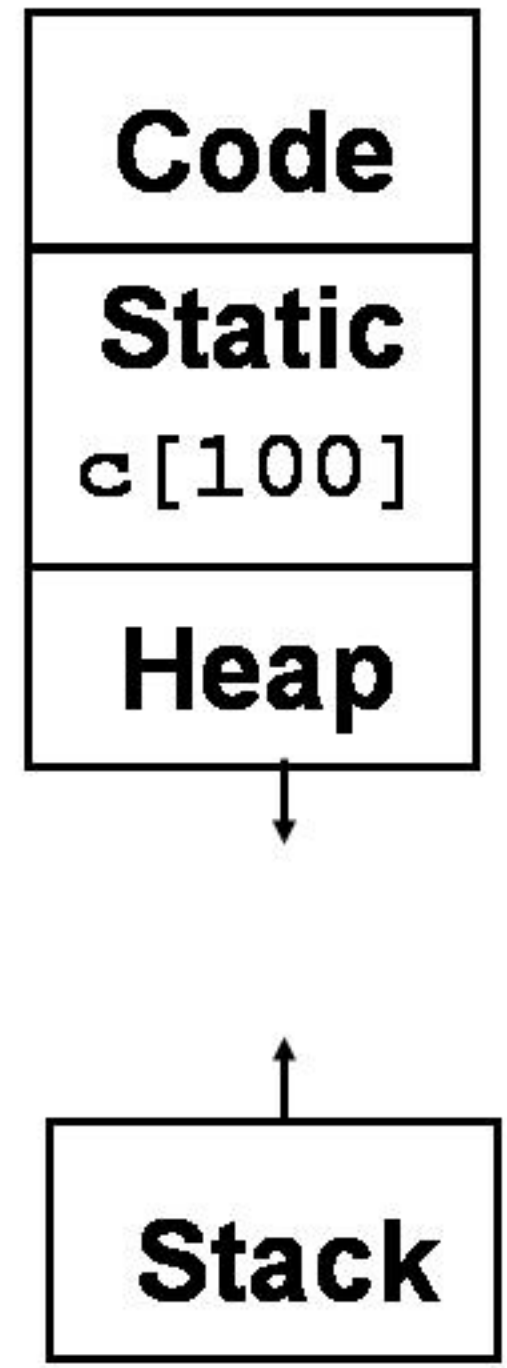
Verze 3: přeložený kód

```
addi   $t0,$a0,400 # beyond end of a[]
addi   $sp,$sp,-12 # space for regs
sw     $ra, 0($sp) # save $ra
sw     $a0, 4($sp) # save 1st arg.
sw     $a1, 8($sp) # save 2nd arg.
addi   $a0,$zero,400
    jal    malloc
    addi   $t3,$v0,0    # ptr for c
    lw     $a0, 4($sp) # restore 1st arg.
    lw     $a1, 8($sp) # restore 2nd arg.
Loop:    beq  $a0,$t0,Exit
    ... (smyčka jako na předcházejícím snímku )
    j      Loop
Exit:lw    $ra, 0($sp)  # restore $ra
    addi   $sp, $sp, 12 # pop stack
    jr     $ra
```

Verze 4

```
int * sumarray(int a[],int b[])
{
     int i;
     static int c[100];

     for(i=0; i<100; i=i+1)
               c[i] = a[i] + b[i];
     return c;
}
```



- Kompilátor přidělí jednou pro funkci, prostor je znovu využit
  - Změní se při příštím volání sumarray
  - Používáno v knihovnách C

Přehled datových oblastí

MIPS operands		
Name	**Example**	**Comments**
	`$s0-$s7, $t0-$t9, $zero,`	Fast locations for data. In MIPS, data must be in registers to perform
32 registers	`$a0-$a3, $v0-$v1, $gp,`	arithmetic. MIPS register $zero always equals 0. Register $at is
	`$fp, $sp, $ra, $at`	reserved for the assembler to handle large constants.
	Memory[0],	Accessed only by data transfer instructions. MIPS uses byte addresses, so
2^{30} **memory**	**Memory[4], ...,**	sequential words differ by 4. Memory holds data structures, such as arrays,
words	**Memory[4294967292]**	and spilled registers, such as those saved on procedure calls.

Přehled instrukcí

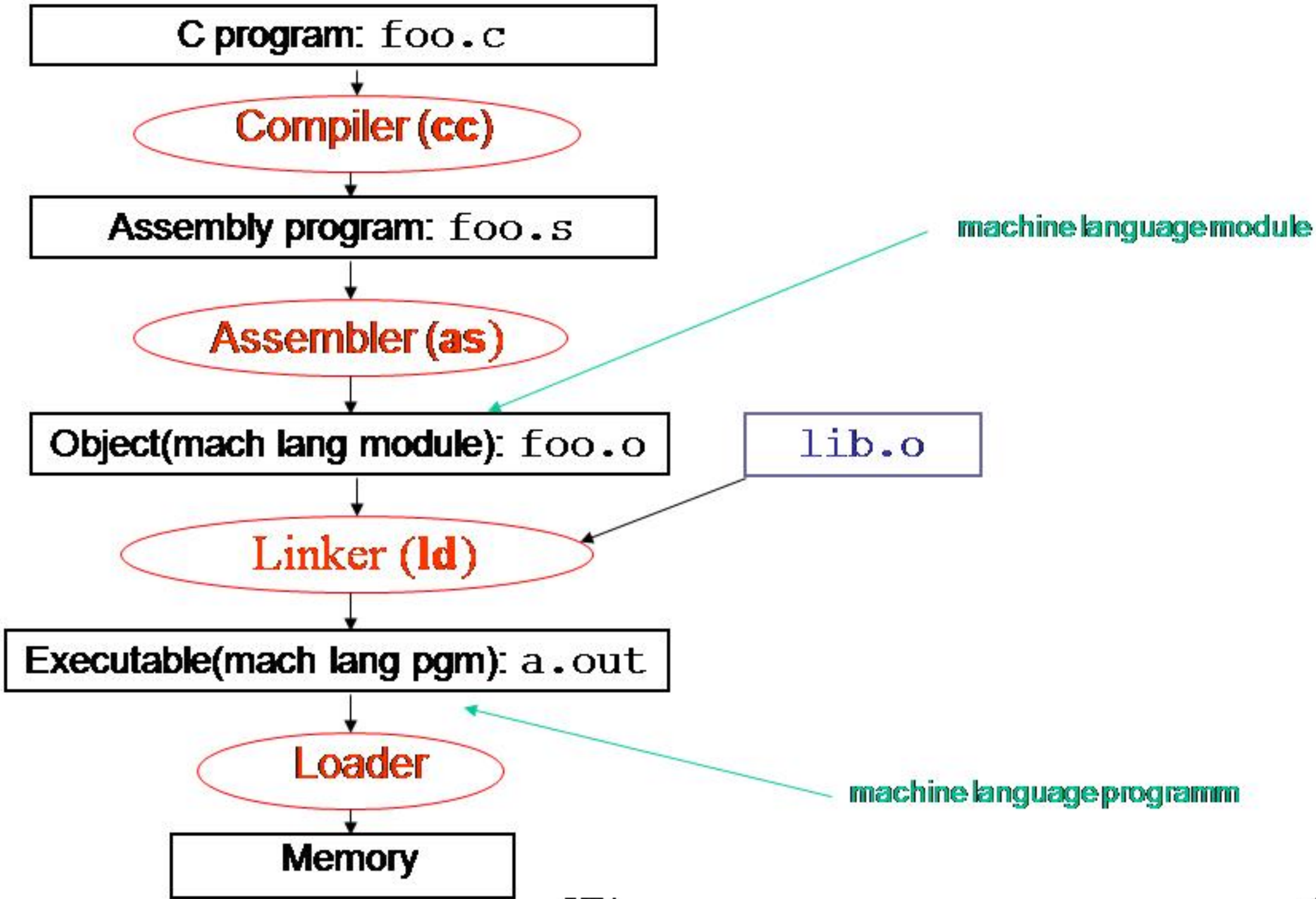
MIPS assembly language

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
Uncondi-tional 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

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



Assembler

- Čte a používá direktivy
- Nahrazuje makroinstrukce
  - subu $sp,$sp,32
  - sd $a0, 32($sp)
  - mul $t7,$t6,$t5
  - la $a0, 0xAABBCCDD

```
addiu  $sp,  $sp,  -32
sw     $a0,  32($sp)
sw     $a1,  36($sp)
mult   $t6,$t5
mflo   $t7
lui    $at,  0xAABB
ori    $a0, $at, 0xCCDD
```

- Generuje strojní jazyk
- Vytváří objektový soubor (*.o)

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 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
  - 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

- **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

- Hlavička objektového souboru: velikost a poloha ostatních částí objektového souboru
- Kódový segment: strojní kód, binární reprezentace dat zdrojového 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 (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

- 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

- 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

Příklad : C => Asm => Obj => Exe => Run

```
#include <stdio.h>
int main (int argc, char *argv[]) {
 int i;
 int sum = 0;
 for (i = 0; i <= 100; i = i + 1)
      sum = sum + i * i;
 printf ("The sum from 0 .. 100 is %d\n",
  sum);
}
```

Příklad : c => Asm => Obj => Exe => Run

```
   .text
   .align    2
   .globl    main
main:
   subu $sp,$sp,32
   sw    $ra, 20($sp)
   sd    $a0, 32($sp)
   sw    $0,  24($sp)
   sw    $0,  28($sp)
loop:
   lw    $t6, 28($sp)
   mul   $t7, $t6,$t6
   lw    $t8, 24($sp)
   addu $t9, $t8,$t7
   sw    $t9, 24($sp)
   addu $t0, $t6, 1
   sw    $t0, 28($sp)
   ble   $t0,100, loop
   la    $a0, str
   lw    $a1, 24($sp)
   jal  printf
   move $v0, $0
   lw    $ra, 20($sp)
   addiu $sp,$sp,32
   jr    $ra
   .data
   .align    0
str:
   .asciiz   "The sum
 from 0 .. 100 is %d\n"
```

Příklad: C => Asm => Obj => Exe => Run

Nahradí makroinstrukce; přiřadí adresy (start na 0x00)

```
00 addiu    $29,$29,-32          30 addiu    $8,$14, 1
04 sw       $31,20($29)          34 sw       $8,28($29)
08 sw       $4, 32($29)          38 slti     $1,$8, 101
0c sw       $5, 36($29)          3c bne      $1,$0, loop
10 sw       $0, 24($29)          40 lui      $4, hi.str
14 sw       $0, 28($29)          44 ori      $4,$4,lo.str
18 lw       $14,28($29)          48 lw       $5,24($29)
1c mult     $14,$14              4c jal      printf
20 mflo     $15                  50 add      $2, $0, $0
24 lw       $24,24($29)          54 lw       $31,20($29)
28 addu     $25,$24,$15          58 addiu    $29,$29,32
2c sw       $25,24($29)          5c jr       $31
                                 60 The
```

Tabulka symbolů a relokační tabulka

- Tabulka symbolů
  - Návěští Adresa

```
main:     0x00000000
loop:     0x00000018
str:      0x10000430
printf:   0x004003b0
```

- Relokační informace
  - **Adresa** **Typ instr.** **Závislost**
  - `0x00000040 HI16 str`
  - `0x00000044 LO16 str`
  - `0x0000004c jal printf`

Příklad : C => Asm => Obj => Exe => Run

```
00 addiu    $29,$29,-32
04 sw       $31,20($29)
08 sw       $4,32($29)
0c sw       $5,36($29)
10 sw       $0, 24($29)
14 sw       $0, 28($29)
18 lw       $14, 28($29)
1c multu    $14, $14
20 mflo     $15
24 lw       $24, 24($29)
28 addu     $25,$24,$15
2c sw       $25, 24($29)
```

```
30 addiu    $8,$14, 1
34 sw       $8,28($29)
38 slti     $1,$8, 101
3c bne      $1,$0, -9
40 lui      $4, 4096
44 ori      $4,$4,1072
48 lw       $5,24($29)
4c jal      1048812
50 add      $2, $0, $0
54 lw       $31,20($29)
58 addiu    $29,$29,32
5c jr       $31
```

Příklad : C => Asm => Obj => Exe => Run

```
0x00400000    00100111101111011111111111100000
0x00400004    10101111101111110000000000010100
0x00400008    10101111101001000000000000100000
0x0040000c    10101111101001010000000000100100
0x00400010    10101111101000000000000000011000
0x00400014    10101111101000000000000000011100
0x00400018    10001111101011100000000000011100
0x0040001c    10001111101110000000000000011000
0x00400020    00000001110011100000000000011001
0x00400024    00100101110010000000000000000001
0x00400028    00101001000000100000000001100101
0x0040002c    10101111101010000000000000011100
0x00400030    00000000000000001111000000010010
0x00400034    00000011000011111110010000100001
0x00400038    00010100001000001111111111110111
0x0040003c    10101111101110010000000000011000
0x00400040    00111100000001000001000000000000
0x00400044    10001111101001010000000000011000
0x00400048    00001100000100000000000011101100
0x0040004c    00100100100001000000010000110000
0x00400050    10001111101111110000000000010100
0x00400054    00100111101111010000000000100000
0x00400058    00000011111000000000000000001000
0x0040005c    00000000000000000100000100001
```

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.

Závěr

- Objekty jsou v počítači reprezentovány jako bitové vzory: *n* bitů => 2^n různých vzorů
- Dekadická čísla - konvence lidí
- Binární čísla – konvence u počítačů (fyzikálně opodstatněná !!!)
- Dvojkový doplněk – nejrozšířenější ve výpočetní technice: budeme ho dále používat
- Operace počítače nad číselnou reprezentací je *abstrakce* reálných operací nad reálnými objekty
- Přetečení:
  - Čísel je nekonečný počet
  - Počítače jsou "omezené"

Závěr

- Data mohou znamenat cokoliv
- Datové typy MIPS : Číslo, řetězec, boolean
- Adresování: Pointery, hodnoty
  - Mnoho adresních módů (adresa přímá, nepřímá,…)
  - Přístup k hlavní paměti – bázové adresování
- Pole: *velké „kusy" paměti*
  - Pointery versus stack
  - Pozor na díry v paměti!

Alternativní architektury
Aritmeticko/logické operace

Aritmeticko-logická jednotka (ALU) a podpora ALU u MIPS

Přehled koncepce (opakování z minulé hodiny)

- Přidělení paměti proměnným
  - Stackové rámce - frame (volající/volaný), statické, heap
- Pointery
  - Adresování paměti
  - Předávání argumentů: pole, struktury
  - Efektivní použití pointerů (aritmetika pointerů)
  - Problémy
    - Chybné paměťové reference (segmentation fault)
    - „Díry“ v paměti

Přehled

- Alternativní návrhy ISA
- Principy návrhu (konvence)
- Rovnice výkonu CPU
- RISC vs. CISC
- Historická perspektiva
- PowerPC a 80x86

Přehled (pokračování)

- Hardwarové komponenty (bloky)
- Návrh ALU
- Implementace ALU
  - 1-bitová ALU
  - 32-bitová ALU

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

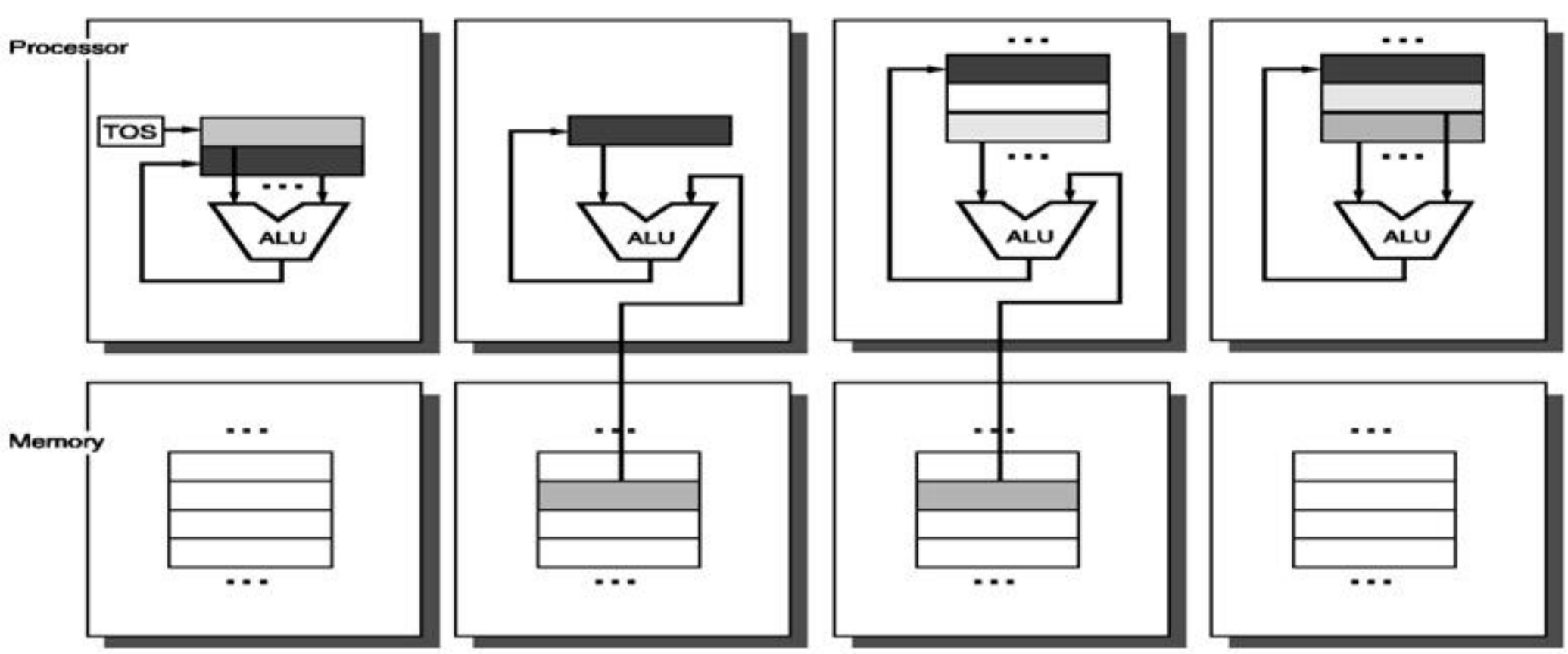
Orientace architektury na:

- 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

Ilustrace typů architektur (ISA)

Assembler pro C:=A+B:

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`



Ilustrace typů architektur (ISA)

C = A + B;

Accumulator

```
Load   AddressA
Add    AddressC
Store  AddressC
```

Stack

```
Push   AddressA
Push   AddressB
Add
Pop    AddressC
```

Load-Store

```
Load   R1, AddressA
Load   R2, AddressB
Add    R3, R1, R2
Store  R3, AddressC
```

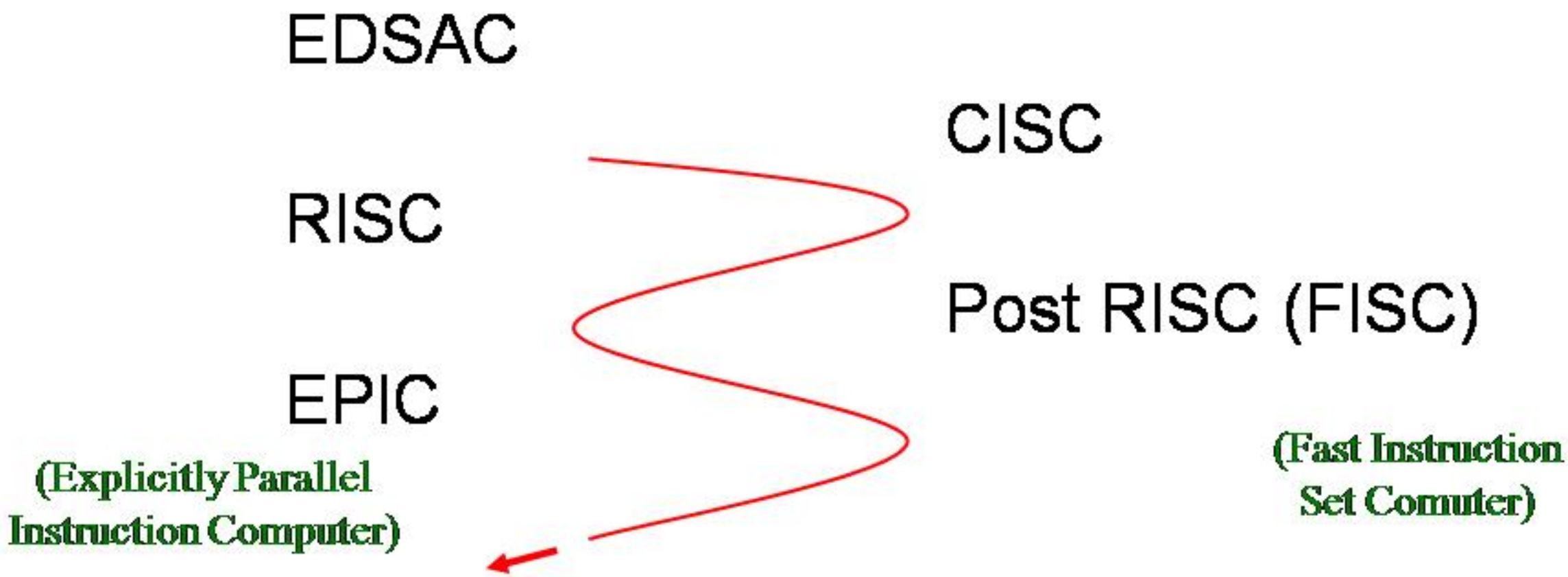
Memory-Memory

```
Add   AddressC, AddressB, AddressA
```

$$CPU_{time} = IC * CPI * Cycle\_time$$

Trendy ISA

- Technologické trendy HW a kompilátorů
  - Mez mezi hardware/software „osciluje“



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 z architektury CDC 6600 (1963)

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]`

    totéž u 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)`

    totéž u MIPS:
    ```
    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
    ```

    totéž u MIPS:
    ```
    addi, $t0, $t0, -1;
    bne  $t0, $zero, Loop
    ```

Mezníky architektury 80x86

- 1978: 8086, 16 bit architektura (64KB), žádné GPR
- 1980: 8087 FP koprocesor, 60 + instrukcí, 80-bit stack, žádné GPR
- 1982: 80286, 24-bitová adresa, ochrany paměti
- 1985: 80386, 32 bitů, nové adresní režimy, 8 GPR
- 1989-1995: 80486, Pentium, Pentium Pro - přidáno několik nových instrukcí (pro zvýšení výkonu)
- 1997: MMX + 57 instrukcí (SIMD)
- 1999: PIII + 70 „multimediálních“ instrukcí
- 2000: P4 +144 „multimediálních“ instrukcí

Zlatá pouta kompatibility vzhůru => *„tuto architekturu je těžké vysvětlit a nemožné mít rád“ (citát)*

Architektura X86

- Dvouadresní architektura
  - Jeden z operandů je zároveň cílem

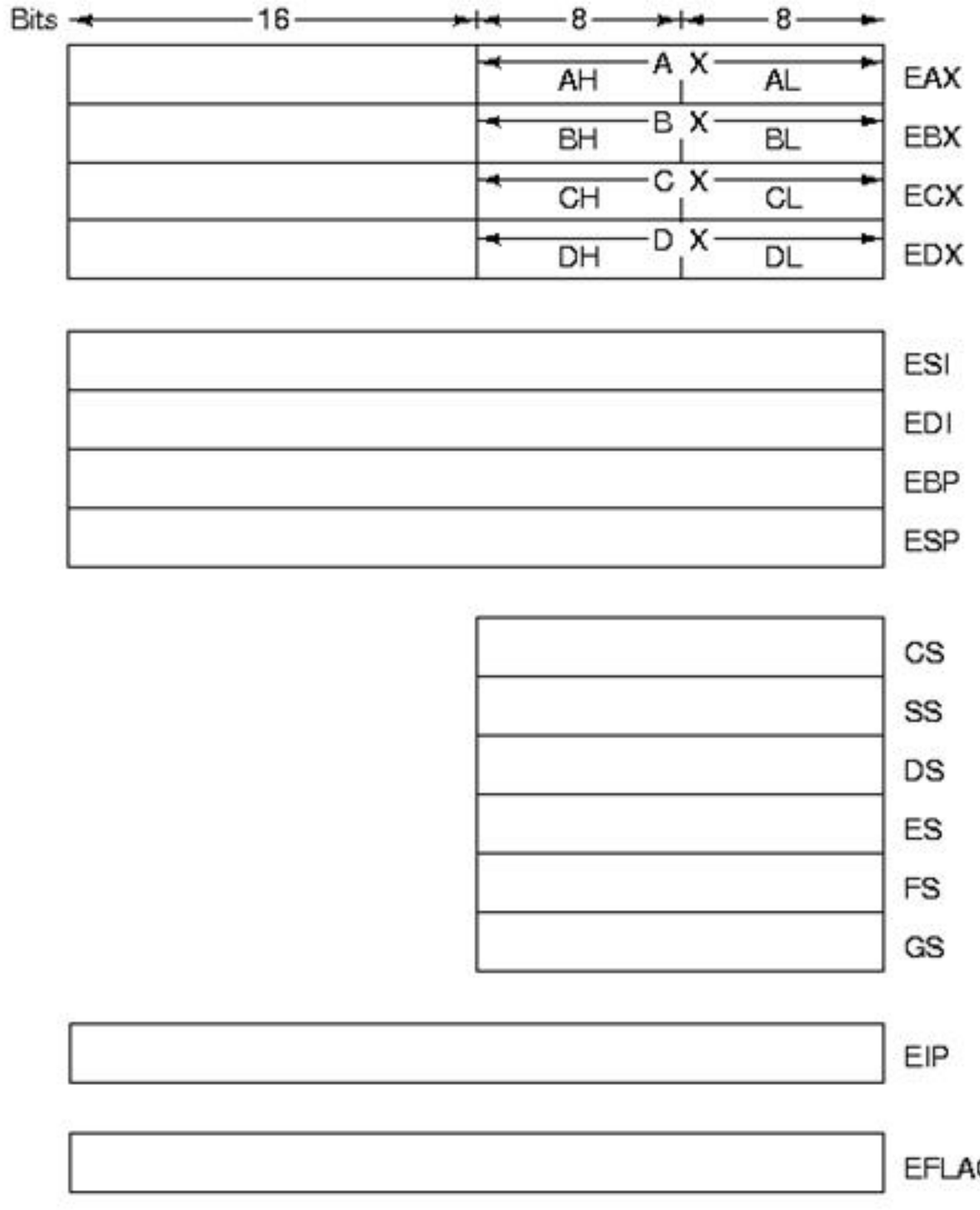
    add $s1,$s0 # s0=s0+s1 (**C**: a += b;)
  - Výhoda: malé instrukce → malý kód → **rychlejší**
- Architektura typu Register-Memory
  - Jeden operand může být v paměti; druhý je v registru

    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 přímé operandy

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
  - Signed/Unsigned integer (8, 16, a 32 bitů)
  - Binary Coded Decimal integer čísla
  - Floating Point (32 a 64 bitů)
- Floating point jednotka používá oddělený stack

Registry X86



Main arithmetic register

Pointers (memory addresses)

Loops

Multiplication and division

Pointer to source string

Pointer to destination string

Base of the current stack frame ($fp)

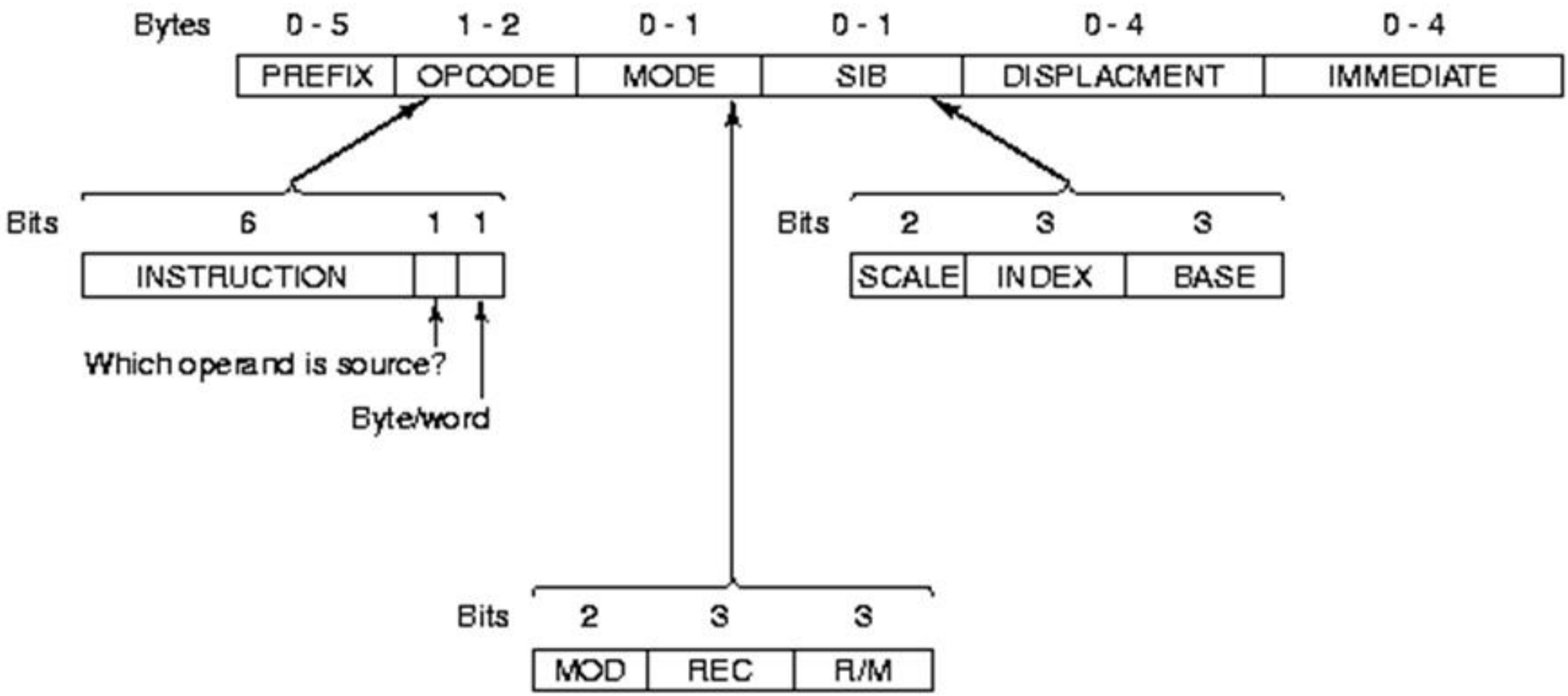
Stack pointer

Support for 8088 attempt to address 2^{20} bytes using 16-bit addresses

Program counter

Processor State Word

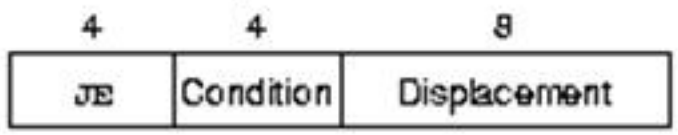
Instrukční formáty X86



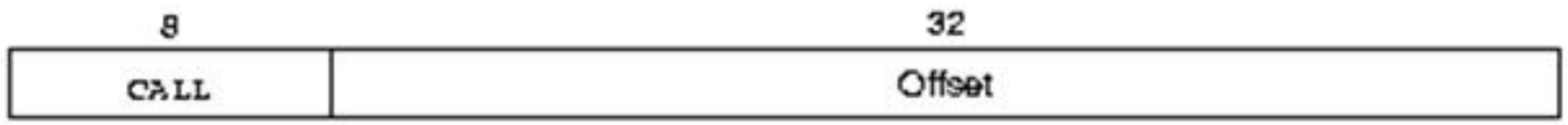
- Velmi složité a nepravidelné
  - Šest polí s proměnnou délkou
  - Dalších pět polí se může vyskytnout

Příklady instrukčních formátů X86

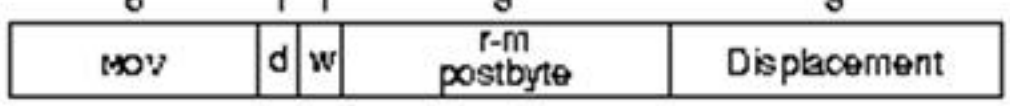
a. JE EIP + displacement



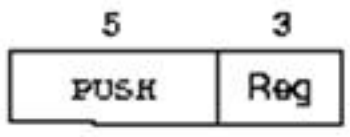
b. CALL



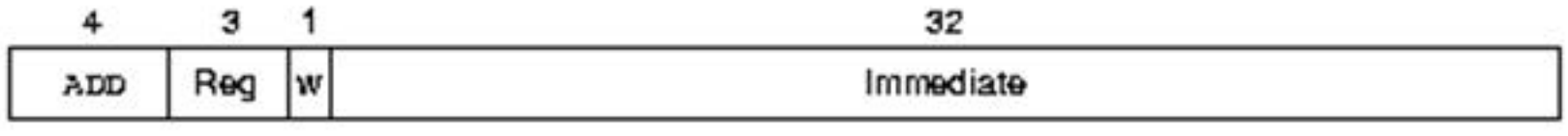
c. MOV EBX, [EDI + 45]



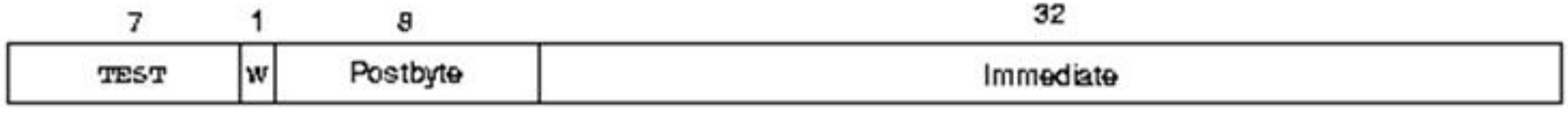
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



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

	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)

- 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

```
while (save[i]==k)
    i = i + j;
```

X86

(*i,j,k* => *%edx, %esi, %ebx*)

```
        leal   -400(%ebp),%eax
.Loop:  cmpl %ebx,(%eax,%edx,4)
        jne    .Exit
        addl  %esi,%edx
        j      .Loop
.Exit:
```

MIPS

(*i,j,k* => *$s3, $s4, $s5*)

```
Loop:  sll   $t1, $s3, 2
       add  $t1, $t1, $s6
       lw   $t0, 0($t1)
       bne  $t0, $s5, Exit
       add  $s3, $s3, $s4
       j      Loop
Exit:
```

PIII, P4 a AMD

- PC World magazine, Nov. 20, 2000
  - WorldBench 2000 benchmark (aplikace obchodního charakteru)
  - P4 score @ 1.5 GHz: 164 (větší hodnota znamená lepší)
  - PIII score @ 1.0 GHz: 167
  - AMD Athlon @ 1.2 GHz: 180
  - (Mediální aplikace vycházejí lépe na P4 vs. PIII)

- Proč? => rovnice výkonu CPU
  - Čas = Počet instrukcí x CPI x 1/Frekvence_hodin
  - Počet instrukcí je stejný jako pro x86
  - Frekvence_hodin : P4 > Athlon > PIII
  - Proč může být P4 pomalejší?
  - Střední CPI procesoru P4 musí být horší než Athlonu a PIII

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!

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

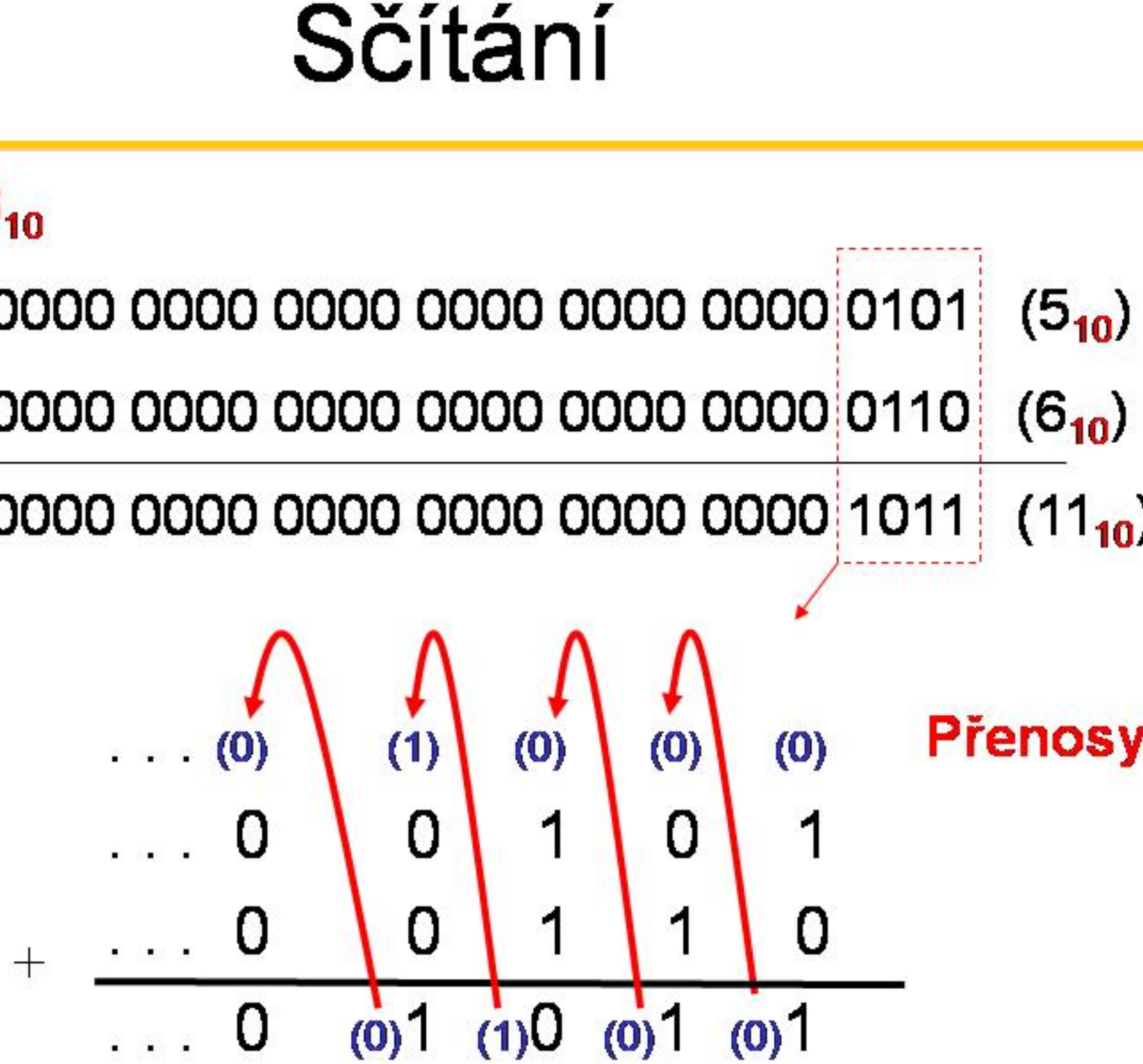
Souhrn

N decimal	(+N) Positive	(-N) Sign/magnitude	(-N) 1's complement	(-N) 2's complement
0	00000000	10000000	11111111	00000000
1	00000001	10000001	11111110	11111111
2	00000010	10000010	11111101	11111110
3	00000011	10000011	11111100	11111101
4	00000100	10000100	11111011	11111100
5	00000101	10000101	11111010	11111011
6	00000110	10000110	11111001	11111010
7	00000111	10000111	11111000	11111001
8	00001000	10001000	11110111	11111000
9	00001001	10001001	11110110	11110111
10	00001010	10001010	11110101	11110110
20	00010100	10010100	11101011	11101100
50	00110010	10110010	11001101	11001110
100	01100100	11100100	10011011	10011100
127	01111111	11111111	10000000	10000001
128	NA	NA	NA	10000000

Sčítání

- $5_{10} + 6_{10}$

	0000 0000 0000 0000 0000 0000 0000 0101	(5_{10})
+	0000 0000 0000 0000 0000 0000 0000 0110	(6_{10})
=	0000 0000 0000 0000 0000 0000 0000 1011	(11_{10})



Odčítání

- $12_{10} - 5_{10}$

0000 0000 0000 0000 0000 0000 0000 1100 (12_{10})

\- 0000 0000 0000 0000 0000 0000 0000 0101 (5_{10})

= 0000 0000 0000 0000 0000 0000 0000 0111 (7_{10})

- $12_{10} - 5_{10} = 12_{10} + (-5_{10})$

0000 0000 0000 0000 0000 0000 0000 1100 (12_{10})

\+ 1111 1111 1111 1111 1111 1111 1111 1011 (-5_{10})

= 0000 0000 0000 0000 0000 0000 0000 0111 (7_{10})

Přetečení

- Množina čísel vzhledem k operacím + - * / není uzavřená
- Přetečení
  - Výsledek operace nelze zobrazit v původním rozsahu (na 32 bitech)
- Přetečení nemůže nastat
  - Sčítání: mají-li sčítanci opačná znaménka
  - Odčítání: mají-li operandy stejná znaménka
- Detekce přetečení
  - Výsledek potřebuje 33 bitů
  - Sčítání: liší se c_{n+1} a c_n
  - Odčítání: promyslete sami

Příklady

- 4 bity (na rozdíl od 32 u MIPS) => lze zobrazit čísla [-8 : 7]

7 + 6

	0	1	1	1 (7_{10})
+	0	1	1	0 (6_{10})
	1	1	0	1 (13_{10})

-7 + -6

	1	0	0	1 (-7_{10})
+	1	0	1	0 (-6_{10})
	0	0	1	1 (-13_{10})

-7 – 6 = -7 + -6

	1	0	0	1 (-7_{10})
+	1	0	1	0 (-6_{10})
	0	0	1	1 (-13_{10})

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

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

Detekce přetečení

C_{n-1}	A_{n-1}	B_{n-1}	S_{n-1}	C_n	OF
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	0	1	1
1	0	0	1	0	1
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	1	1	0

Discarded

- Test MSB
- P: kladné; N: záporné
- N + N = N
  - P + P = P
  - P+N nebo N+P vždy „spadne“ do zobrazovaného intervalu
    - Např. -128+P nemůže být menší než -128 nebo větší než 127
- Problém nastává pro:
  - N+N = P
  - P+P = N

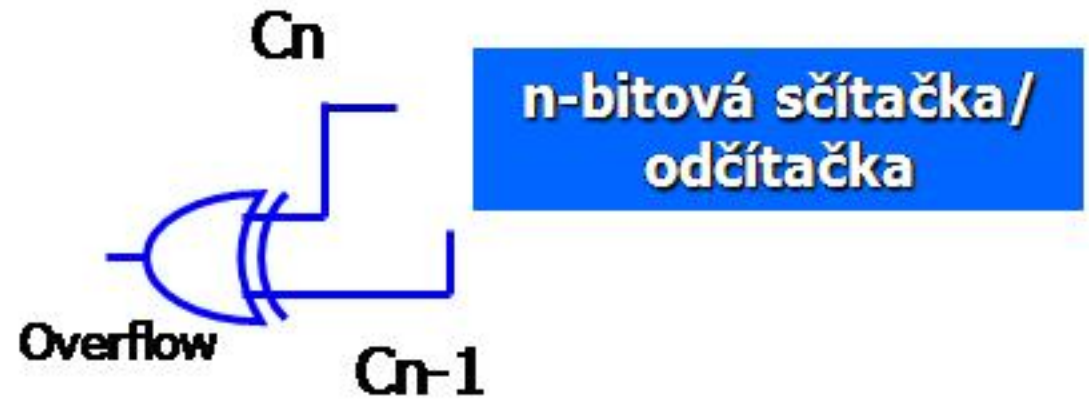
Detekce přetečení

C_{n-1}	A_{n-1}	B_{n-1}	S_{n-1}	C_n	OF
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	0	1	1
1	0	0	1	0	1
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	1	1	0

$$OF = \overline{C}AB + C\overline{A}\overline{B}$$

nebo

$$OF = C_{n-1} \oplus C_n$$



Co způsobí přetečení (Overflow)

- Nastane výjimka (interrupt)
  - **Řadič způsobí skok na předem stanovenou adresu obsluhy výjimky**
  - **Adresa přerušení (kde bylo přerušeno) se uloží kvůli možnému dalšímu pokračování**
- Detaily závisejí na software (role OS)
- Detekce přetečení není požadována vždy
  — nové instrukce MIPS: addu, addiu, subu

Pozn.: addiu *pracuje s rozšířením znaménka!*

- *stejně tak addi, vyjma no overflow exception*

Pozn.: **sltu, sltiu** *pro porovnání bez znaménka*

Podmíněné skoky při přetečení

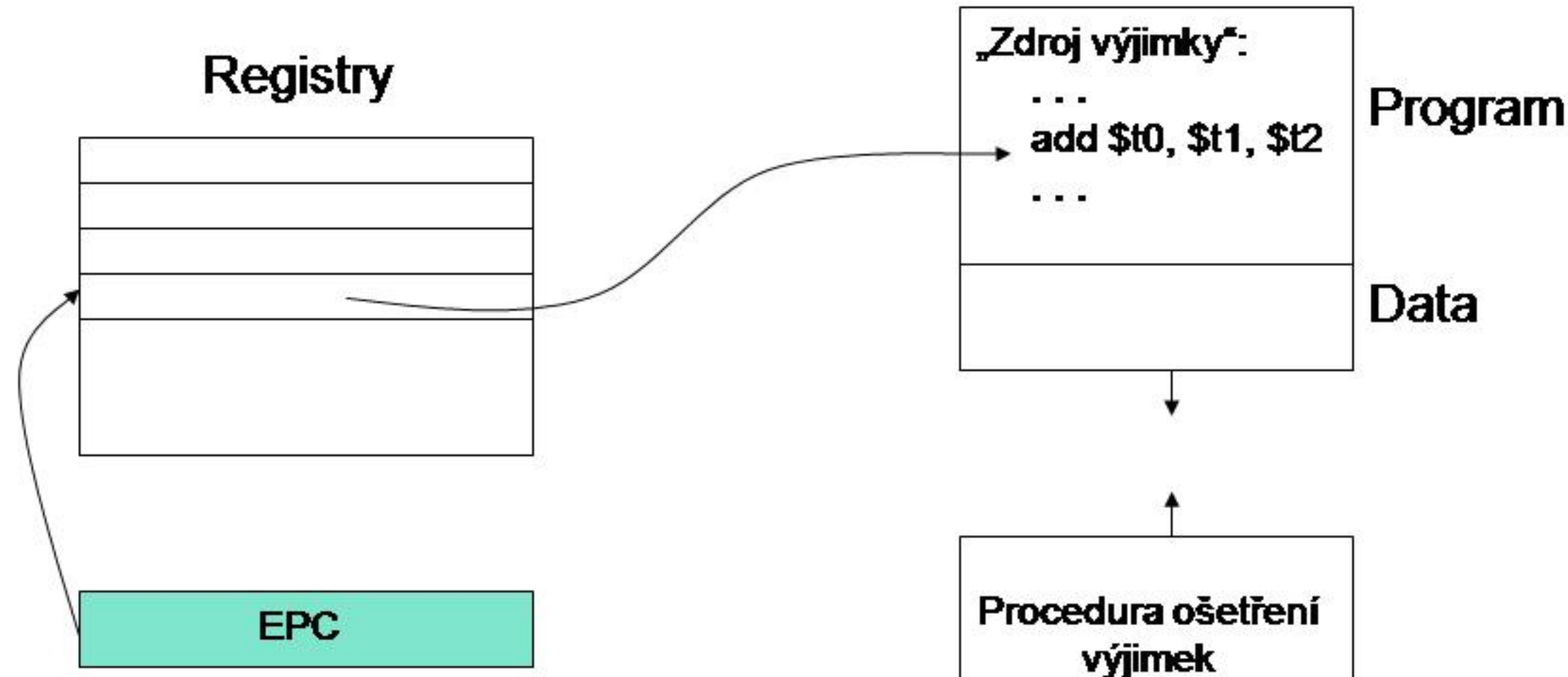
Sčítání se znaménkem – softwarová detekce přetečení

```
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

Logické operace

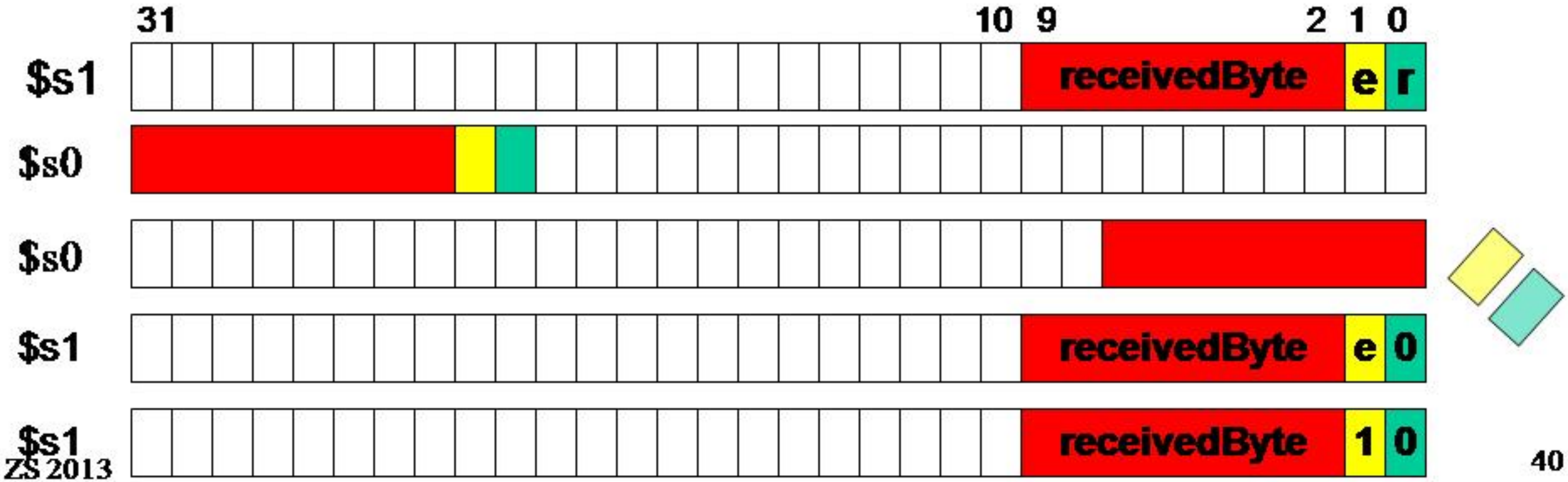
- Operace nad poli bitů v rámci 32-bitových slov
  - Znaky (8 bitů)
  - Bitová pole (v C)
- Logické operace
  - sll posuv vlevo
  - srl posuv vpravo
  - and, andi bitové AND
  - or, ori bitové OR
- Bitové operátory - operandy jsou *bitové vektory*

Bitová pole v C

```
struct {
    unsigned int ready:          1;
    unsigned int enable:         1;
    unsigned int receivedByte:   8;
} receiver;
int data = receiver.receiverByte;
receiver.ready = 0;
receiver.enable = 1;
```

```
#$s0: data;  $s1: receiver

sll     $s0, $s1, 22
srl     $s0, $s0, 24
andi    $s1, $s1, 0xfffe
ori     $s1, $s1, 0x0002
```



Hardwarové komponenty (bloky)

- ALU se staví z komponent nízké úrovně (implementace)
  – z logických hradel

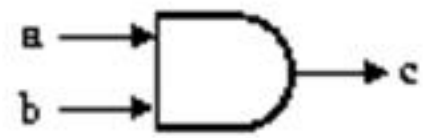
- **Hradlo** (přehled)
  – **Hardwarový prvek, který zpracovává několik vstupů a generuje jeden výstup**
  – **Může být reprezentován pravdivostní tabulkou a nebo logickou rovnicí**
  – **Hradla se vytvářejí z tranzistorů na křemíku**

- Různé hardwarové komponenty:
  – **Hradlo And**
  – **Hradlo Or**
  – **Invertor (not)**
  – **Multiplexor (mux)**

Poznámka:
Označení „hradlo" není zcela korektní, protože všechny jeho vstupy jsou rovnocenné.

Základní elektronické prvky

1. AND gate ($c = a \cdot b$)



a	b	$c = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

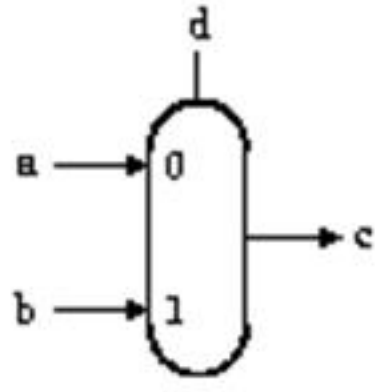
2. OR gate ($c = a + b$)

a	b	$c = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter ($c = \bar{a}$)

a	$c = \bar{a}$
0	1
1	0

4. Multiplexor (MUX)
   (if d == 0
      c = a;
   else
      c = b)



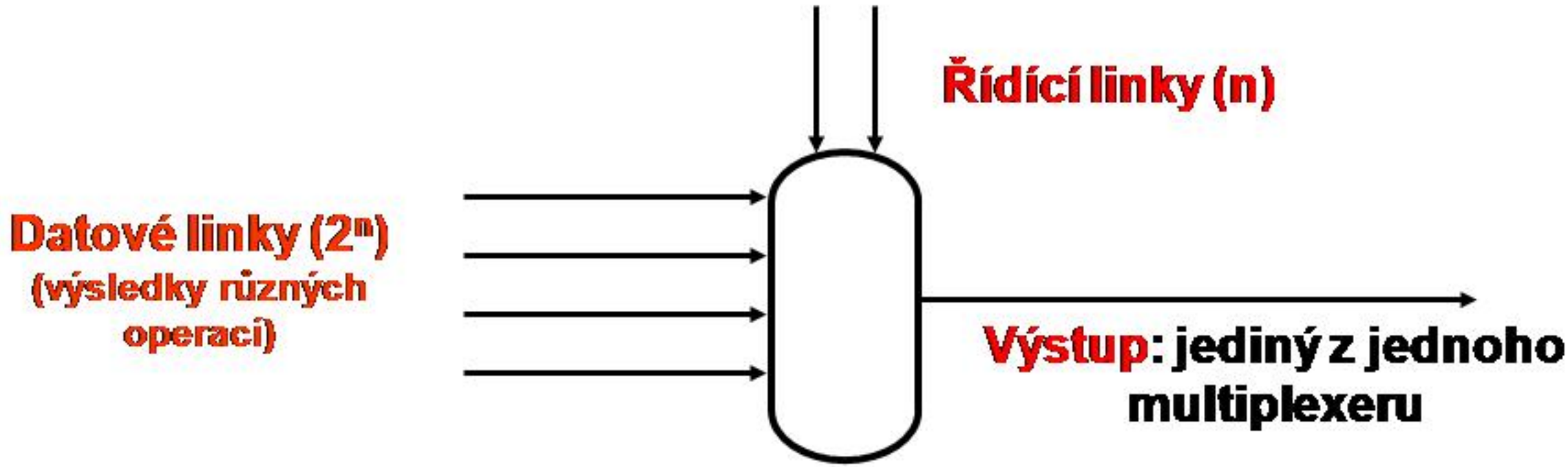
d	c
0	a
1	b

n control bits select between 2^n data bits

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

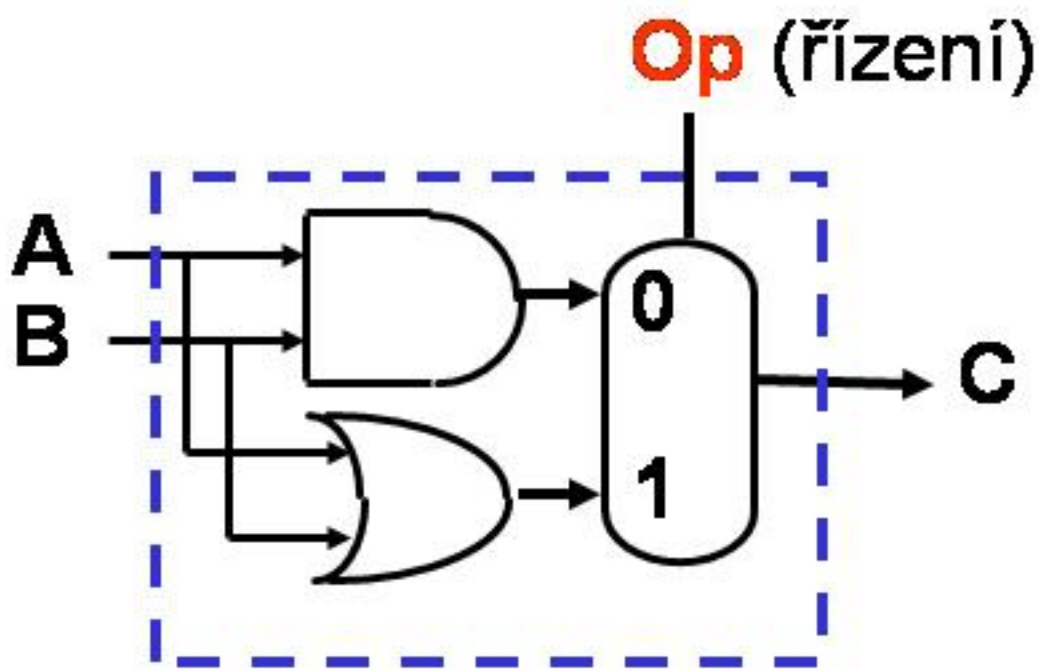
Implementace ALU



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.

Jednobitové logické instrukce

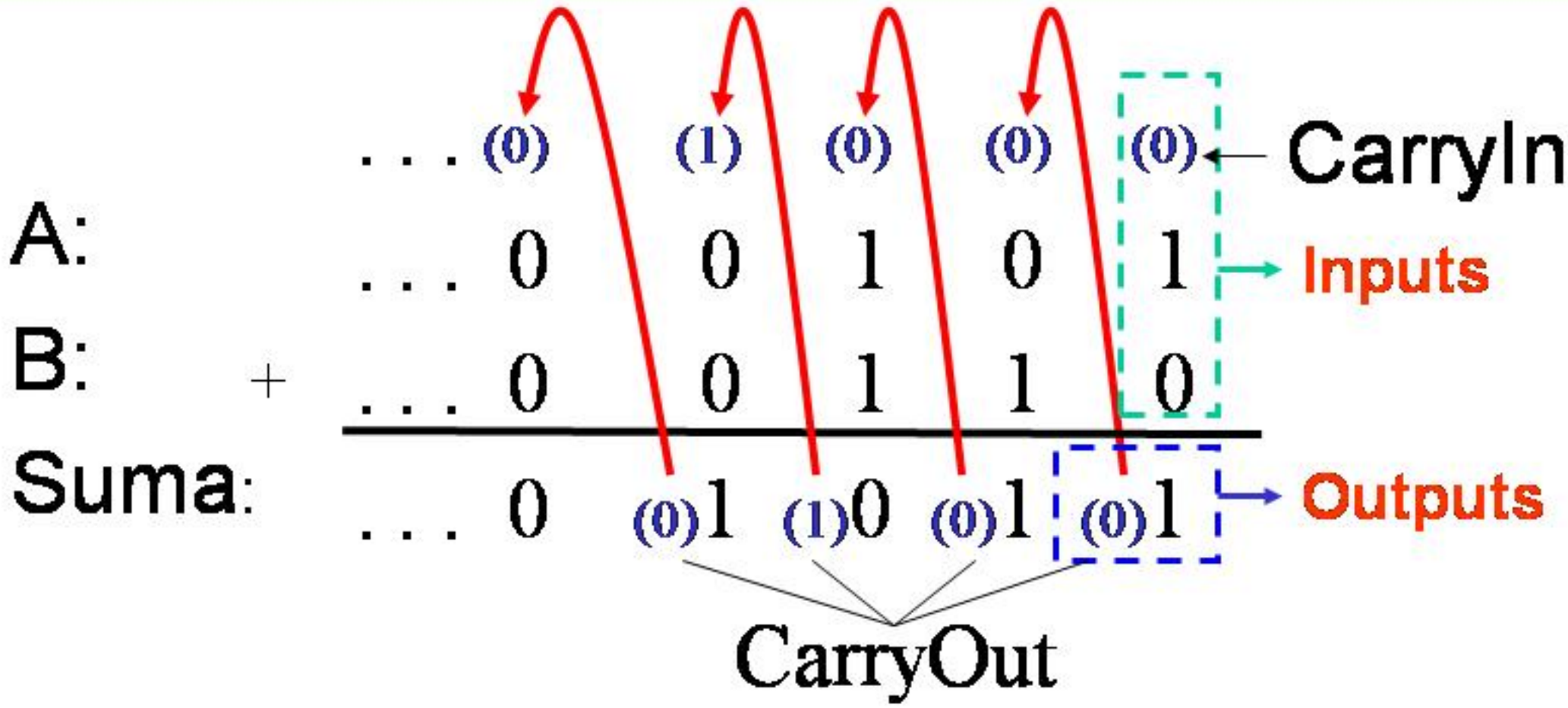
- Přímé mapování na hardwarové komponenty
  - instrukce AND
    - **Jedna datová linka pochází z pouhého hradla AND**
  - instrukce OR
    - **Další datová linka pochází z pouhého hradla OR**



Definice

Op	C
0	A and B
1	A or B

Jednobitová úplná sčítačka

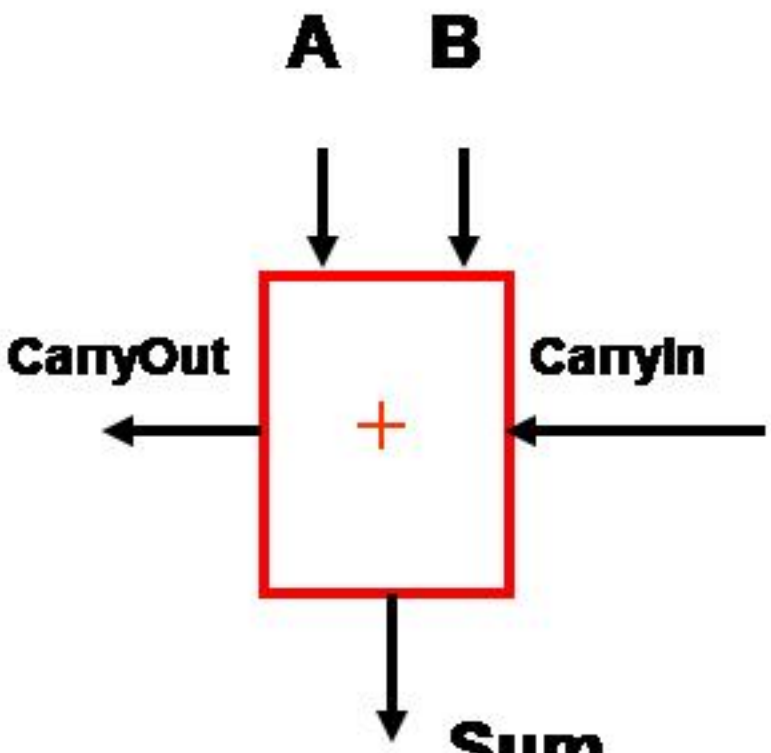


- Každý „bit“ sčítačky má:
  - Tři vstupní signály: A_i, B_i, $CarryIn_i$
  - Dva výstupní signály: Sum_i, $CarryOut_i$

    ($CarryIn_{i+1} = CarryOut_i$)

Pravdivostní tabulka úplné sčítačky

Symbol



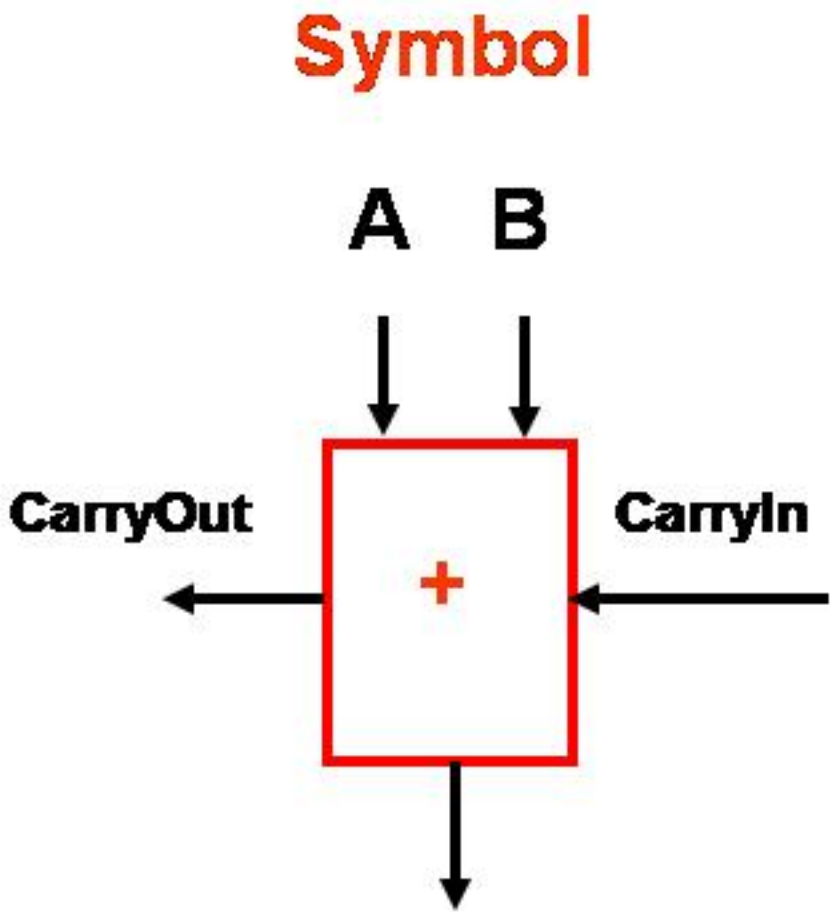
Definice

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



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

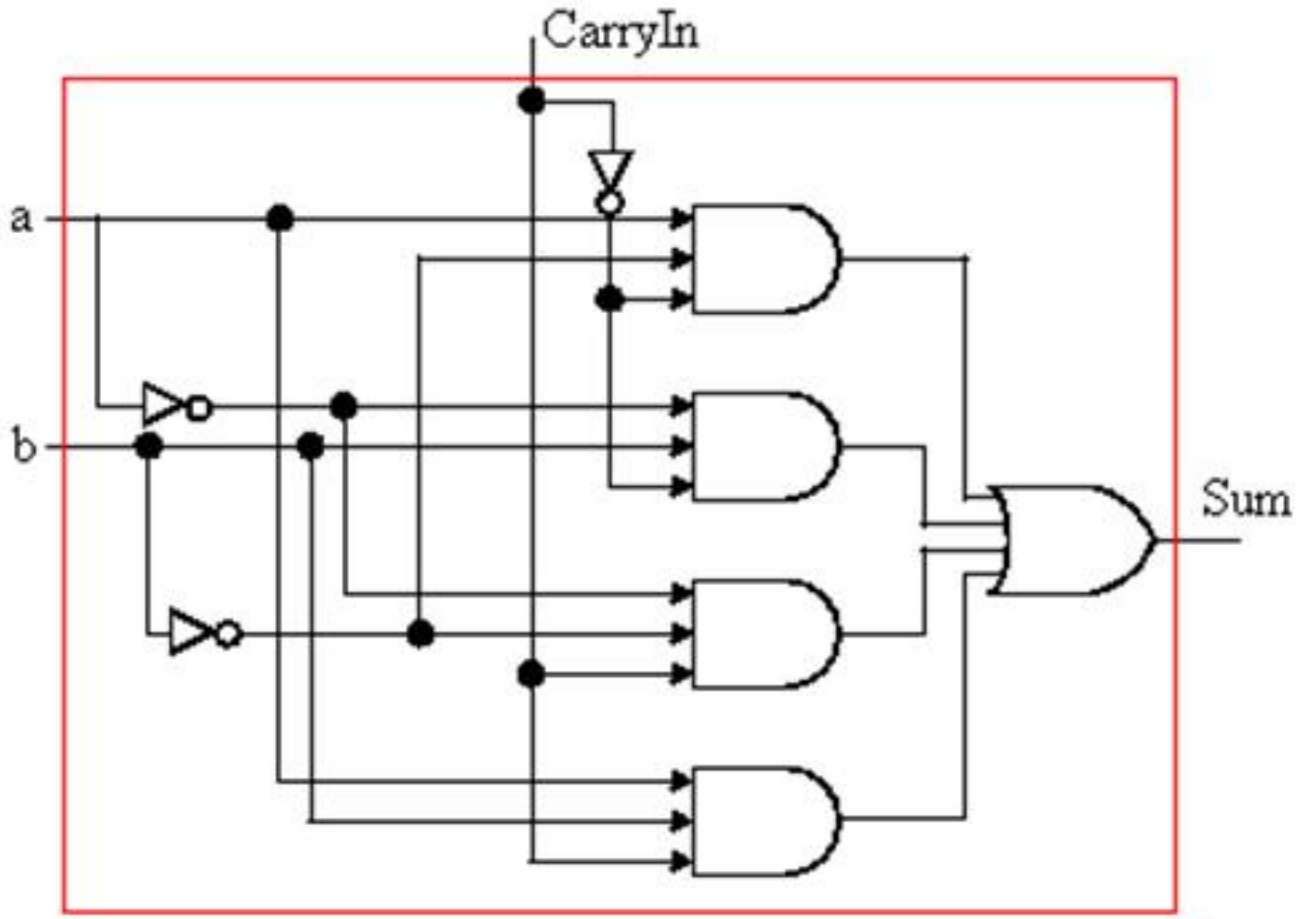
Generační funkce G

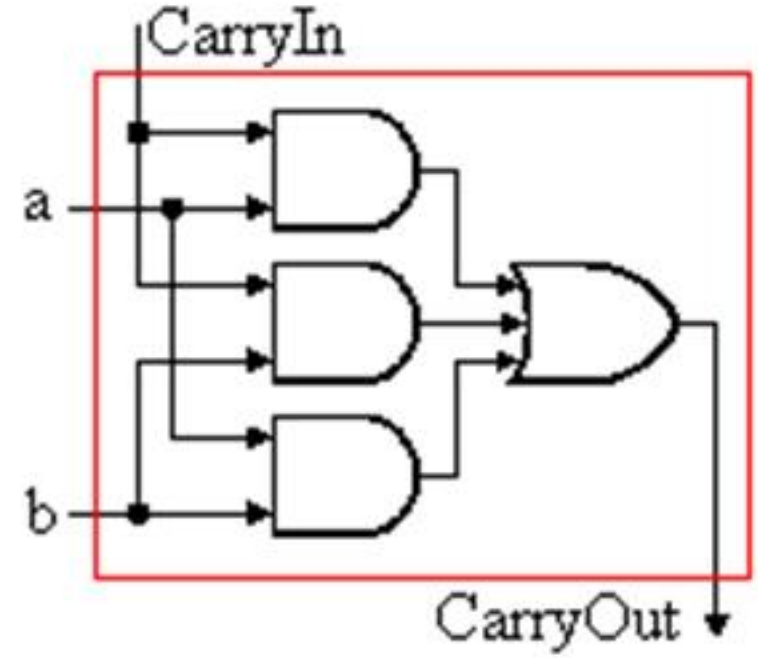
Přenosová funkce P

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

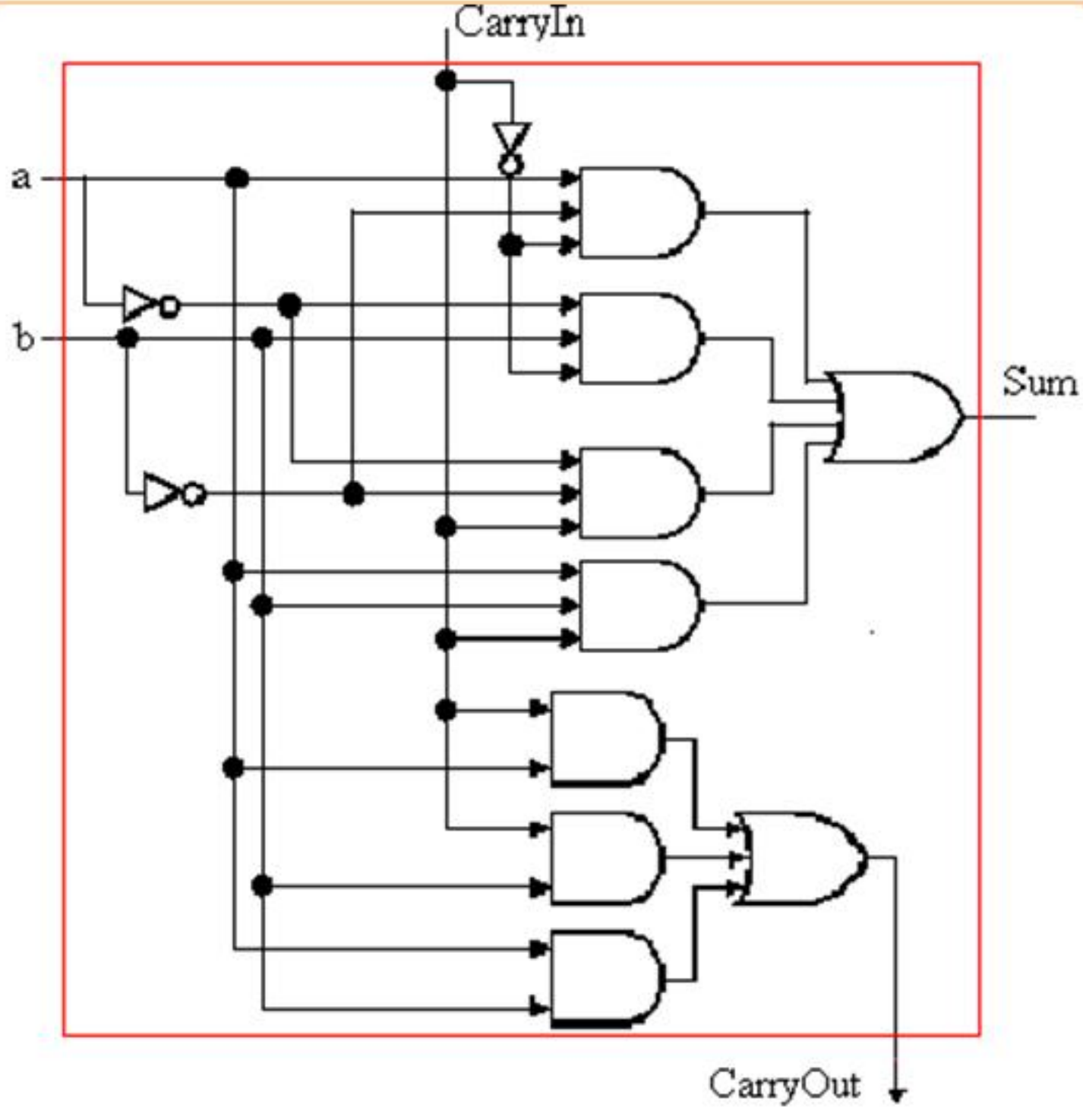
Obvody úplné sčítačky (1/2)

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

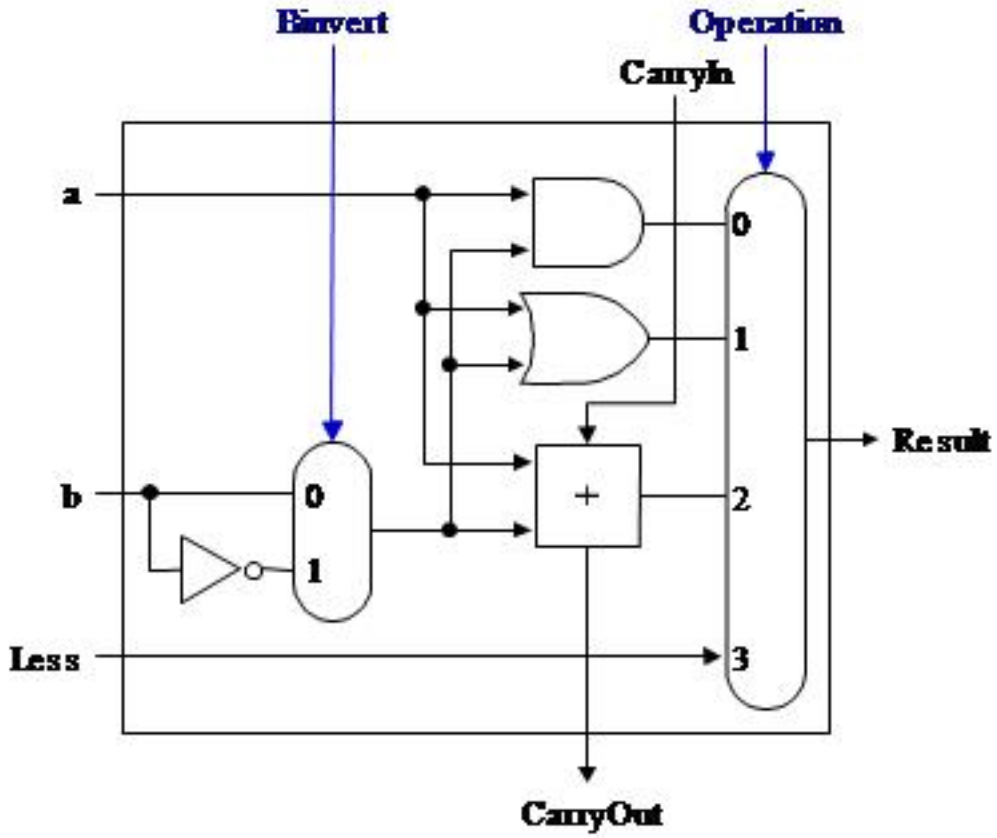




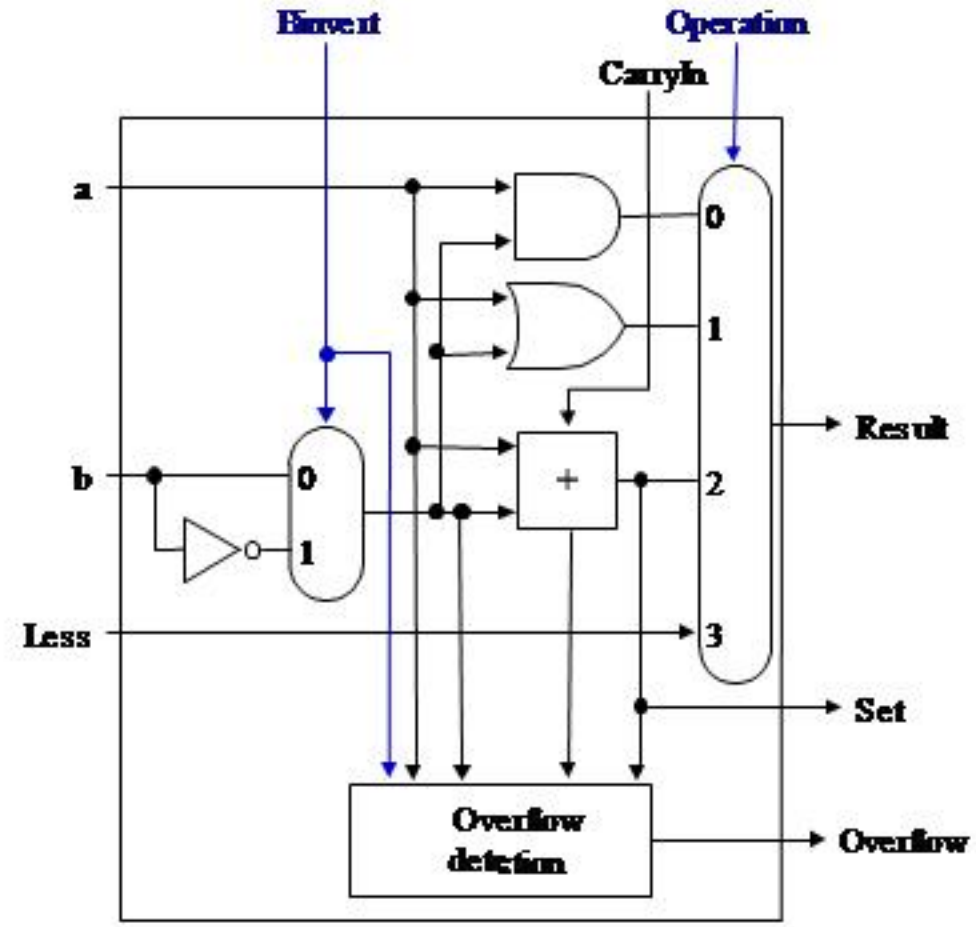
Obvody úplné sčítačky (2/2)



Jednobitová ALU

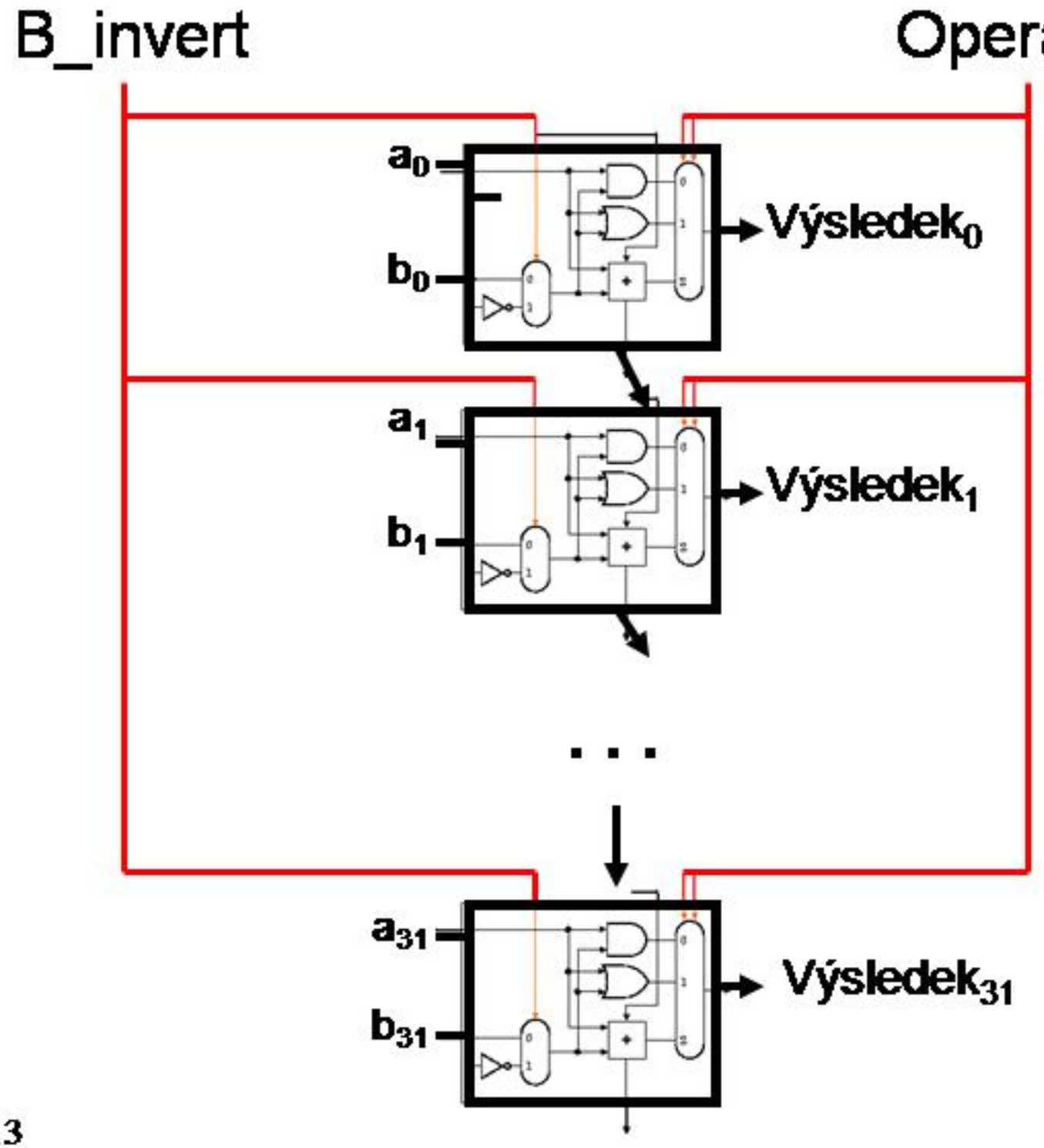


Bit s nejnižší váhou



Ostatní bity

32-bitová ALU



Souhrn

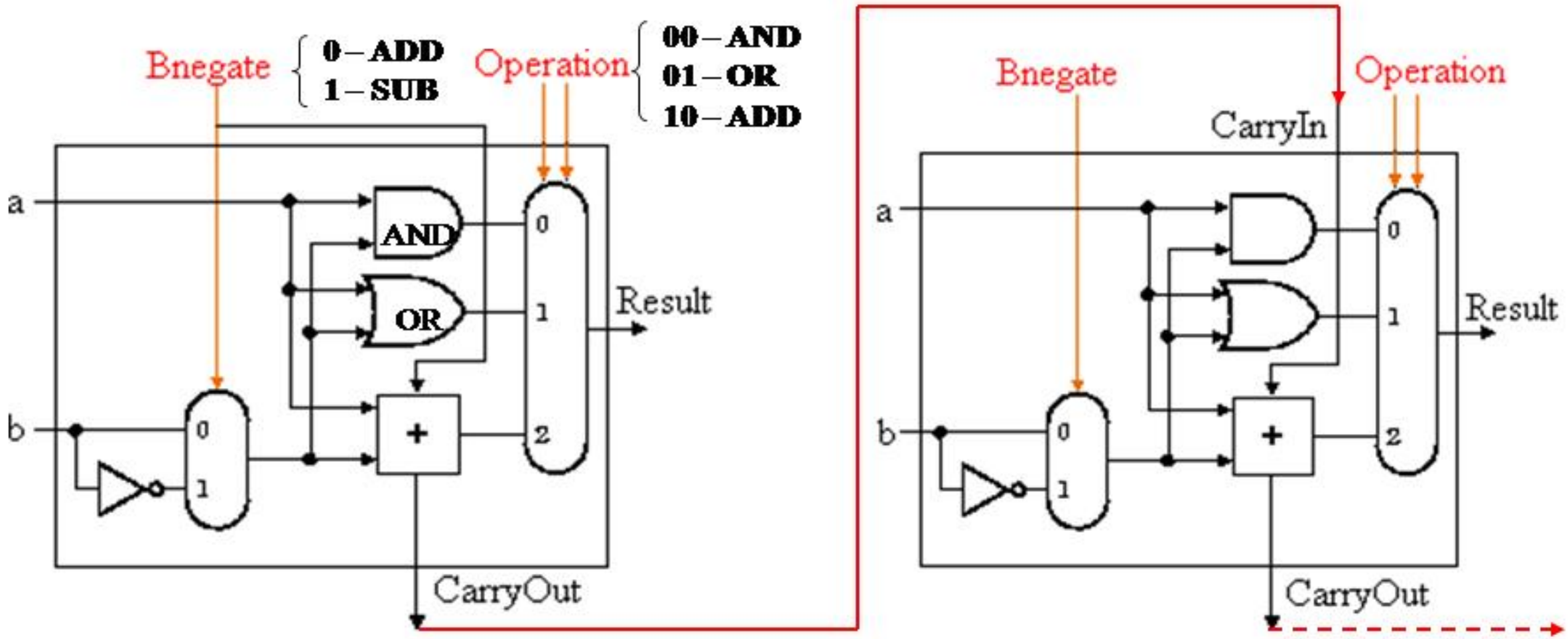
- Stavební bloky: základní hradla (AND, OR a NOT)
- Modulární návrh a implementace
  - Hradla mají větší počet vstupů a jen jeden výstup
  - ALU zpracovává 32-bitová slova (integer)
  - v ALU je implementována řada operací *paralelně*

    => Nejdříve konstrukce 1-bitové ALU
  - *Mux* vybere jednu z mnoha různých operací ALU
  - Podle instrukčního souboru se doplní základní operace ALU tak, aby bylo možno implementovat všechny instrukce
  - *Reprezentace dvojkového doplňku dovoluje použít tentýž hardware pro sčítání i odčítání*

Aplikace na ALU - MIPS

- *Rozšíření* MIPS ALU
- Detekce přetečení
- Instrukce Slt
- Instrukce větvení
- Posuvové instrukce
- Instrukce s přímými operandy
- Výkonnost ALU
  - Výkon vs. cena
  - Sčítačka s urychlením typu „Carry lookahead“
- Alternativy implementace

Opakování: Generická jednobitová ALU



První bit (LSB)

Ostatní bity

Operace: AND, OR, ADD, SUB

Řídící linky: 000 001 010 110

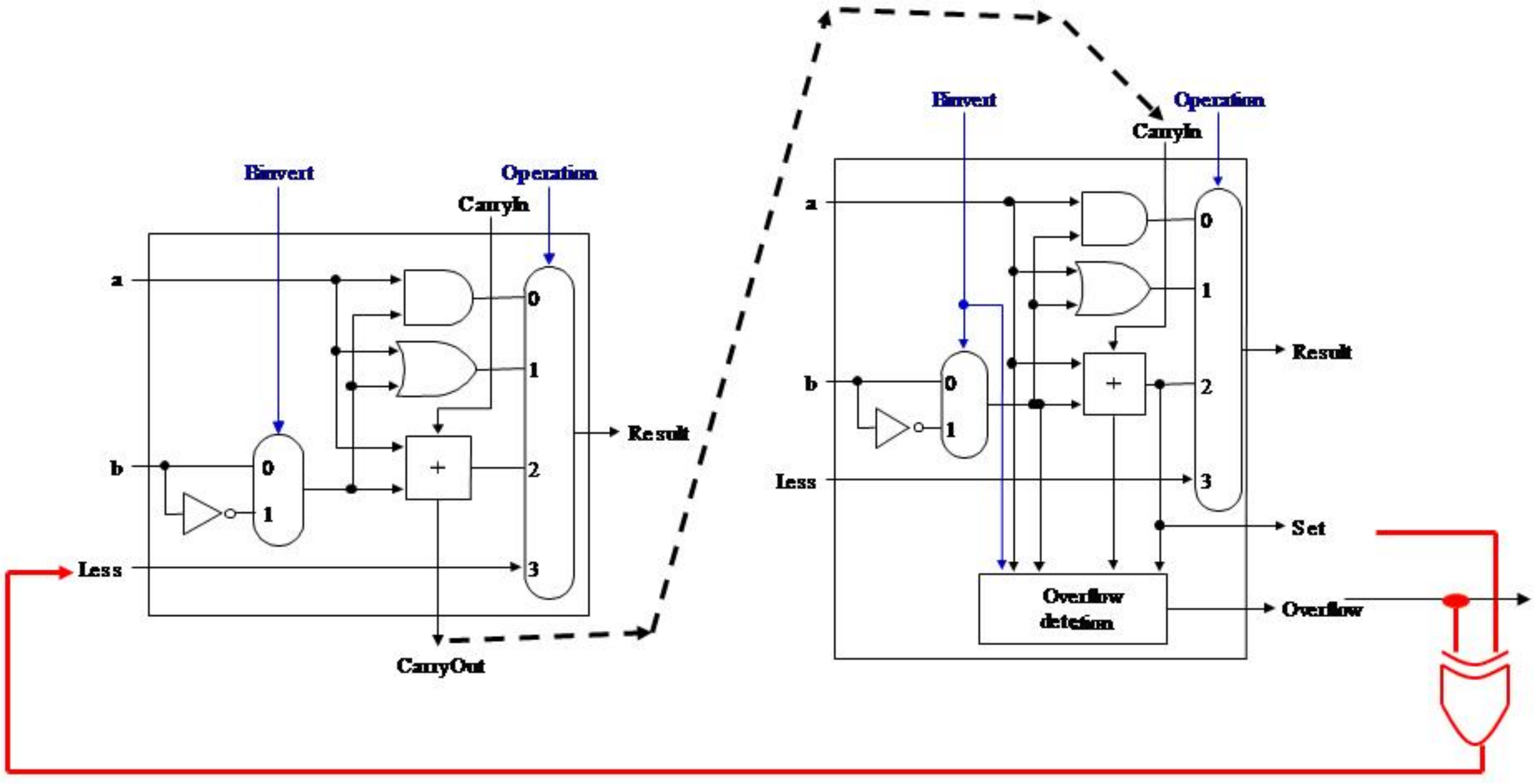
Instrukce Slt

- **Slt rd, rs, rt**

rd: 0000 0000 0000 0000 0000 0000 0000 000r → { 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 (MSB)

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

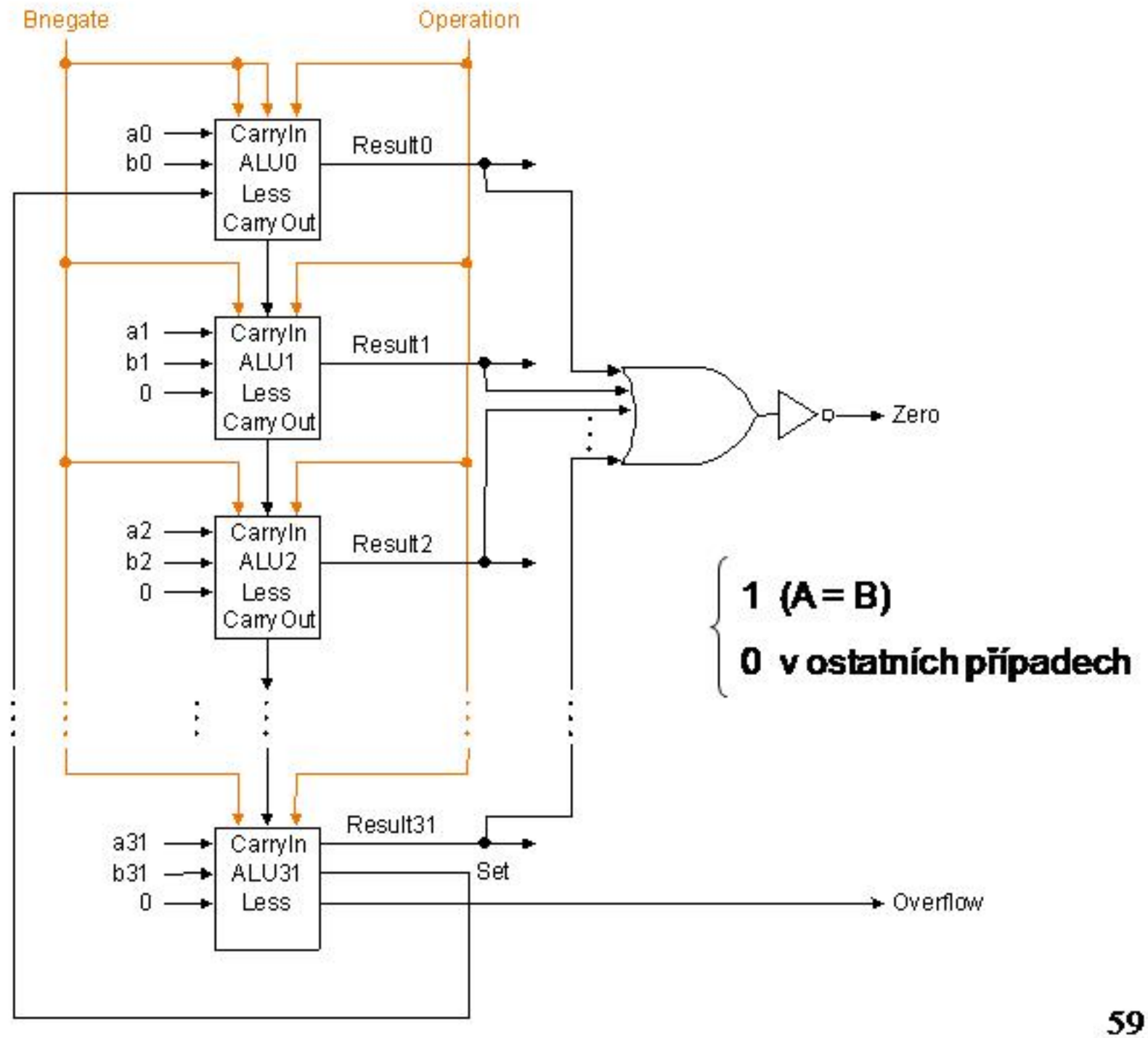
$$Zero = (Result_1 + Result_2 + .. + Result_{31})$$

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

HW podpora větvení

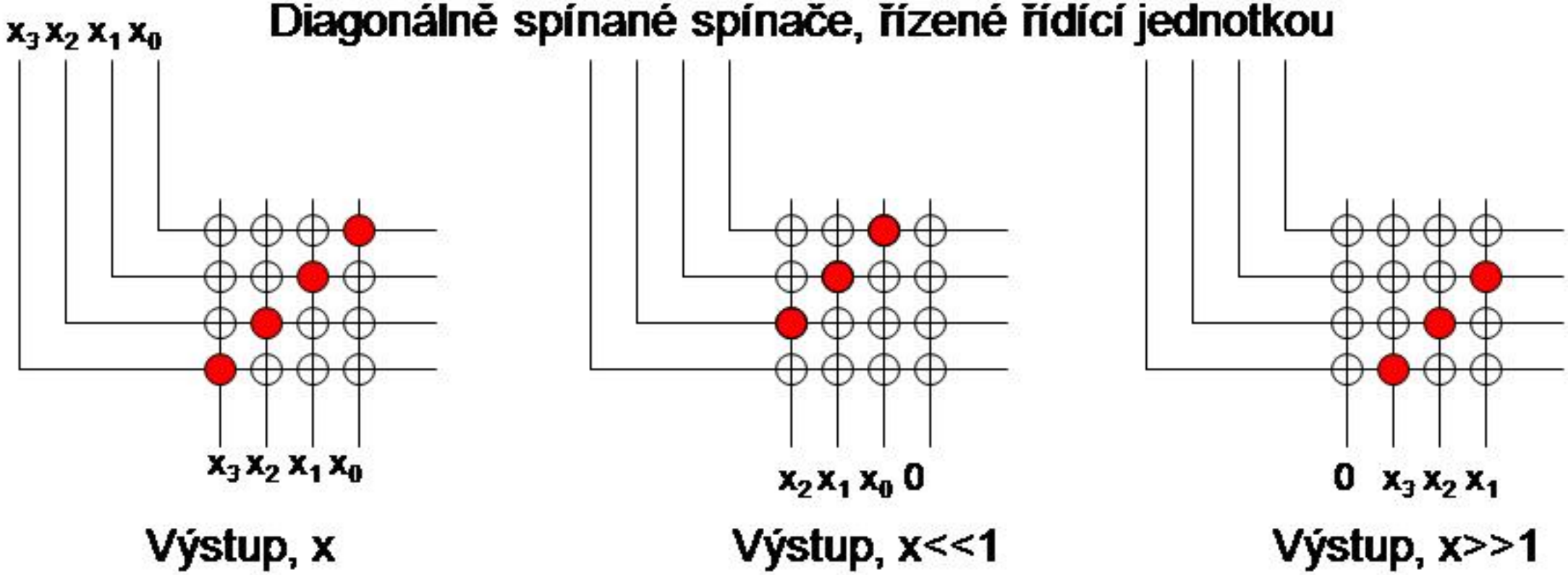
- Řídící linky:

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

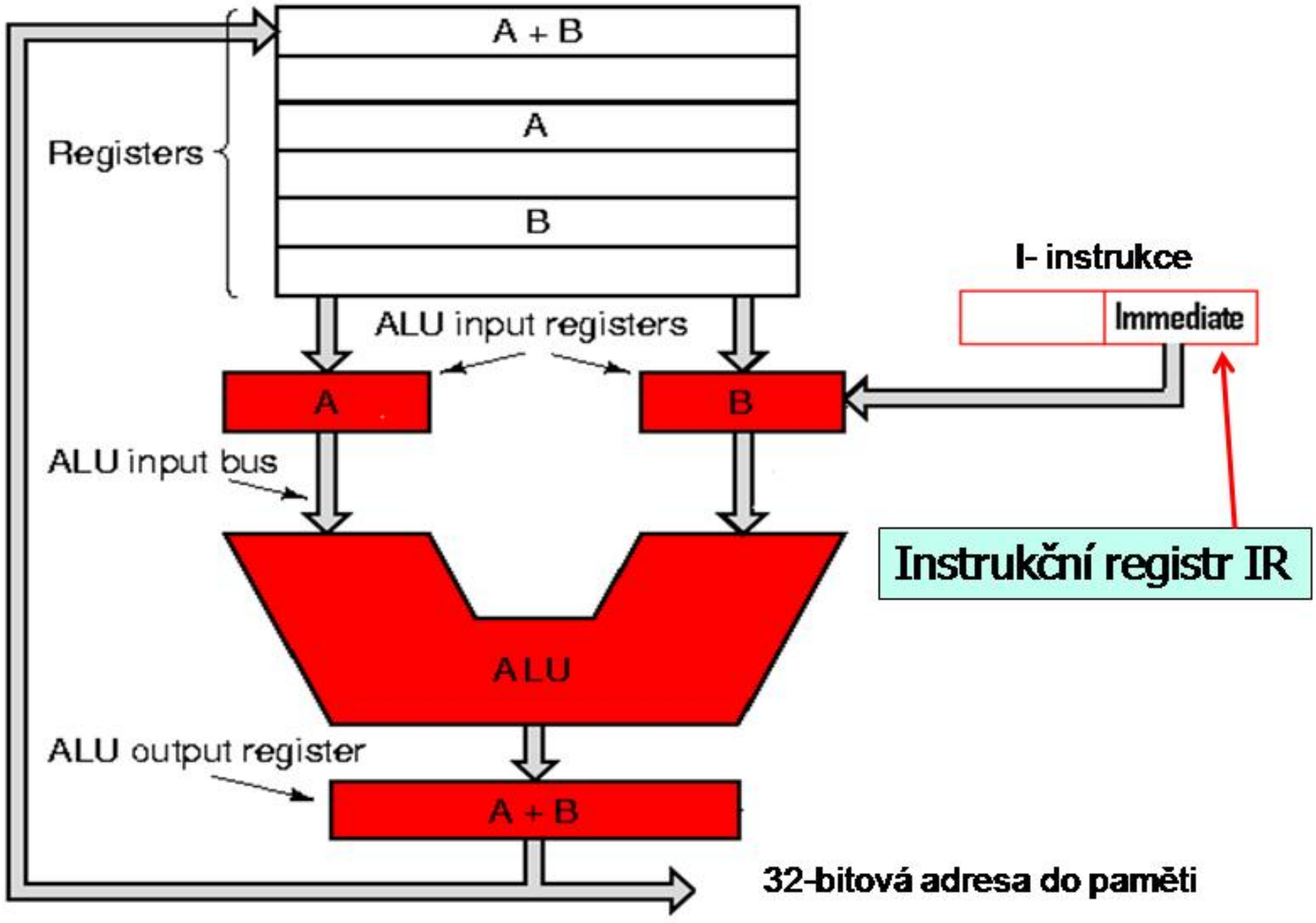


Instrukce posuvu

- 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“

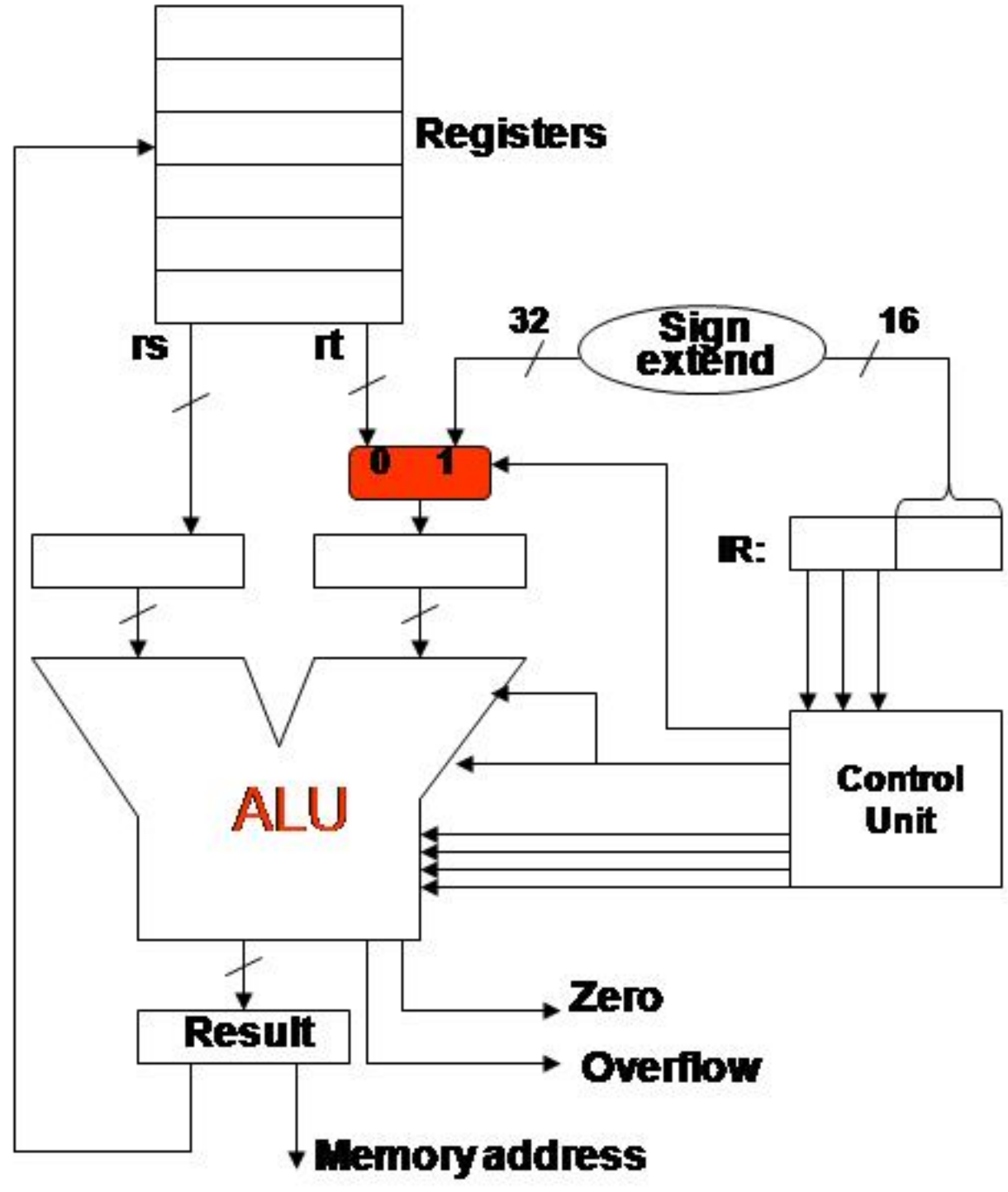


Přehled

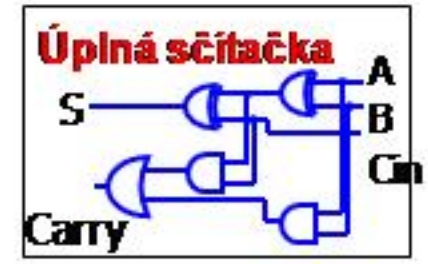

A + B
A
B
Registers
ALU input registers
A
B
I- instrukce
Immediate
ALU input bus
Instrukční registr IR
ALU
ALU output register
A + B
32-bitová adresa do paměti

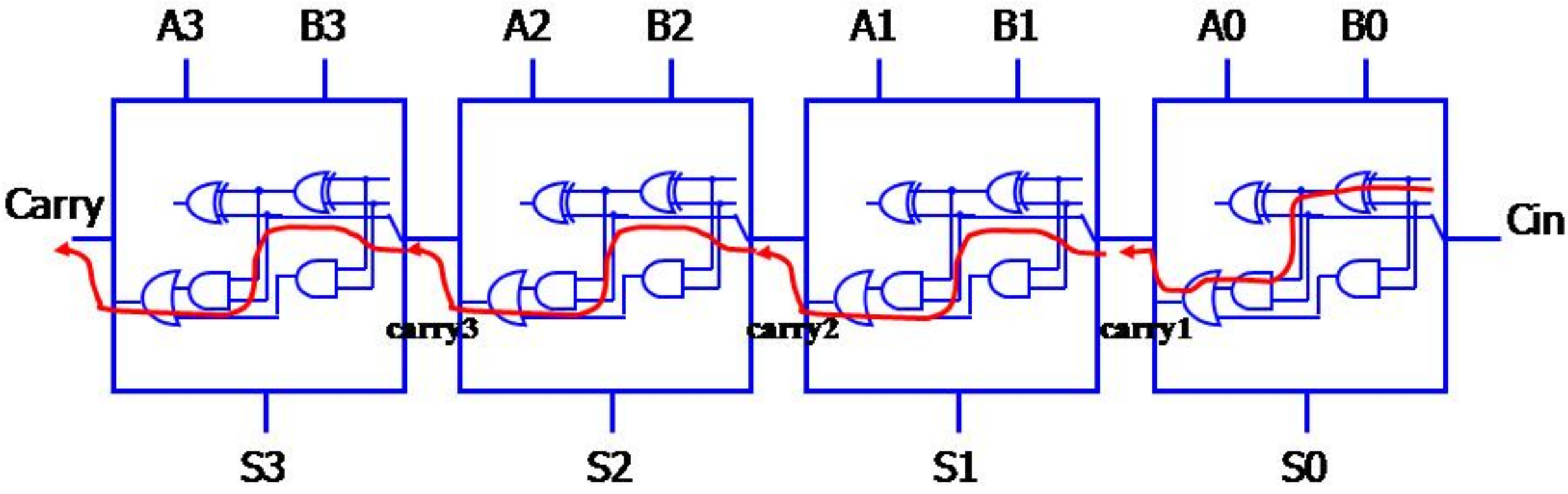
Instrukce s přímým operandem

- 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



4-bitová „Ripple Carry“ sčítačka





Kritická cesta = D_{XOR}+4*(D_{AND}+D_{OR}) pro 4-bitovou ripple carry sčítačku (9 úrovní hradel)

Pro N-bitovou ripple carry sčítačku:
Zpoždění kritické cesty ~ 2(N-1)+3 = (2N+1) * zpoždění hradla

Výkonnost ALU

- Je 32-bitová ALU stejně rychlá jako 1-bitová ALU?
  - Šíření signálu při vyhodnocování přenosů?
- Hardware provádí vyhodnocení paralelně (???)
- Rychlost vs. cena
  - Méně sekvenčních hradel vs. celkový počet hradel
- Dva extrémy jak provést součet
  - „Ripple carry“ a součet součinů
- Jak se zbavit problému s přenosem
  - Dvě úrovně logiky

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$
$$c_2 = b_1c_1 + a_1c_1 + a_1b_1 \qquad c_2 =$$
$$c_3 = b_2c_2 + a_2c_2 + a_2b_2 \qquad c_3 =$$
$$c_4 = b_3c_3 + a_3c_3 + a_3b_3 \qquad c_4 =$$

Sčítačka Carry Lookahead

$$C_{i+1} = g_i + p_i C_i$$

$$C_{i+1} = A_i B_i + C_i (A_i + B_i)$$

$g_i = A_i B_i$ (generace)

$p_i = A_i + B_i$ (propuštění)

$$C_1 = g_0 + p_0 C_0$$

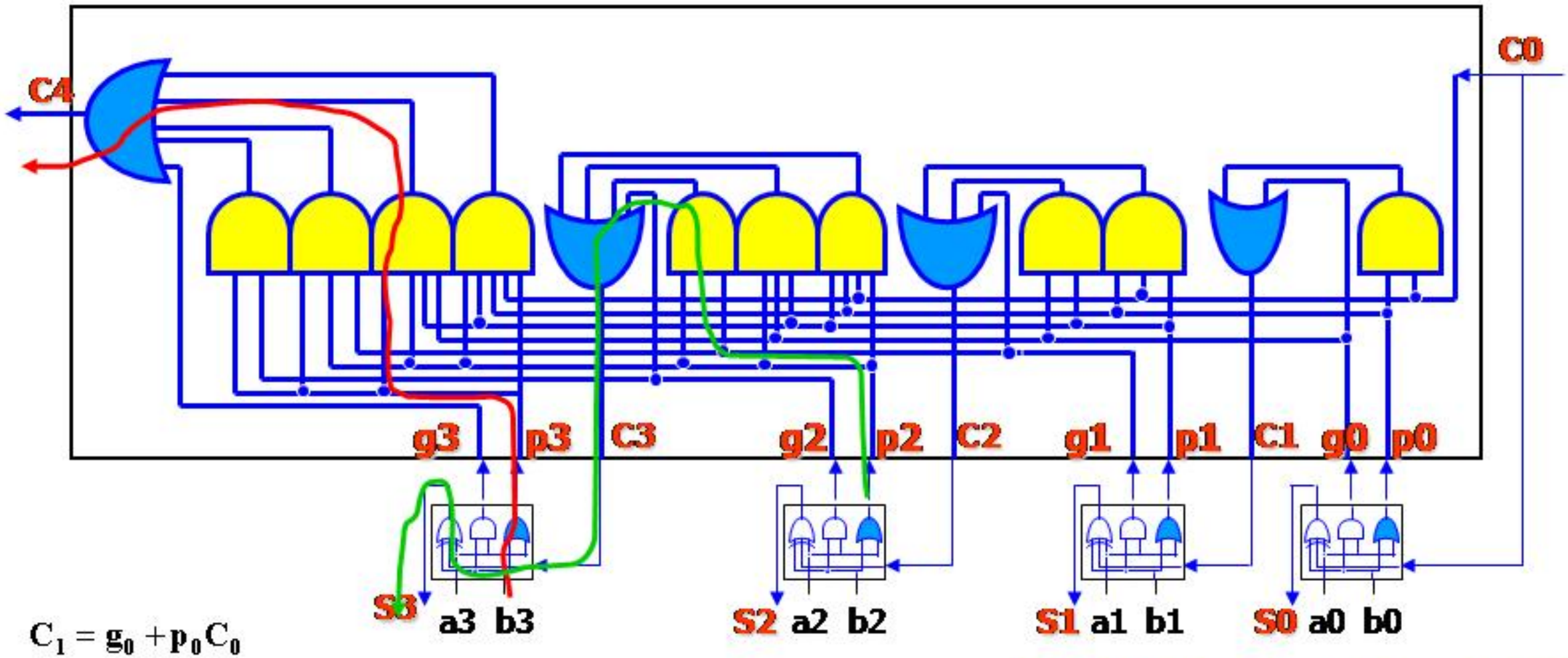
$$C_2 = g_1 + p_1 C_1 = g_1 + p_1 g_0 + p_1 p_0 C_0$$

$$C_3 = g_2 + p_2 C_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0$$

$$C_4 = g_3 + p_3 C_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0$$

Poznámka: Přenosy závisejí pouze na vstupech A, B a C !!!

4-bitová sčítačka Carry Lookahead



$$C_1 = g_0 + p_0 C_0$$

$$C_2 = g_1 + p_1 C_1 = g_1 + p_1 g_0 + p_1 p_0 C_0$$

$$C_3 = g_2 + p_2 C_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0$$

$$C_4 = g_3 + p_3 C_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0$$

Zpoždění jen 3 hradel pro každý C_i
$= D_{AND} + 2*D_{OR}$

4 zpoždění pro každý S_i
$= D_{AND} + 2*D_{OR} + D_{XOR}$

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?

$$c_1 = g_0 + p_0 c_0$$

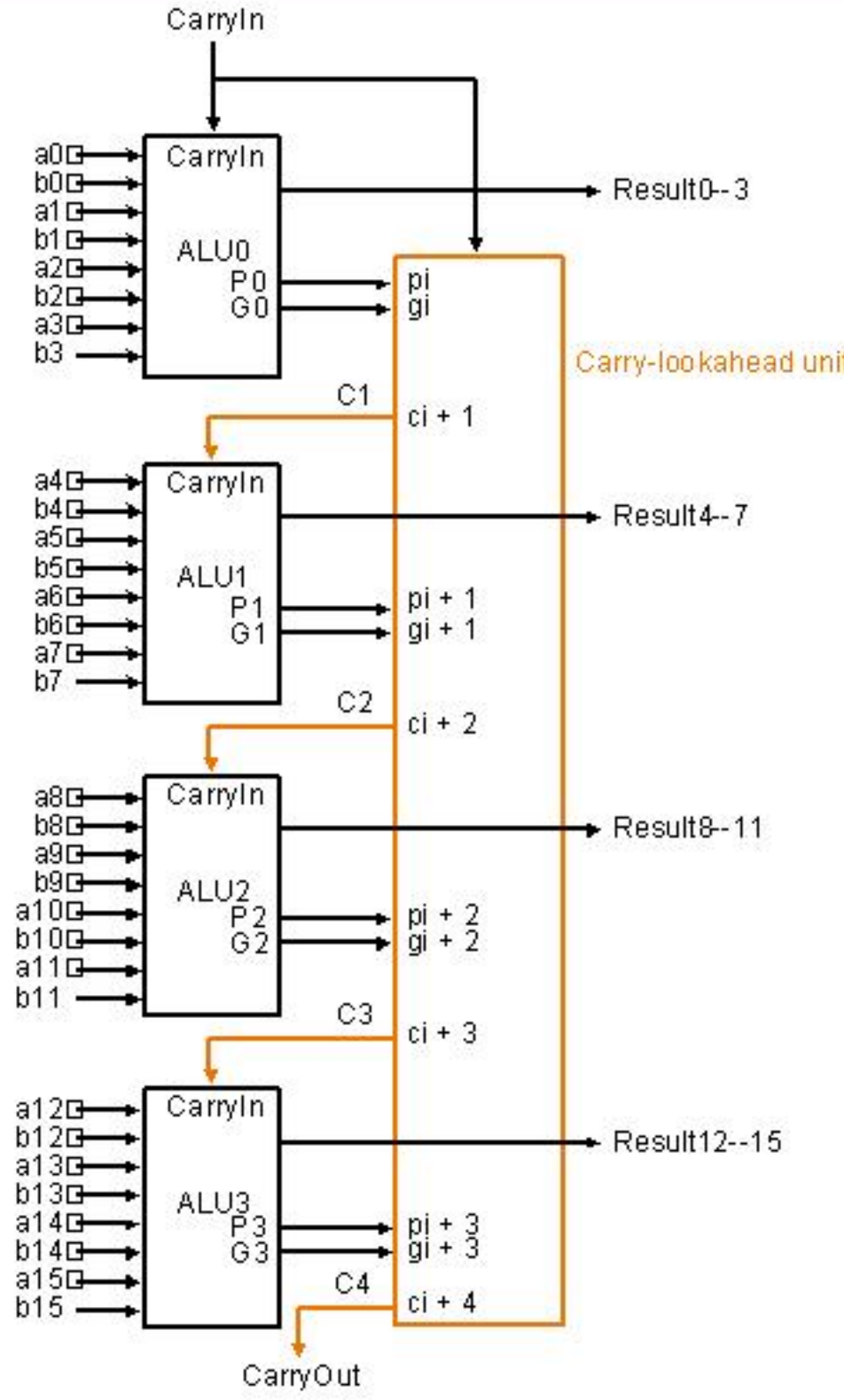
$$c_2 = g_1 + p_1 c_1 \qquad c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

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

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

Hierarchická stavba sčítaček typu Carry-Lookahead

- **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!**



Funkce G a P druhé úrovně

$$P_0 = p_3 + p_2 + p_1 + p_0$$
$$P_1 = p_7 + p_6 + p_5 + p_4$$
$$P_2 = p_{11} + p_{10} + p_9 + p_8$$
$$P_3 = p_{15} + p_{14} + p_{13} + p_{12}$$

$$G_0 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$
$$G_1 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4$$
$$G_2 = g_{11} + p_{11} g_{10} + p_{11} p_{10} g_9 + p_{11} p_{10} p_9 g_8$$
$$G_3 = g_{15} + p_{15} g_{14} + p_{15} p_{14} g_{13} + p_{15} p_{14} p_{13} g_{12}$$

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

- 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

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

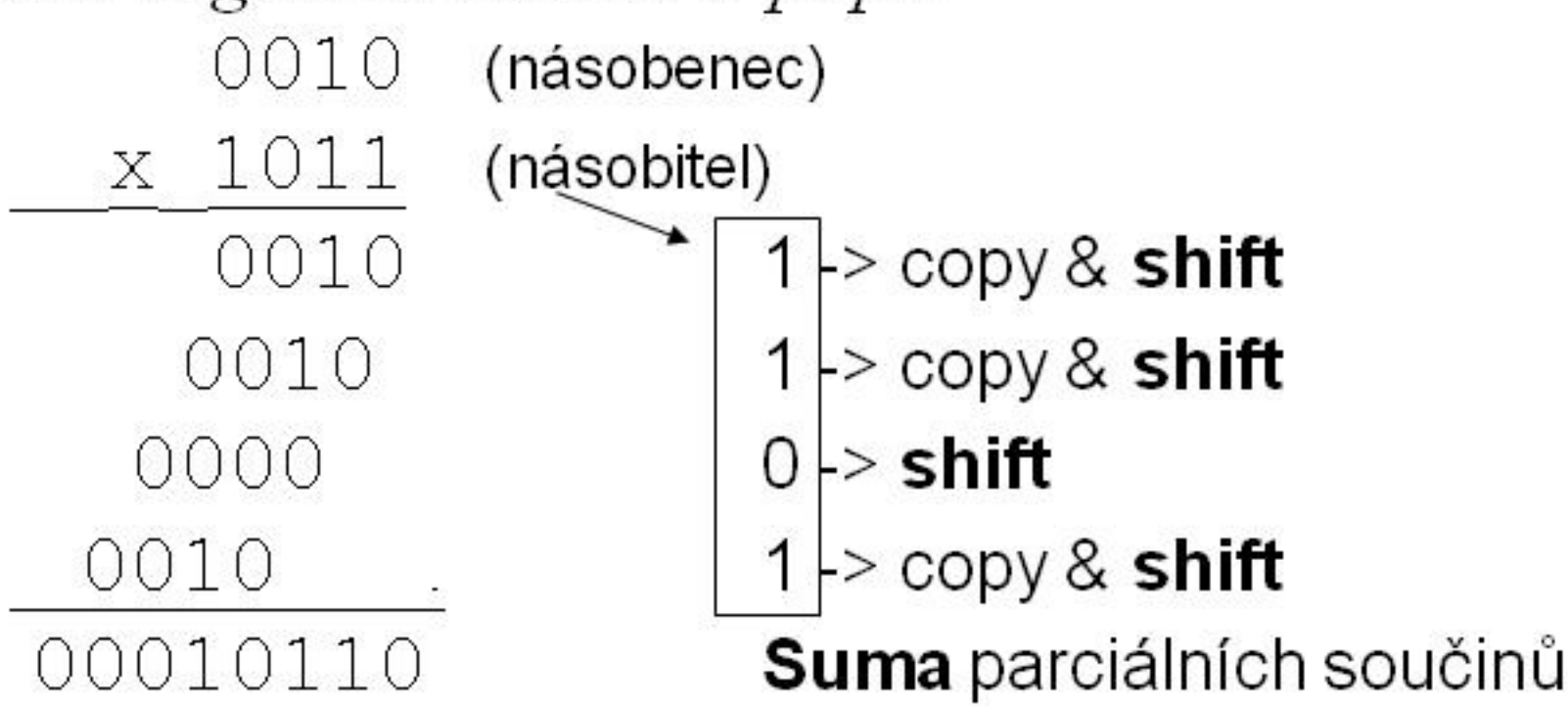
- Můžeme postavit ALU tak, aby vyhověla MIPS ISA
  - **Klíčová myšlenka**: Použít multiplexer pro výběr výstupu ALU
  - Pro odčítání se používá sčítání dvojkového doplňku
  - Pro sestavení 32-bitové ALU opakovaně použít 1-bitovou ALU
- Důležité poznámky k hardware
  - Všechna *hradla* v ALU *pracují 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)

Úvod do organizace počítače

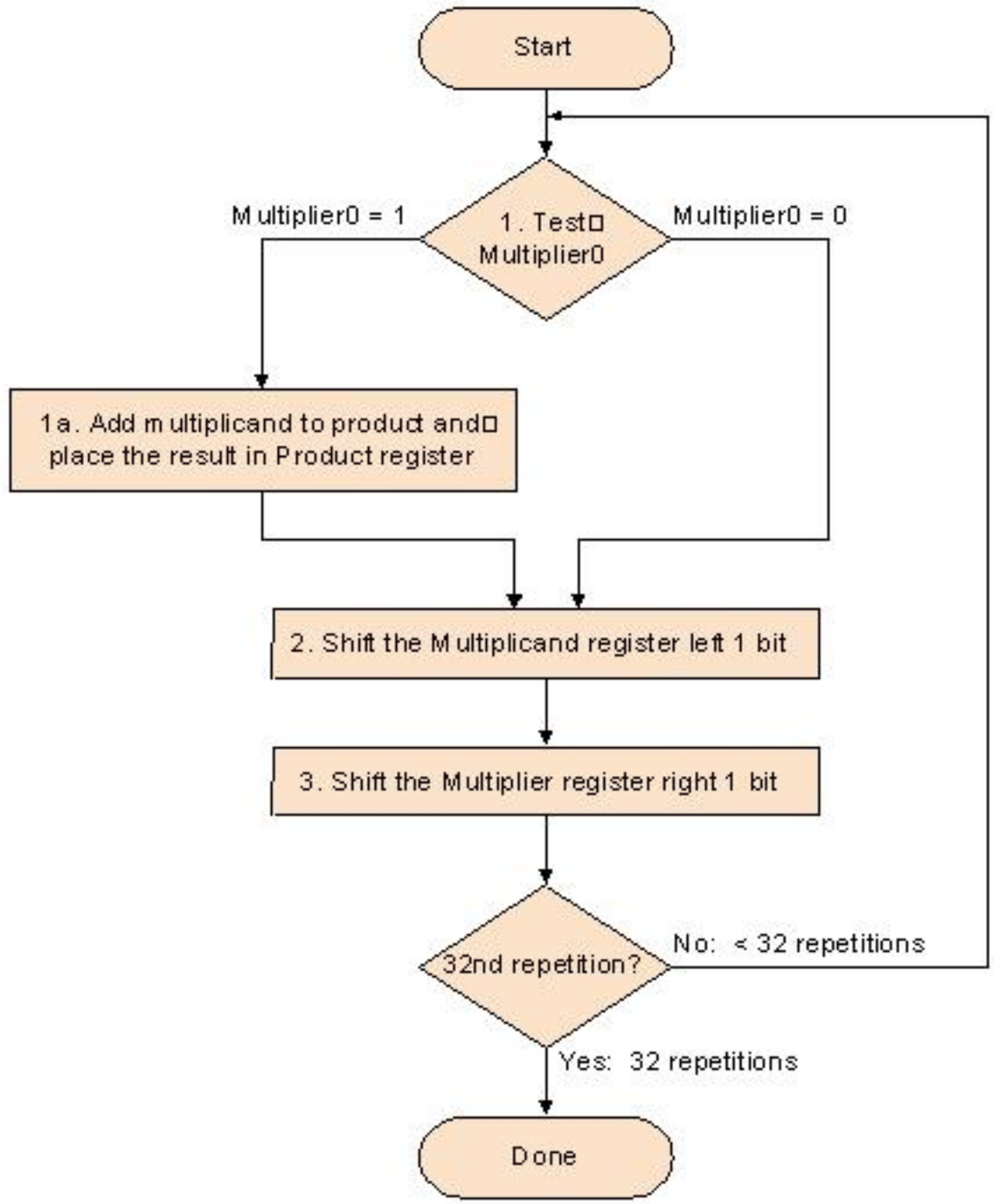
Operace násobení a dělení

Násobení

- Algoritmus je podobný algoritmu ze základní školy
- Složitější operace než sčítání
  - Realizace pomocí posuvů a sčítání
  - Vyžaduje více času a zabere větší plochu čipu
- 3 verze *algoritmu tužka-a-papír*

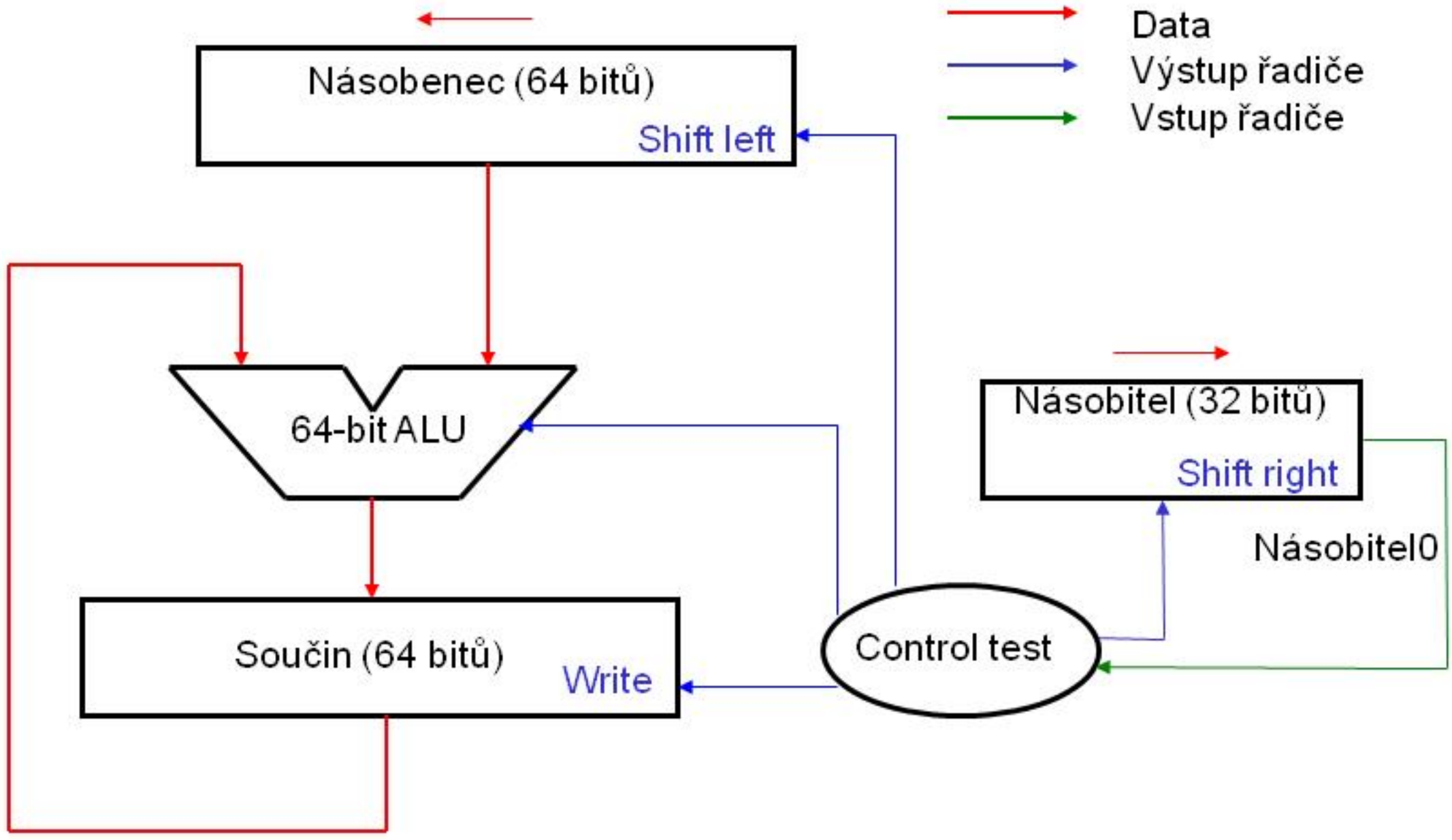


První verze (V.1)

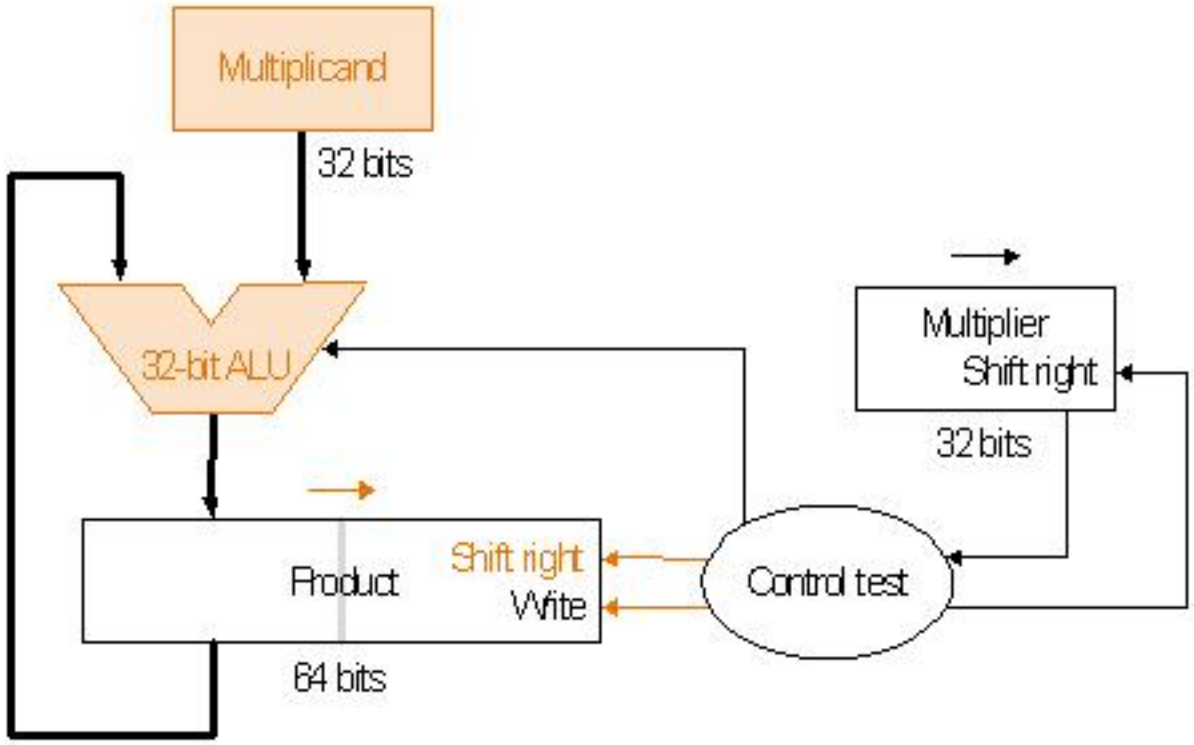


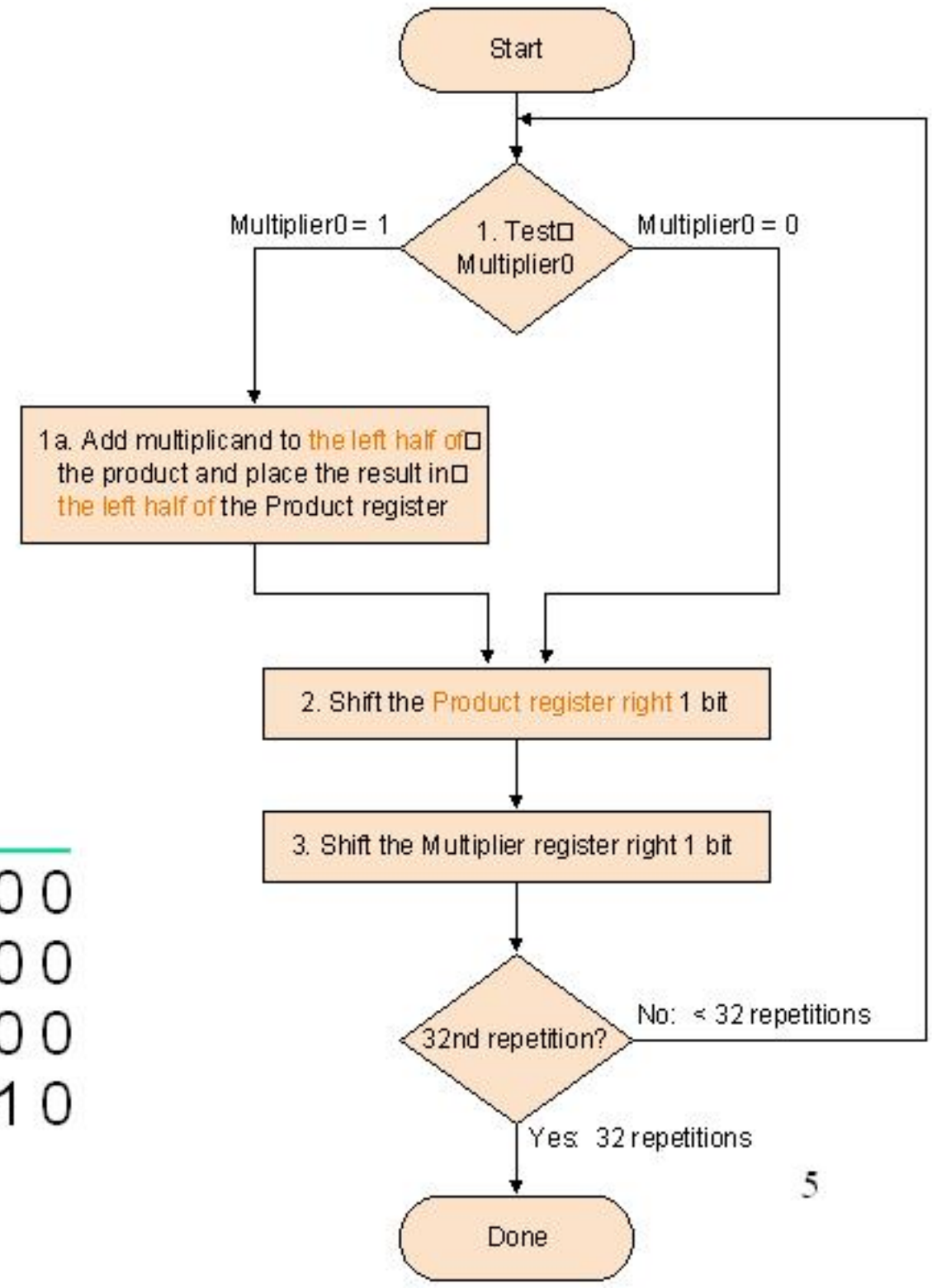

```
← 0 0 1 0
→ 1 0 1 1
-------------
  0 0 0 0 0 0 1 0
  0 0 0 0 0 1 0 0
  0 0 0 0 1 0 0 0
  0 0 0 1 0 0 0 0
-------------
  0 0 0 1 0 1 1 0
```

Hardware (V.1)


Data
Výstup řadiče
Vstup řadiče
Násobenec (64 bitů)
Shift left
64-bit ALU
Součin (64 bitů)
Write
Násobitel (32 bitů)
Shift right
Násobitel0
Control test

Druhá verze (V.2)

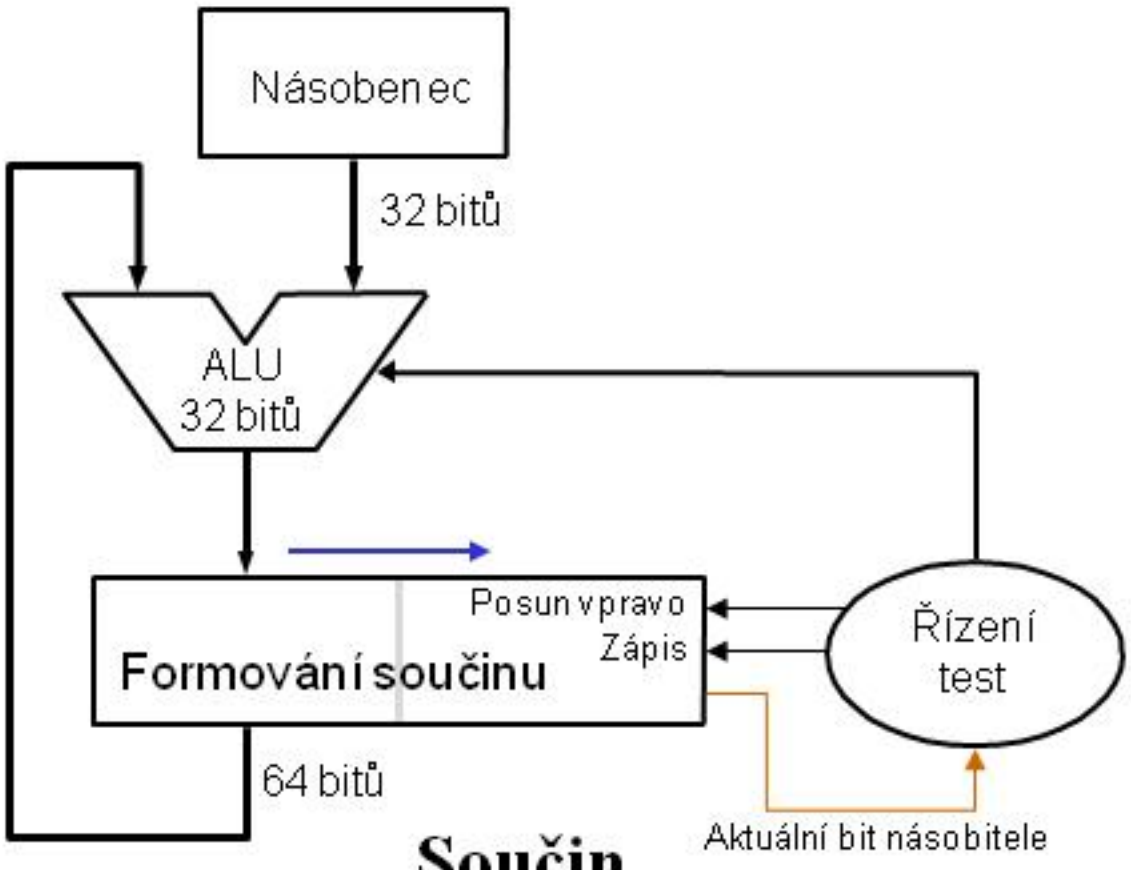




Součin

Násobitel0	0 0 0 0 0 0 0 0
1	0 0 1 0 0 0 0 0 → 0 0 0 1 0 0 0 0
1	0 0 1 1 0 0 0 0 → 0 0 0 1 1 0 0 0
0	0 0 0 1 1 0 0 0 → 0 0 0 0 1 1 0 0
1	0 0 1 0 1 1 0 0 → 0 0 0 1 0 1 1 0

Konečná verze (V.3)



Součin

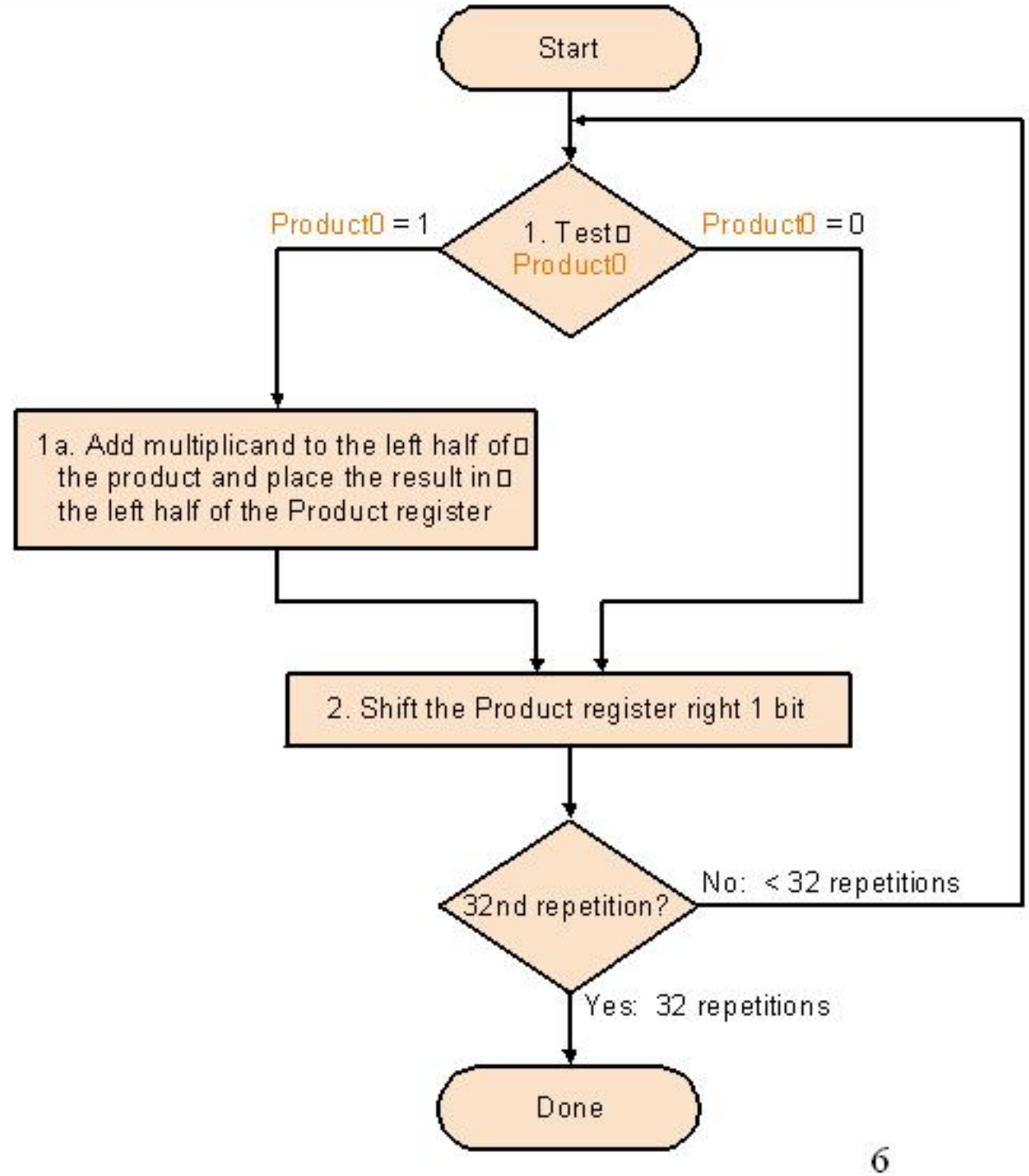
0 0 0 0 1 0 1 1

0 0 1 0 1 0 1 **1** → 0 0 0 1 0 1 0 1

0 0 1 1 0 1 0 **1** → 0 0 0 1 1 0 1 0

0 0 0 1 1 0 1 **0** → 0 0 0 0 1 1 0 1

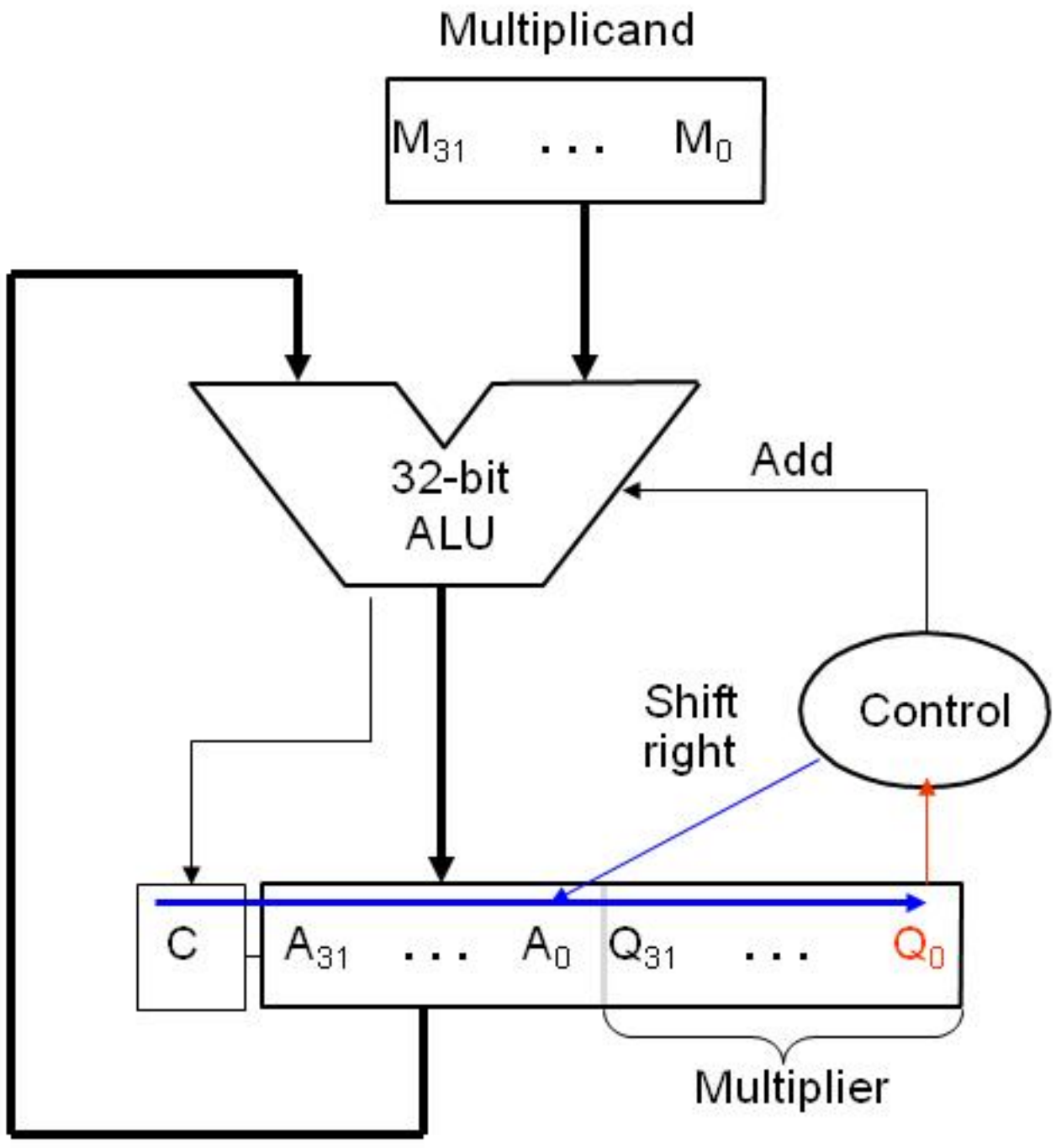
0 0 1 0 1 1 0 **1** → 0 0 0 1 0 1 1 0



Souhrn

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

Celkový pohled



1011 Multiplicand (11)

x 1101 Multiplier (13)

Product (143)

		C	A	Q	M
Initial values		0	0000	1101	1011
1	Add	0	1011	1101	1011
	Shift	0	0101	1110	1011
2	Shift	0	0010	1111	1011
3	Add	0	1101	1111	1011
	Shift	0	0110	1111	1011
4	Add	1	0001	1111	1011
	Shift	0	1000	1111	1011

Aritmetika se znaménkem

- Sčítání a odčítání se znaménkem
  - S operandy se zachází jako s čísly bez znaménka
  - Používá stejné algoritmy i hardware upoužívané pro odpovídající operace bez znaménka

	Unsigned	Signed
1001	9	-7
+ 0011	3	3
1100	12	-4

- Pro násobení nelze použít!

Příklad

	Unsigned	Signed
1011	11	-5
x 1101	13	-3
10001111	143	~~-113~~

- Částečné řešení, je-li násobenec záporný

1001	(9)		1001	(-7)
x 0011	(3)		x 0011	(3)
00001001	1001×2^0		11111001	$(-7) \times 2^0 = (-7)$
00010010	1001×2^1		11110010	$(-7) \times 2^1 = (-14)$
00011011	(27)		11101011	(-21)

- Nelze použít, je-li násobitel záporný

Záporný násobitel

- Bity násobitele nekorespondují s parciálními součiny
- Příklad: (-3) = 1101
  - Parciální součiny by se generovaly podle stavu bitů násobitele:

1: -1×2^0

0: -0×2^1

1: -1×2^2

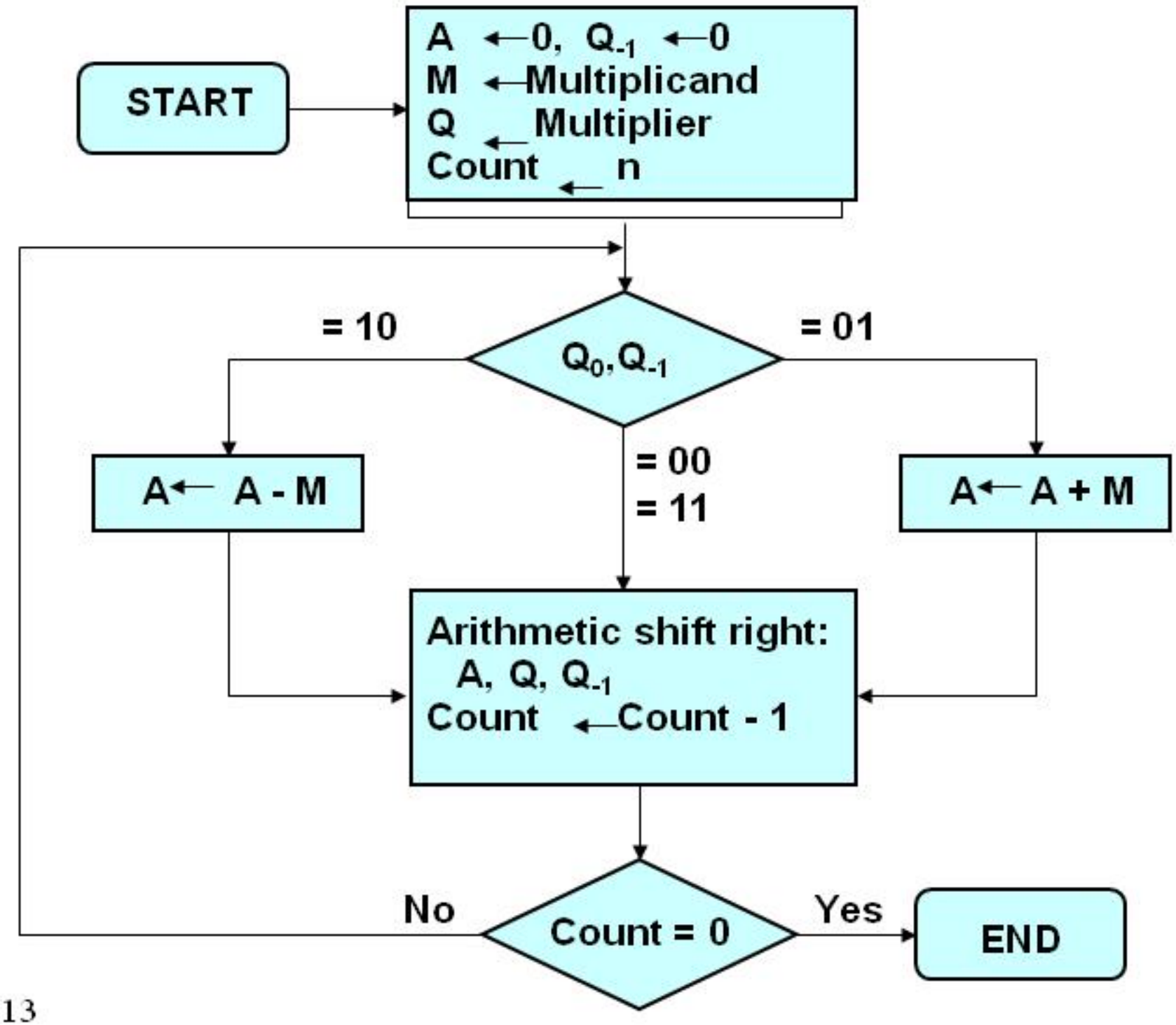
1: -1×2^3

  - Měly by však být generovány s použitím následujících mocnin 2:

-1×2^0

-1×2^1

Boothův algoritmus



Boothův algoritmus

Boothův algoritmus pro dvojkový komplement

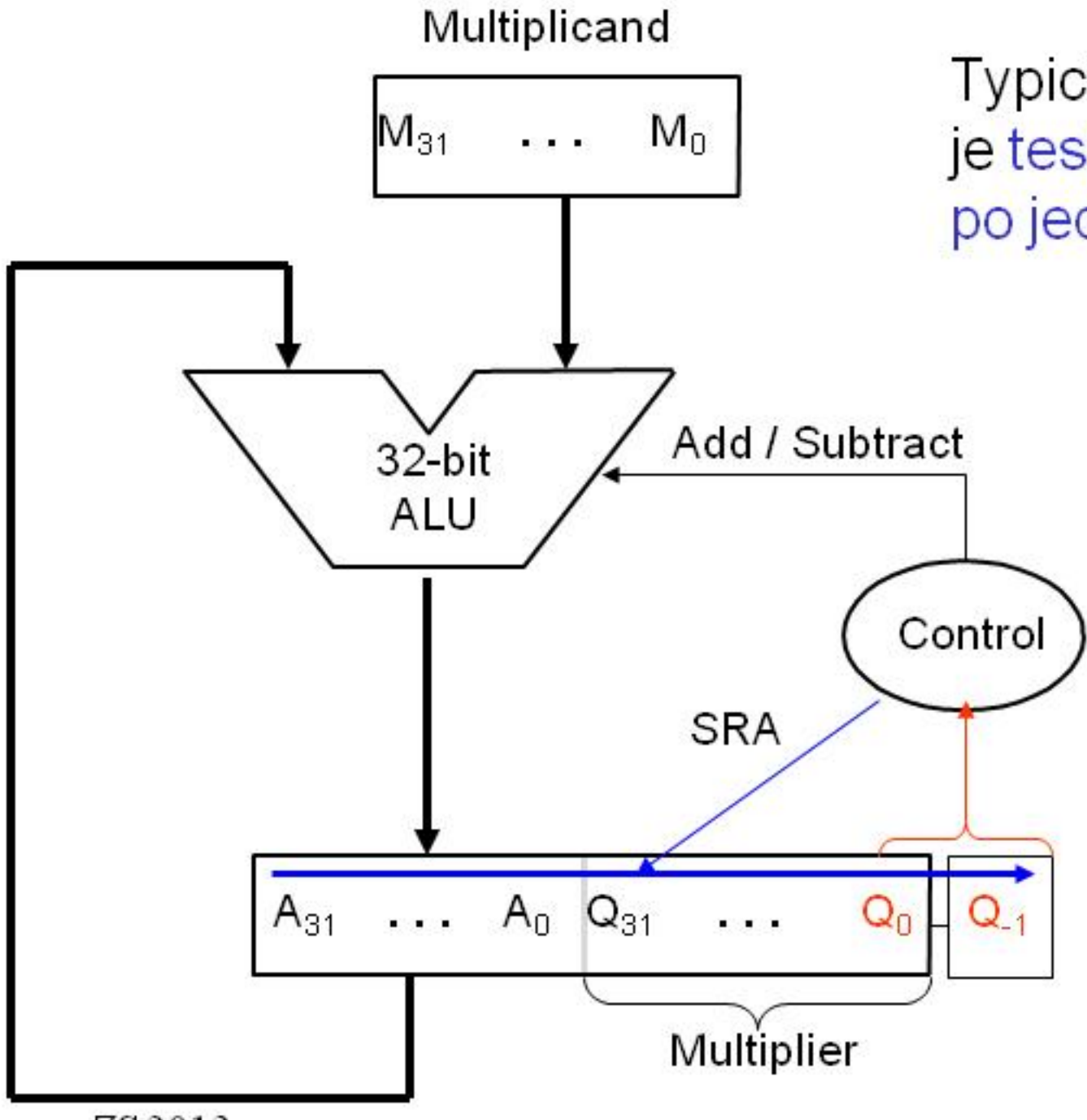
- Zpracovává kladné i záporné operandy
- Mnohem efektivnější než konvenční algoritmus

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

- 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



Typickým znakem Boothova algoritmu je test dvou bitů násobitele a postup po jednom bitu

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

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

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

  $= M \times 30$

  5 4 3 2 1 0

- 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} . . . x_1 x_0 }

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

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

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

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

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

(1-0) tato změna znamená operaci odečtení

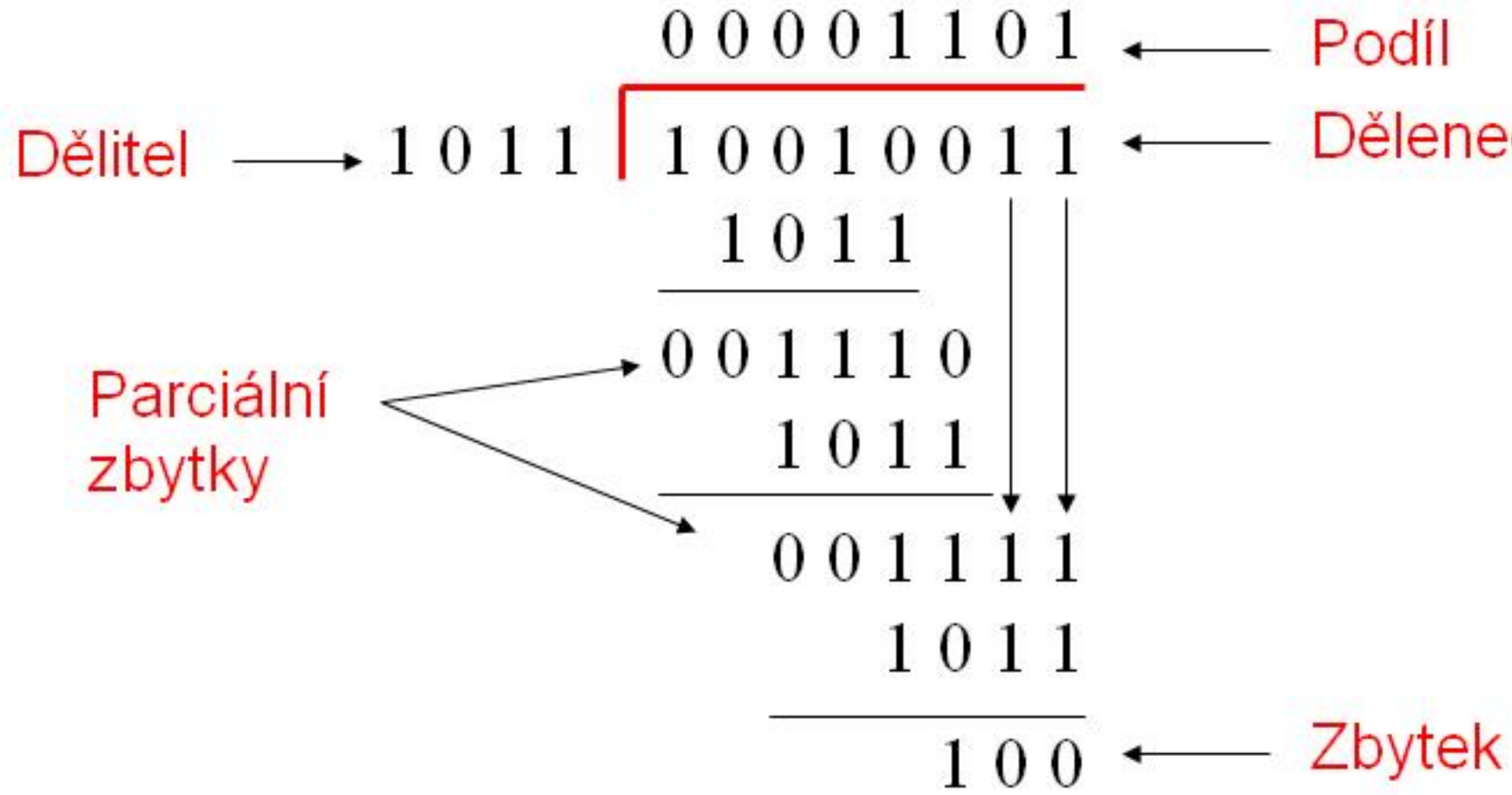
Násobení u MIPS

- 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

- 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

Některé metody dělení

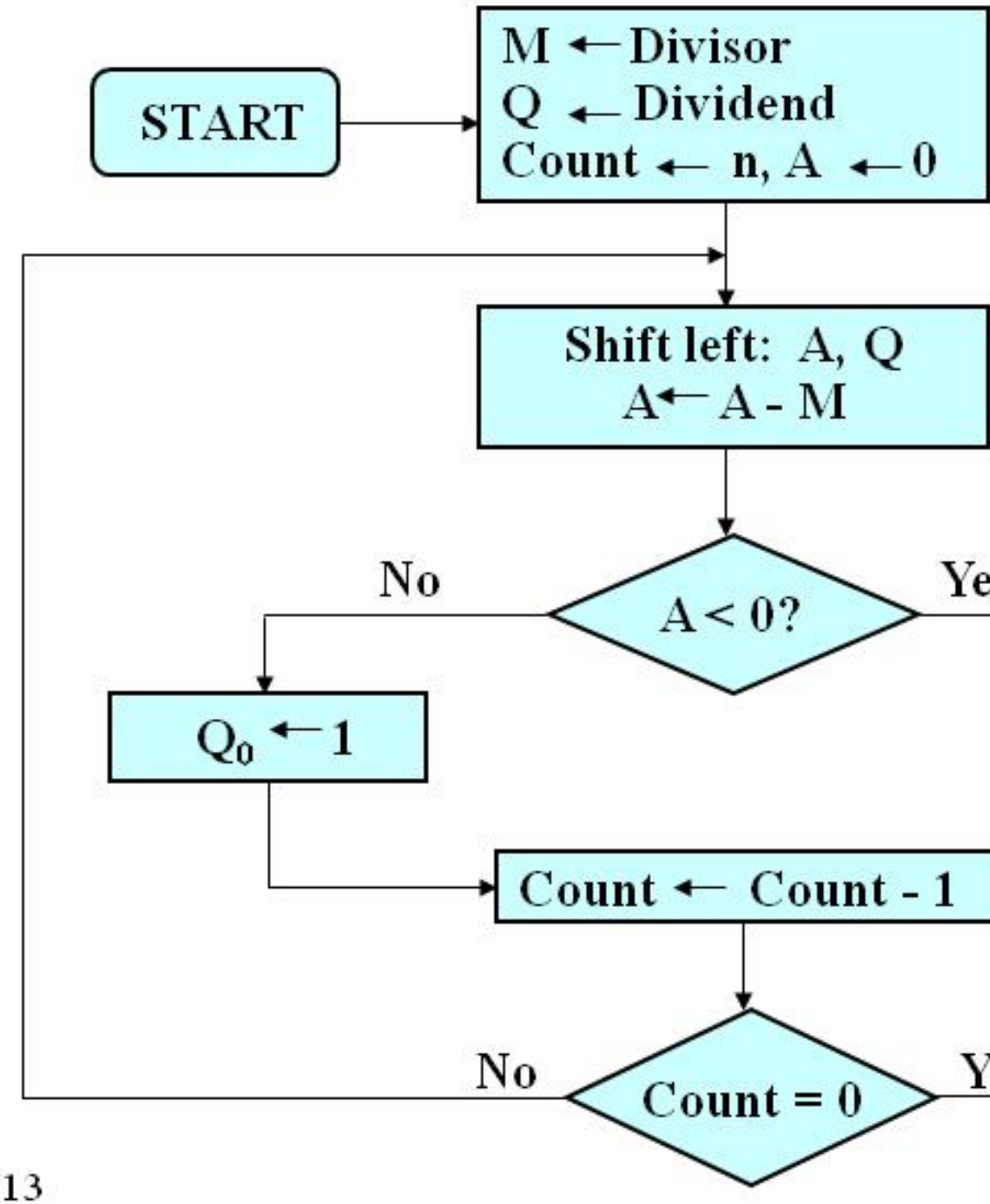
Sekvenční metody binárního dělení se hlavně odlišují množinou generovaných cifer:

$z = 2$

$q_j \in \{0, 1\}$ … 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

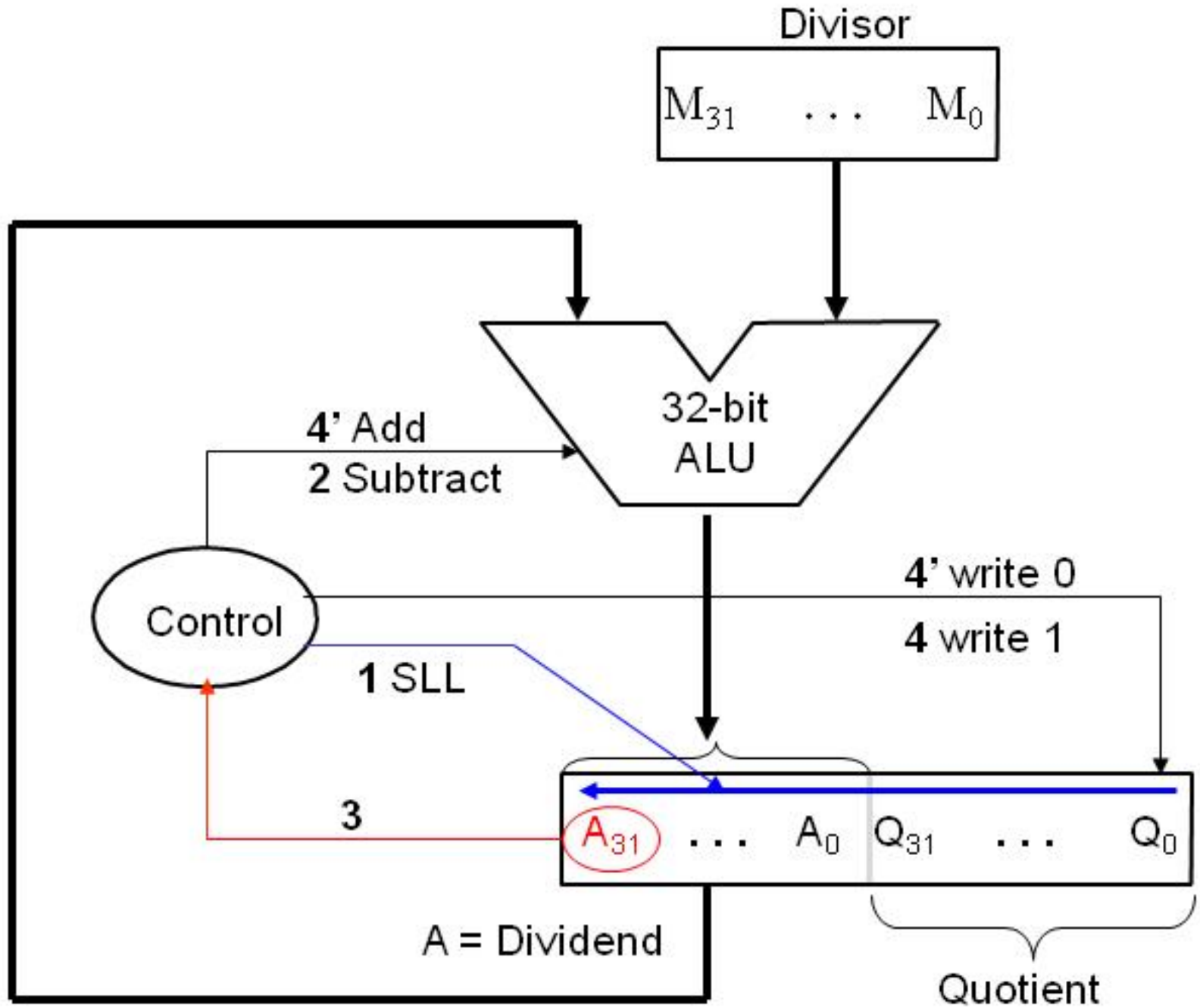


Dělenec přechází do A postupně z Q, který je na počátku volný

Metoda s obnovou zbytku

Quotient in Q
Remainder in A

Hardware pro operaci dělení



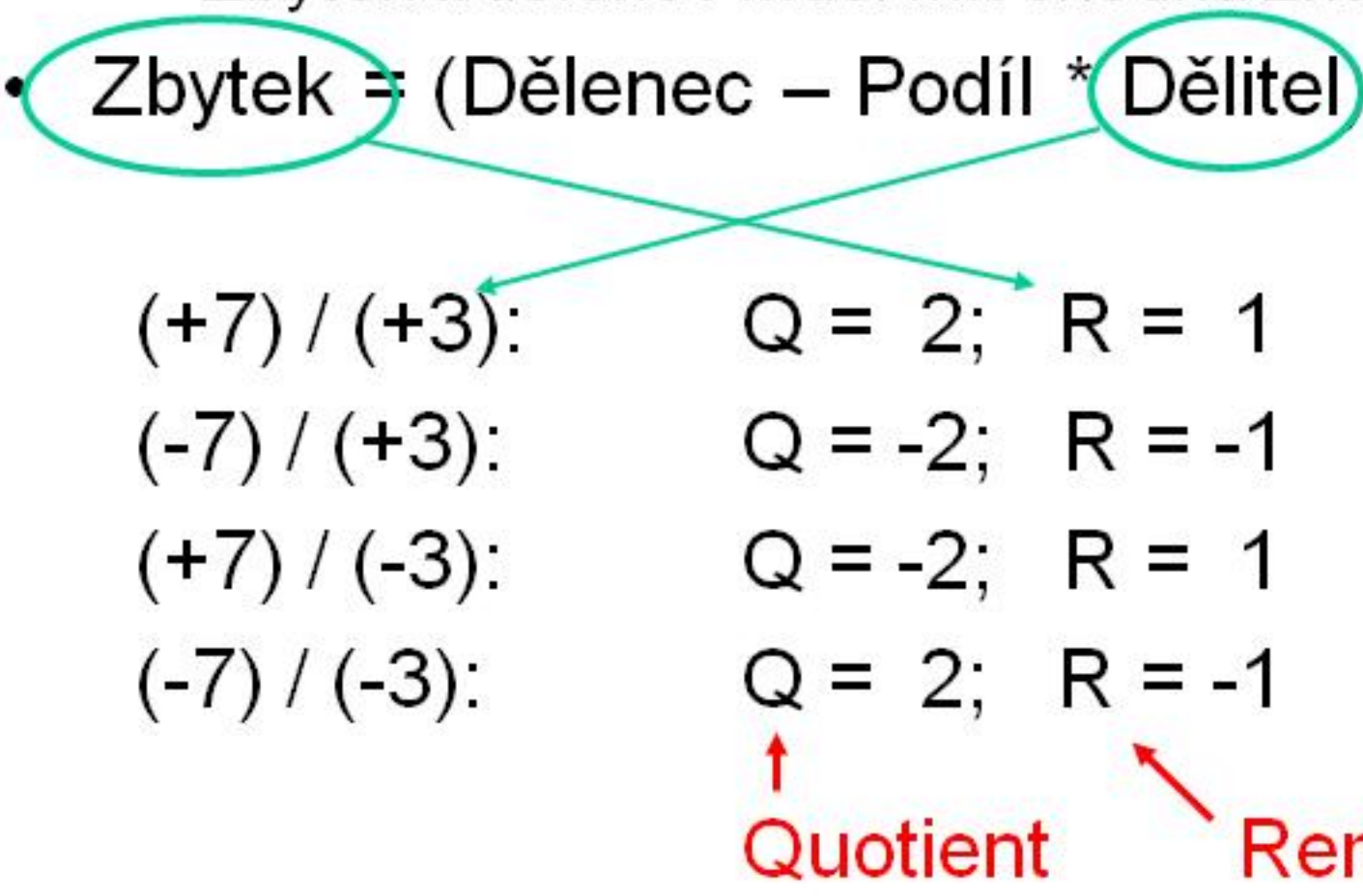
Příklady

7 / 3 :

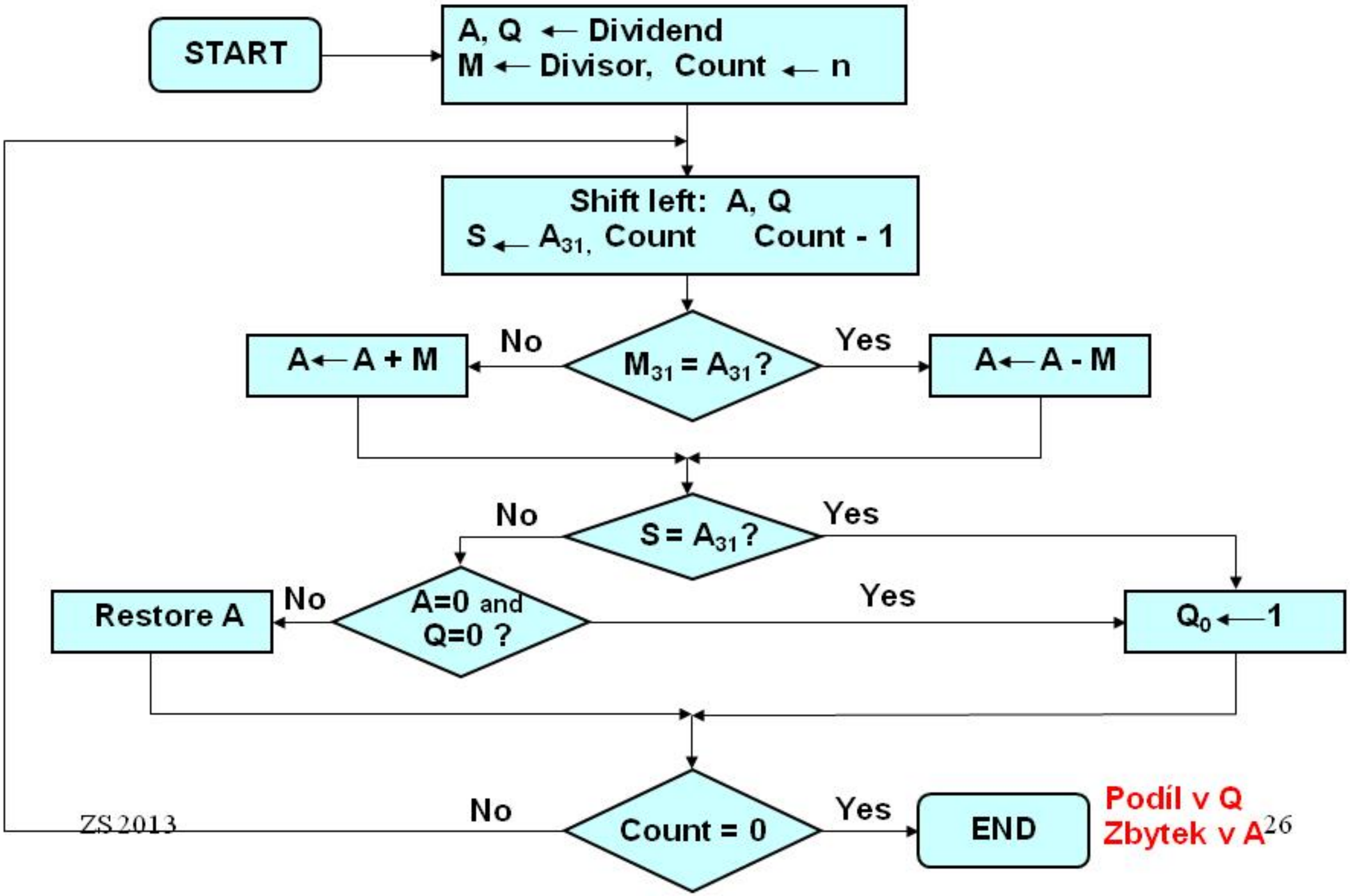
A	Q	M = 0011	
0000	0111	Initial values	
0000	1110	Shift	
1101	1110	A = A - M	1
0000	1110	A = A + M	
0001	1100	Shift	
1110	1100	A = A - M	2
0001	1100	A = A + M	
0011	1000	Shift	
0000	1000	A = A - M	3
0000	1001	$Q_0 = 1$	
0001	0010	Shift	
1110	0010	A = A - M	4
0001	0010	A = A + M	

Operace dělení se znaménkem

- Jednoduché řešení
  - Negovat podíl, jestliže znaménka dělitele a dělence nejsou shodná
  - Zbytek a dělenec musí mít shodná znaménka
- Zbytek = (Dělenec – Podíl * Dělitel



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)

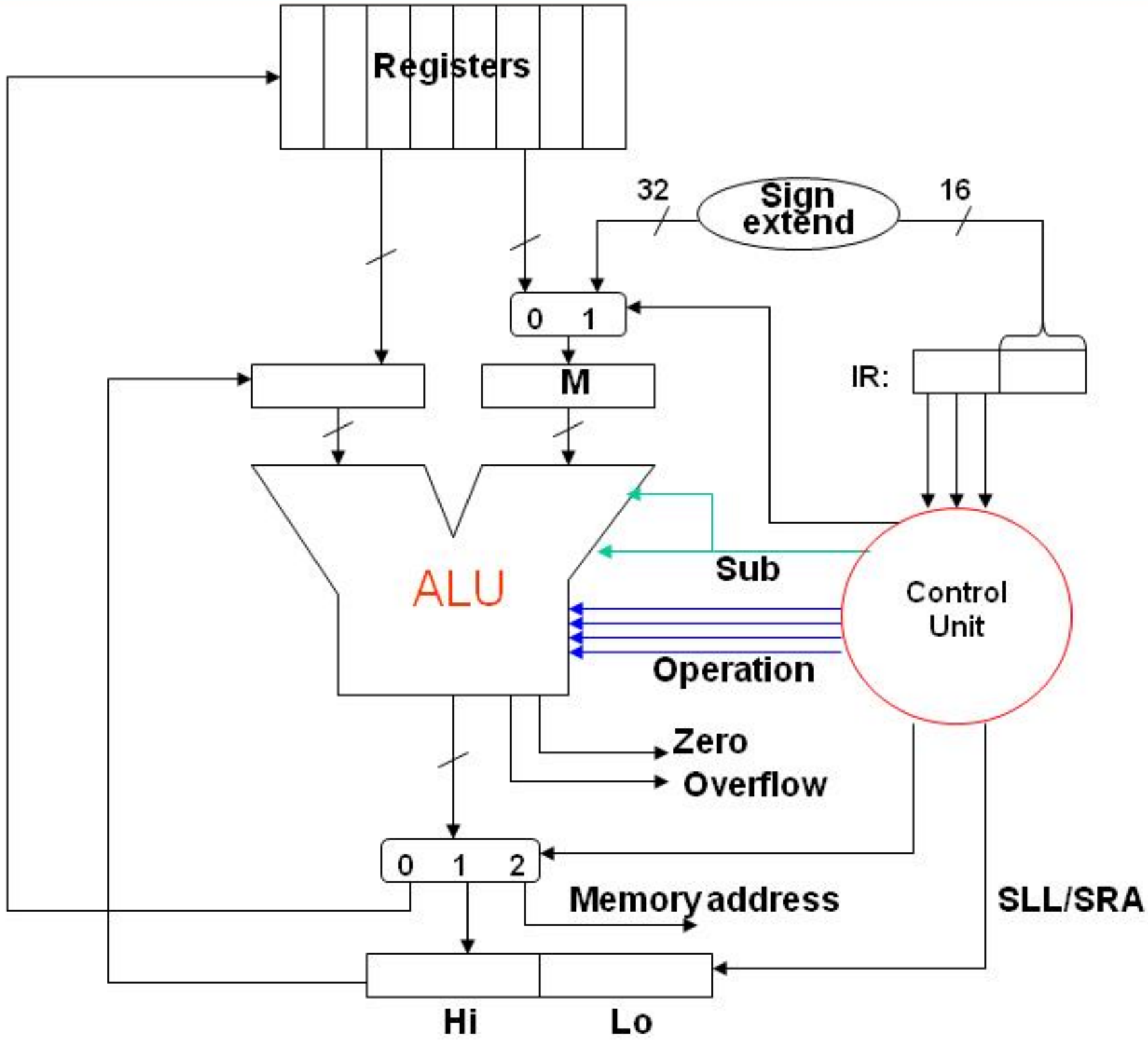
A	Q	**M = 0011**	
1111	1001	Initial values	
1111	0010	Shift	1
0010		Add	
1111	0010	Restore	
1110	0100	Shift	2
0001		Add	
1110	0100	Restore	
1100	1000	Shift	3
1111		Add	
1111	1001	$Q_0 = 1$	
1111	0010	Shift	4
0010		Add	
1111	0010	Restore	

(-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)

Procesor MIPS



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 !

Úvod do organizace počítače

Pohyblivá řádová čárka (Floating Point)

Konvence zápisu, MIPS

Literatura

Normy:

- IEEE 754
- IEEE 854

FP = Floating Point

Knihy:

Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefévre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres:

Handbook of Floating-Point Arithmetic

ISBN 978-0-8176-4704-9 e-ISBN 978-0-8176-4705-6

Přehled

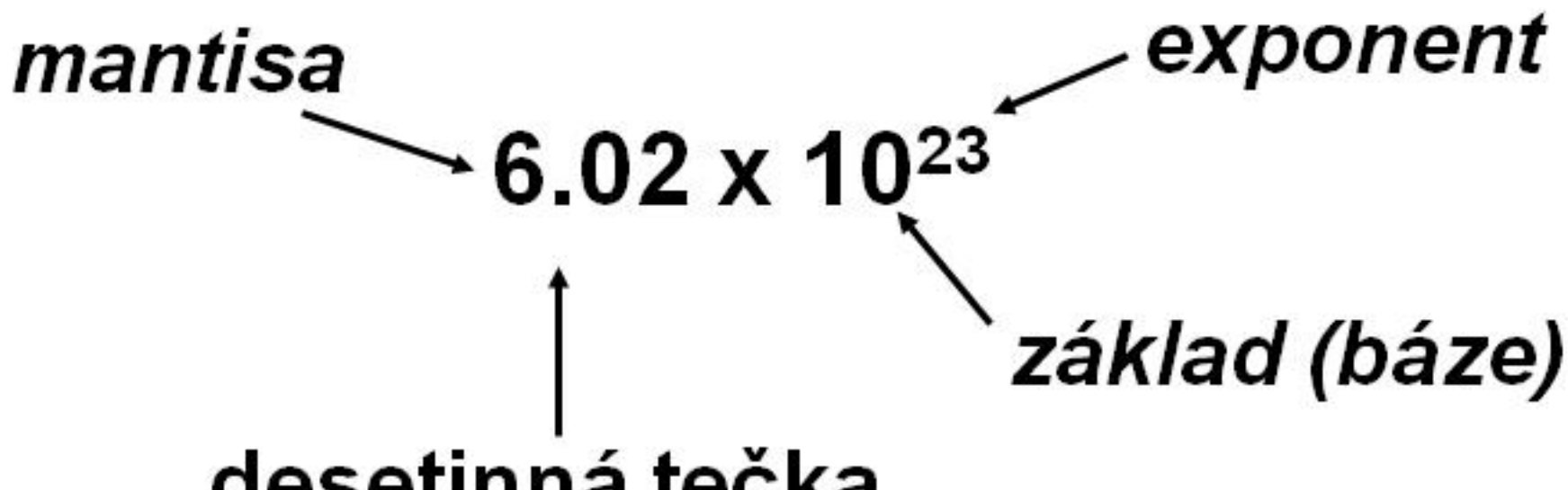
- Čísla v pohyblivé řádové čárce
- Zápis čísel
  - Desítková notace
  - Binární notace
- Standard IEEE 754 FP
- Interní reprezentace FP čísel v počítači
  - Větší rozsah vs. přesnost zobrazení
- Konverze desítkového zápisu na FP
- Typ není asociován s daty
- FP instrukce MIPS, registry

FP = Floating Point

Čísla a počítače

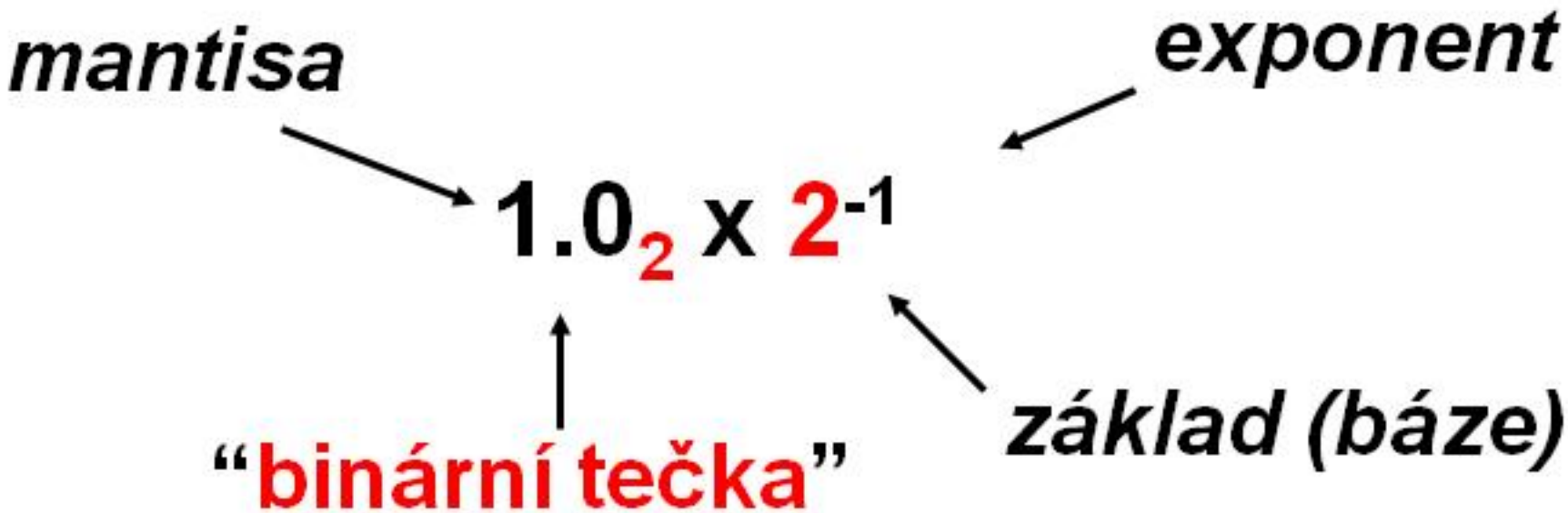
- Co lze zobrazit na n bitech ?
  - Čísla integer bez znaménka: 0 až $2^n - 1$
  - Čísla integer se znaménkem: $-2^{(n-1)}$ až $2^{(n-1)} - 1$
- Ostatní čísla?
  - Velmi velká čísla? (počet částic hmoty)
    $3,155,760,000_{10}$ ($3.15576_{10} \times 10^9$)
  - Velmi malá čísla? (rozměry částic hmoty)
    0.00000001_{10} ($1.0_{10} \times 10^{-8}$)
  - Racionální čísla (popř. čísla s periodou)
    2/3 (0.666666666. . .)
  - Iracionální čísla: $2^{1/2}$ (1.414213562373. . .)
  - Transcendetní čísla: e (2.718...), π (3.141...)

Notace



- Normalizovaný tvar: nemá úvodní nuly (přesně první číslice vlevo od desetinné tečky)
- Alternativní reprezentace 1/1,000,000,000
  - Normalizováno: 1.0×10^{-9}
  - Nenormalizováno: 0.1×10^{-8}, 10.0×10^{-10}

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 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

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

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

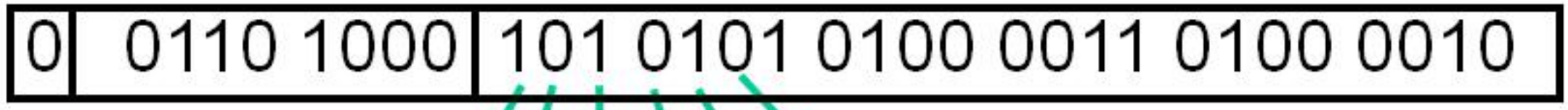
Exponent – kód s posunutou nulou

- 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

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

Převod z binárního do desítkového tvaru FP



- 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)

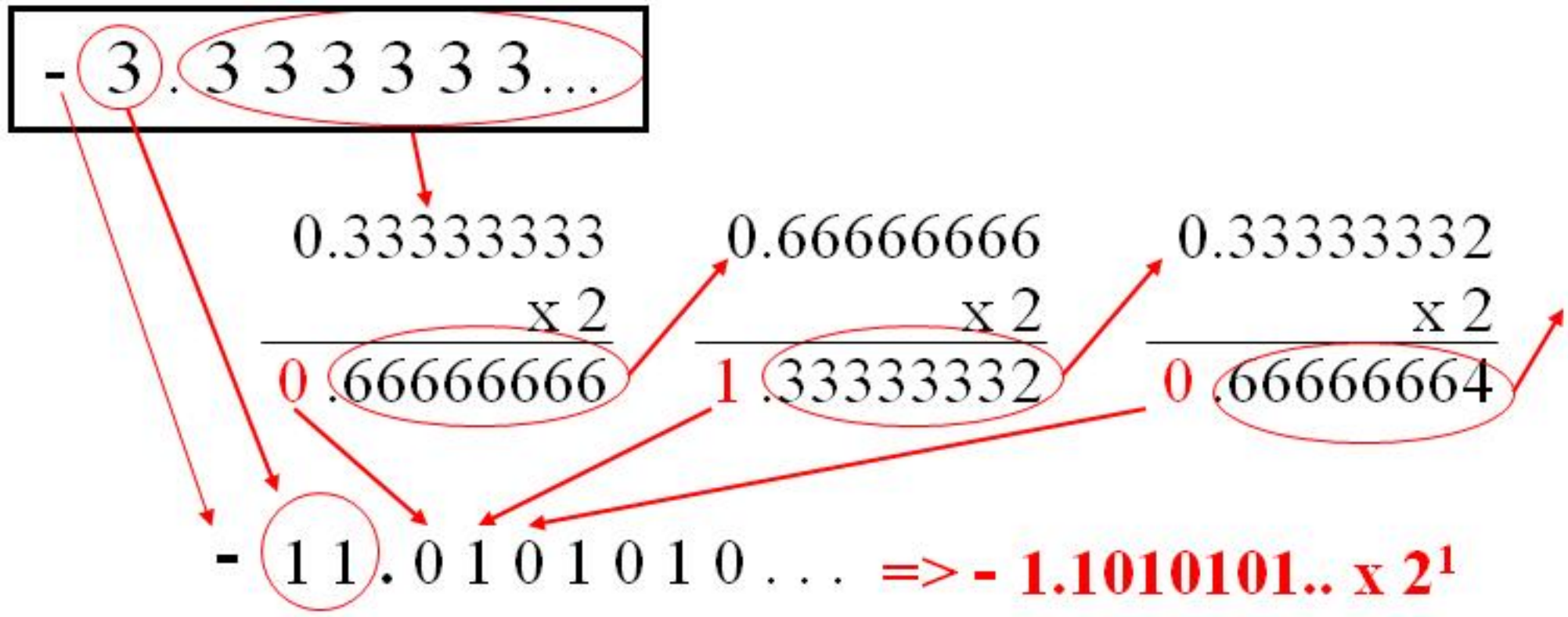
- Jednoduchý případ: Je-li jmenovatel mocninou 2 (2, 4, 8, 16, atd.), je to snadné.
- Binární FP reprezentace čísla -0.75
  - -0.75 = -3/4
  - $-11_2/100_2 = -0.11_2$
  - Normalizováno na $-1.1_2 \times 2^{-1}$
  - $(-1)^S \times (1 + \text{Mantisa}) \times 2^{(\text{Exponent}-127)}$
  - $(-1)^1 \times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

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 oříznete (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

Hlediska návrhu formátu

- Pro uložení FP-čísla musíme uložit následující tři složky informace …
  - Znaménko (sign) kladné/záporné
  - Exponent
  - Mantisa
- Je-li dán pevný počet bitů pro uložení čísla (např. slovo), jak zvolit velikost pole pro mantisu a pro exponent?
  - Zvětšováním mantisy roste přesnost zobrazení.
  - Zvětšováním exponentu narůstá rozsah zobrazovaných čísel.
  - **Jde o kompromis** (ostatně jako u mnoha dalších podobných rozhodnutí).

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.

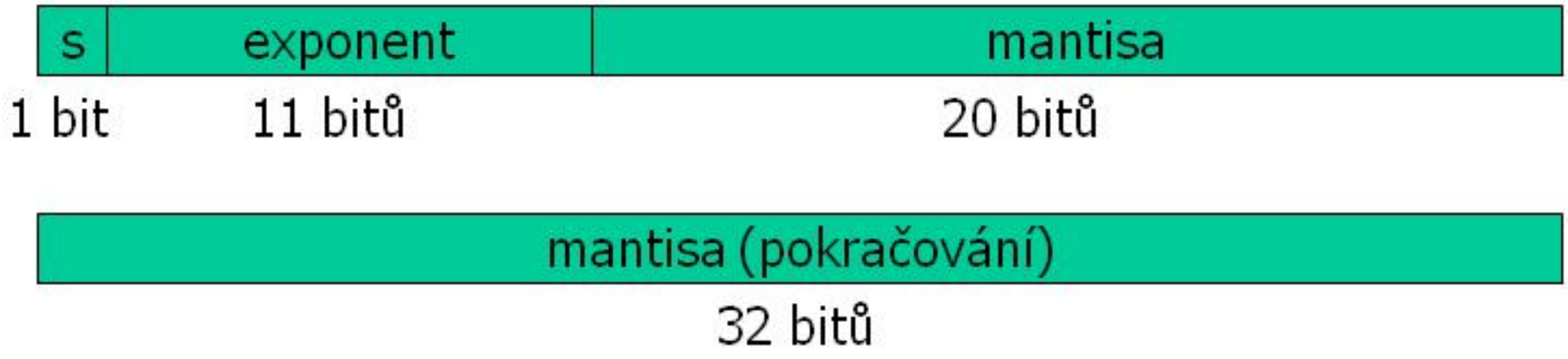
Reprezentované číslo = $(-1)^S \times F \times 2^E$
kde S, F a E jsou pole znaménka, exponentu a mantisy
(1 v poli s znamená zápornou hodnotu čísla)

FP výjimky

- Standard IEEE 754 pokrývá velmi velký rozsah reálných čísel, která mohou být vyjádřena, od nejmenších $2.0x10^{-38}$ k největším $2.0x10^{38}$.
- ! Je třeba podotknout, že rozsah je velký, ale nikoliv nekonečný…
  - Přetečení v pohyblivé řádové čárce nastává, jestliže vypočtený exponent výsledku je příliš velký a nelze ho vyjádřit v poli exponentu (příliš velké číslo). ($> 2.0x10^{38}$)
  - 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 ($>0, < 2.0x10^{-38}$)

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…



- 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 !)

Výhody dvojité přesnosti

- Tento formát dovoluje vyjádřit čísla ve větším rozsahu a to od $2.0x10^{-308}$ do $2.0x10^{308}$.
- Přestože primárním důvodem rozšíření je podstatné zvýšení přesnosti zobrazení, bylo zvětšeno i pole pro zobrazení exponentu.

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:

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

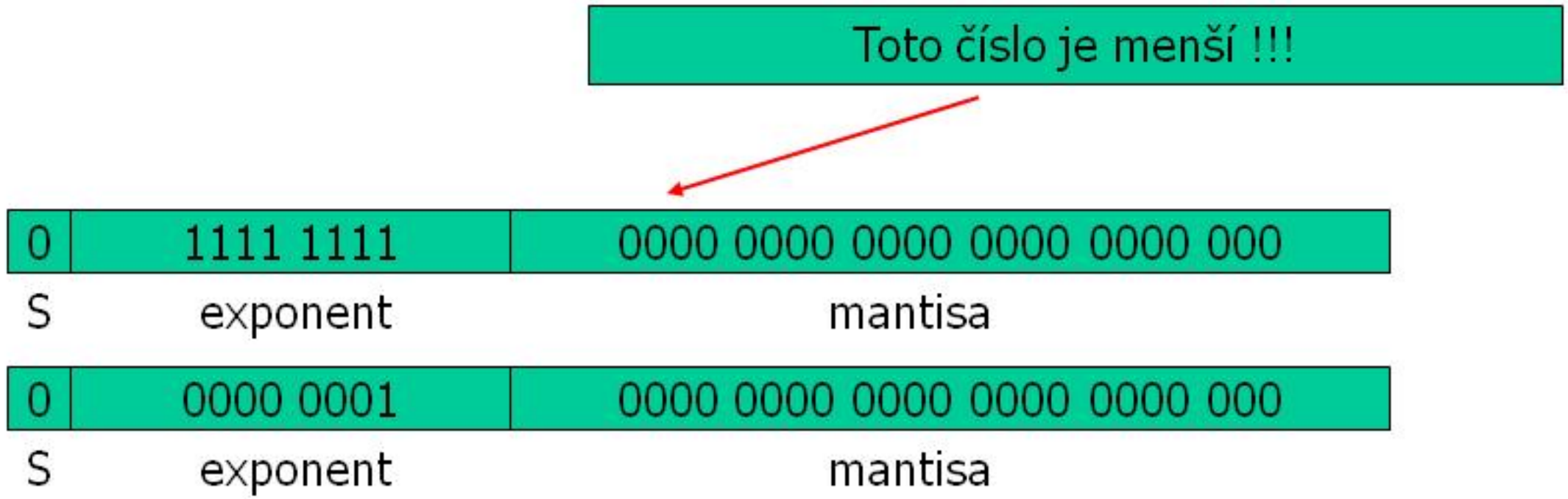
  kde bity mantisy představují zlomek mezi nulou a jedničkou.
- Pro zjednodušení a zrychlení komparace čísel bylo vhodně zvoleno i pořadí jednotlivých polí v zobrazení čísla.
  - To je hlavní důvod proč znaménkový bit leží v MSB.

Optimalizace porovnání

- Exponent leží vlevo od mantisy – to ulehčuje porovnání, které lze provést pomocí integer operace.
  - Není to tak snadné jako v případě čísel v doplňkovém kódu, protože je třeba vyšetřit znaménkový bit a amplitudu exponentu.
  - Tento postup je korektní, pokud jsou oba exponenty kladné. Jak tomu bude v případě záporných exponentů? Jak mají být kódovány? Uvědomte si, že je třeba jednoduše porovnat hodnoty dvou exponentů abychom určili jejich vzájemný vztah.

Kódování exponentu

- Kdybychom zakódovali exponenty v doplňkovém kódu, záporný exponent by se jevil jako velké kladné číslo (jednička v MSB).
- Například, zakódujeme-li 1.0×2^{-1} a 1.0×2^{1} s použitím doplňkového kódu pro exponent, dostaneme...



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.

Příklad: dekódování IEEE 754

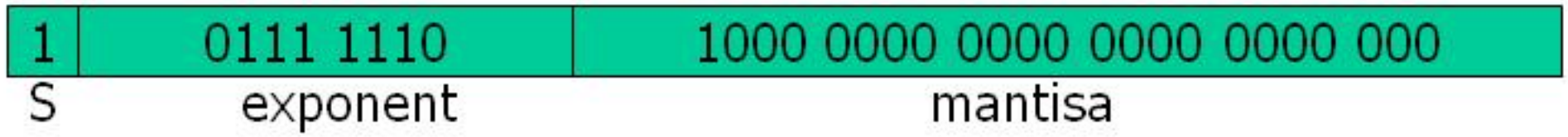
- Zakódujeme $(-0.75)_{10}$ podle IEEE 754.
- Zápis ve dvojkové soustavě… $(-0.11)_2$.
- Normalizovaná forma… $(-1.1)_2 \times 2^{-1}$.
- Požadovaný tvar…

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

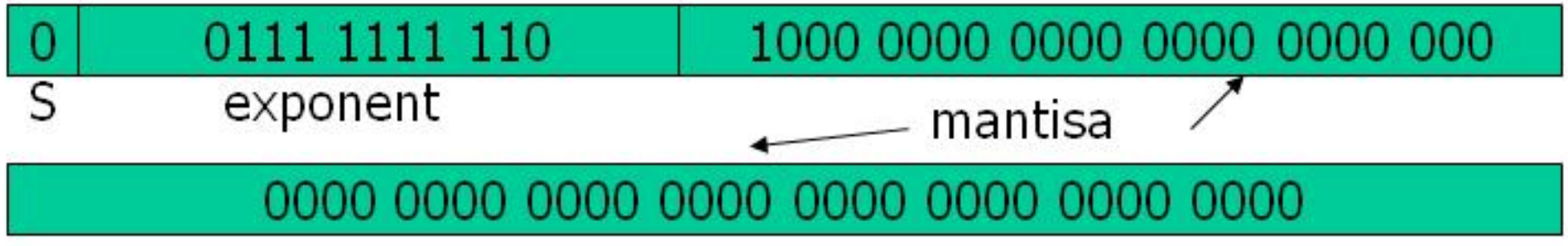
- Převod do požadovaného tvaru…

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

- Pro jednoduchou přesnost podle IEEE 754…

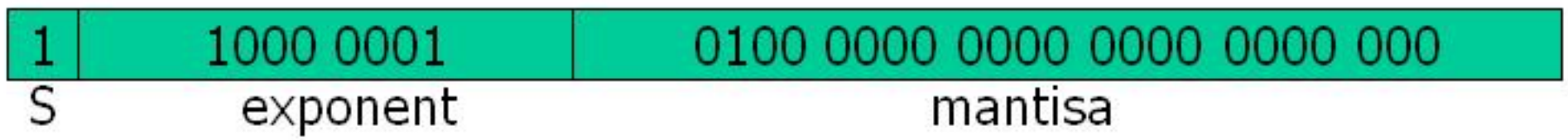


- Pro dvojnásobnou přesnost podle IEEE 754…



Příklad: dekódování IEEE 754

- Budeme dekódovat číslo podle IEEE 754…



- 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 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 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 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 FP čísel

Krok 3: Normalizace součtu

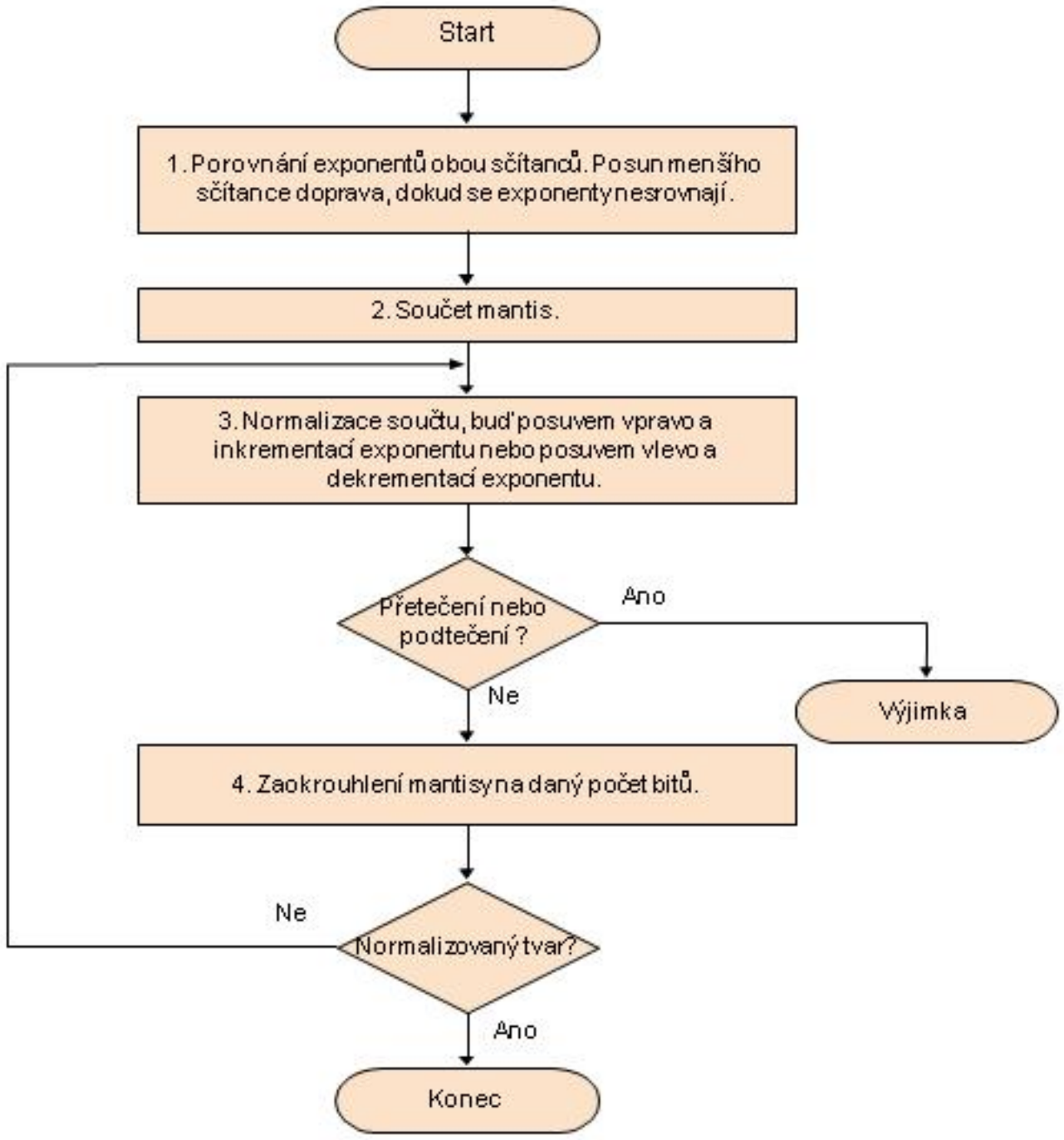
- Nakonec provedeme normalizaci součtu – převedení do standardního tvaru, který byl operací součtu porušen.
- $10.015 \times 10^1 = 1.0015 \times 10^2$
- Nezapomeňte, že i zde je nutné provést kontrolu, zda nenastalo přetečení nebo podtečení. V tomto případě k chybám nedošlo, exponent výsledku je roven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

Sčítání 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 součtu FP-čísel



Není optimalizováno!

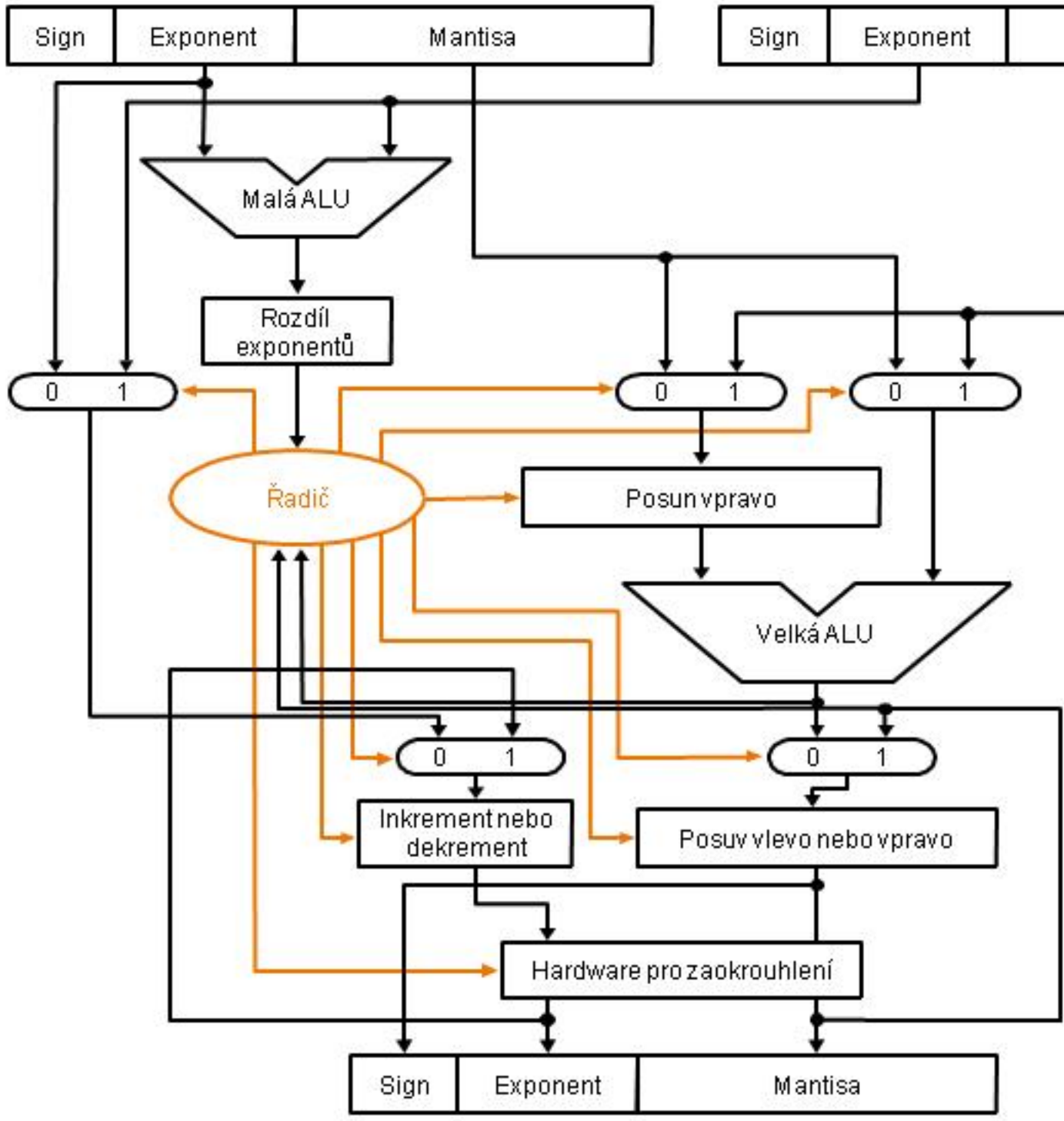
Jistě dokážete najít „rezervy“ tohoto algoritmu.

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



Porovnání exponentů

Posuv menšího čísla vpravo

Součet mantis

Normalizace

Zaokrouhlení

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 1.110×10^{10} a 9.200×10^{-5}.

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

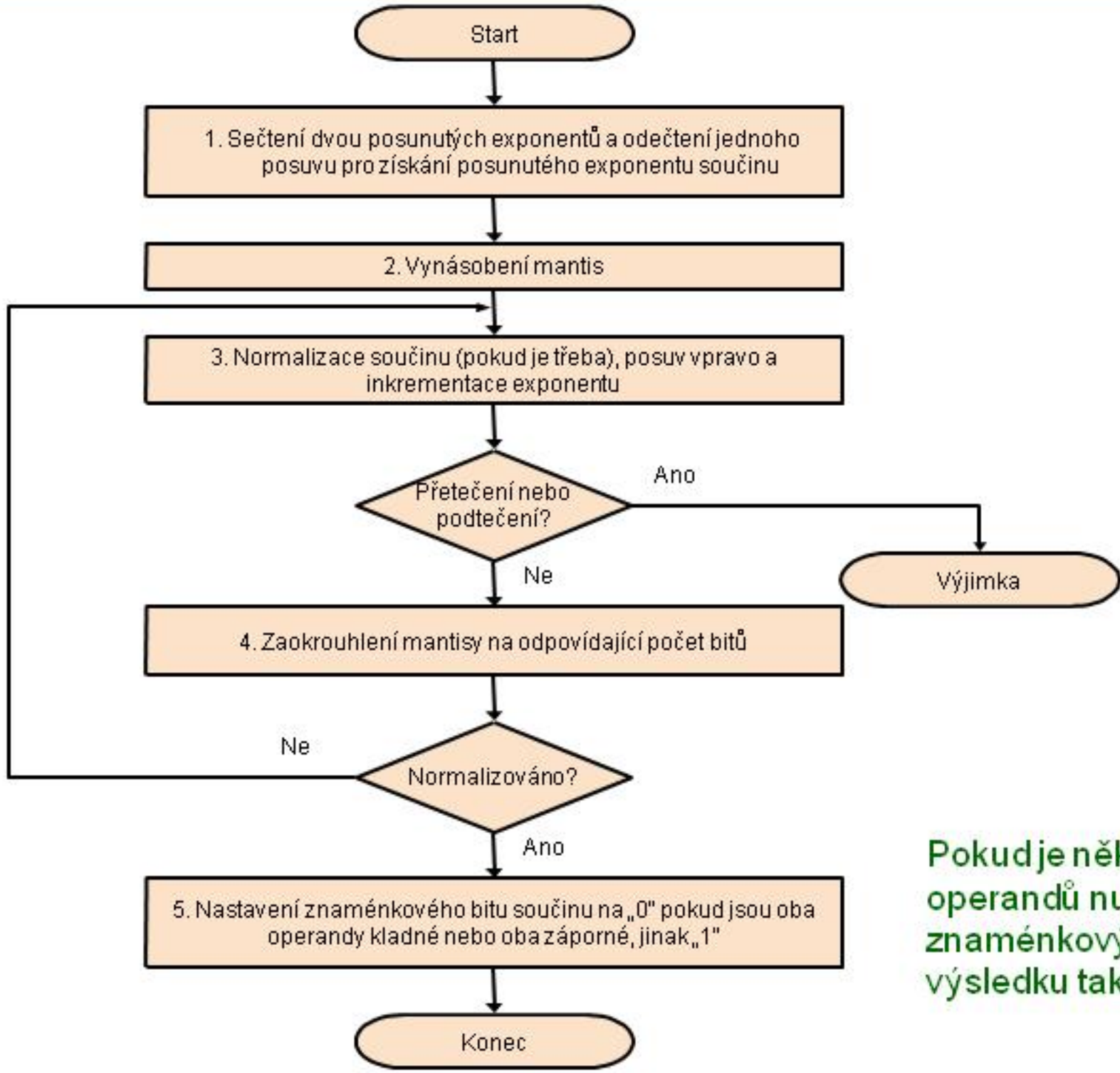
- Protože provedením operace se počet cifer zvýšil, je třeba provést zaokrouhlení výsledku.
- Použitím zaokrouhlovacích pravidel (ze základní školy) dostaneme: 1.0212×10^6 zaokrouhleno dává 1.021×10^6.
- Nakonec je opět třeba ověřit, zda zůstal zachován normalizovaný tvar stejně, jako tomu bylo v případě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

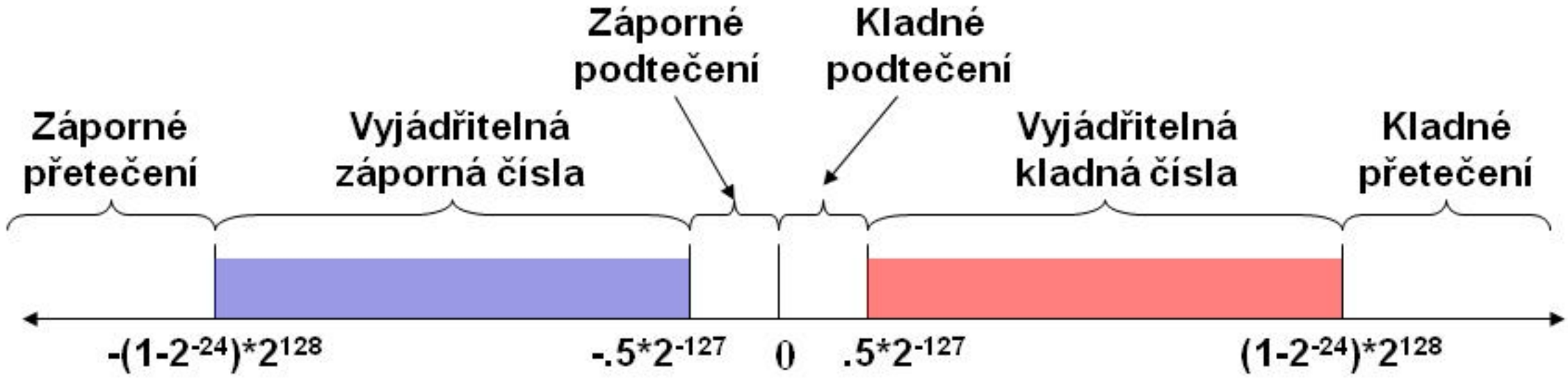


Pokud je některý z operandů nulový, je znaménkový bit výsledku také „0“

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



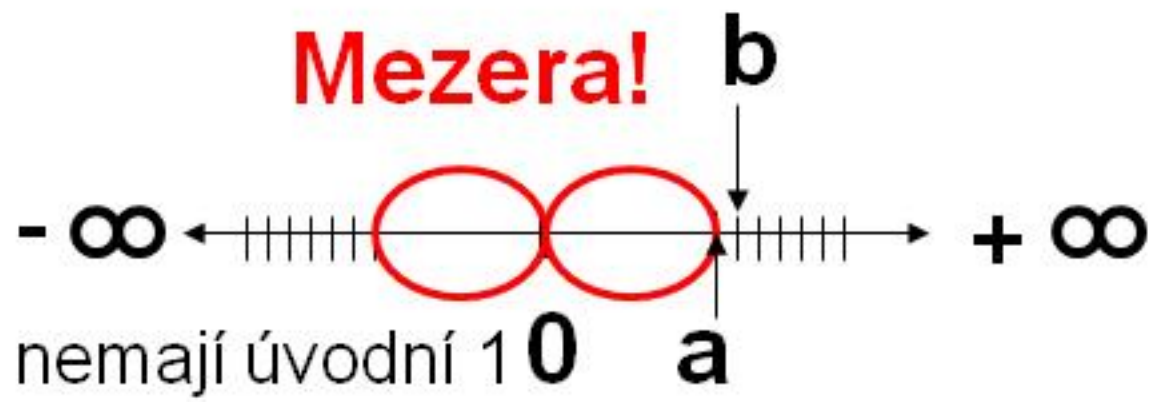
Speciální hodnoty	Exponent	Mantisa
+/- 0	0000 0000	0
denormalizované číslo	0000 0000	nenulová
NaN	1111 1111	nenulová
+/- nekonečno	1111 1111	0

Not a Number

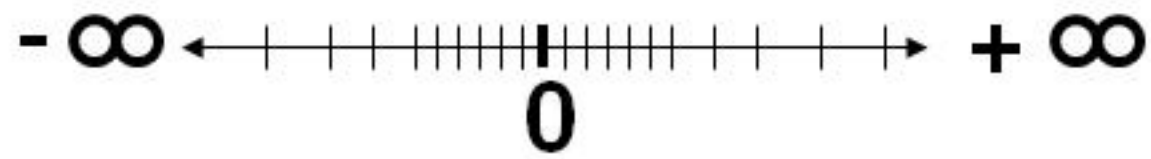
- Co je výsledkem operace: `sqrt(-4.0)` or `0/0`?
  - Jestliže nekonečno není chyba, tohle by také nemělo.
  - Nazývá se Not a Number (NaN)
  - Exponent = 255, mantisa je nenulová
- Aplikace
  - někdy lze „NaNy“ využít při ladění programu
  - šíření v návazných operacích: op(NaN, X) = NaN

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: $b = 1.001_2 * 2^{-126} = 2^{-126} + 2^{-150}$
  - $a - 0 = 2^{-126}$
  - $b - a = 2^{-150}$



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



Častý omyl při práci s FP čísly

- FP operace Add, Sub jsou asociativní: CHYBA!

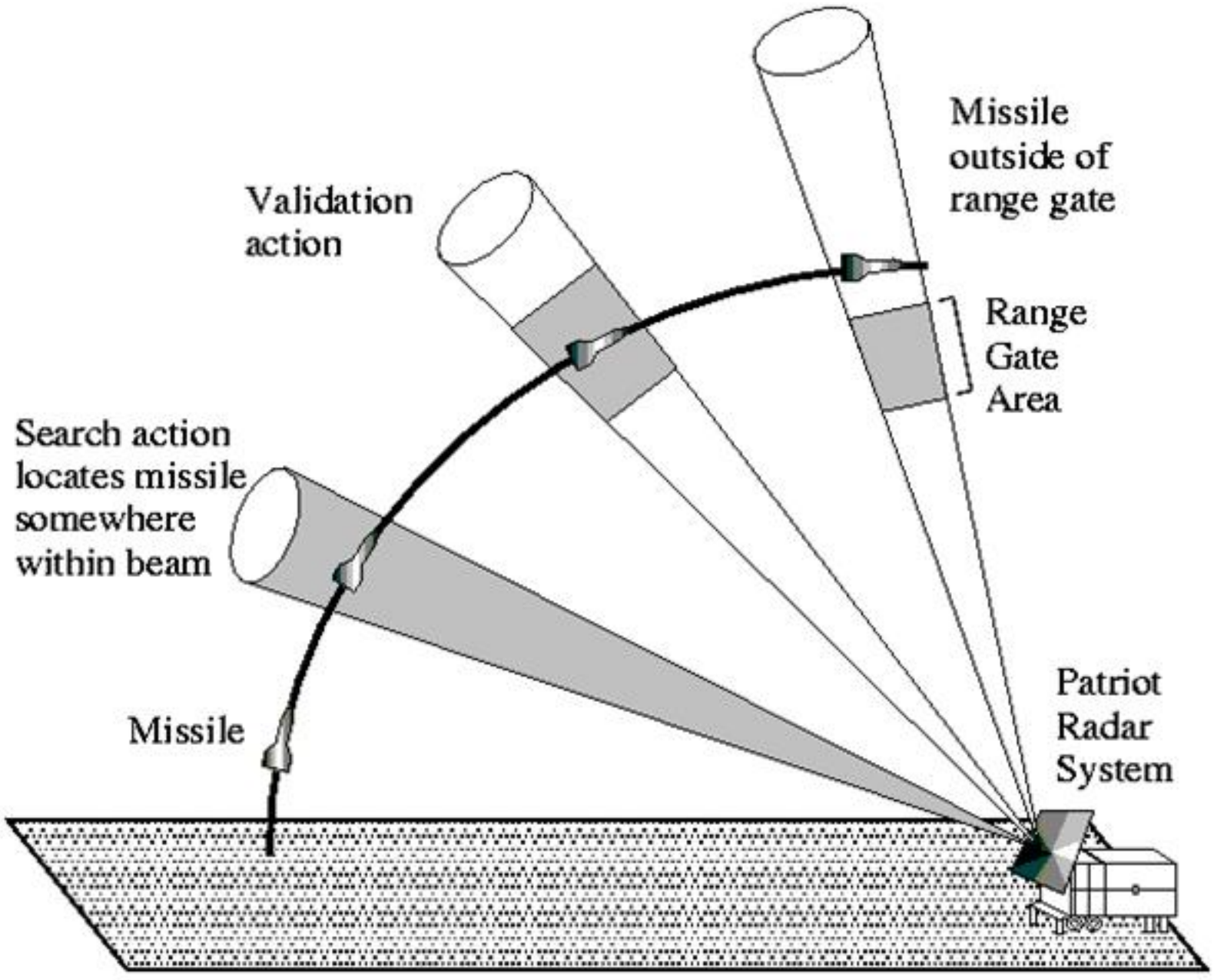
$x = -1.5 \times 10^{38}$ $y = 1.5 \times 10^{38}$ $z = 1.0$

$x + (y + z) = -1.5\text{x}10^{38} + (1.5\text{x}10^{38} + 1.0) = -1.5\text{x}10^{38} + (1.5\text{x}10^{38}) = \underline{0.0}$

$(x + y) + z = (-1.5\text{x}10^{38} + 1.5\text{x}10^{38}) + 1.0 = (0.0) + 1.0 = \underline{1.0}$

- 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}

Chyby u čísel ve formátu FP



Perský záliv – 1990

Chyby v aritmetice protiraketového systému způsobily vážné ztráty

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.

Zaokrouhlovací procesy

- 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

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

- 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 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 ?!

Oddělené FP registry?

- Jedno z důležitých rozhodnutí při návrhu procesorů je, zda pro práci s FP čísly budou použity tytéž registry jako pro čísla integer a nebo vyhrazená sada registrů.
  - Operace integer a FP operace často pracují nad různými daty a proto nevzniká příliš mnoho konfliktů při sdílení registrů.
  - Hlavní důraz je kladen na oddělený soubor instrukcí přenosu dat pro FP.
  - Je-li použita oddělená sada registrů, dostáváte dvojnásobek registrů, aniž by bylo třeba více bitů v instrukčním formátu.
- Rozhodnutí návrháře – stojí to za to ?

Historická hlediska

- Jedním z důvodů, proč některé návrhy používají oddělené registrové banky pro integer a FP čísla je omezení, pocházející ze starších počítačů.
- V 80-tých letech nebylo ještě možné kvůli nízké hustotě integrace jednoduše zahrnout FP hardware na chip s hlavním procesorem. FP jednotka a její registrová banka byly implementovány na dalším chipu, který byl dodáván volitelně. Označoval se jako *akcelerátor* nebo jako *matematický koprocesor*.
- V 90-tých letech se začíná FP hardware integrovat rovnou do procesoru (spolu s mnoha dalšími funkcemi). Od té doby je použití koprocesorů méně časté.

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.

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.

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 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 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.

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

- 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

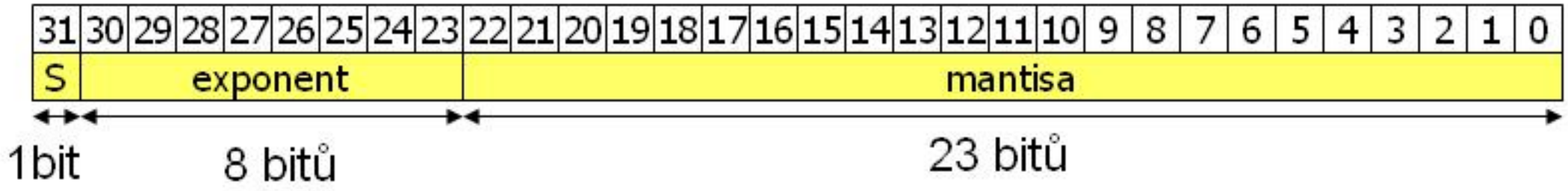
- Zásobník v systému pohyblivé řádové čárky Intel IA-32 je široký 80 bitů => označení *double extended precision*.
- Všechna FP čísla jednoduché i dvojité přesnosti jsou konvertována do tohoto formátu, když se přesouvají z paměti do zásobníku a naopak.
- FP registry mají šířku 80 bitů.
- Leží-li jeden 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).

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, 2×10^{-38}... 2×10^{38})
    - `add.s, sub.s, mul.s, div.s`
  - Dvojitá přesnost (64 bitů, 2×10^{-308}... 2×10^{308})
    - `add.d, sub.d, mul.d, div.d`
- Typ není asociován s daty, význam bitů závisí na kontextu (například *int* vs. *float*)

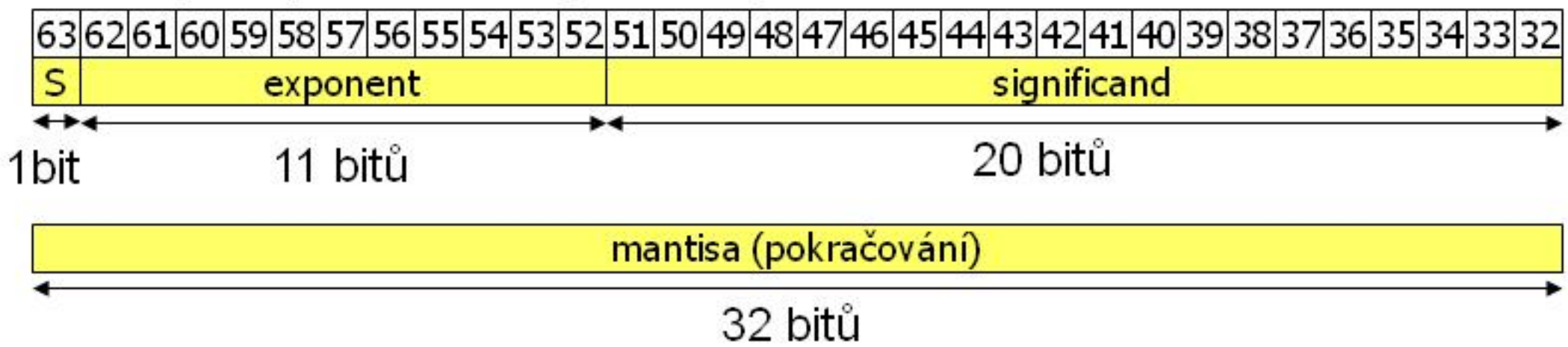
Standardní reprezentace FP čísel IEEE 754

Jednoduchá přesnost (32-bit)



$(-1)^{sign} \times (1 + mantisa) \times 2^{exponent-127}$

Dvojitá přesnost (64-bit)



$(-1)^{sign} \times (1 + mantisa) \times 2^{exponent-1023}$

Úvod do organizace počítače

Organizace procesoru

Návrh datových cest

Opakování

- Konstrukce ALU
  - Stavba bloků (návrh pomocí hradel)
  - Modulární návrh
    - Multiplexor vybírá požadovanou operaci
    - Všechny operace se vykonávají paralelně
    - Sčítačka typu carry lookahead
- Počítačová aritmetika
  - Konečná přesnost
  - Matematické zákony pro reálná čísla „neplatí“ zcela
  - Čísla integer: reprezentace – dvojkový doplněk
  - Čísla FP: IEEE 754 standard

Přehled

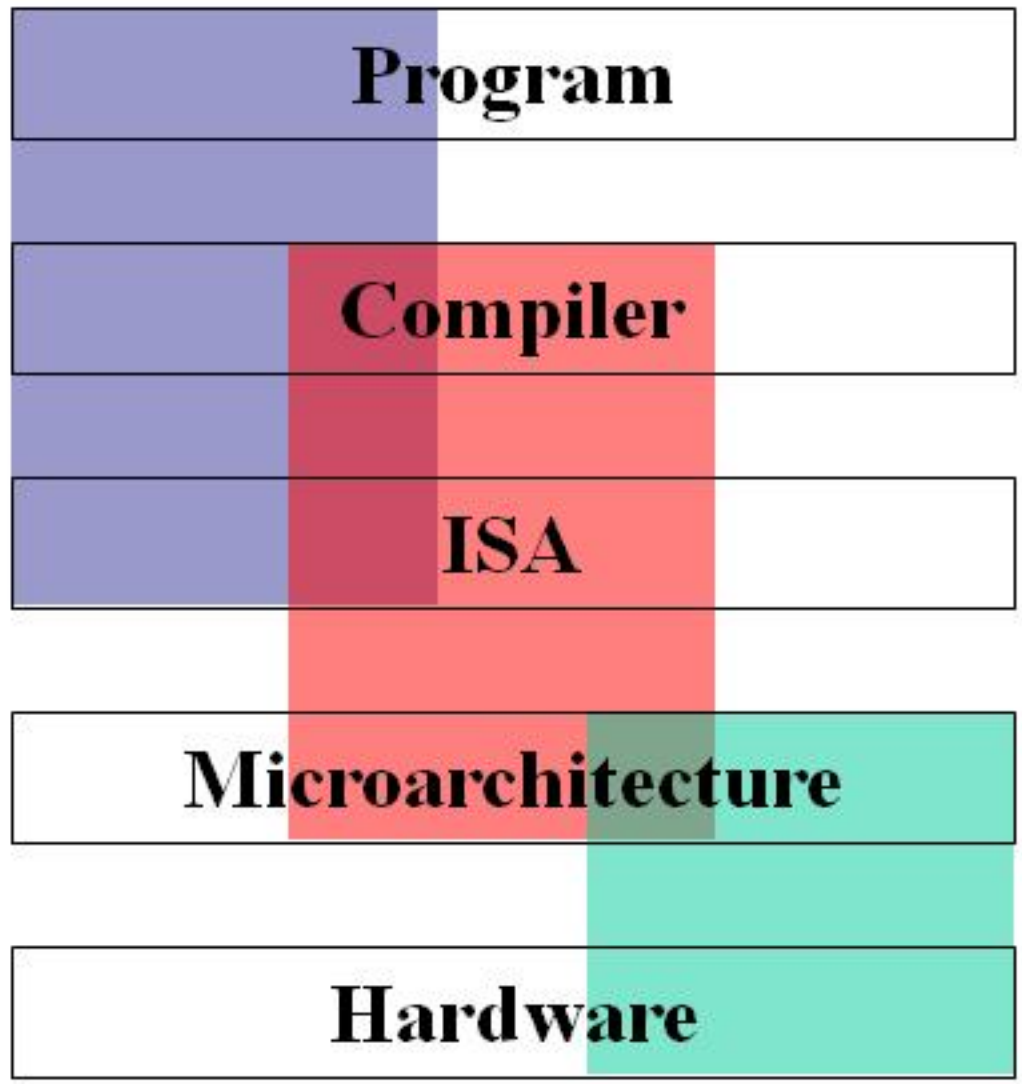
- Organizace počítače (mikroarchitektura)
- Organizace procesoru
- Datové cesty v procesoru
  - Řízení
  - Registrový soubor
- Přehled implementace procesoru
- Metodologie taktování činnosti (Processor Clocking)
- Sekvenční obvody
  - Registry typu Latch
  - Registry

Nové – Návrh datových cest

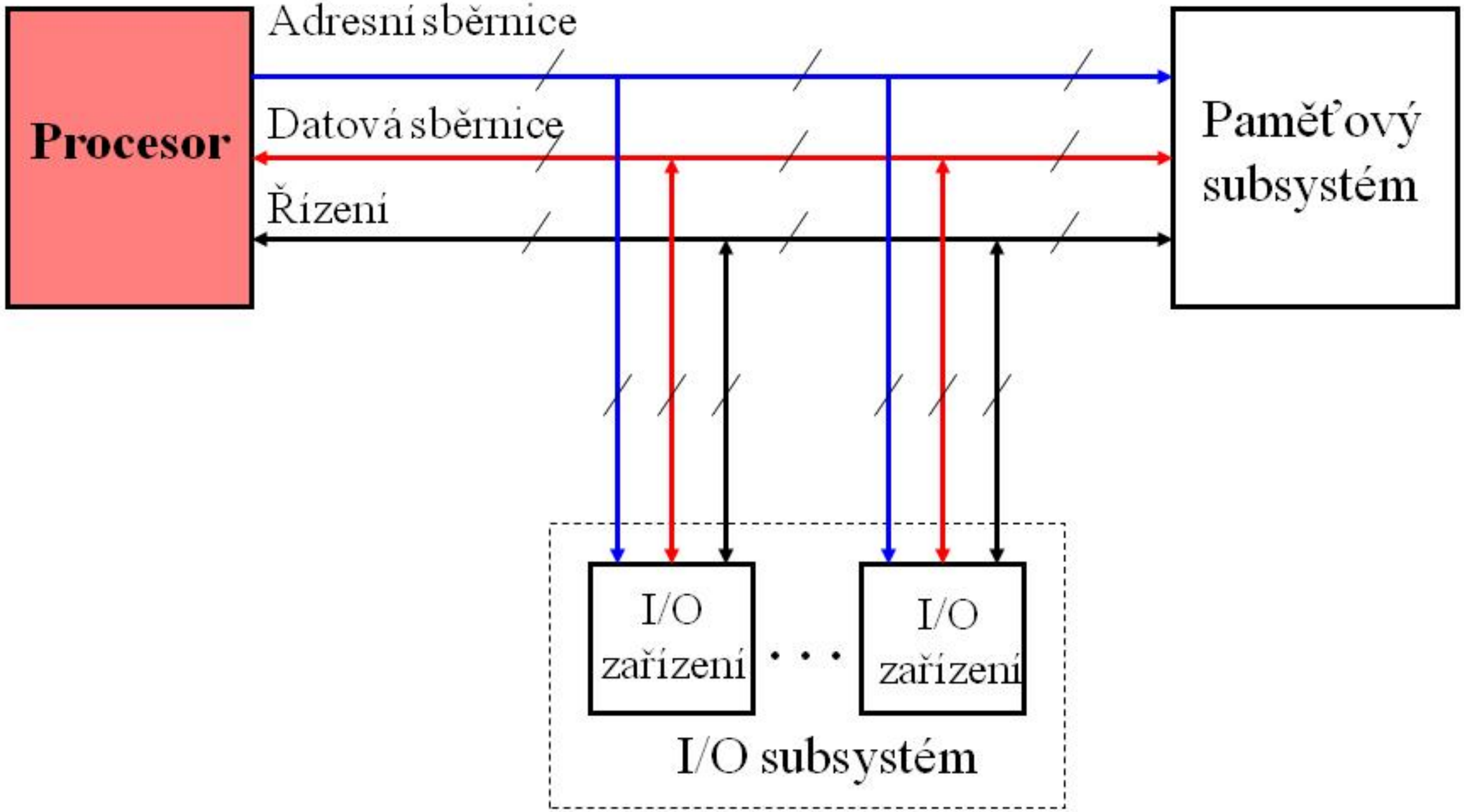
- **Datové cesty -** implementace fází fetch-decode-execute
- **Metodologie návrhu**
  - Určení tříd instrukcí a instrukčních formátů
  - Vytvoření sekcí datových cest pro každý instrukční fmt.
  - Sloučení sekcí tak, aby byly pokryty potřebné přesuny u MIPS
- *Otázka #1*: Co jsou to instrukční třídy?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

Výkon procesoru

CPU time = IC * CPI * Cycle time



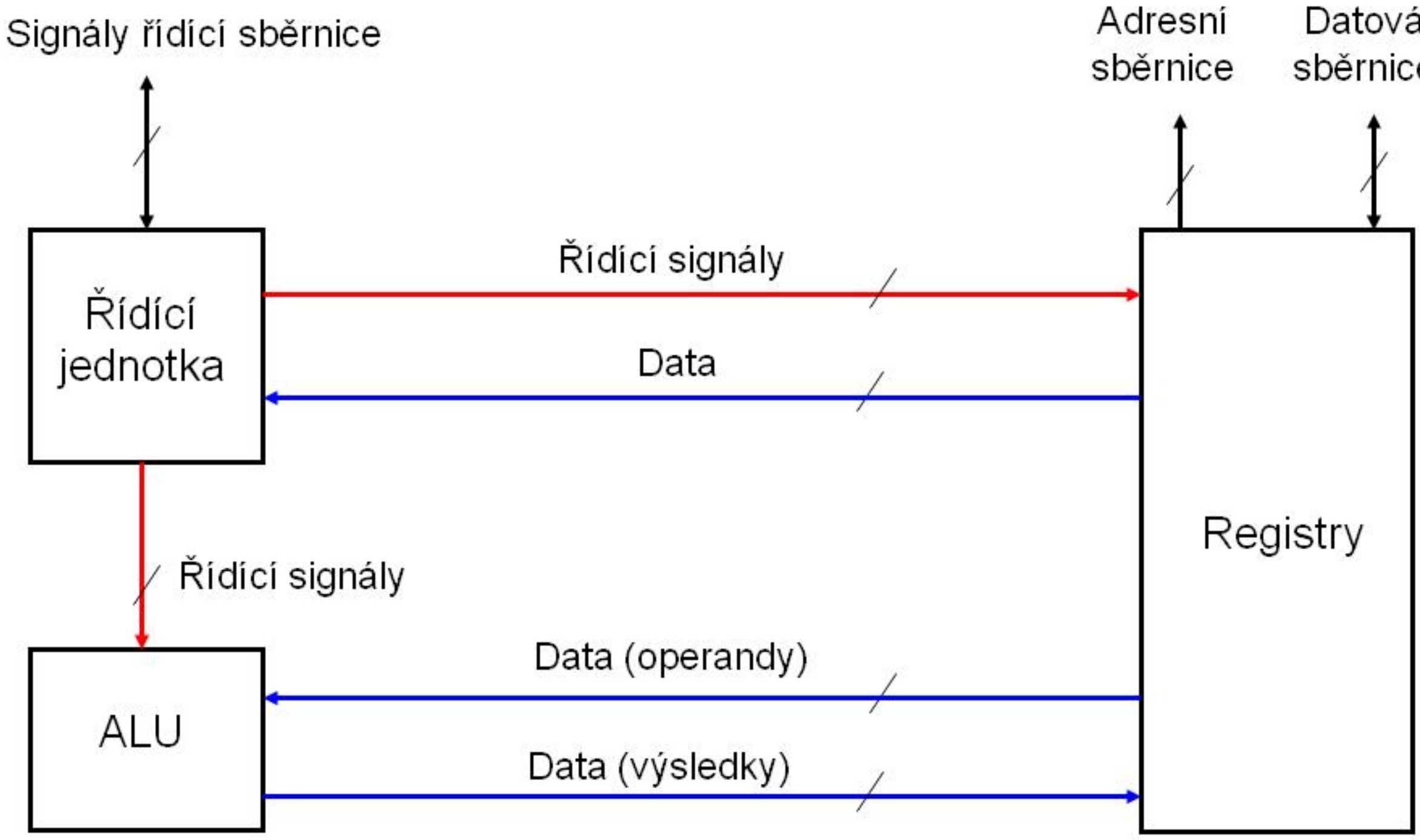
Organizace počítače



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

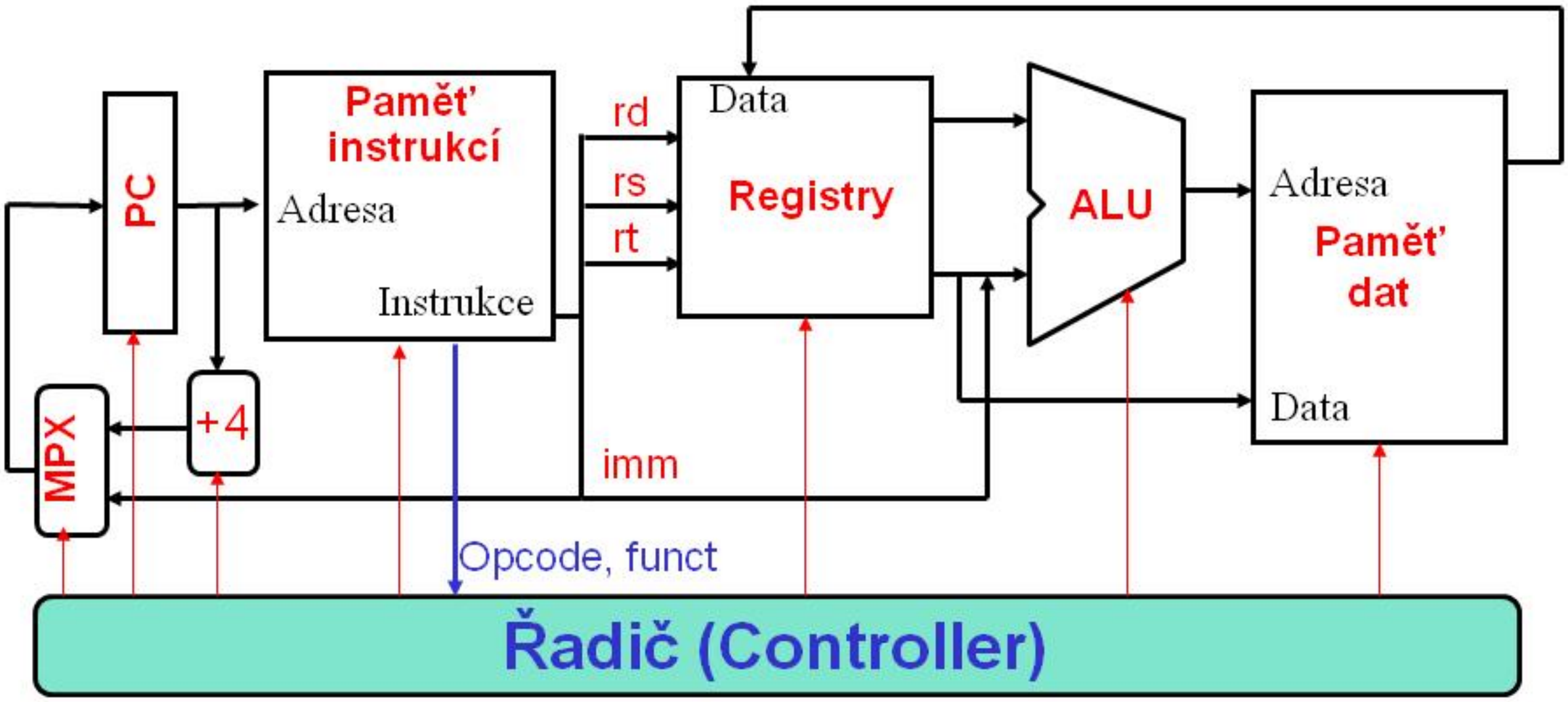
Organizace procesoru



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



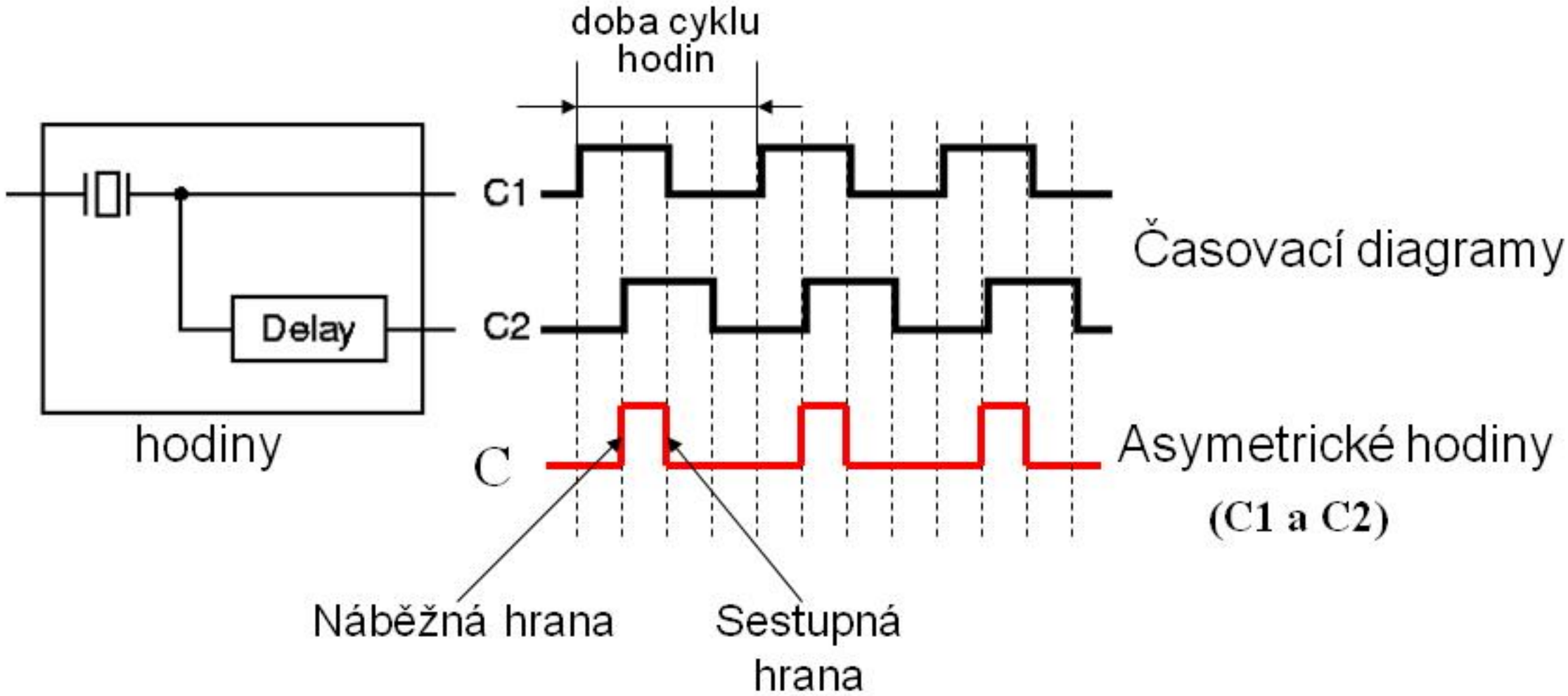
- Datové cesty umožňují přesuny obsahu registrů při provádění instrukcí
- Řízení zajišťuje provedení správných přesunů

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)

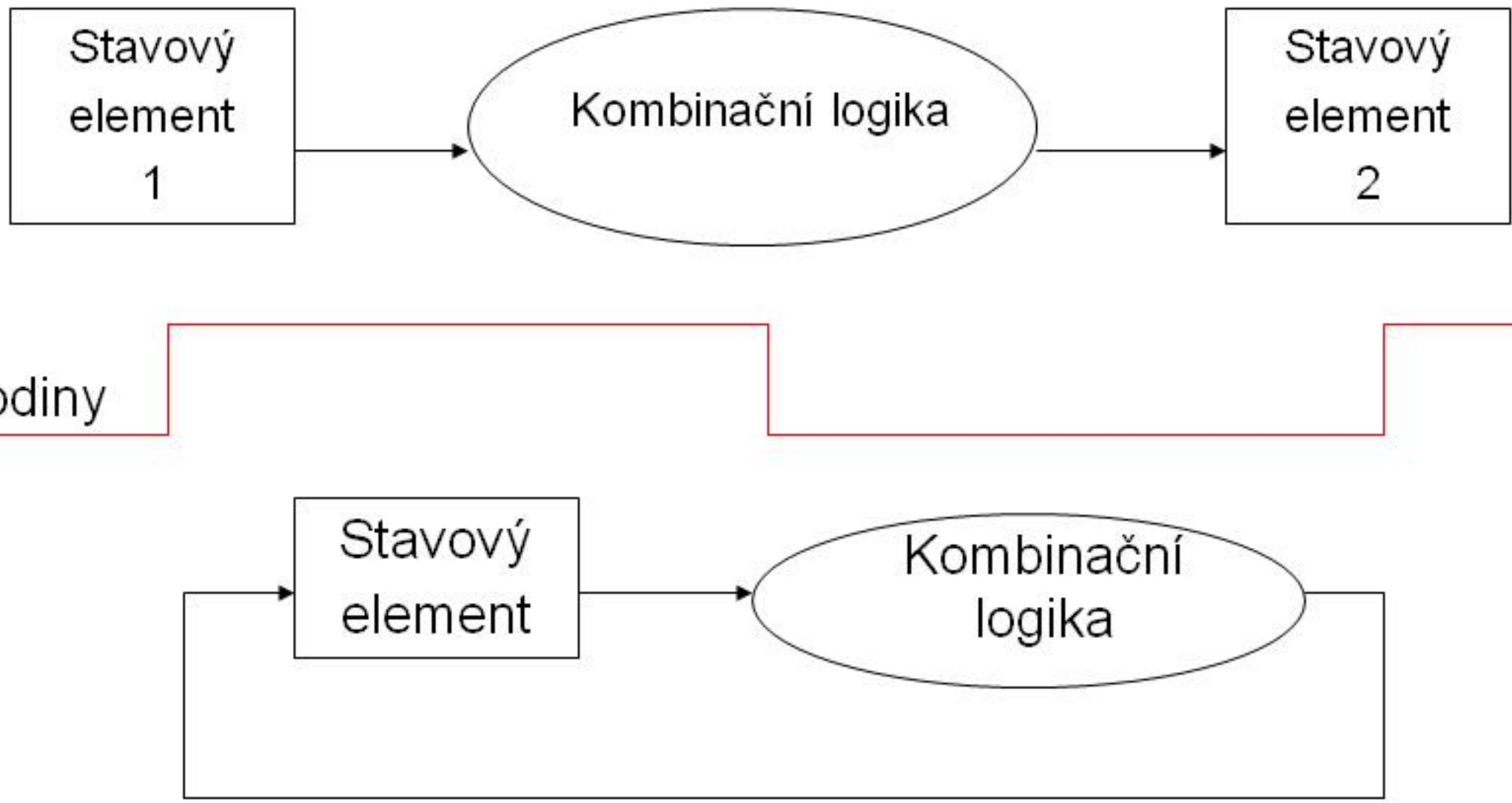
Metodologie taktování

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

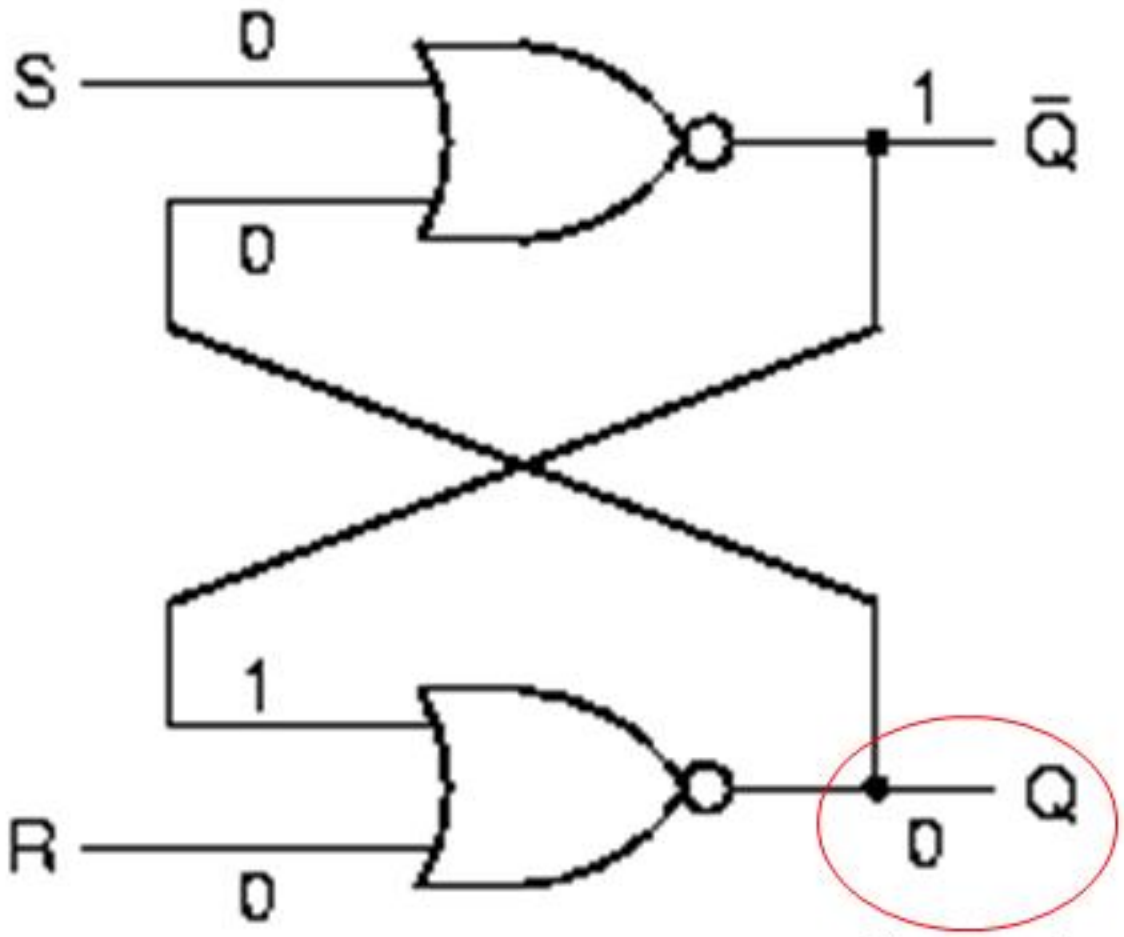


Taktování hranou hodin

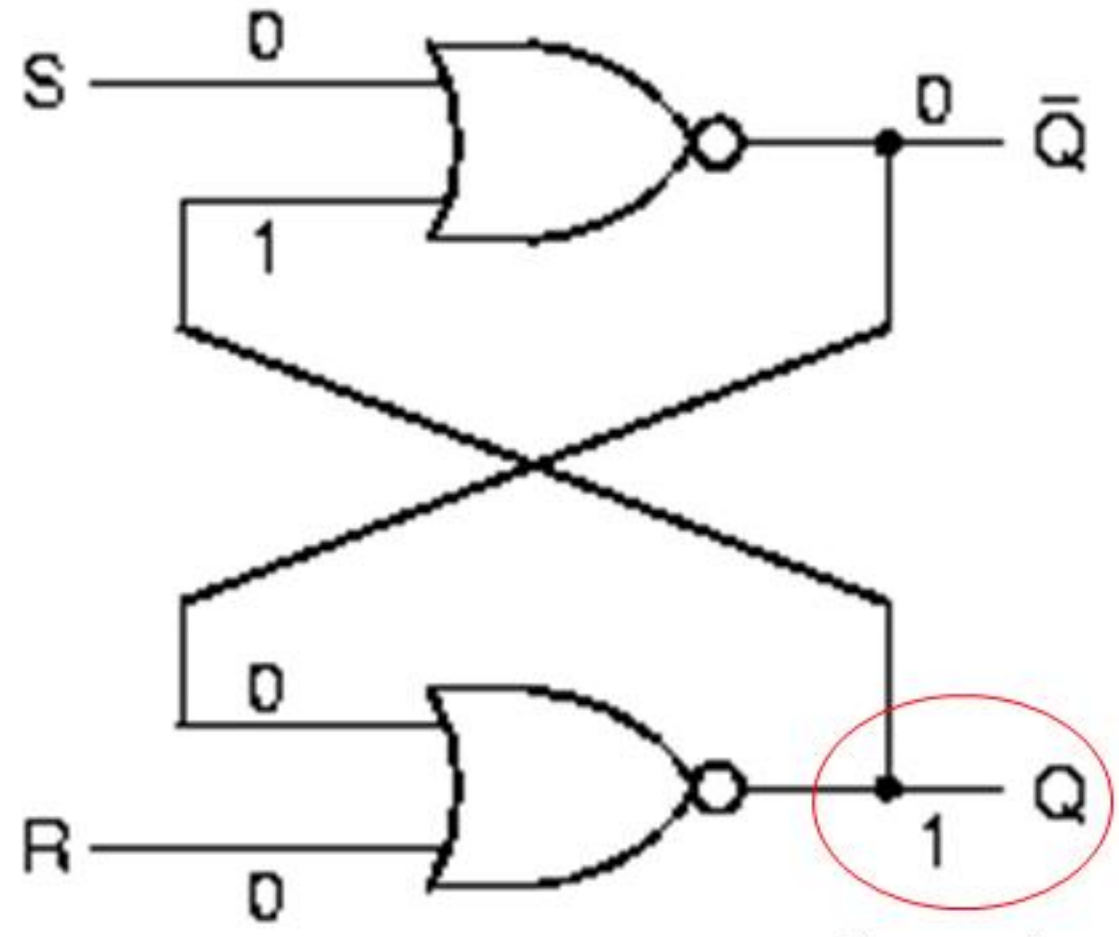
- Aktivní je buď náběžná nebo sestupná hrana hodin
- Ke změnám stavu dochází pouze na aktivní hraně hodin



SR Latch s hradly NOR



State 0

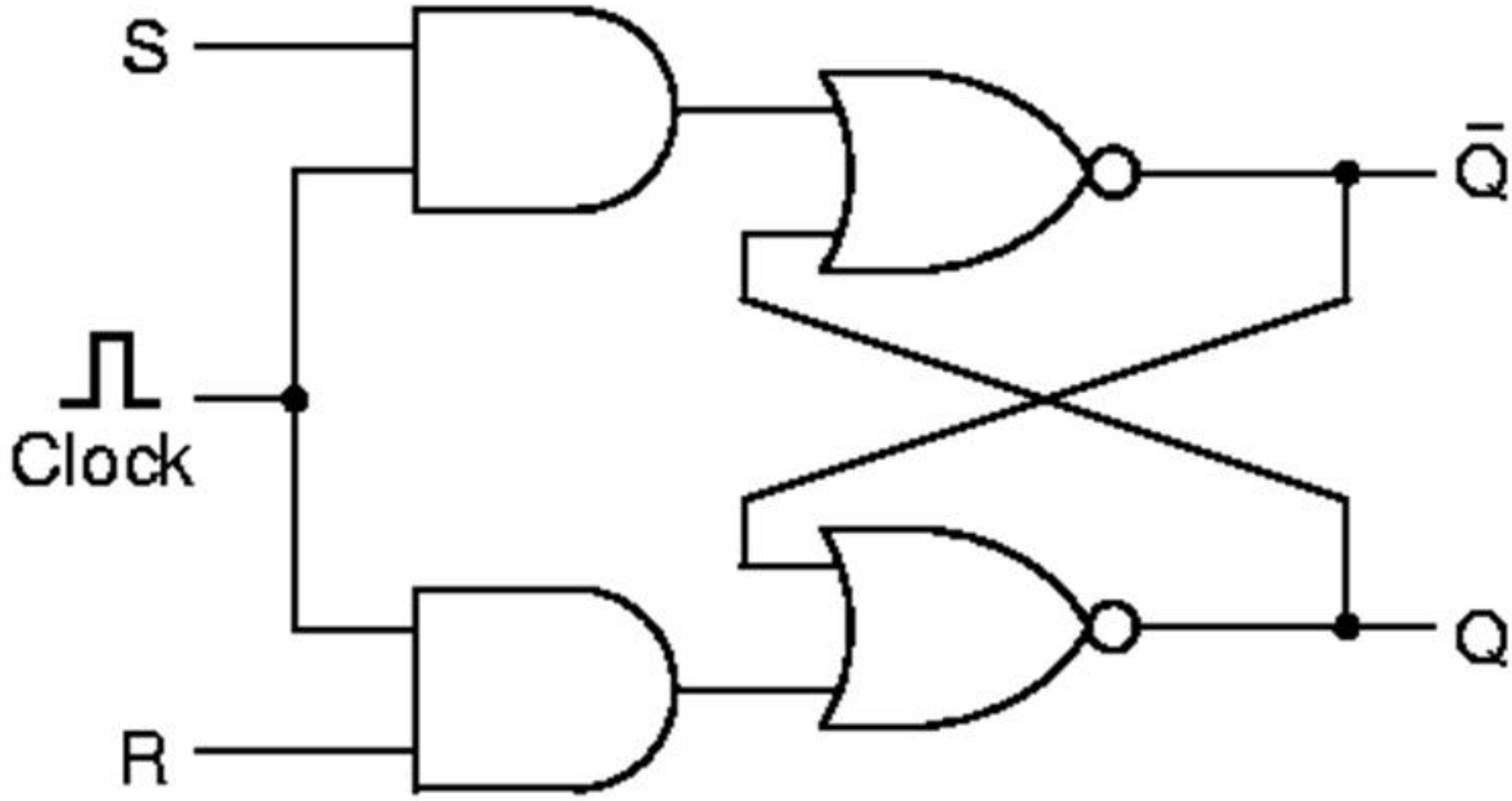


State 1

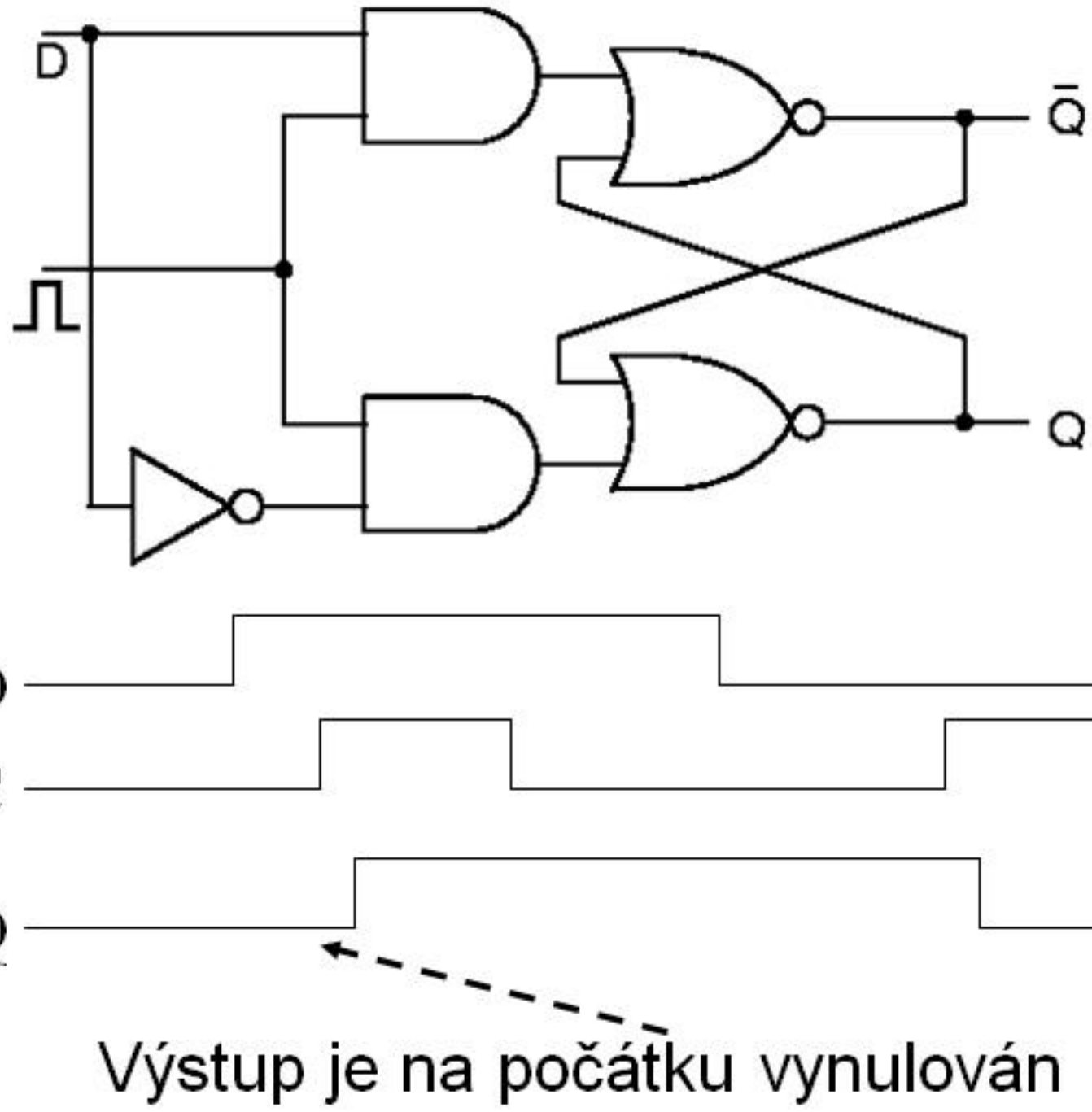
Vstupy { S - set, R - reset

Výstupy: Q and $\bar{Q}$

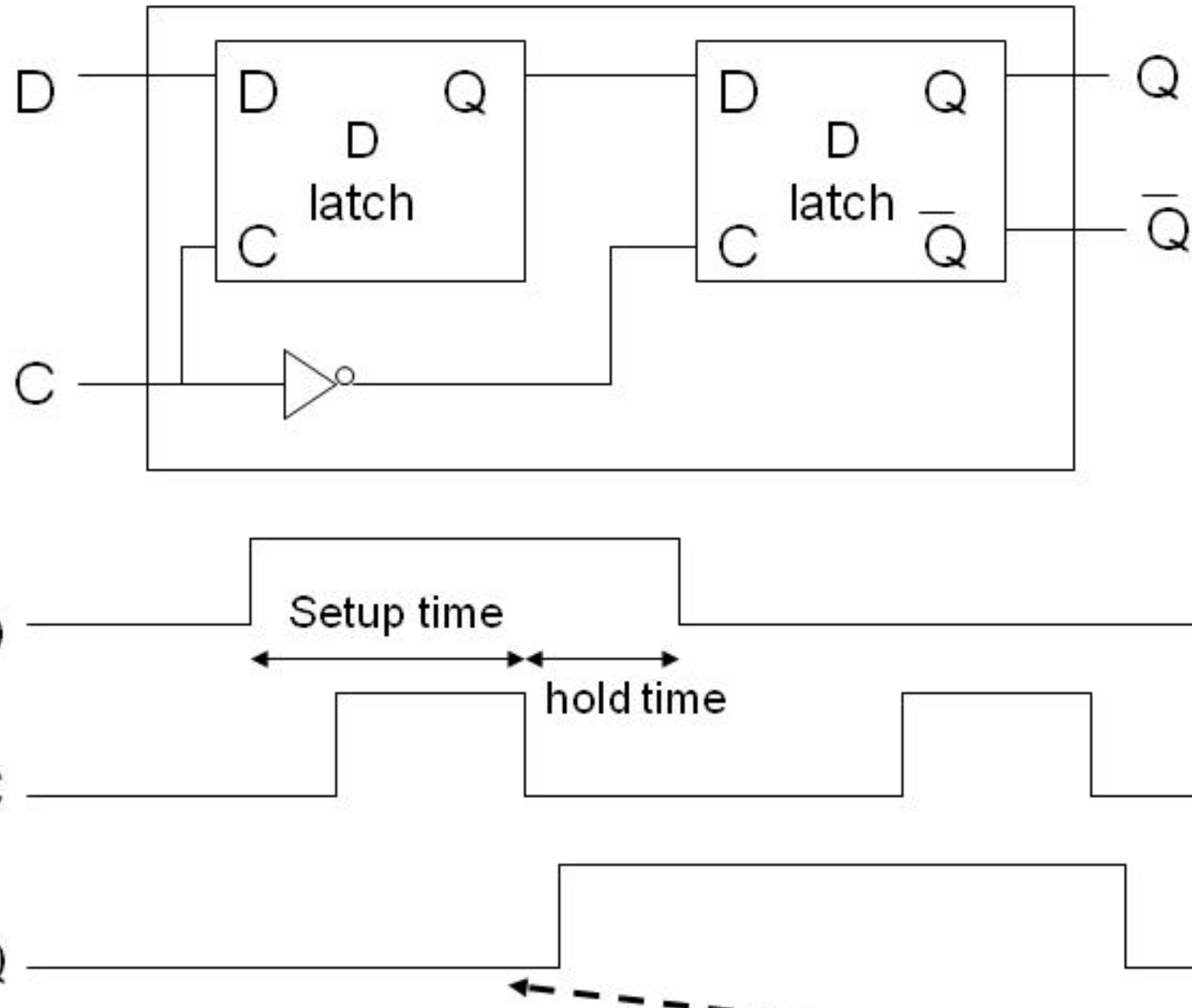
Taktovaný SR Latch



Taktovaný D Latch

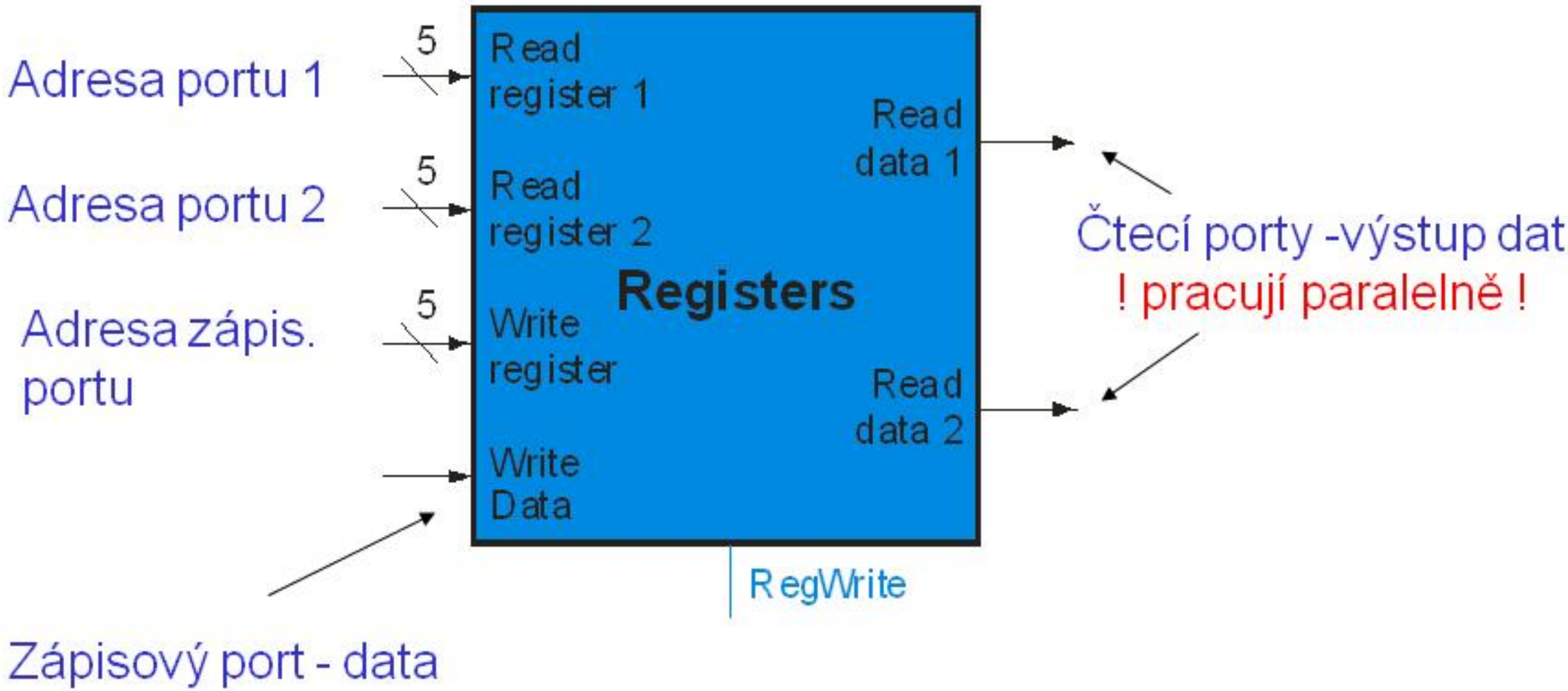


Klopný obvod typu D

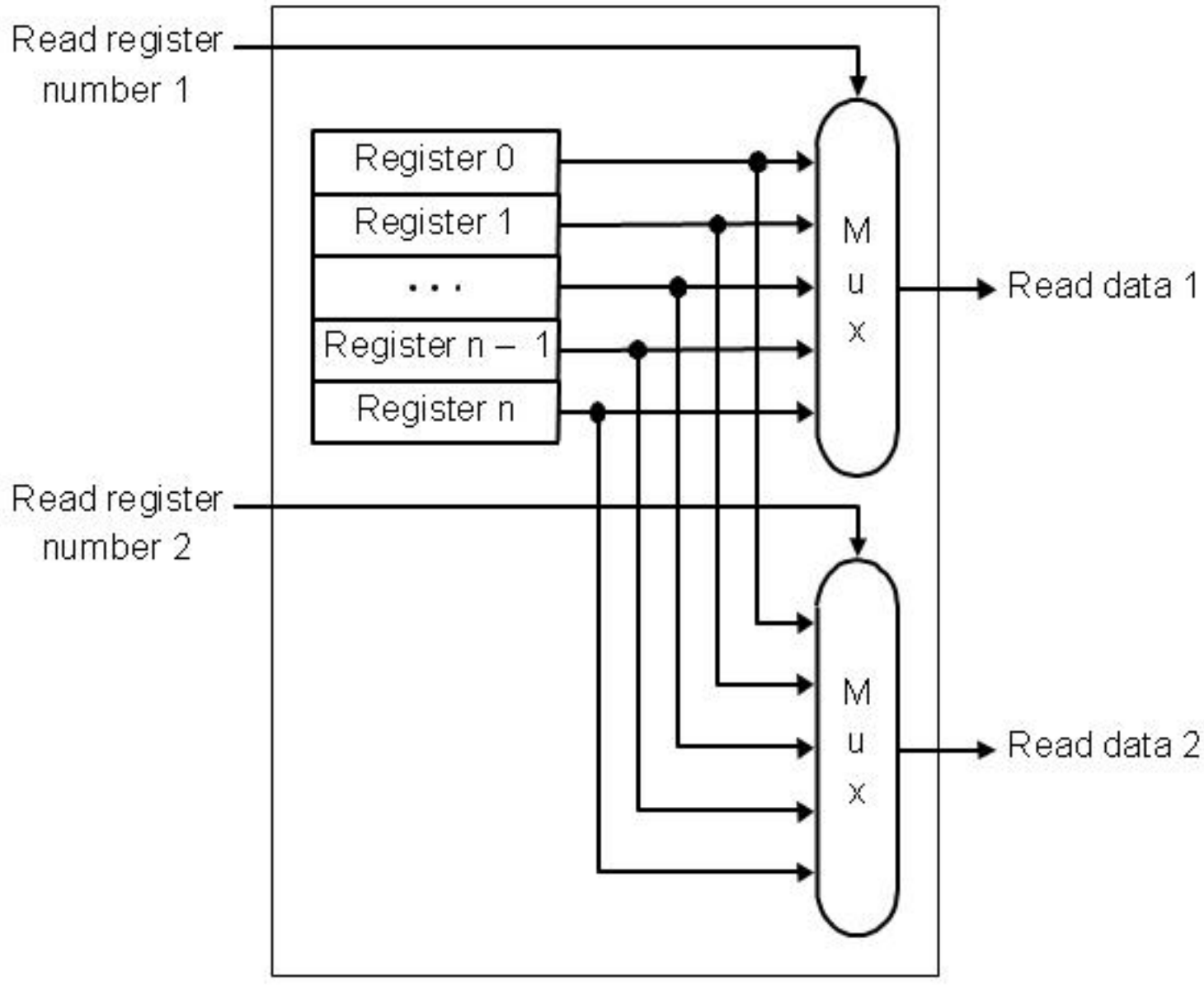


Aktivní sestupná hrana, výstup je na počátku vynulován

Registrový soubor

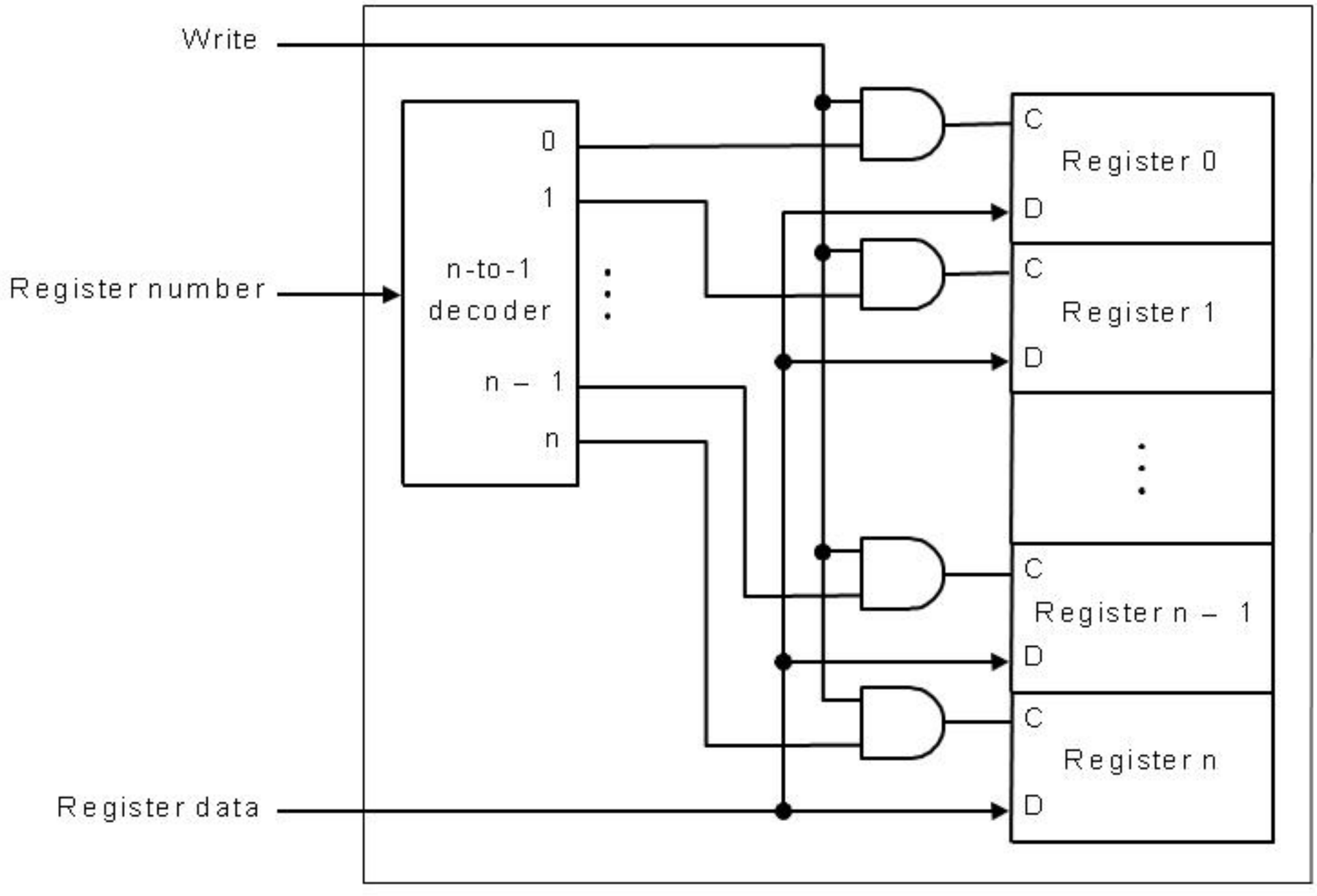


Čtecí porty registrového souboru



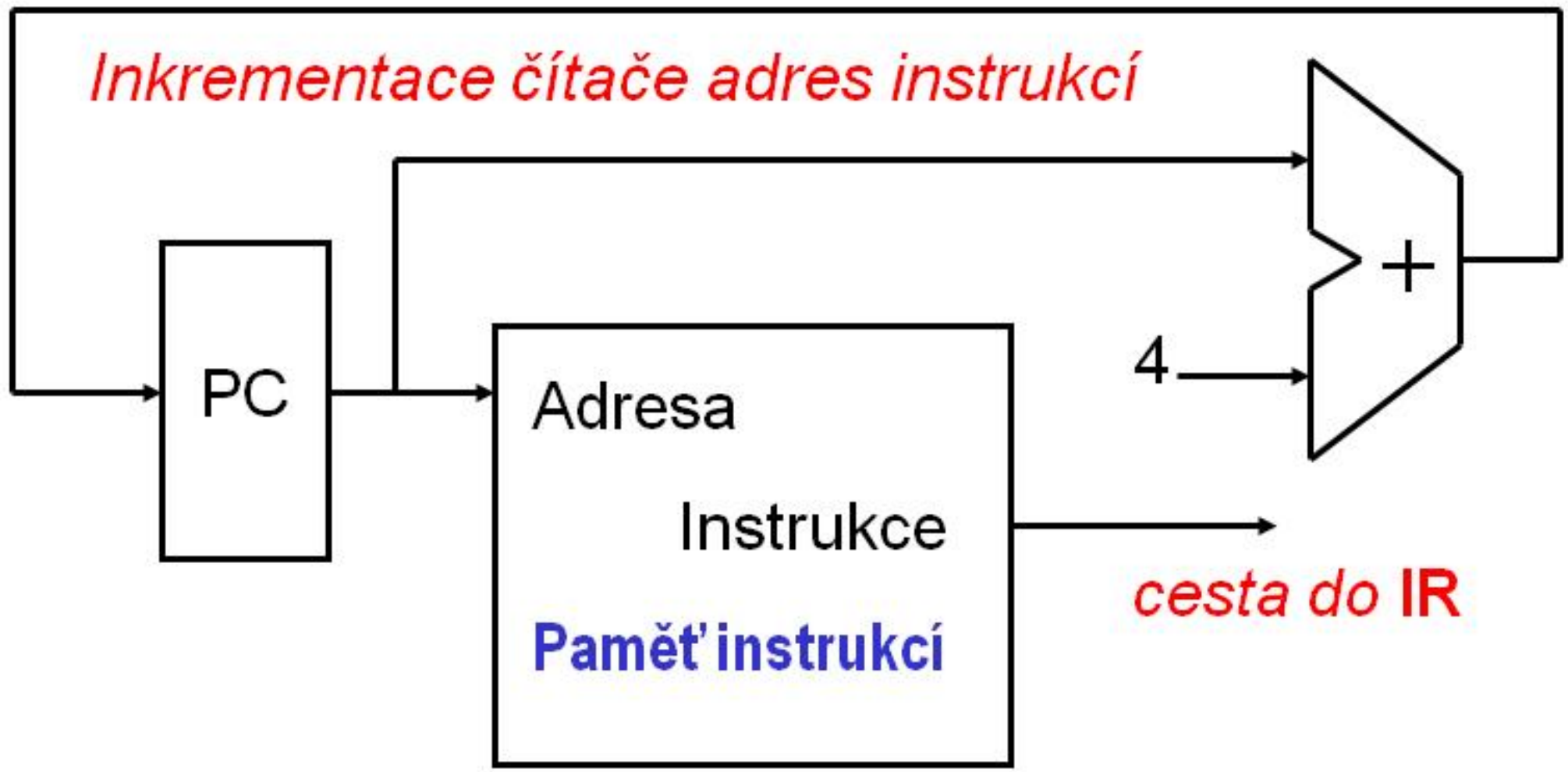
Implementace čtecích portů

Zápisové porty registrového souboru



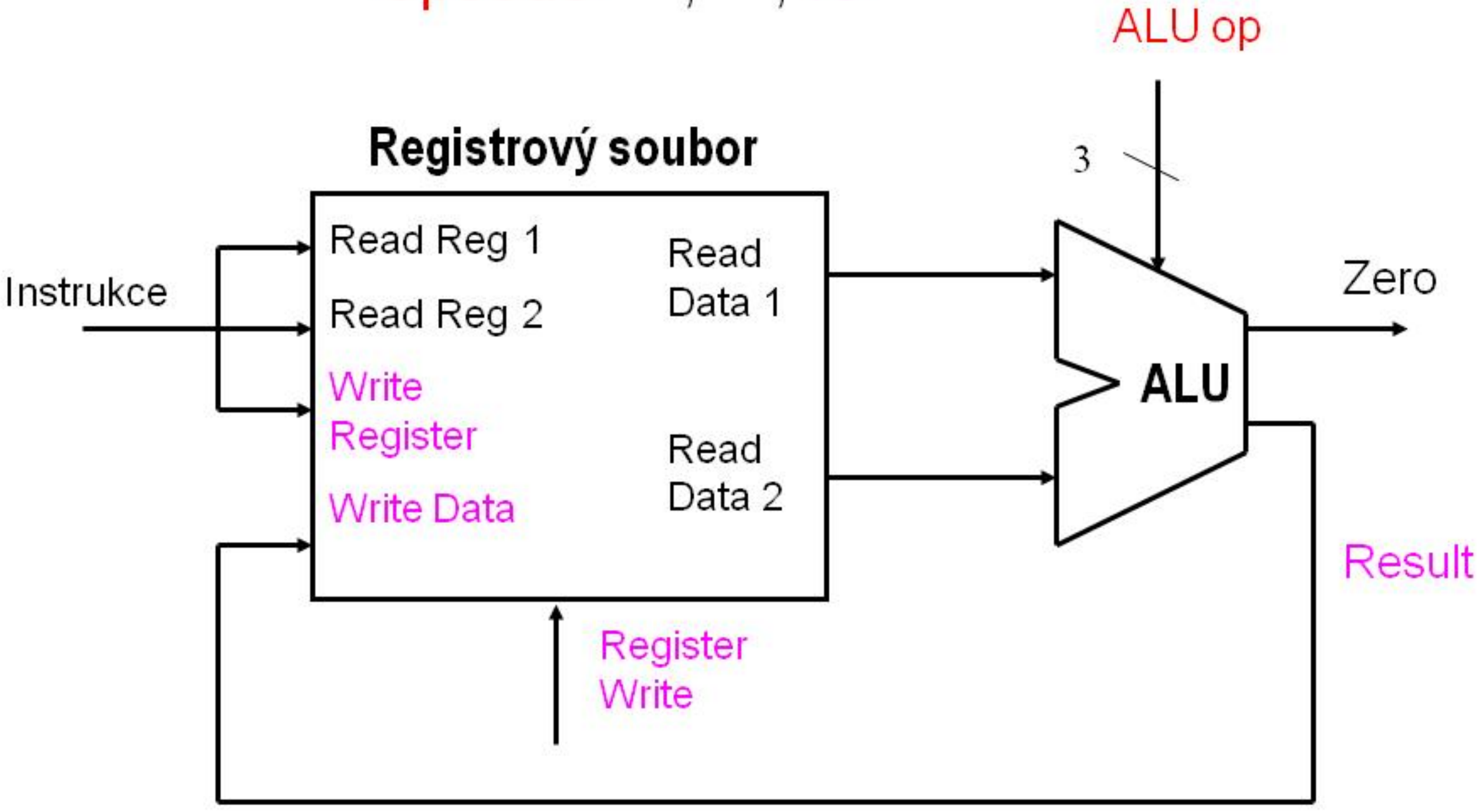
Základní datové cesty

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



Datové cesty pro R-formát

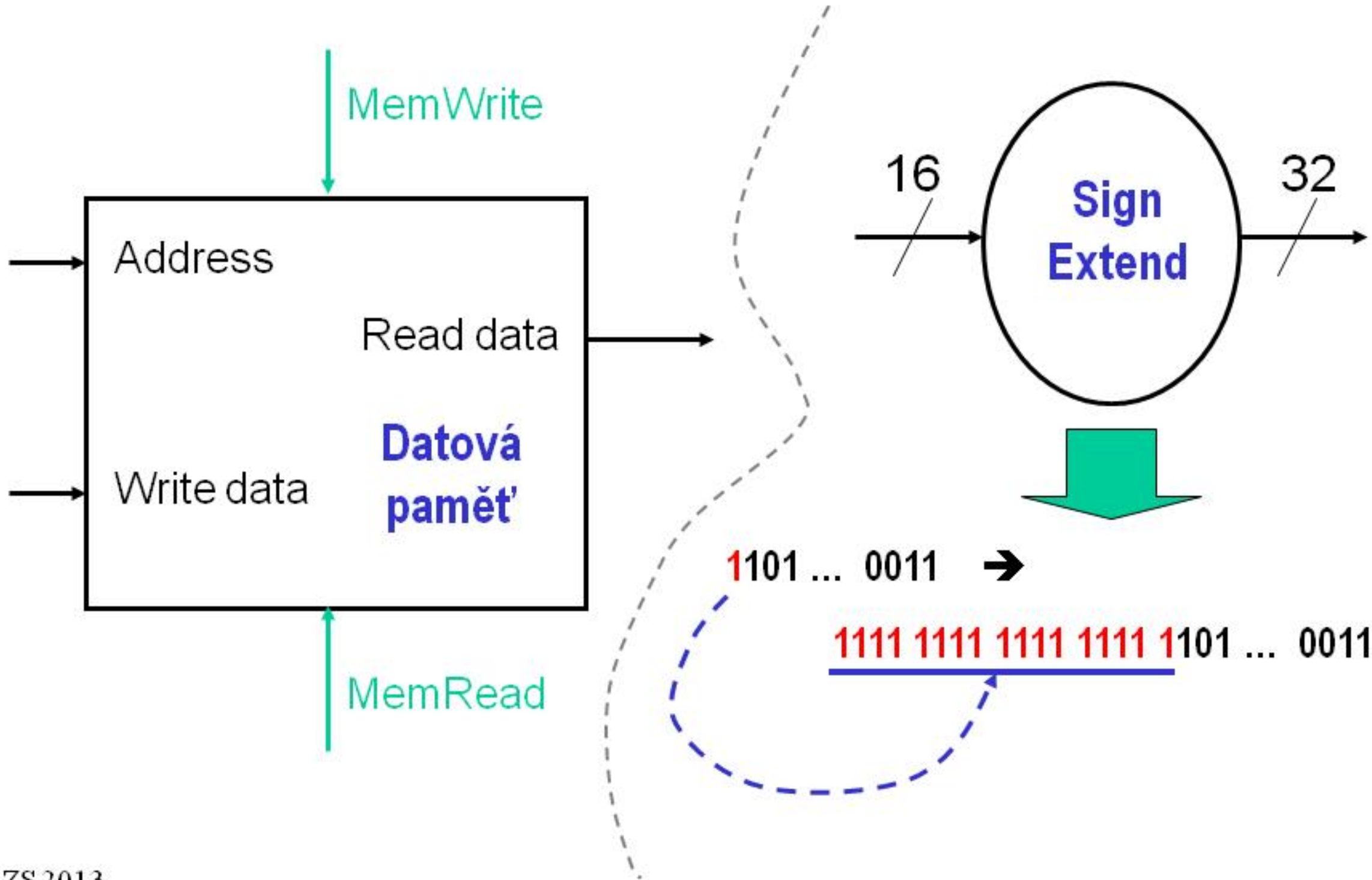
- **Formát:** opcode r1, r2, r3



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

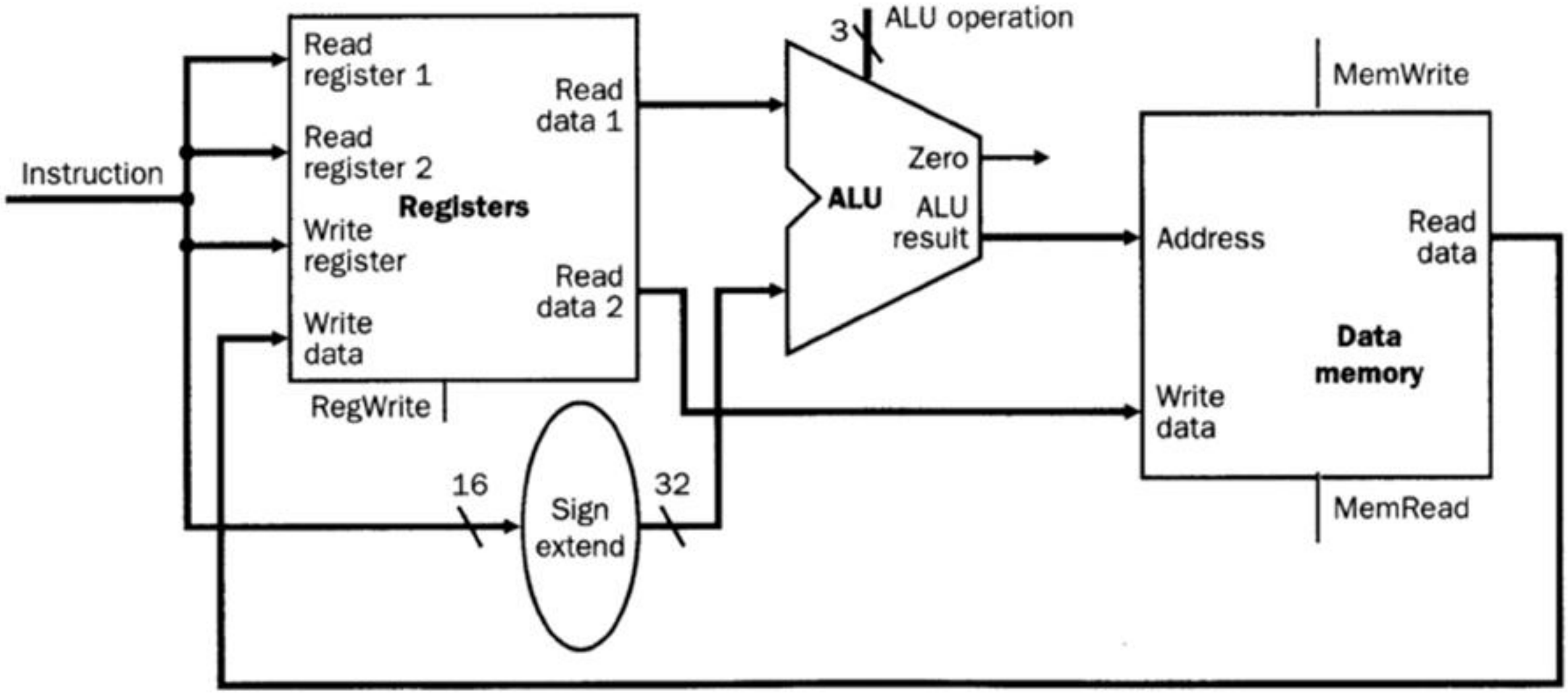
Prvky datové cesty pro Load/Store



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



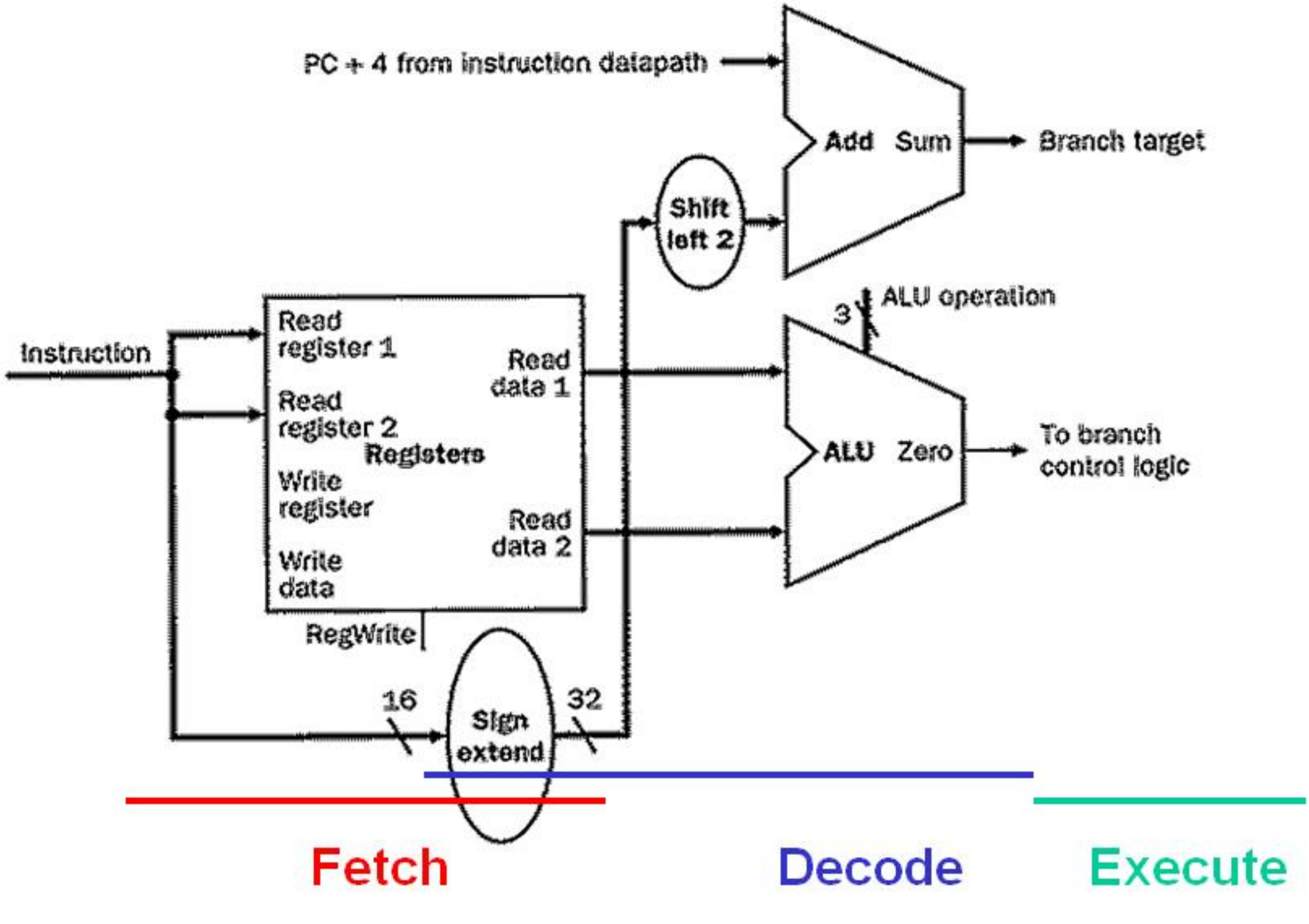
Datové cesty - instrukce větvení (Branch)

- **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 (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)

- **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*

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

Úvod do organizace počítače

Výpočetní struktura pro CPI = 1

Opakování

- **Konstrukce datových cest**
  - Specifické stavební bloky pro formáty (R, I a J)
  - Modulární návrh
    - ALU, registrový soubor, datová paměť
    - ALU nebo sčítačka pro výpočet adresy cíle skoku (BTA)
  - Datové cesty mezi výkonnými bloky, vytvořené podle požadavků jednotlivých instrukčních formátů
- **Instrukční formáty a datové cesty**
  - R: operace ALU
  - I: Load/store – vstup/výstup dat do/z registrového souboru nebo paměti
  - I: Branch (podmíněný skok) – vyhodnocení podmínky, výpočet BTA
  - J: Jump (nepodmíněný skok) – výpočet JTA

Přehled přednášky

- Lze postavit procesor tak, aby se operace provedly vždy v jednom taktu ?
  - Všechny instrukce provedeny tak, že CPI = 1
  - Zvýšení výkonu *software*
- Návrh realizovat z jednoduchých komponent
- Návrh provést iterativně (postupně „nabalováním“)
  - R-formát
  - I-formát
  - J-formát
- Problémy zpracování v jednom cyklu

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

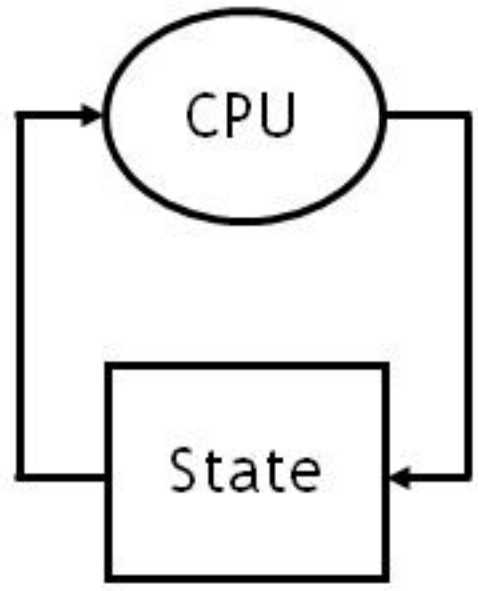
- Popíšeme implementaci instrukčního souboru procesoru na bázi MIPS s následujícími operacemi.

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 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).

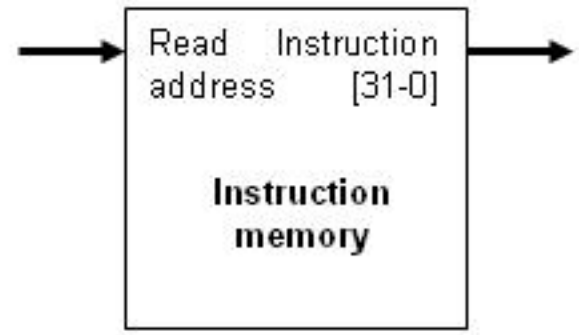


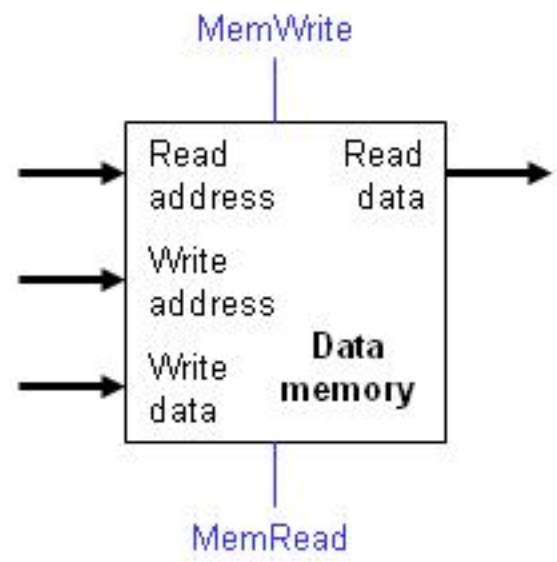
Paměti

- Začneme s jednodušší Harvardskou architekturou, kde data a instrukce leží v *oddělených* pamětech.
- Pro čtení instrukce a čtení & zápis slov, potřebujeme paměti široké 32-bitů (sběrnice reprezentovány tmavou tlustou čarou).

- Modré linky reprezentují řídící signály. MemRead (MemWrite) se nastaví na 1 jestliže se provádí čtení (zápis) z (do) datové paměti (data memory). V opačném případě 0.

- Pro začátek se nebudeme zabývat zápisem do instrukční paměti (instruction memory).
  - Nechť instrukční paměť už obsahuje program a ten se během zpracování nemění.



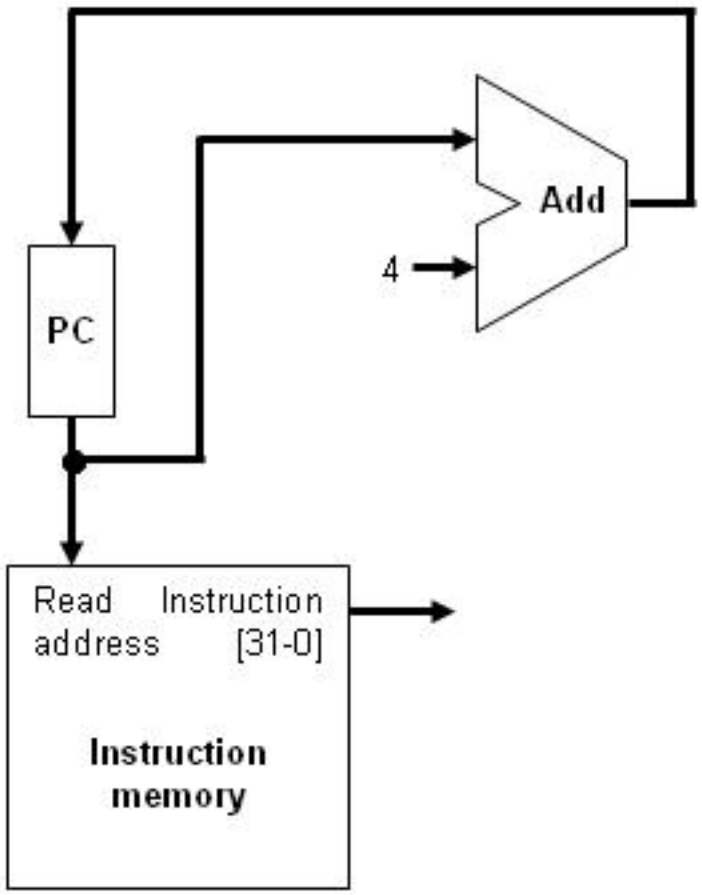


Čtení instrukce

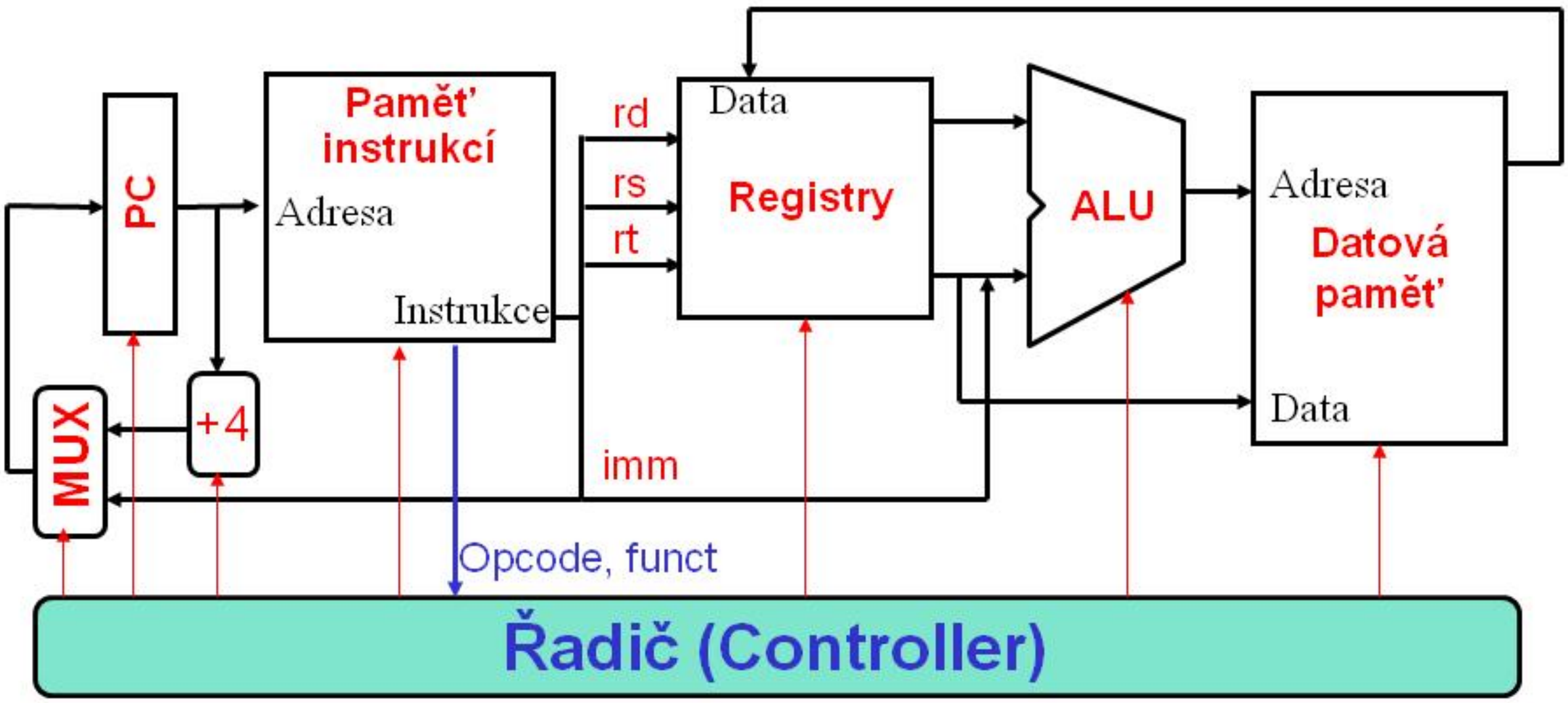
- CPU pracuje v nekonečné smyčce – čte instrukce z paměti a provádí je.

- 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.



Přehled implementace



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

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

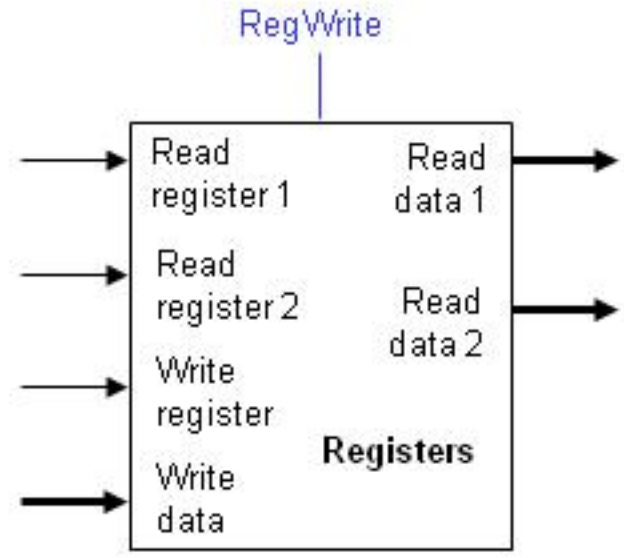
- Příklad instrukce a jejího dekódování:

add $s4, $t1, $t2

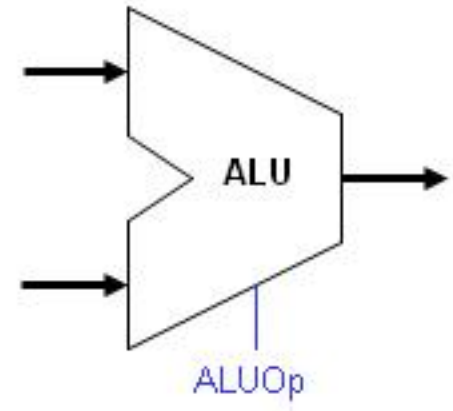
000000	01001	01010	10100	00000	1000000

Registry a ALU

- Instrukce typu R 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 (řídící signály) ALUOp.

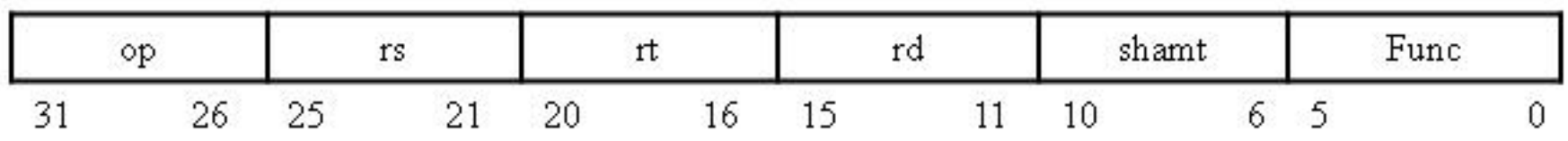


ALUOp	Funkce
000	and
001	or
010	add
110	subtract
111	slt



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.



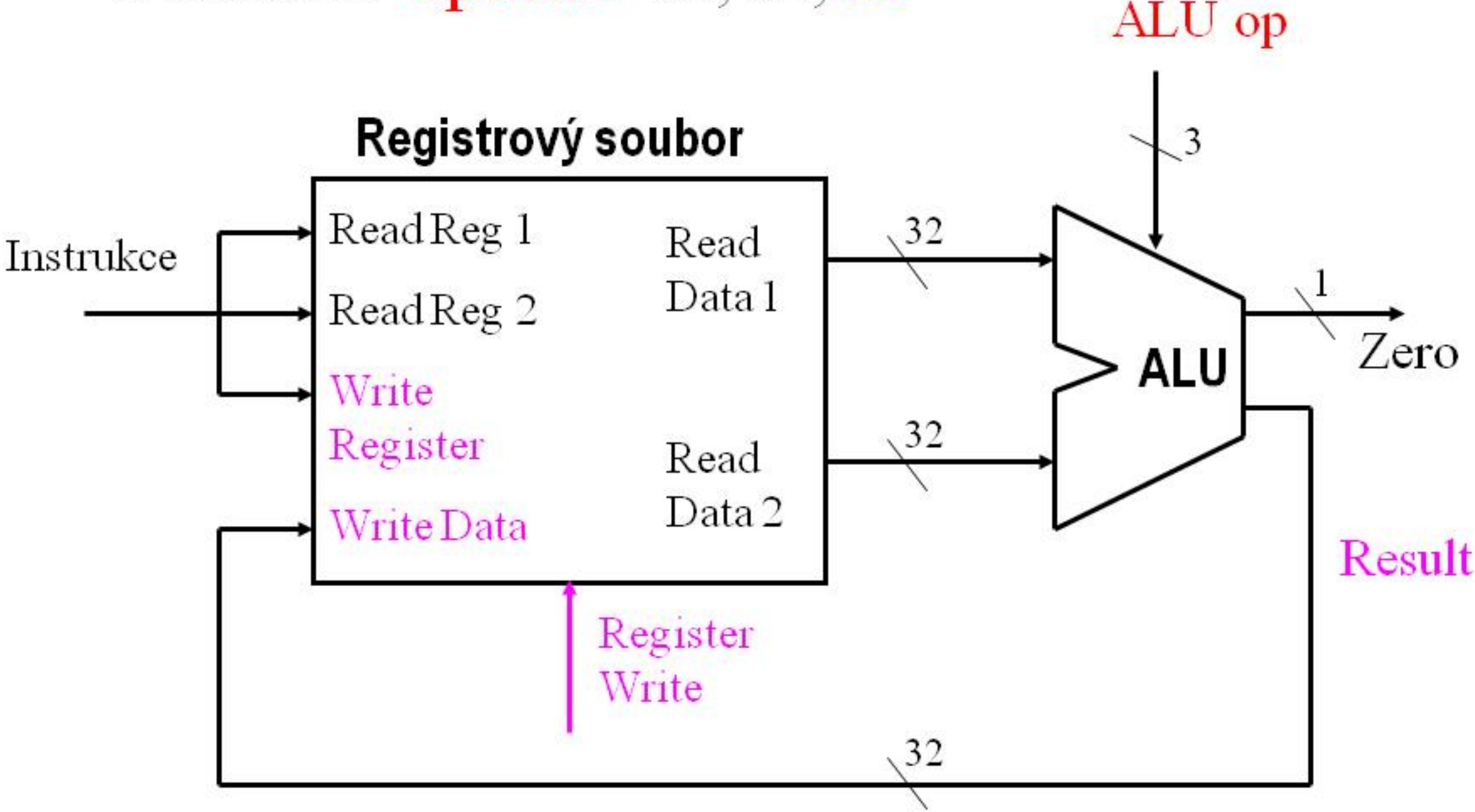
R-formát - provedení

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).

Komponenty: R-formát

- **Formát:** opcode r1, r2, r3



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:

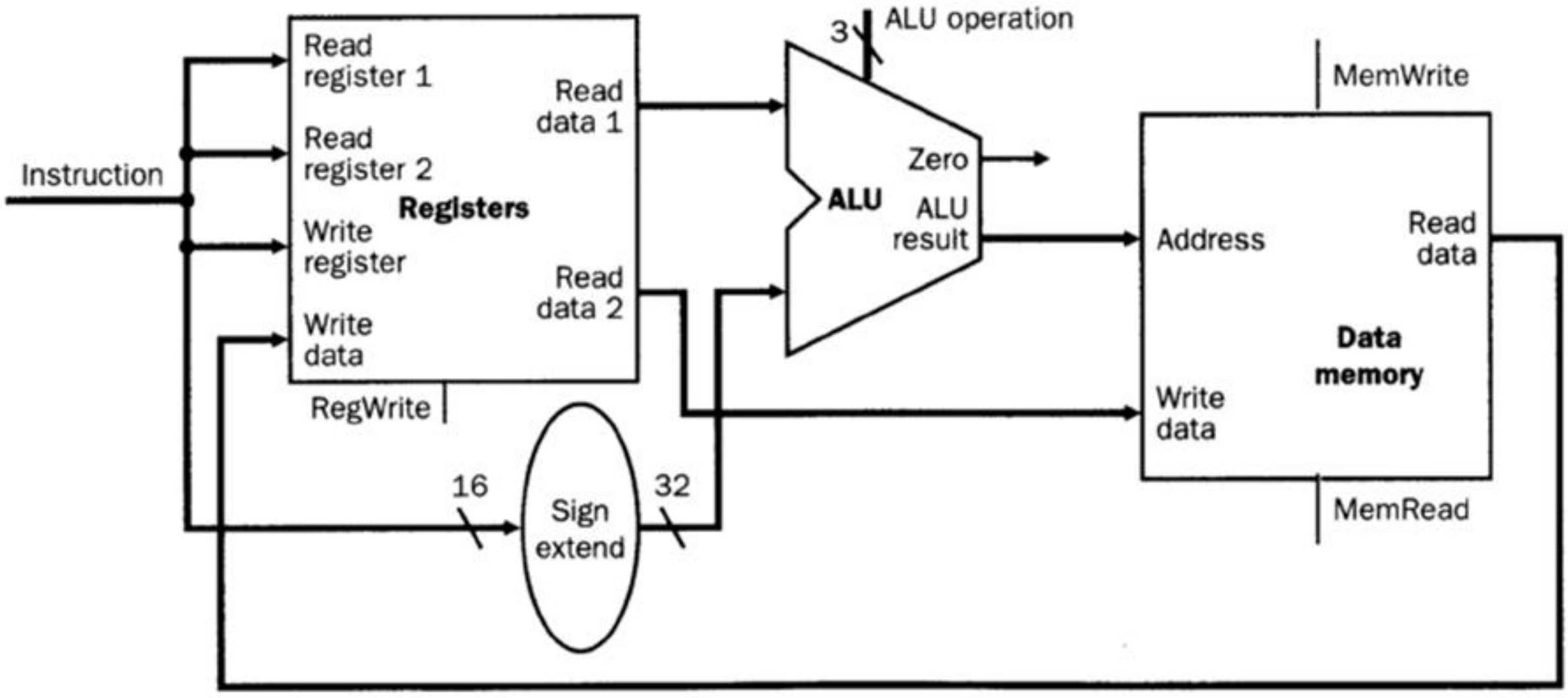
lw	$t0, -4($sp)	100011	11101	01000	1111 1111 1111 1100
sw	$a0, 16($sp)	101011	11101	00100	0000 0000 0001 0000

Load/Store - provedení

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

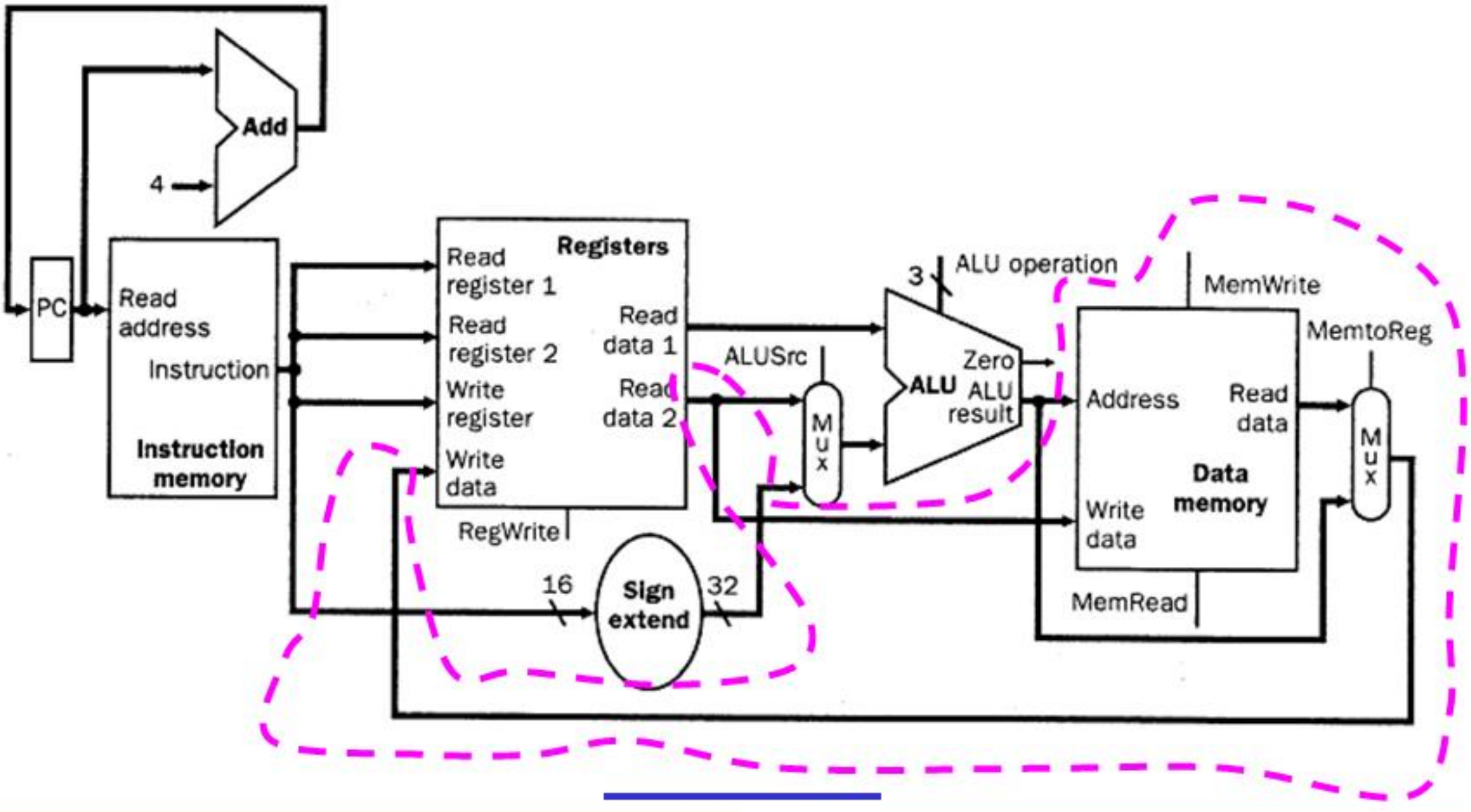
1. Načtení instrukce a inkrement PC
2. Čtení obsahu registru (např., bázová adresa v $t2) z registrového souboru
3. ALU přičte hodnotu z $t2 ke (znaménkem rozšířeným) dolním 16 bitům instrukce (`offset`)
4. Výsledek z ALU = adresa do datové paměti
5. Čtení dat z paměti, zápis do registrové sady, podle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

Komponenty: Load/Store



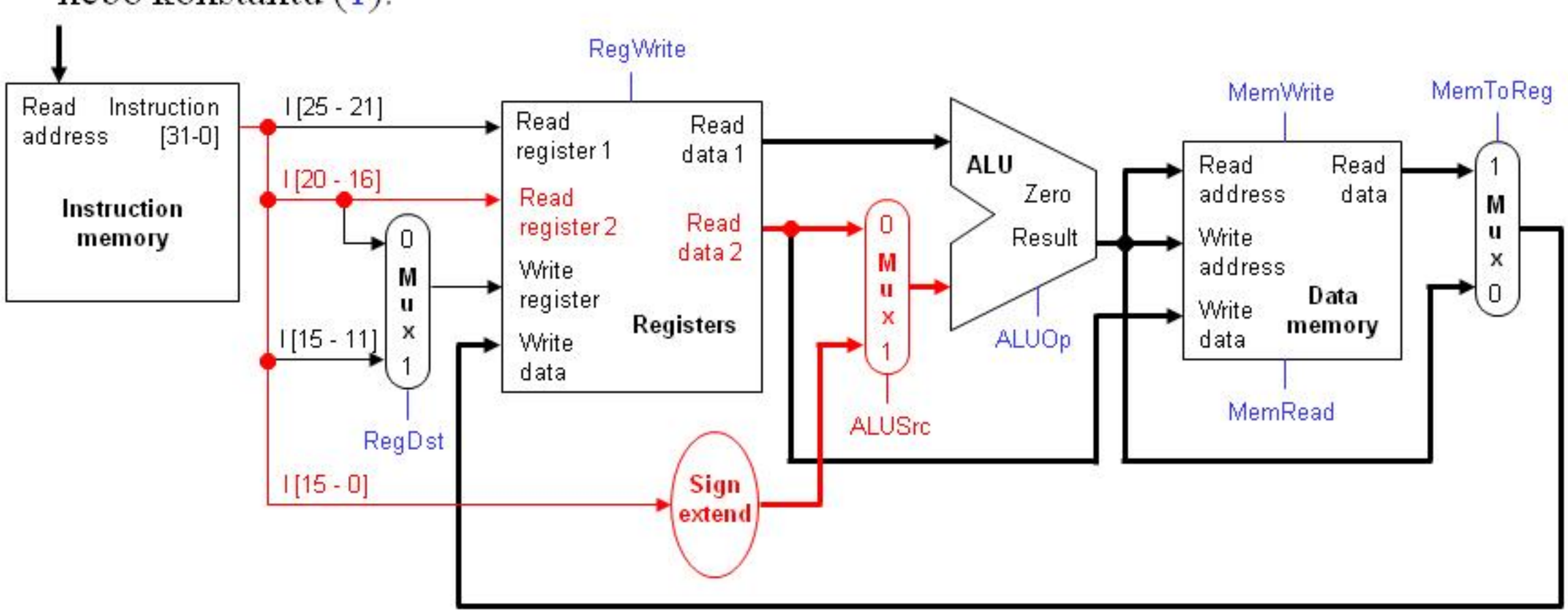
Fetch Decode Execute

R-formát + *Load/Store HW*



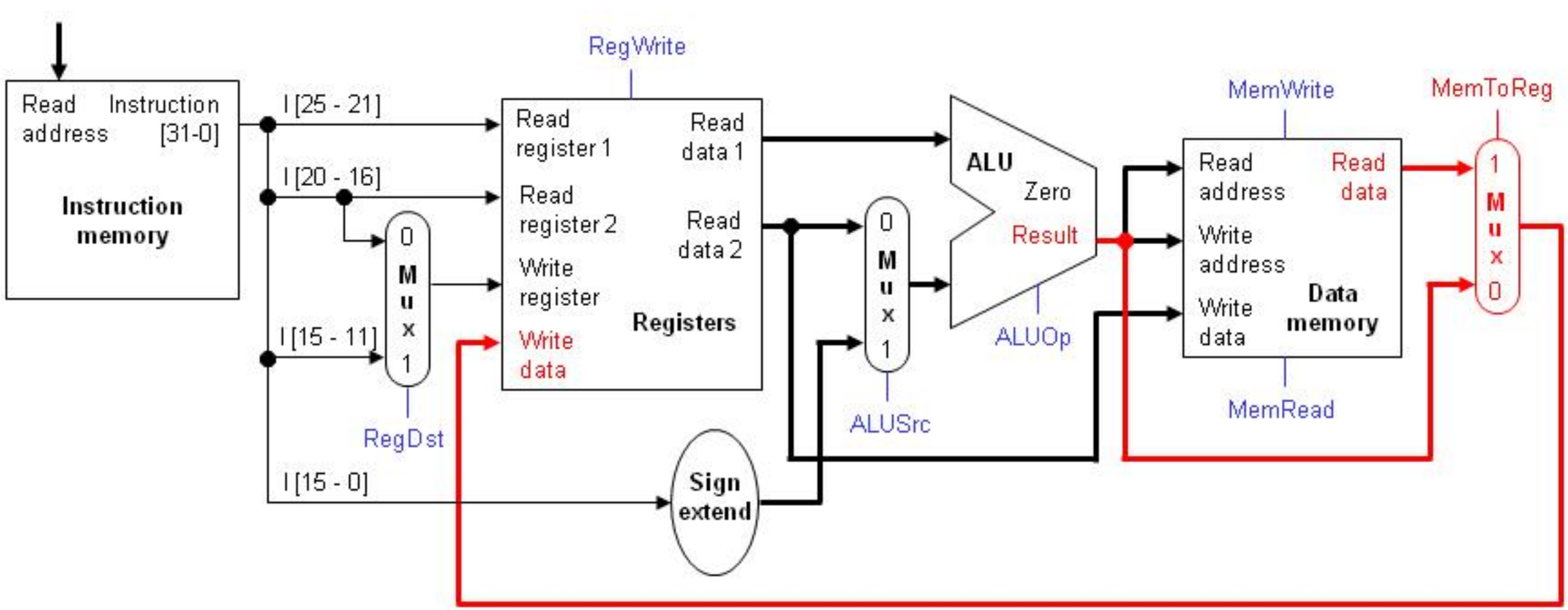
Přístup do datové paměti

- 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 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).



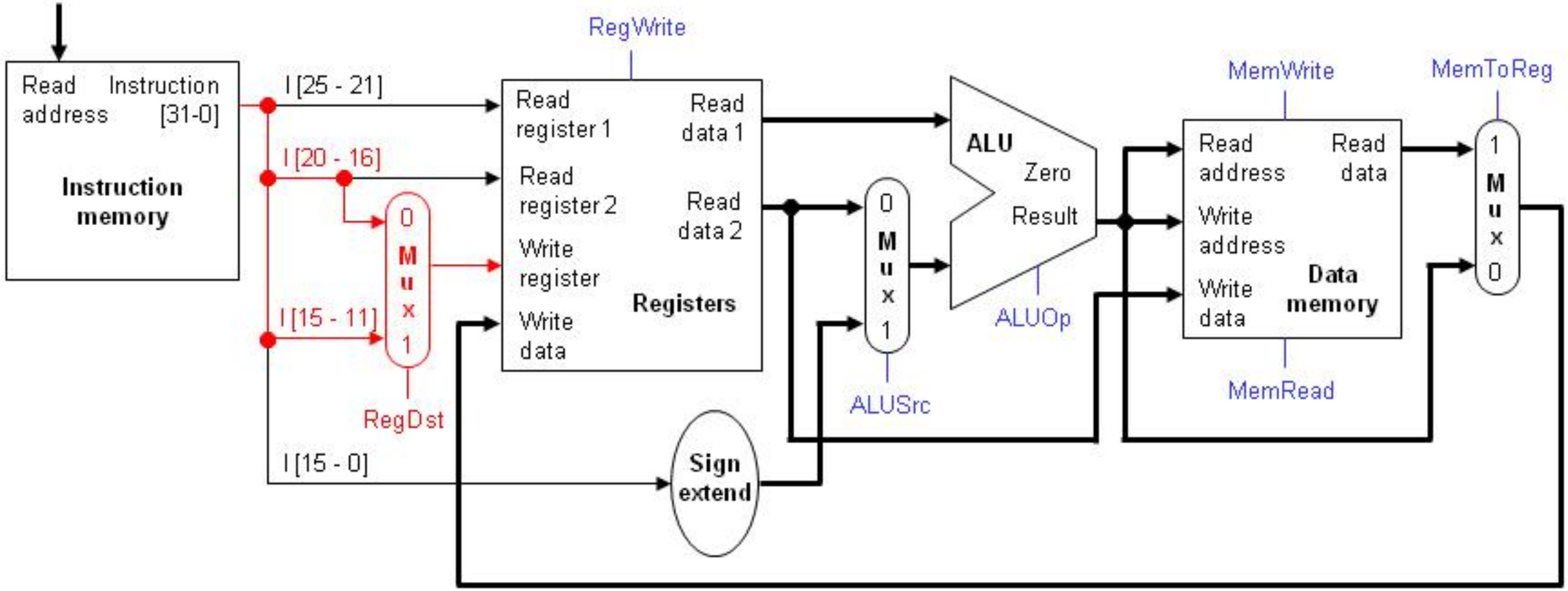
Cílový registr

- 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

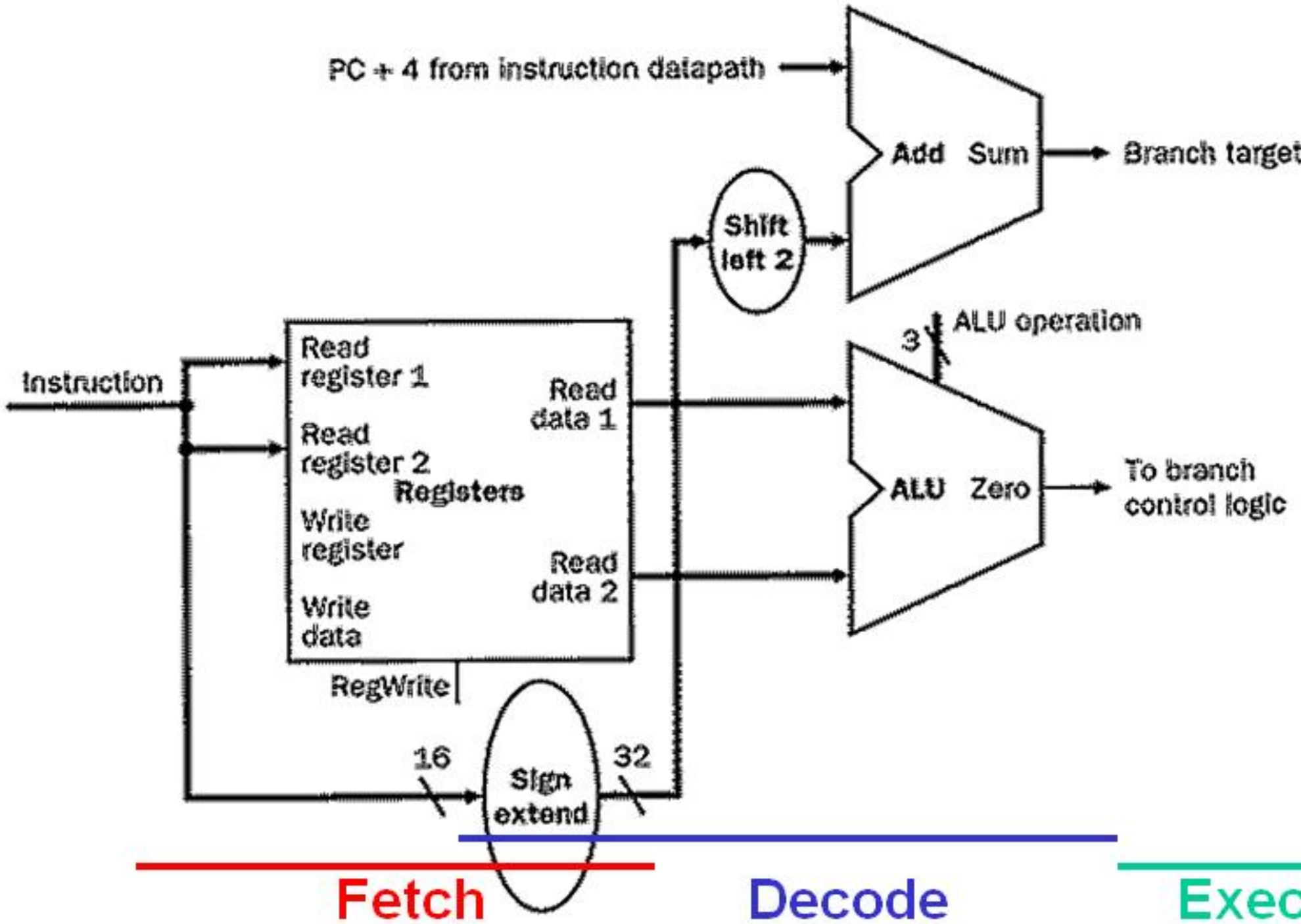
- Pro instrukce větvení nepředstavuje konstanta adresu, ale *offset* registru PC od cílové adresy.

```
        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.

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

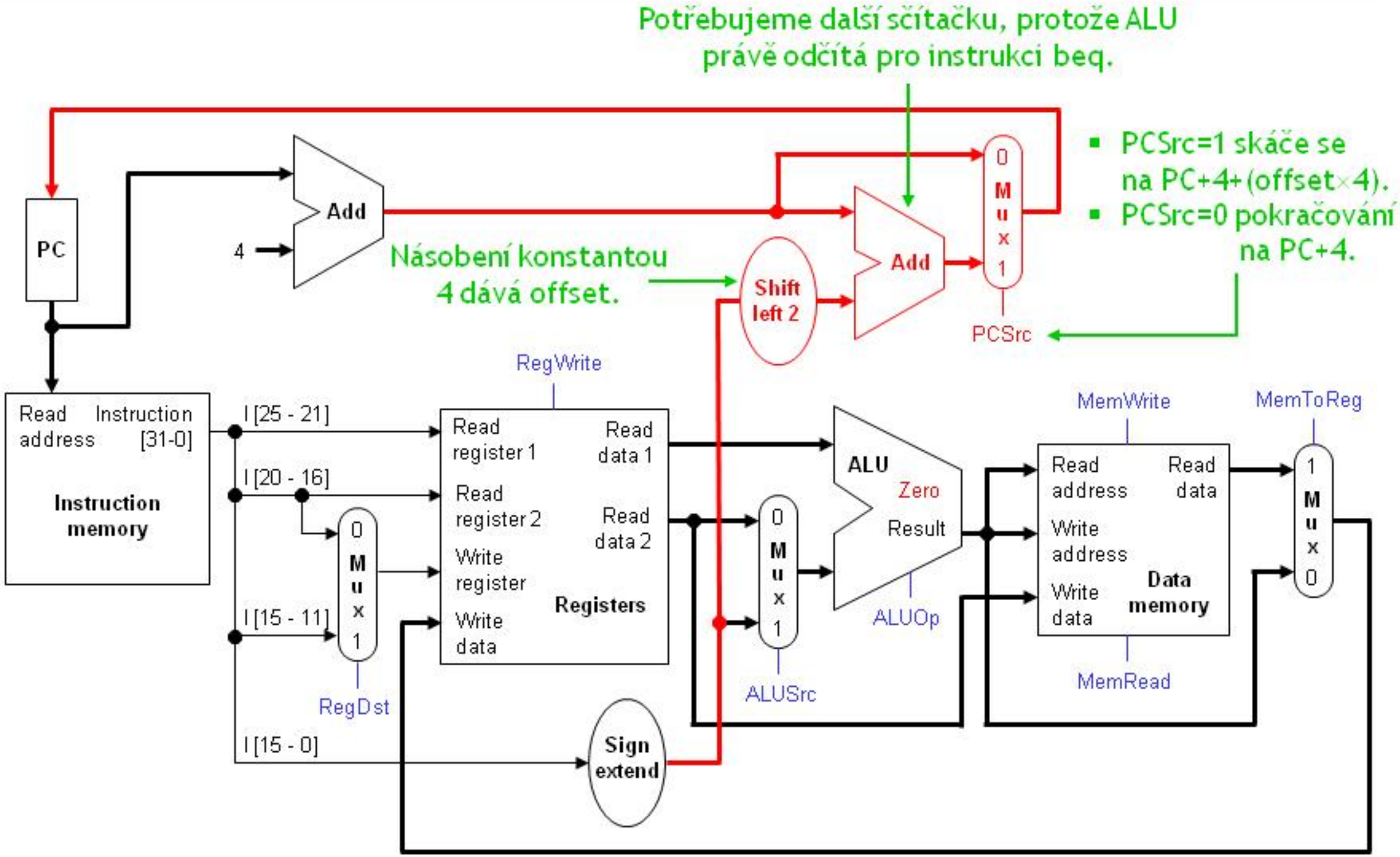
Komponenty: instrukce větvení



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



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

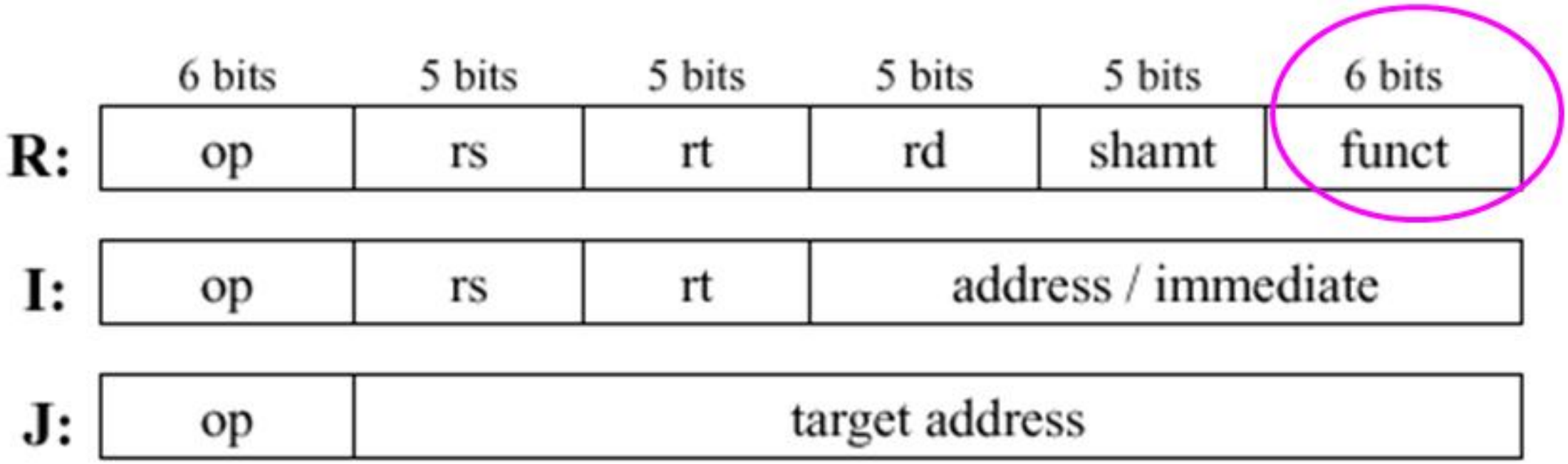
ALU Control Input	Function
000	and
001	or
010	add
110	sub
111	slt

Řídící bity ALU

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

Opakování: Instrukční formáty MIPS



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

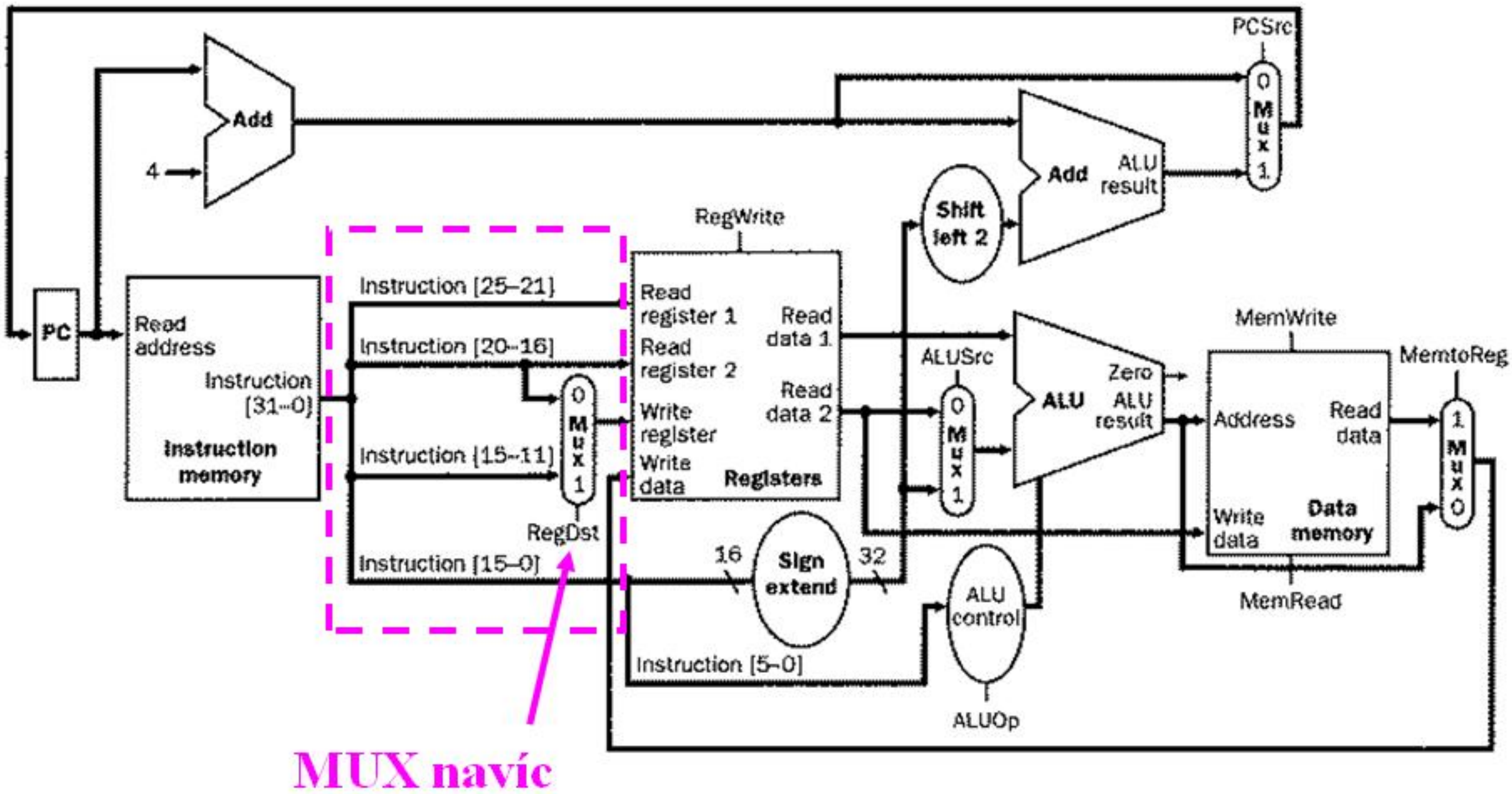
Všimněte si, že vždy platí:

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

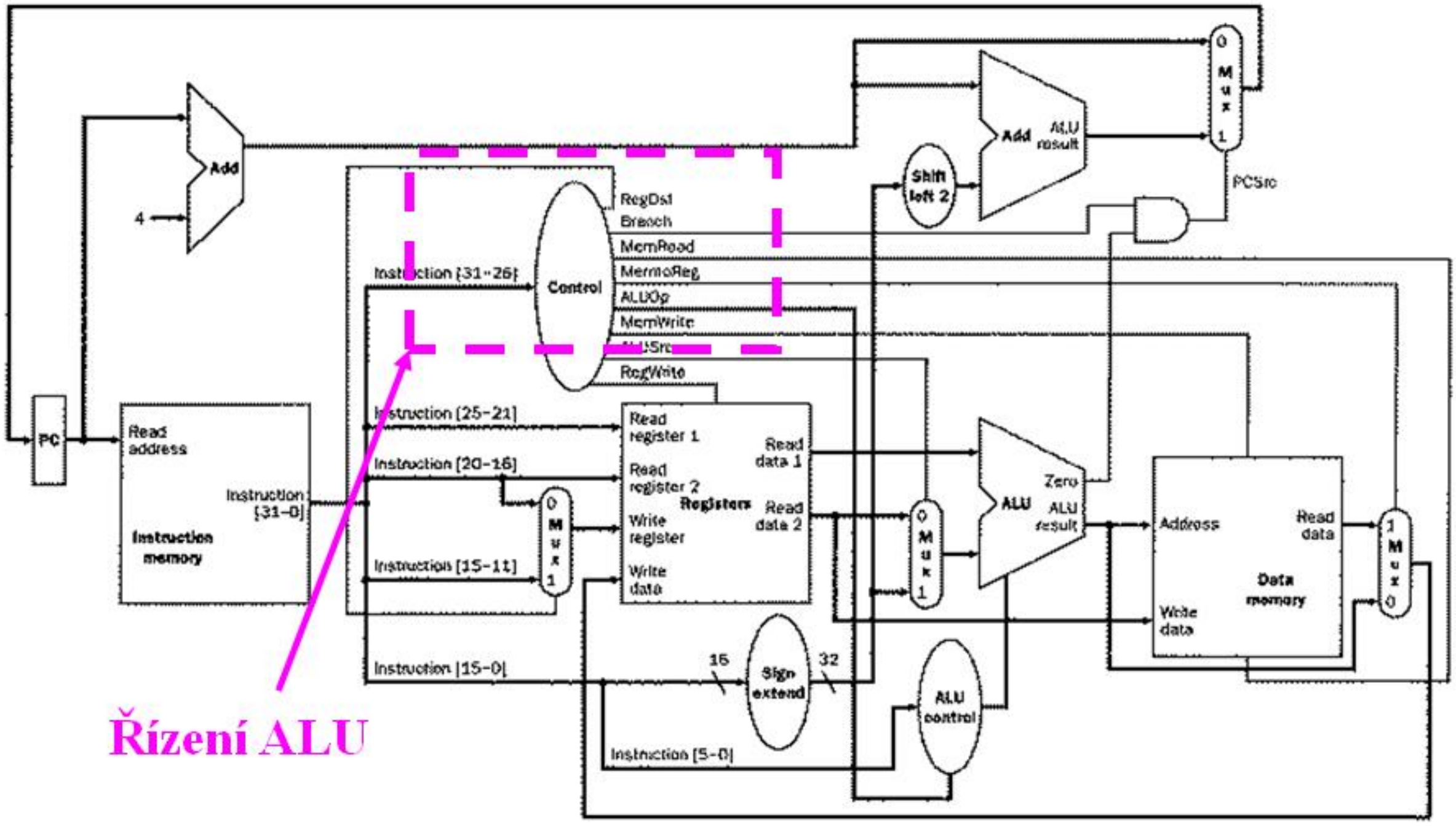
Dále podle typu instrukce platí:

- **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 – řídící signály

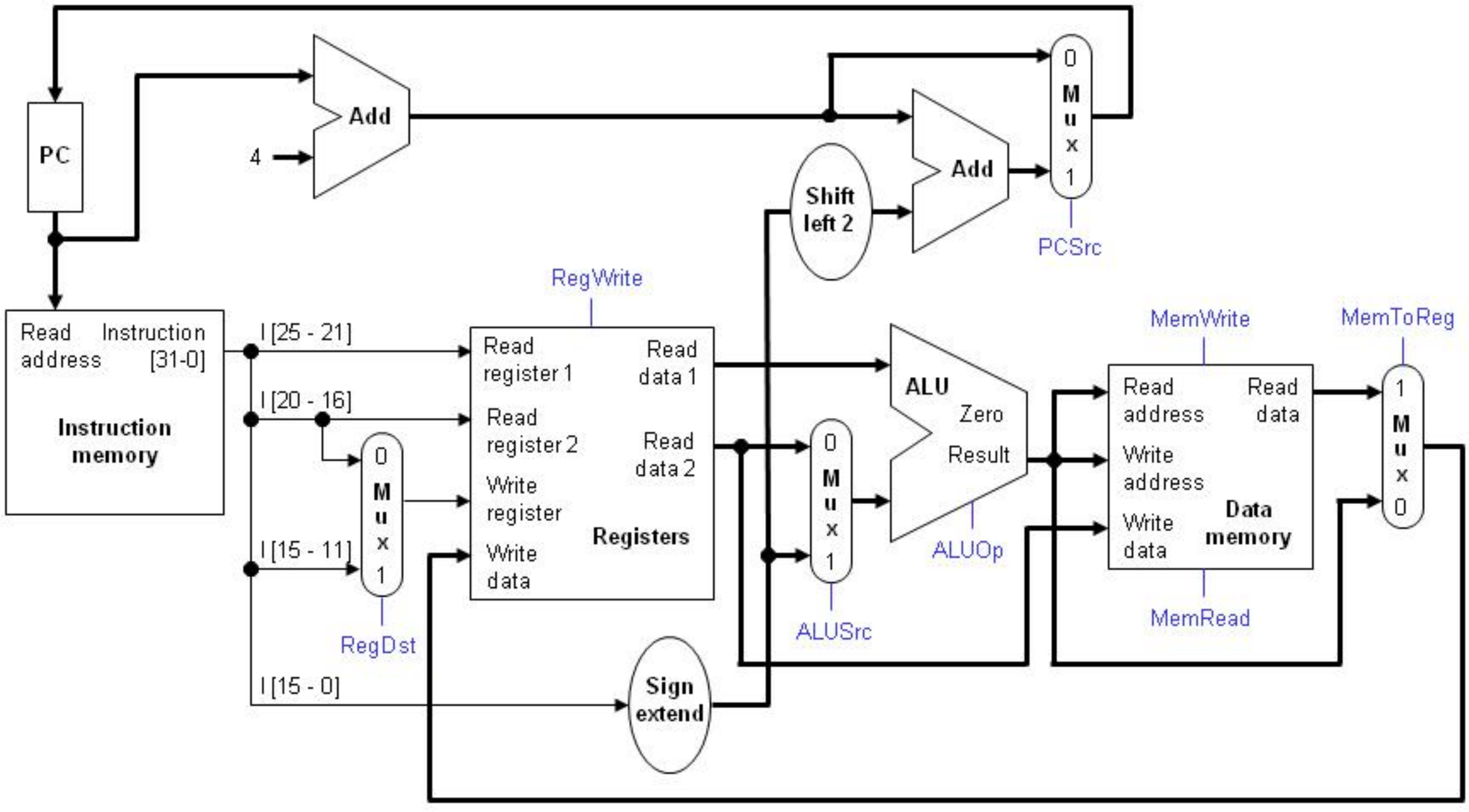


Řízení ALU

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

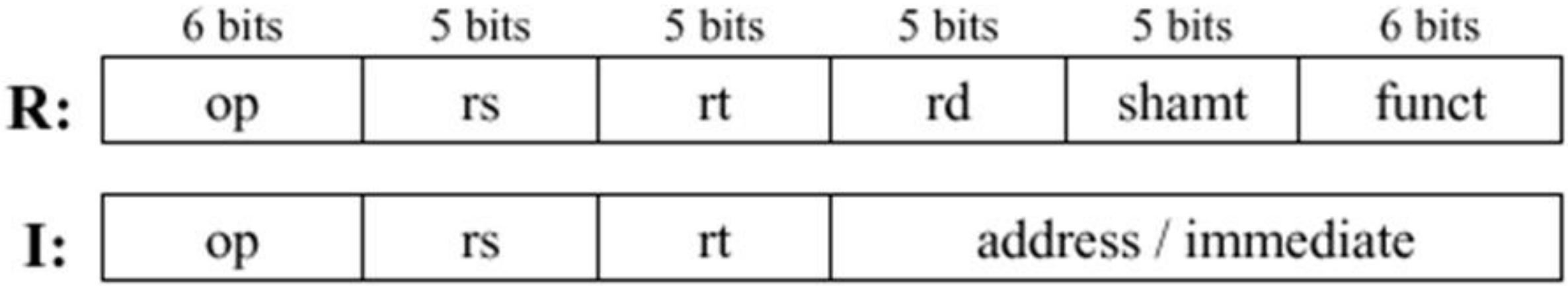


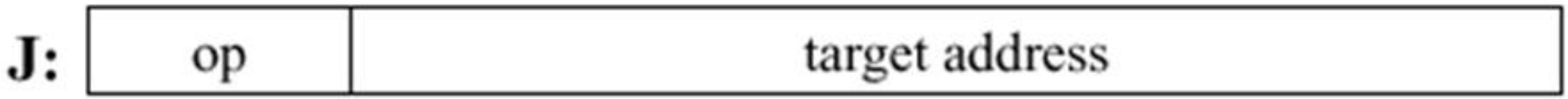
Rozšíření: instrukce Jump

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

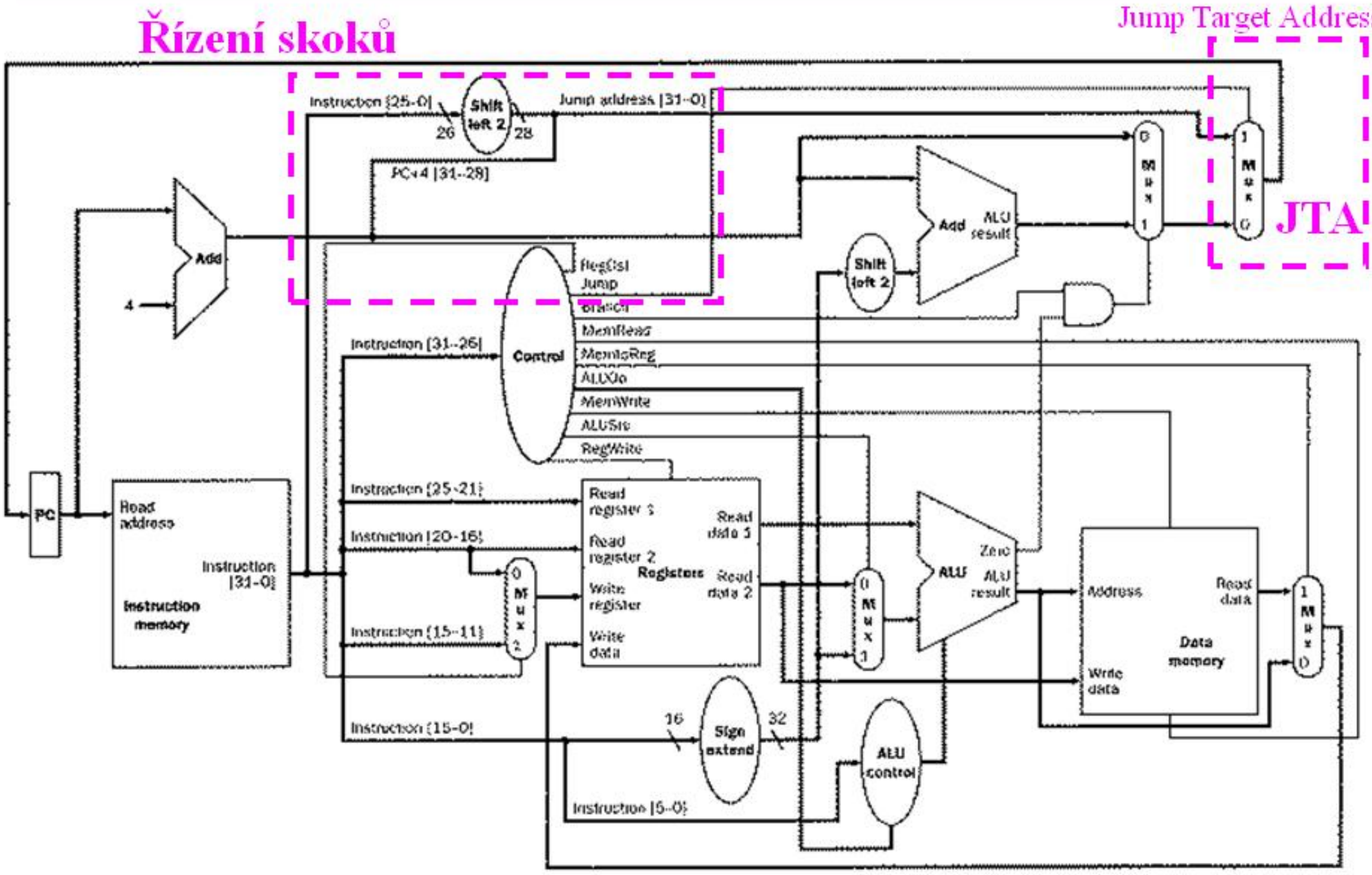
Rozšíření: instrukce Jump





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

Rozšíření: instrukce Jump



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 jednocyklové implementace

- 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 řídících signálů, které ovládají datové cesty.
  - Řídící 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

- 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 jednotky

Přehled – vícecyklové zpracování

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

1. Načtení instrukce
2. Dekódování instrukce / Načtení dat
3. Operace ALU / provedení instrukcí R-formátu
4. Dokončení provedení R-formátu
5. Provedení paměťového cyklu

Používají všechny instrukce

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

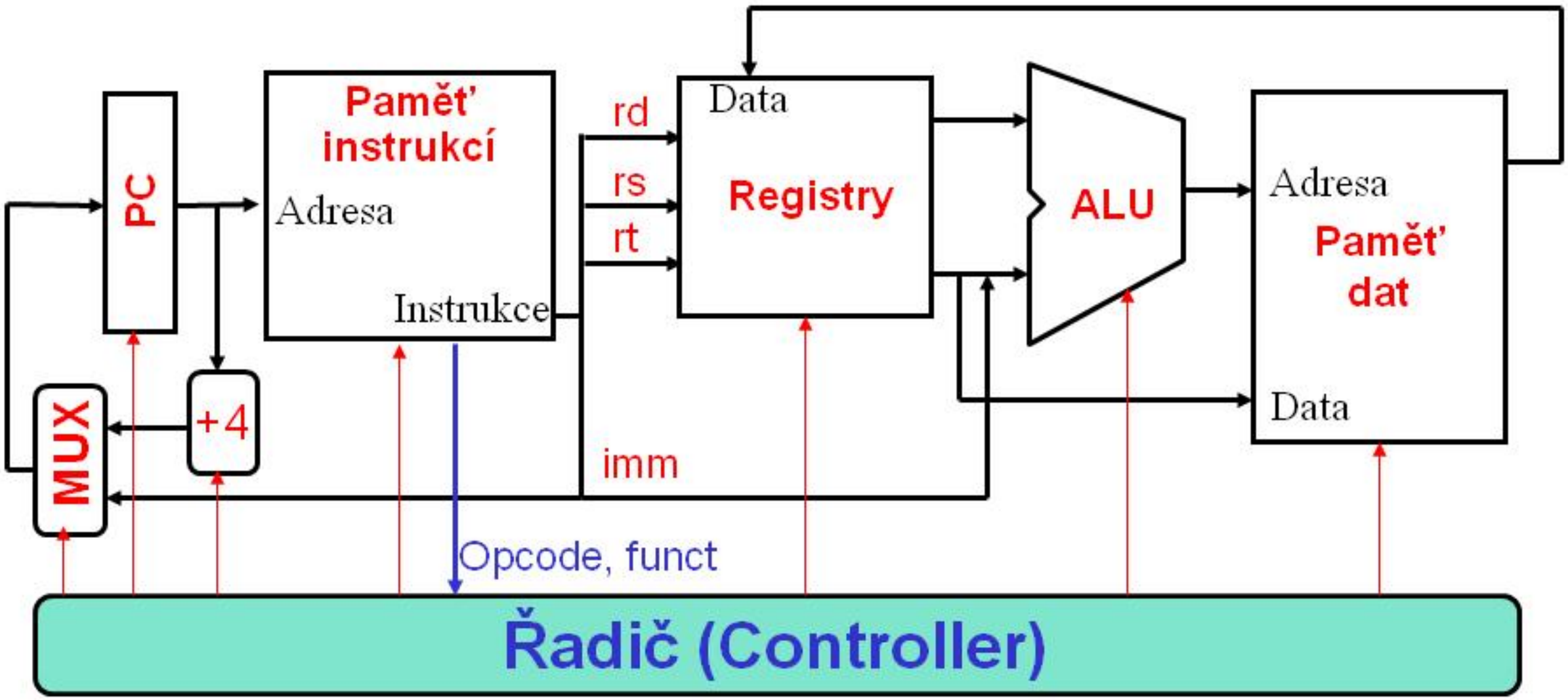
☺ *Efektivnější využití hardware a času*

>> Nový hardware vyžaduje registrový výstup

>> Nové multiplexery (MUX) pro opětné využití hardware

>> Rozšíření řídící jednotky (řadiče) pro nový hardware

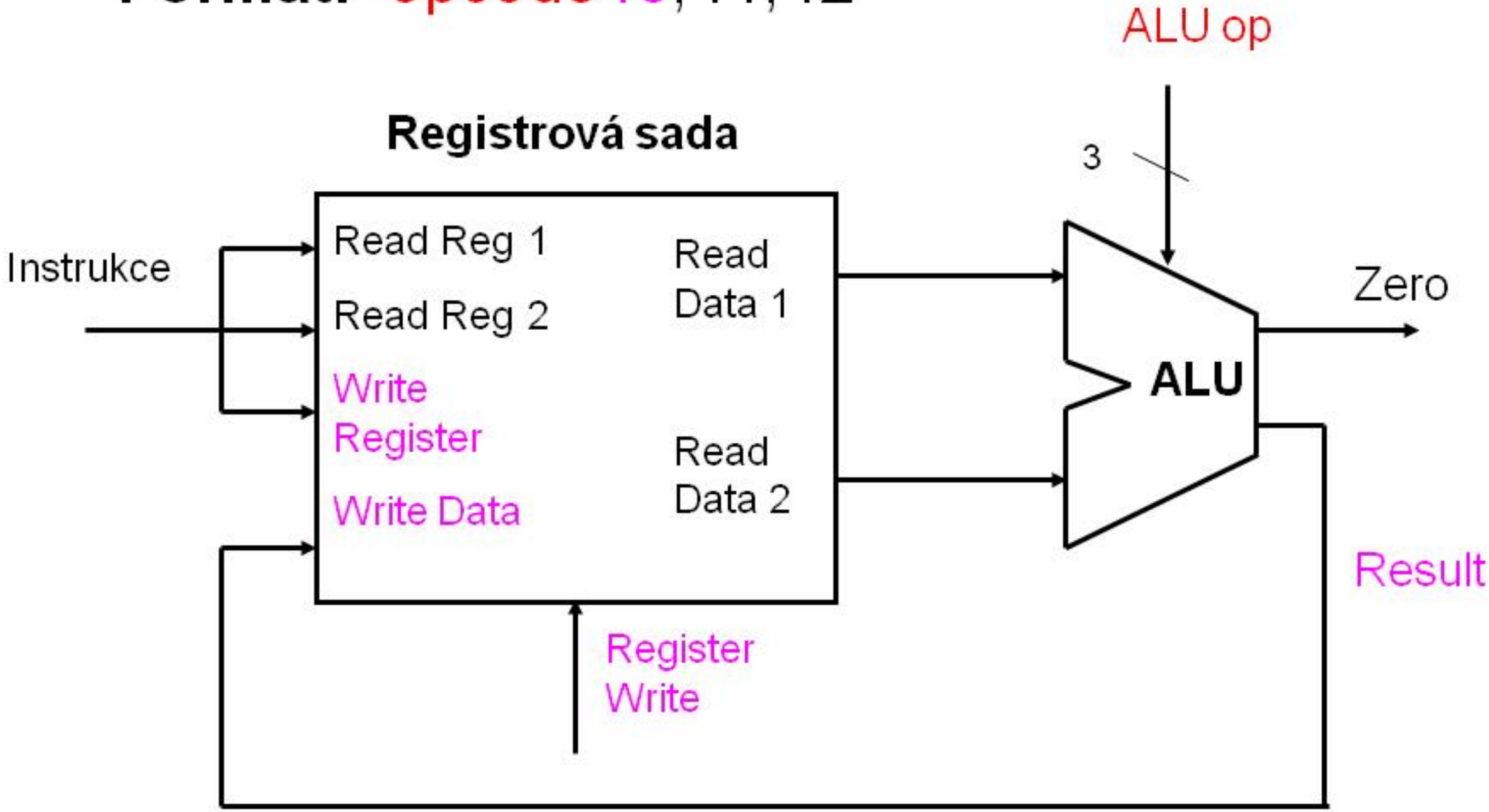
Opakování: Jednoduché datové cesty



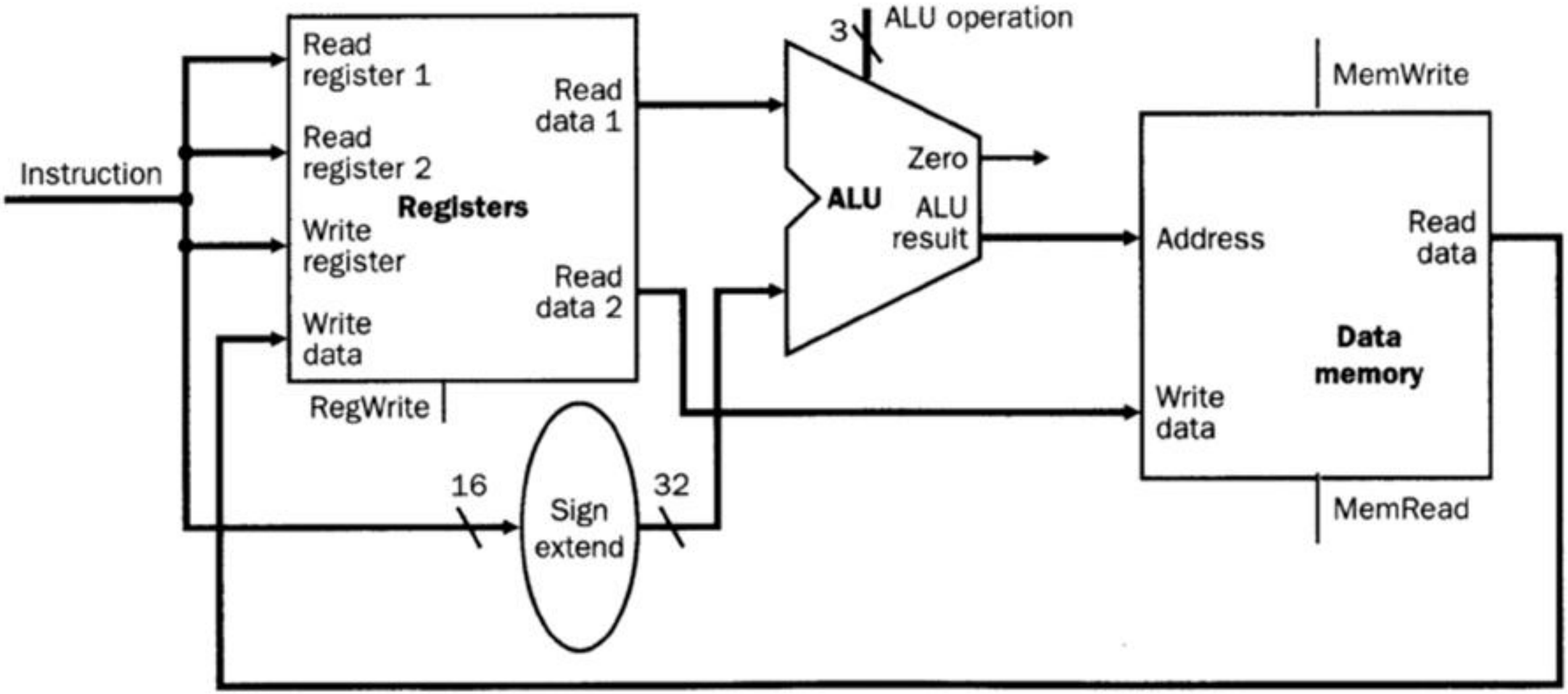
- Datové cesty umožňují přesuny obsahu registrů při provádění instrukcí
- Řízení zajišťuje provedení správných přesunů

Opakování: R-formát - cesty

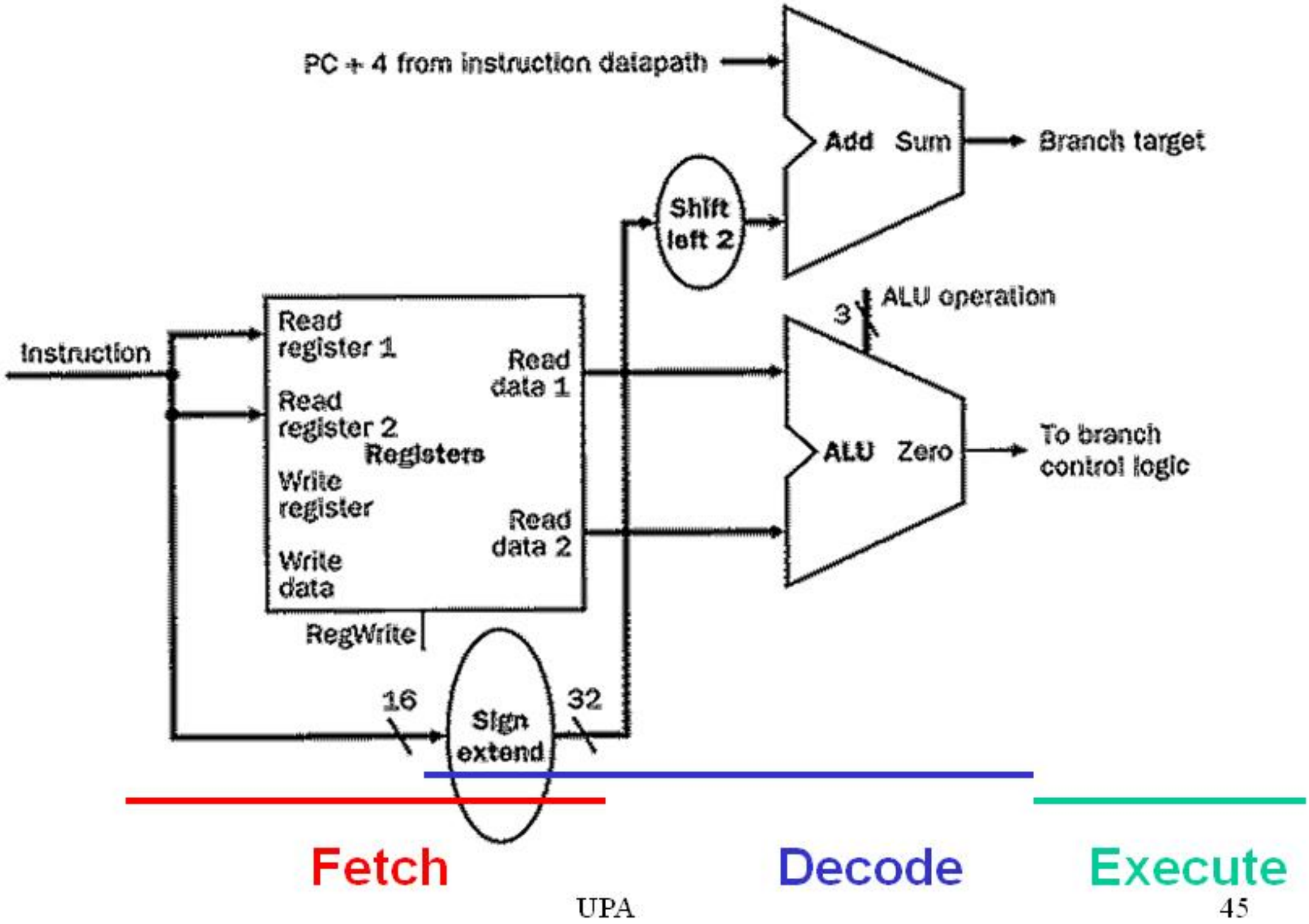
- **Formát:** opcode r3, r1, r2



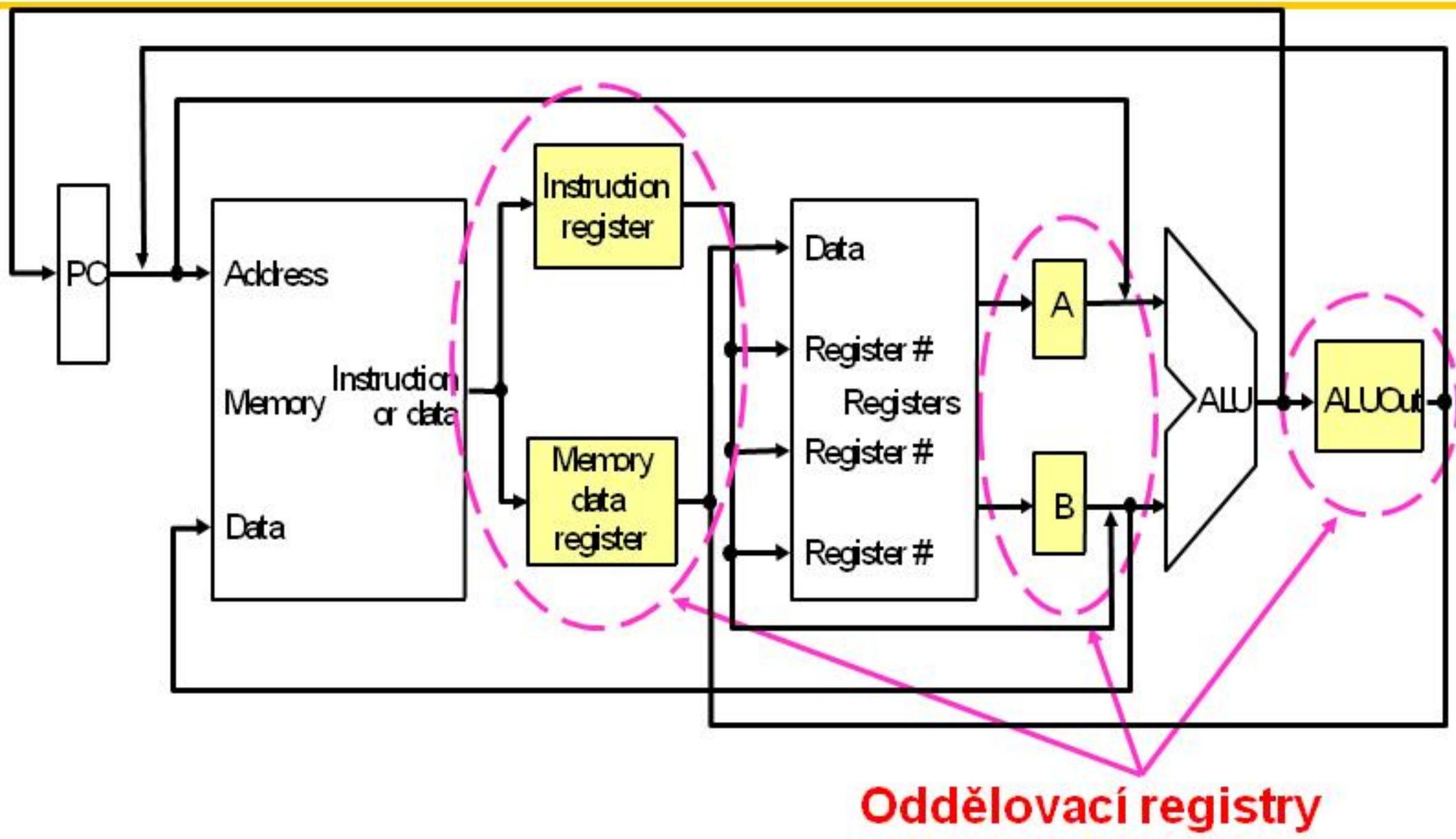
Opakování: Load/Store -cesty



Opakování: Podmíněný skok - cesty



Princip: Vícecyklové datové cesty

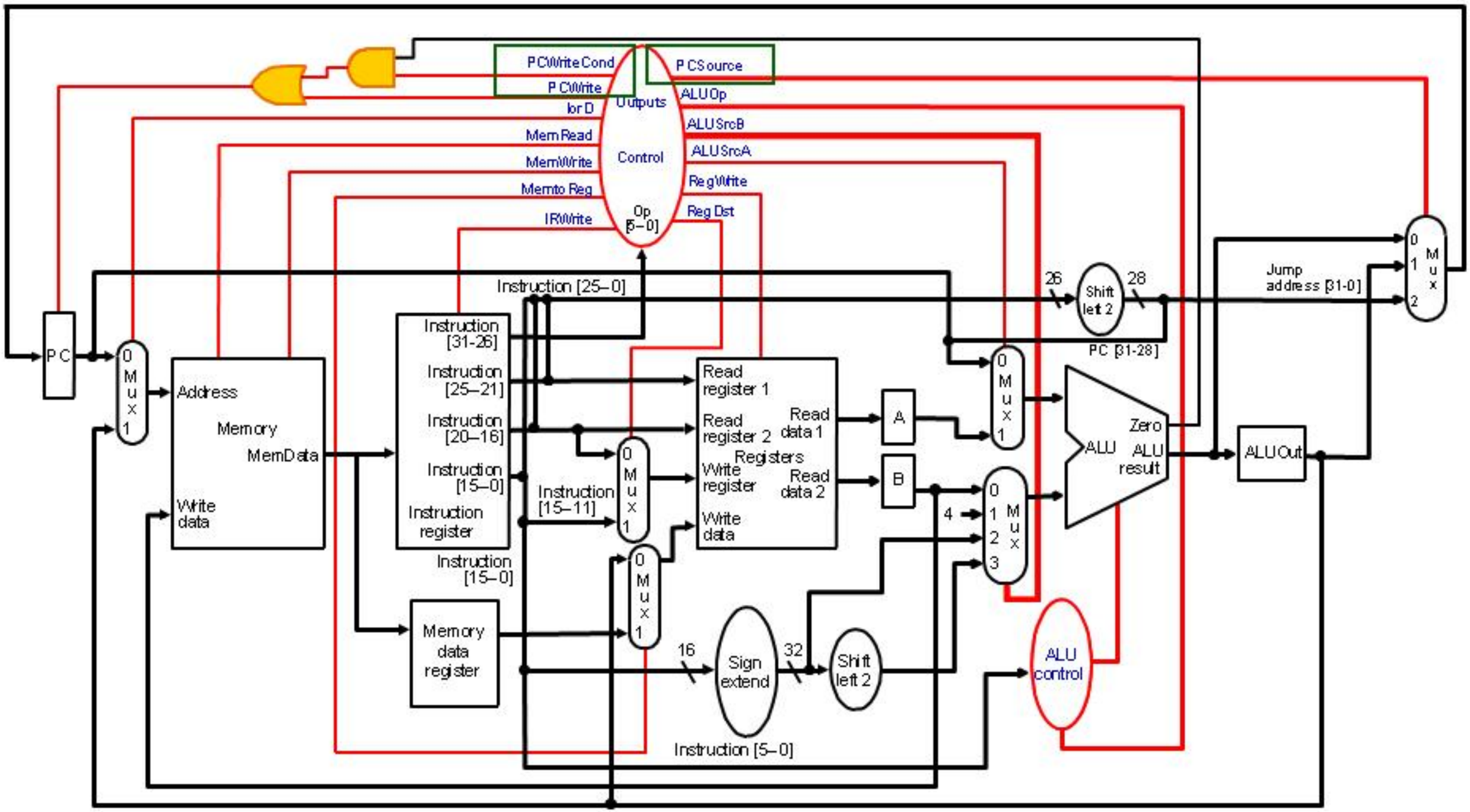


Princip: Multicyklové datové cesty

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 (komplet)



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.

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

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

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 (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 (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: 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: 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

Závěr

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

Cena: Komplikovanější návrh řízení

Úvod do organizace počítače

Řízení multicyklové jednotky

Mikroprogramování a výjimky (přerušení)

Opakování – Multicyklová jednotka

- **Každá instrukce se provádí ve více stupních**
- **Zpracování v jednom stupni trvá jeden cykl**
  1. Načtení instrukce
  2. Dekódování instrukce / načtení dat
  3. Operace ALU provedení R-formátu
  4. Dokončení instrukce typu R
  5. Dokončení přístupu do paměti

Pro všechny instrukce společné

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

☺ *Efektivnější využití hardware a času*

>> Nový hardware + multiplexery pro oddělení stupňů a přepínání datových cest, řízení

>> Navíc řízení nového hardware

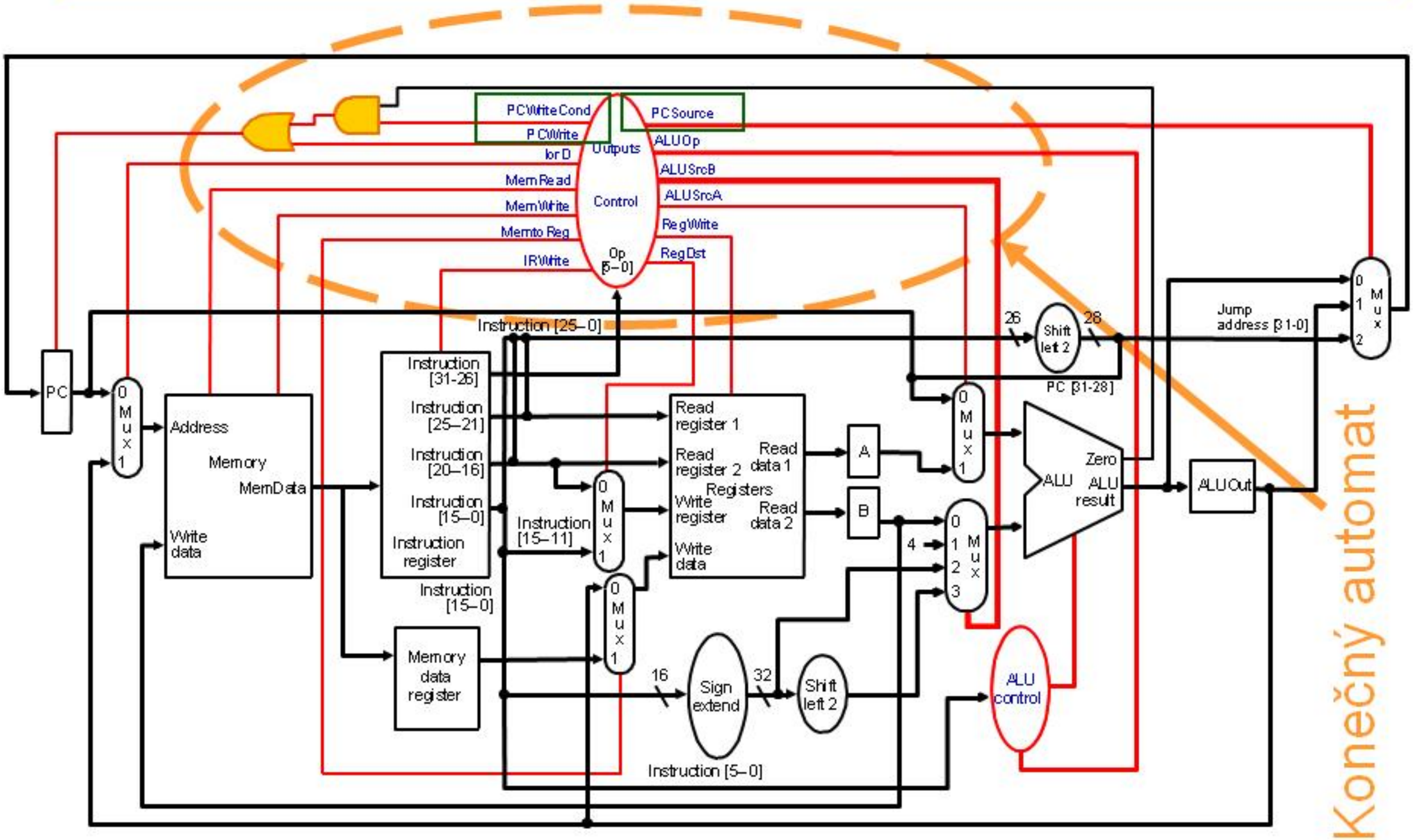
Opakování – Multicyklová jednotka

- 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)

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

Multicyklové jednotky: 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.

Multicyklové jednotky: 2-bitové řídící signály

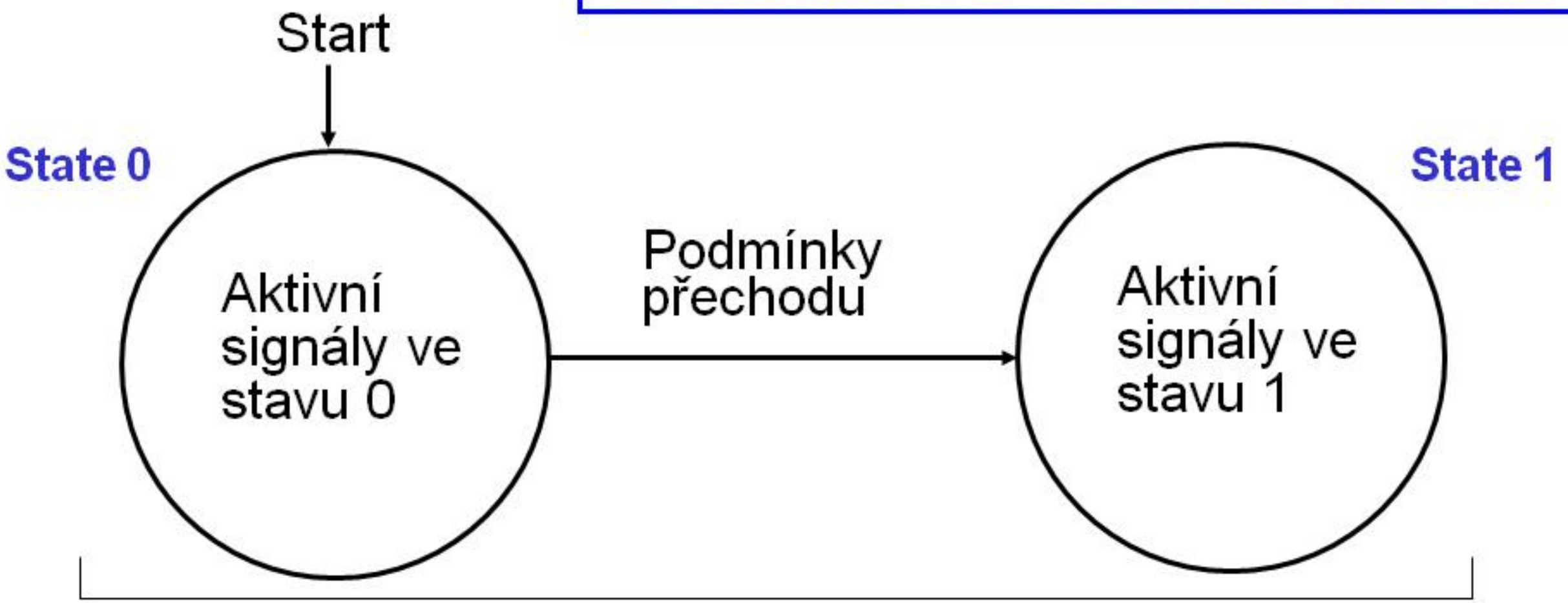
Signal name	Value	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	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.

Základy řízení pomocí FSM

Stav: „Snímek stroje“ (stav paměťových prvků)

Přechod: změna stavu (přechod od jednoho stavu do druhého

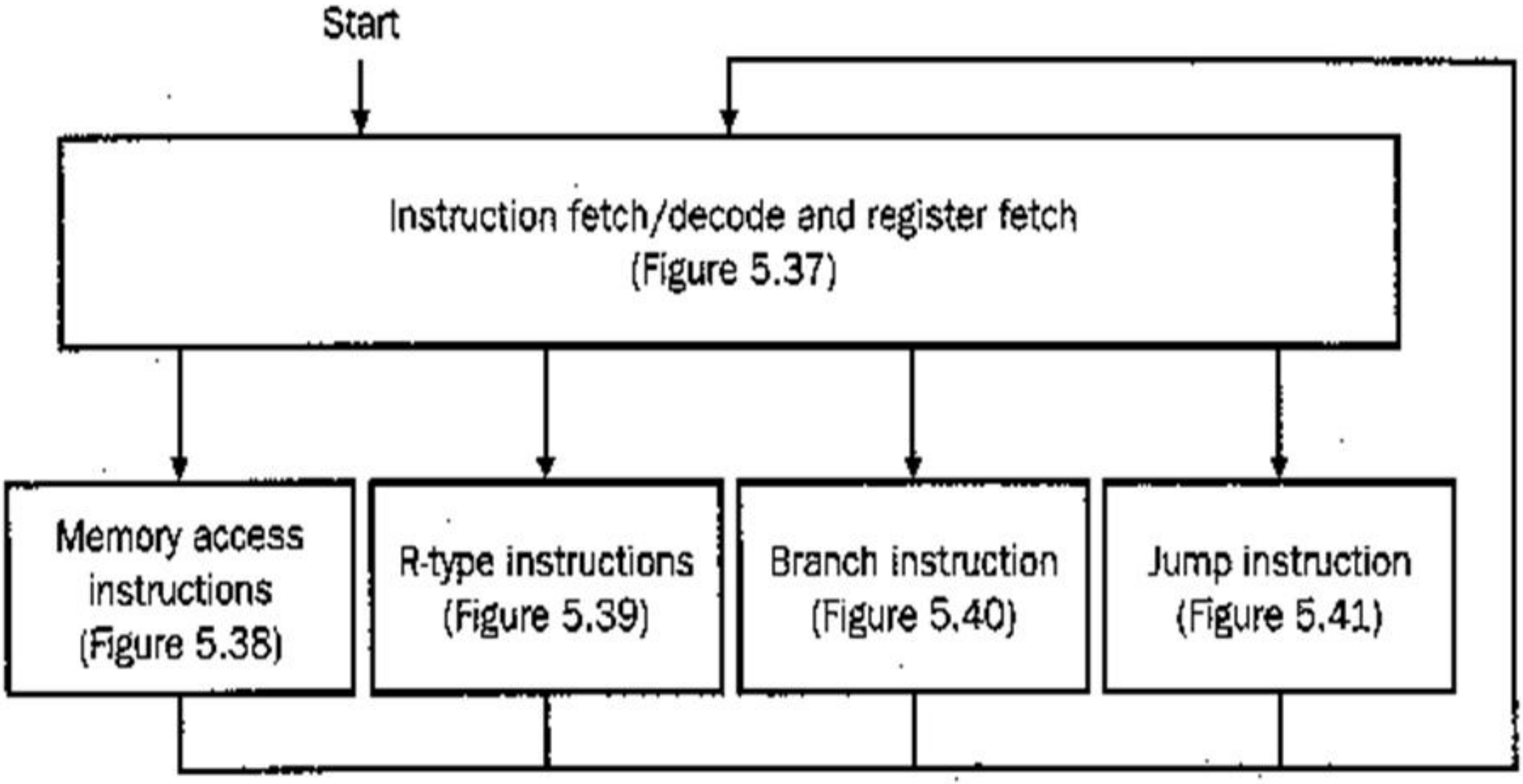
viz kánonický tvar konečného automatu



Finite State Machine – automat s konečným počtem stavů

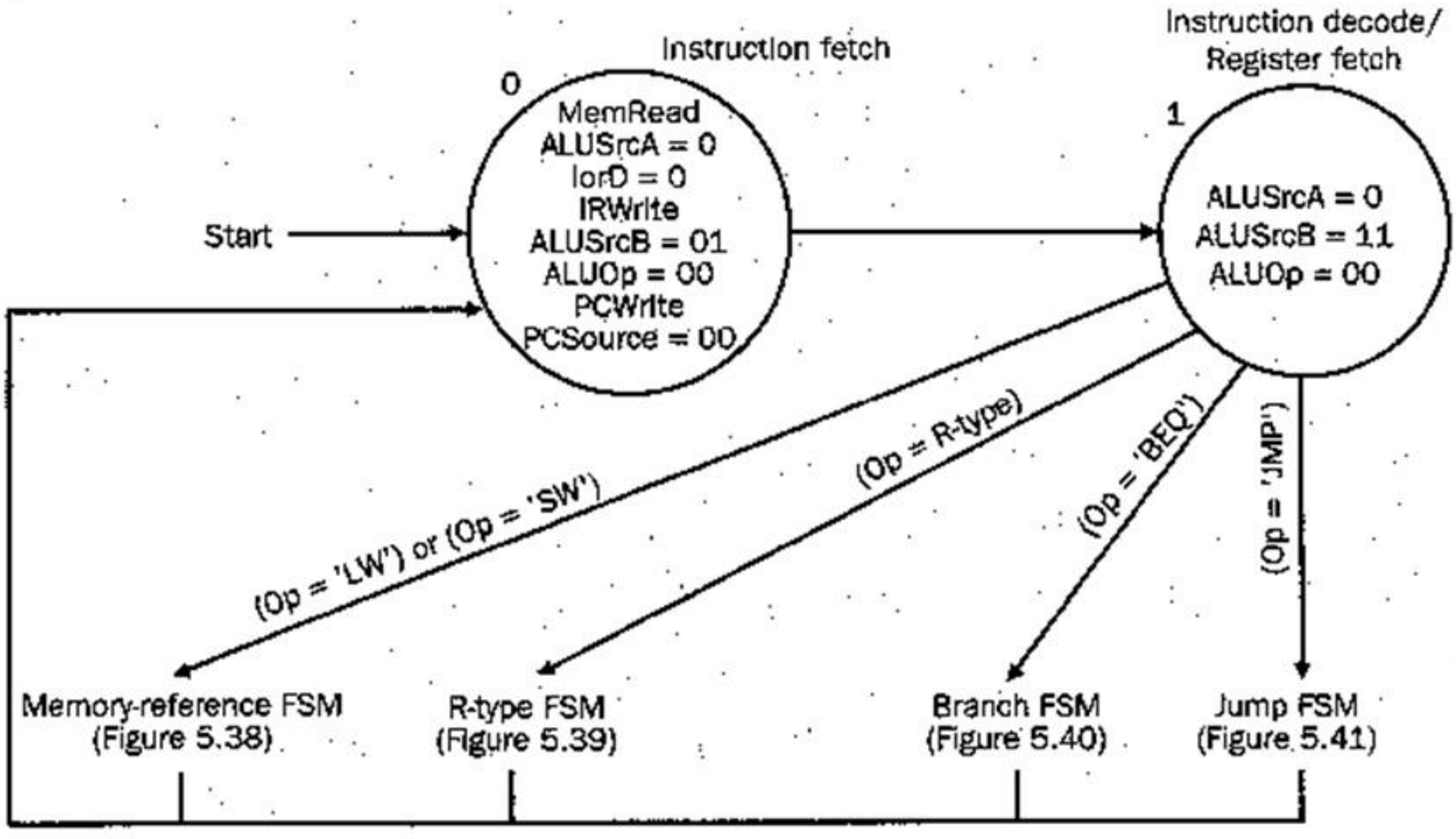
FSC pro multicyklové jednotky

„Pohled shora“ – vhodné pro abstrakci



FSC pro načtení & dekódování instrukce

Společné: Načtení instrukce/dat, dekódování



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)

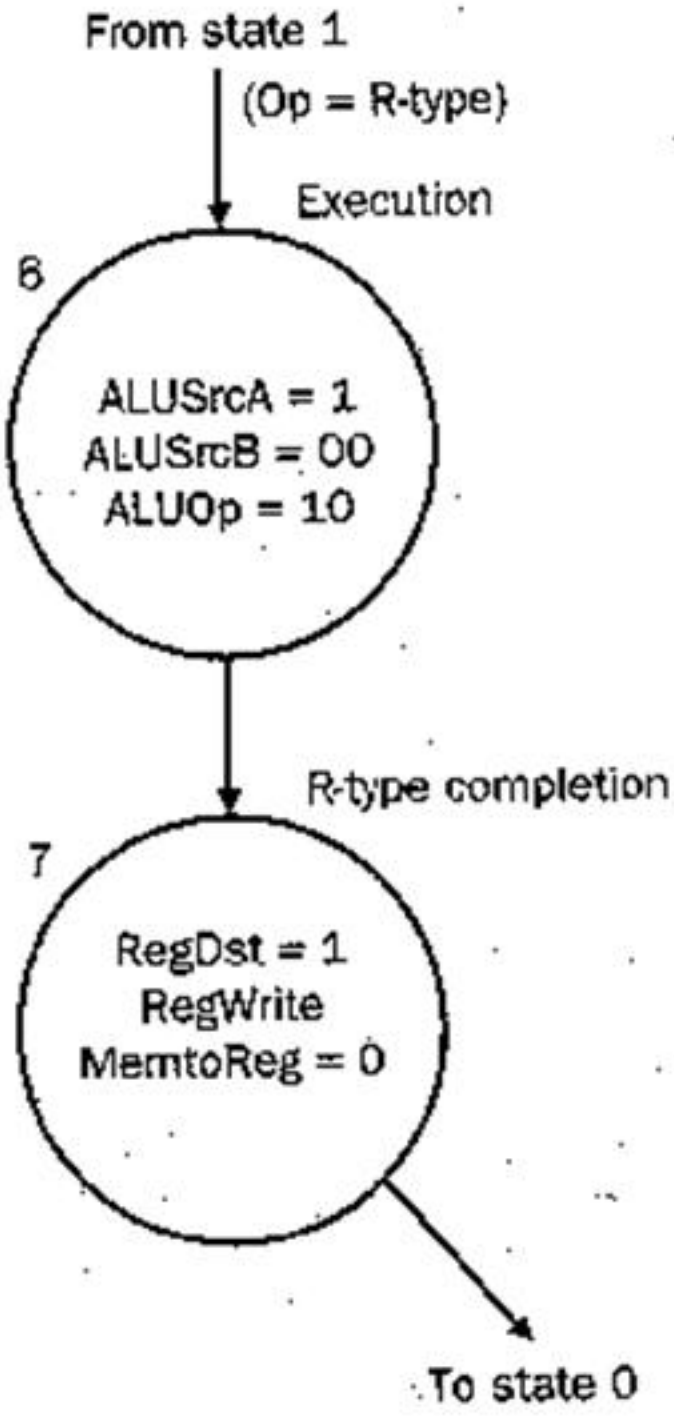
Výstup z ALU je zapsán do registru ALUout

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



Předpokládáme ukončení Fetch/Decode fáze

Určeny vstupy ALU z bufferů A, B

Provedení operace ALU podle opcode

Určení cílového registru podle rd

Zápis bufferu ALUout do sady registrů

Zpět na zpracování další instrukce

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

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

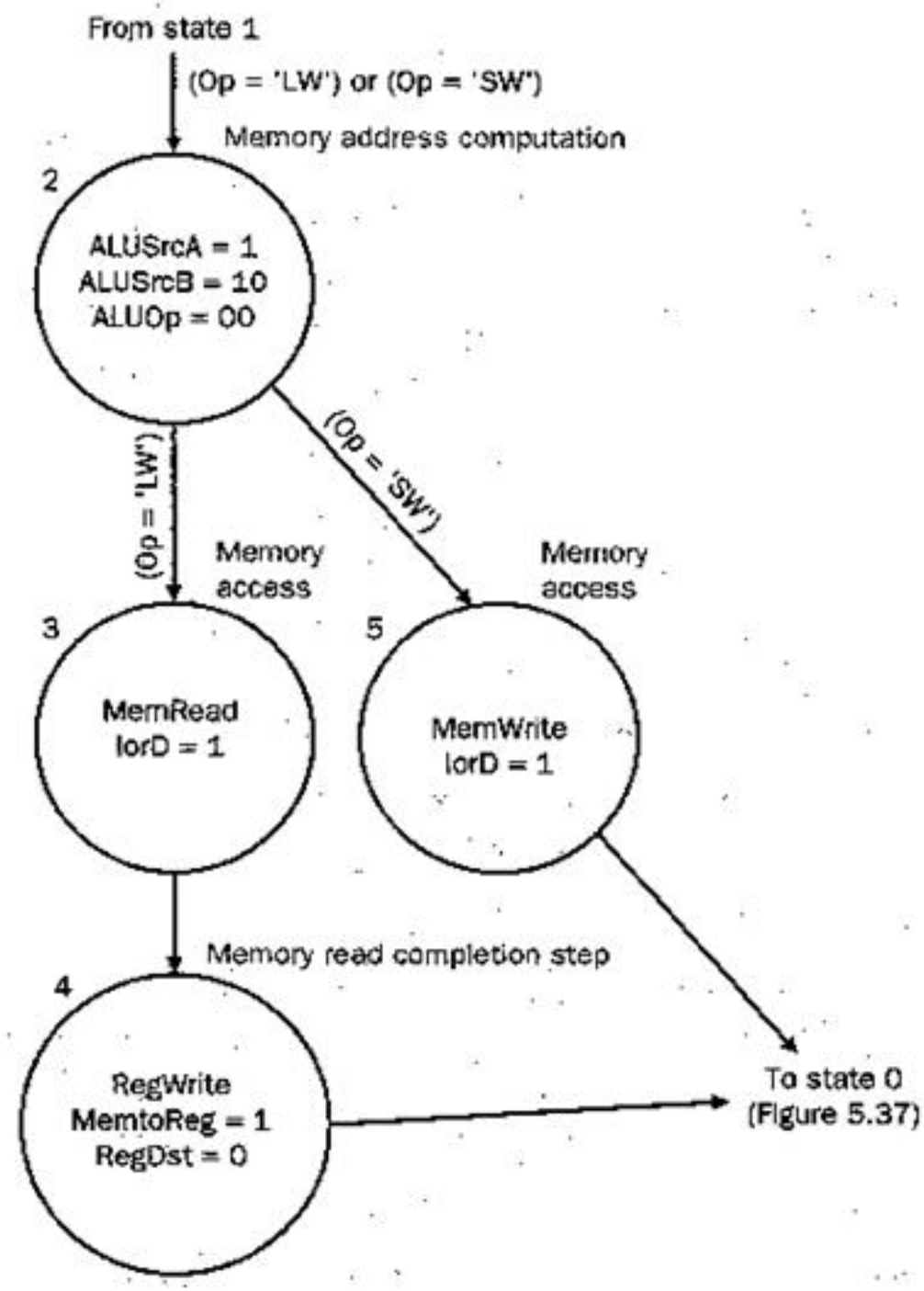
Aktivován signál: MemRead

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



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: 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

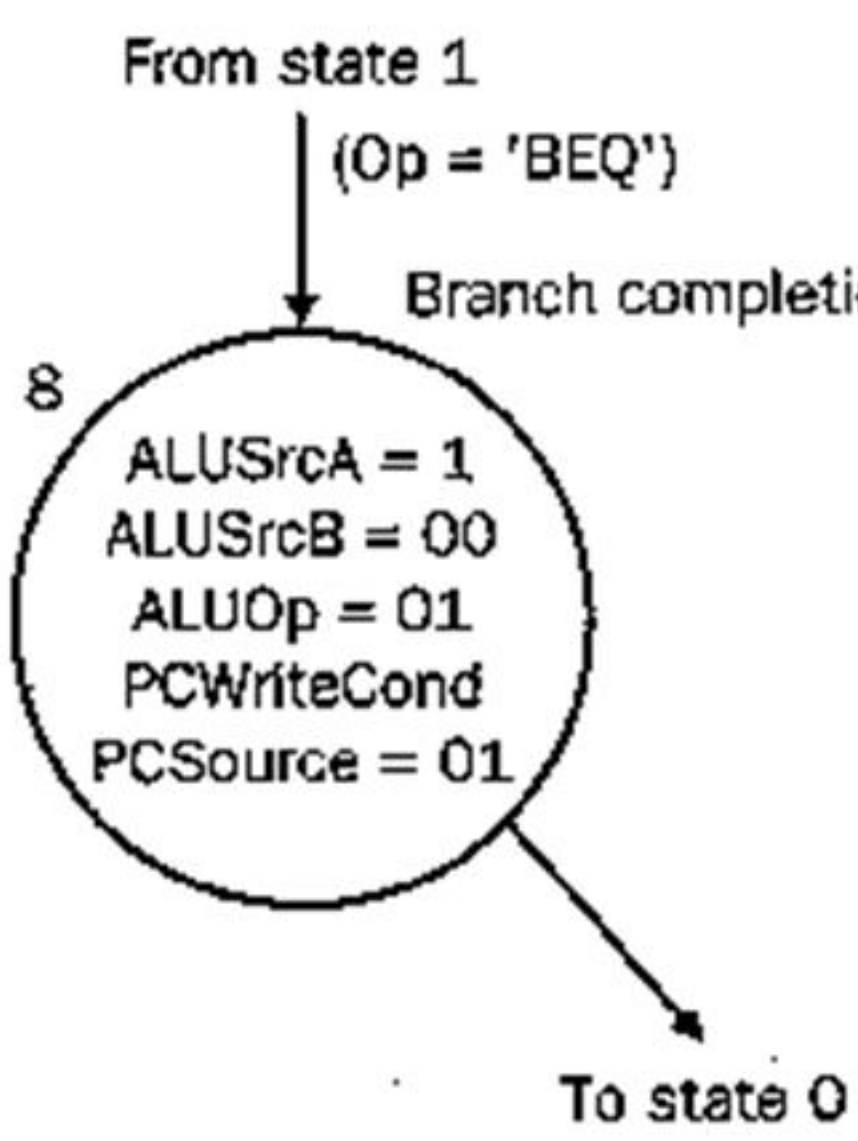
ALU určí PC, offset => BTA

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



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: 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

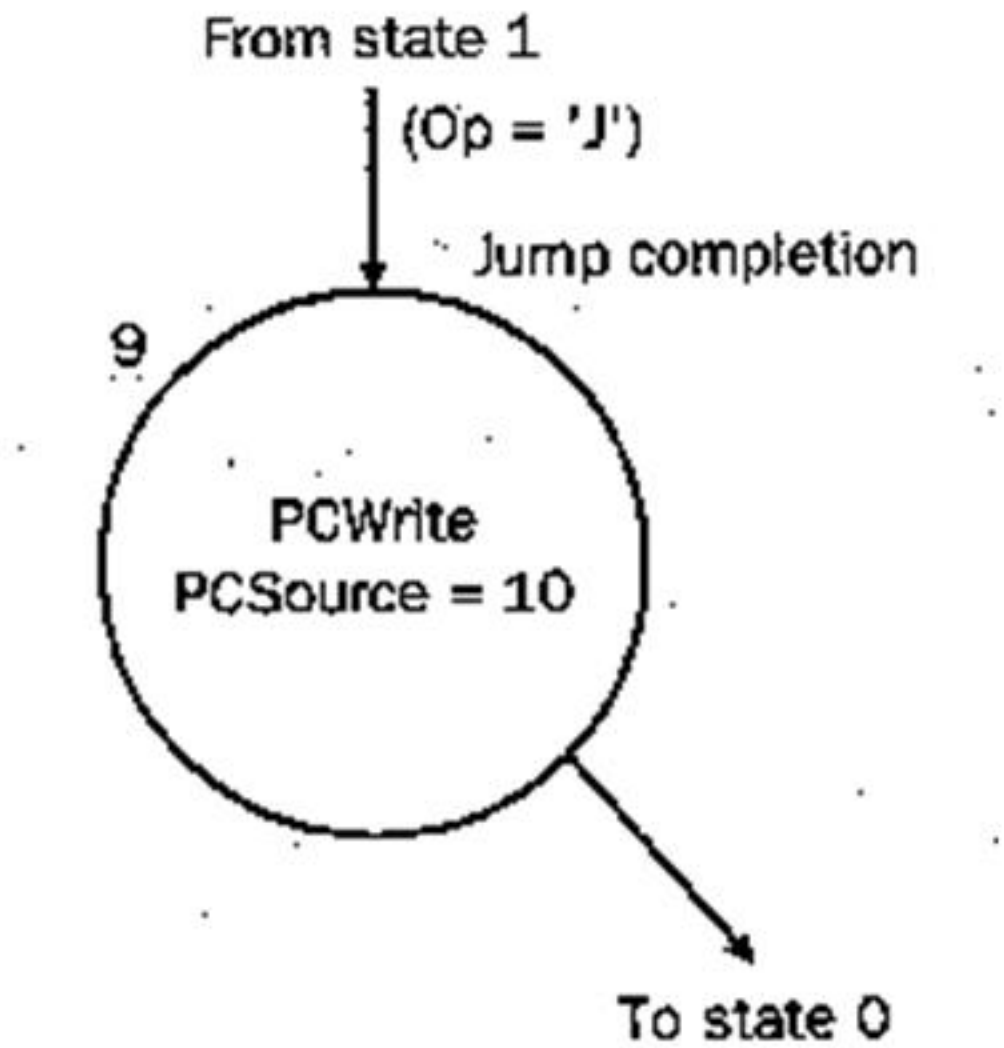
Krok 3: Obsah PC nahrazen cílovou adresou skoku (JTA)

PCsource = 10, *Aktivován signál:* PCWrite

CPI pro Jump = 3 cykly

Pozn.: JTA … adresa cíle instrukce skoku

FSC pro instrukci skoku



Předpokládáme ukončení fáze Fetch/Decode

PC bude přepsán

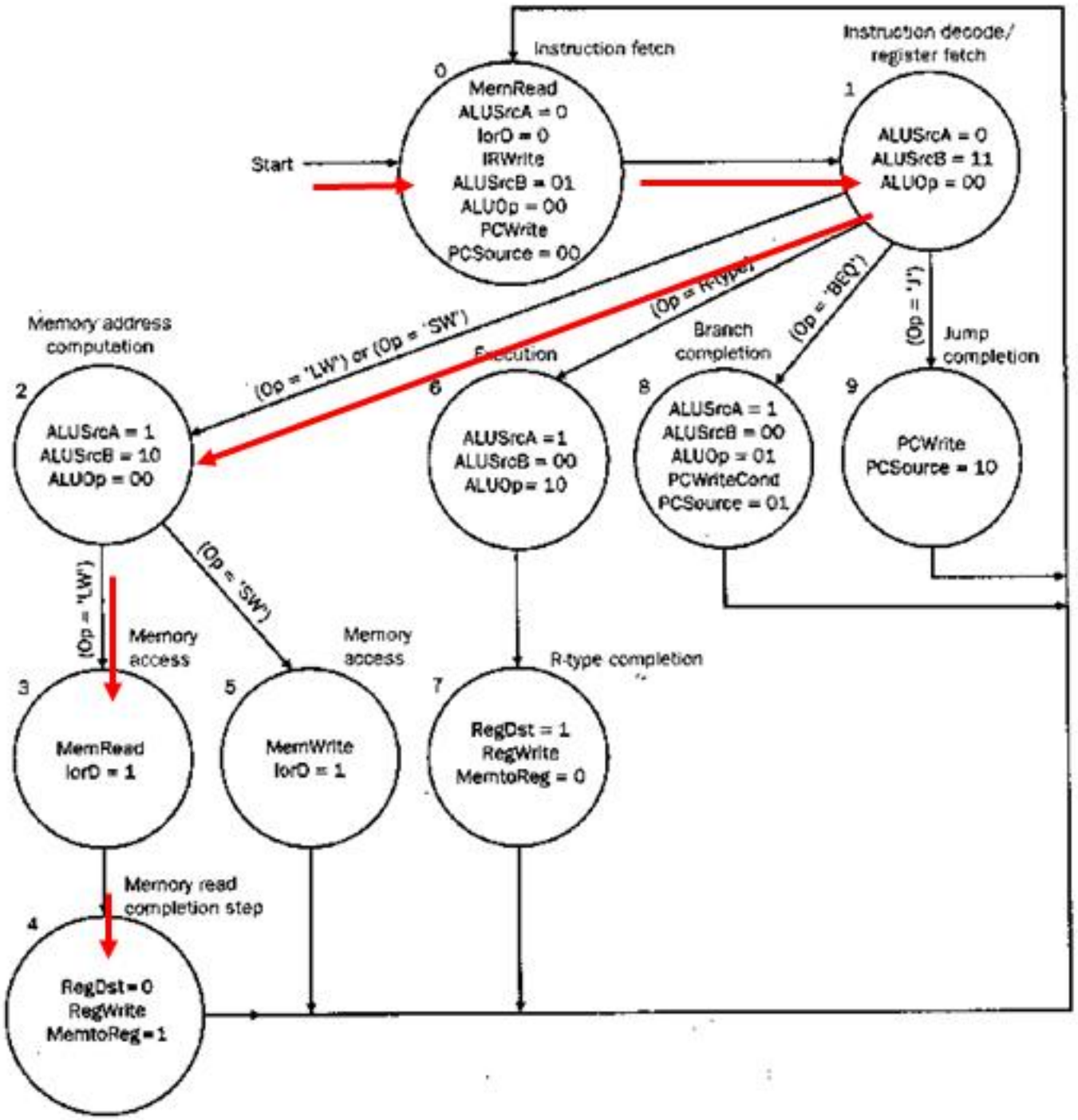
Hardware sestaví JTA

JTA se zapíše do PC

Zpět na zpracování další instrukce

JTA ... **J**ump **T**arget **A**ddress

Kompletní FSC



10 stavů celkem

CPI = počet stavů nutných pro provedení dané instrukce

R-formát = 4 stavy

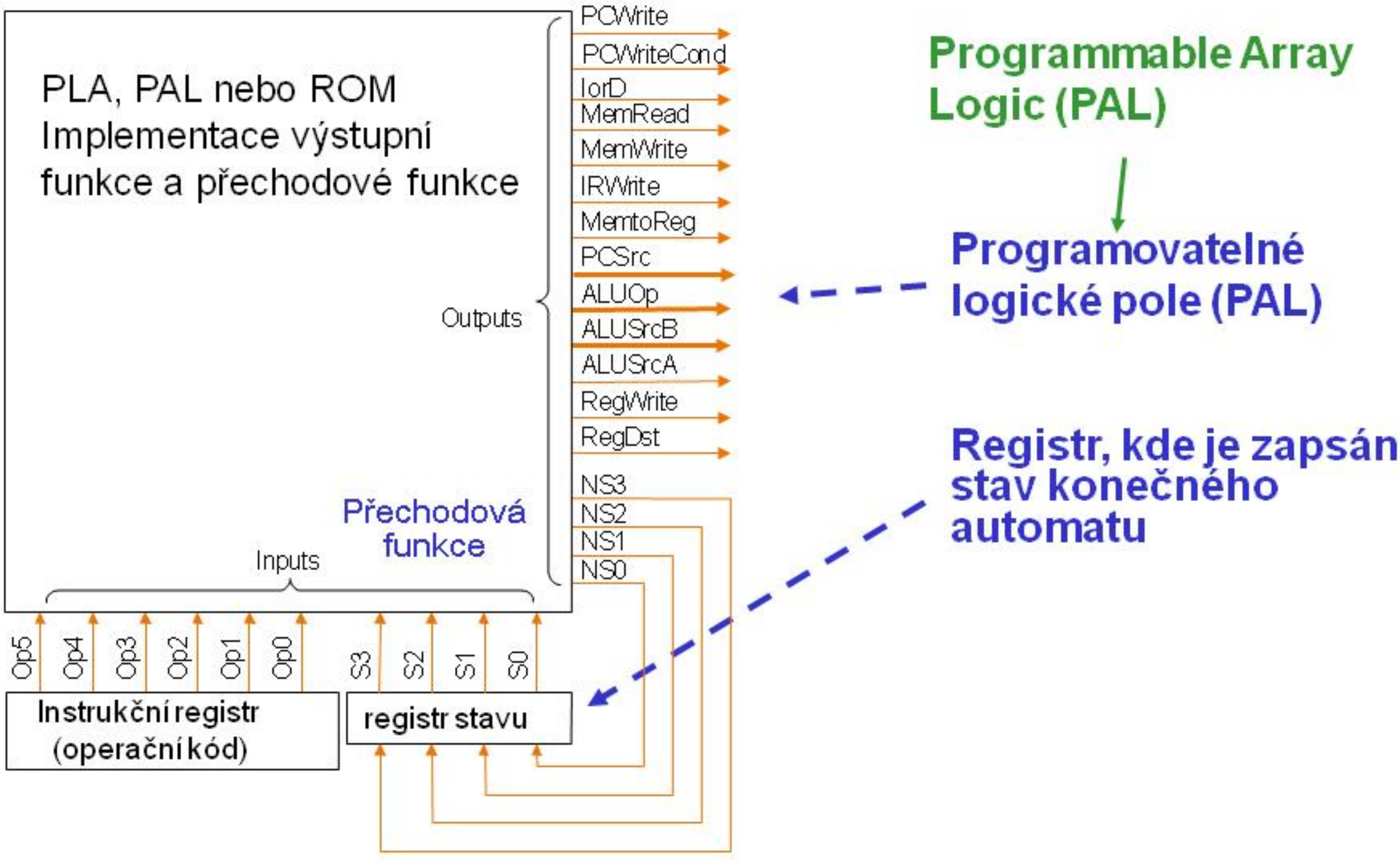
Store = 4 stavy

Load = 5 stavů [0,1,2,3,4]

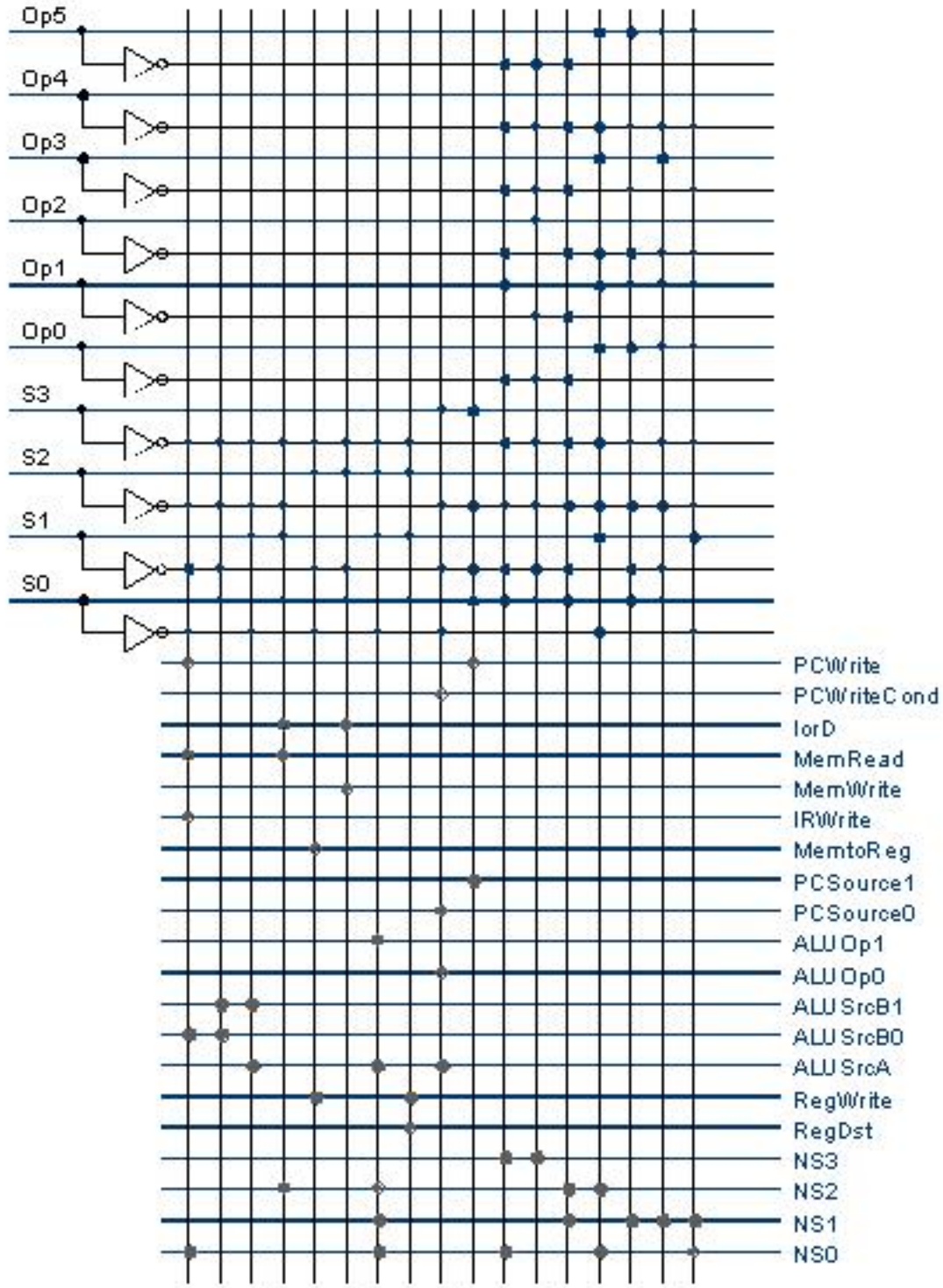
Branch = 3 stavy

Jump = 3 stavy

Hardware pro multicyklovou FSC

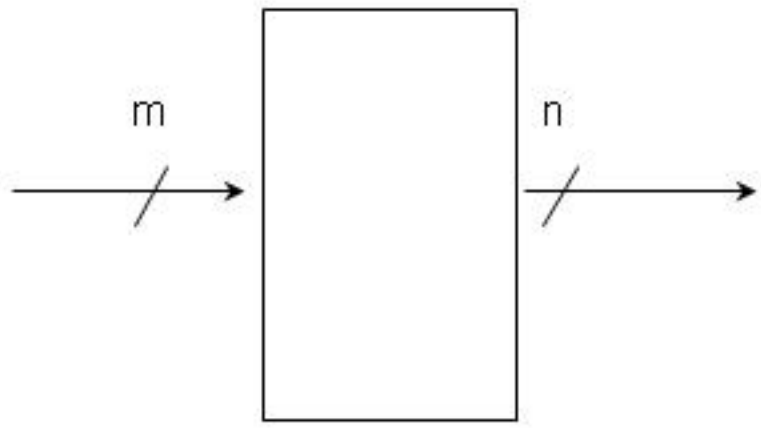


Implementace pomocí PLA



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)

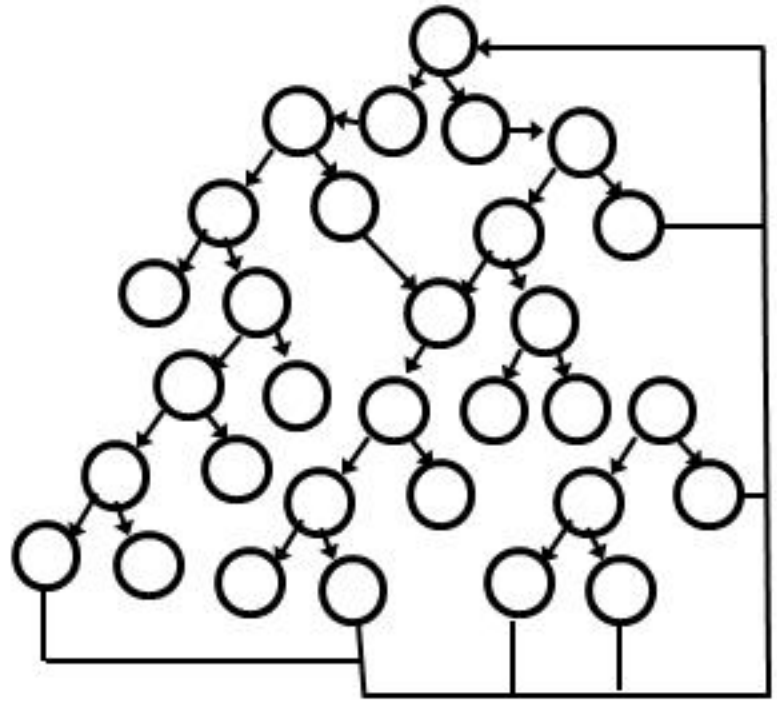
- Velmi nehospodárné vzhledem k velkému počtu situací, které nás nezajímají => výstupy závisejí pouze na stavech, nikoliv na op. kódu.

ROM versus PLA

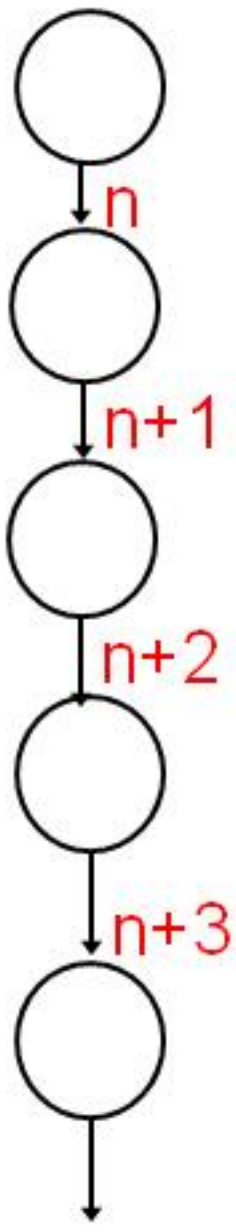
- Rozdělíme tabulku na dvě části
  - — 4 stavové bity určují celkem 16 výstupů, 2^4 x 16 bitů ROM
  - — 10 bitů určuje 4 bity příštího stavu, 2^{10} x 4 bitů ROM
  - — Celkem: 4.3K bitů ROM => poměrně značná úspora.
- PLA je mnohem menší
  - — může sdílet součinové termy
  - — obsahuje jen položky, které 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 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 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

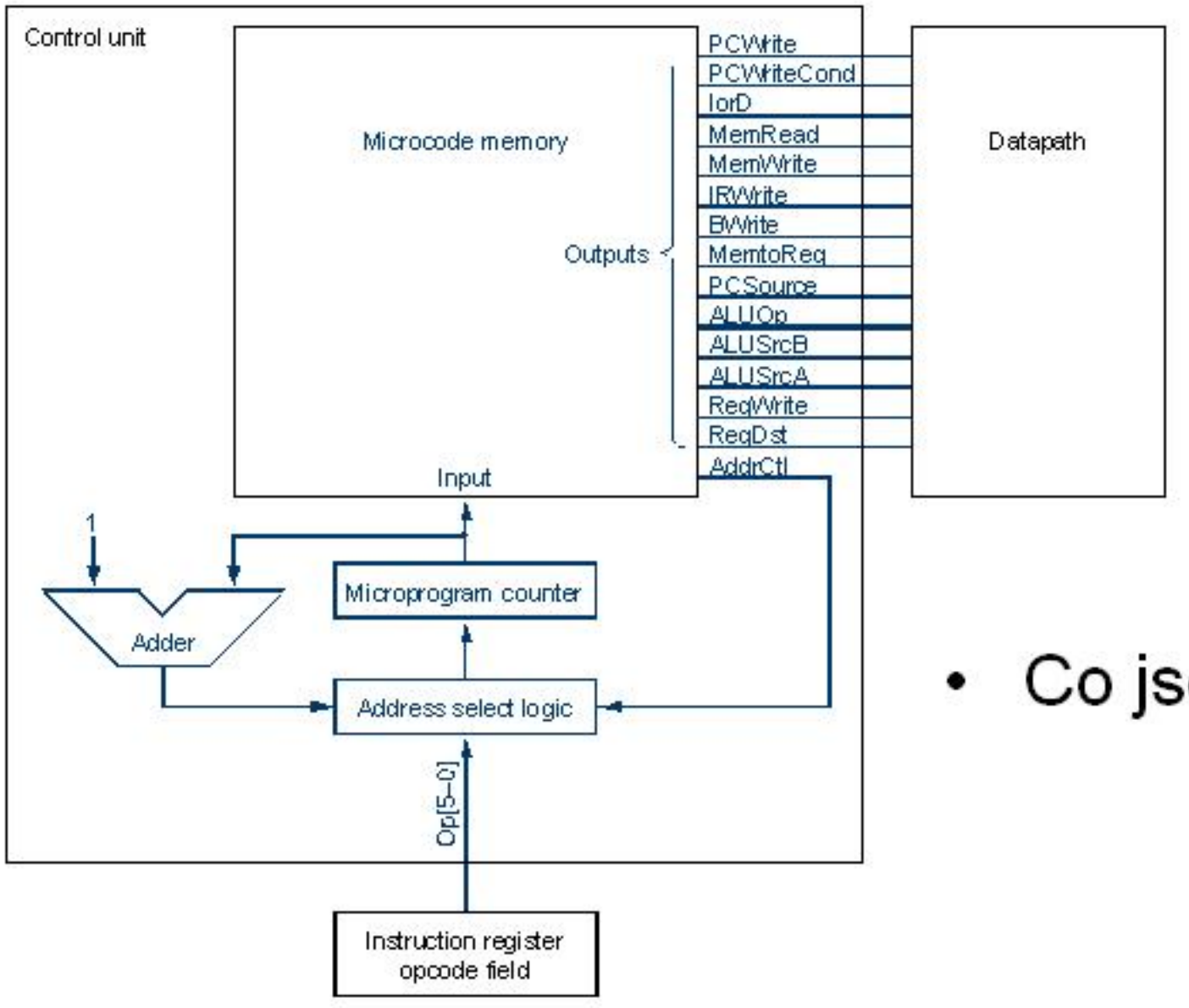
Mikroprogramové řízení

- 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 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



- Co jsou “mikroinstrukce”?

Mikroprogramování

- Specifikační metodologie
  - Vhodné pro stovky operačních kódů a množství adresních režimů
  - Signály 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

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, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 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:
  - 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

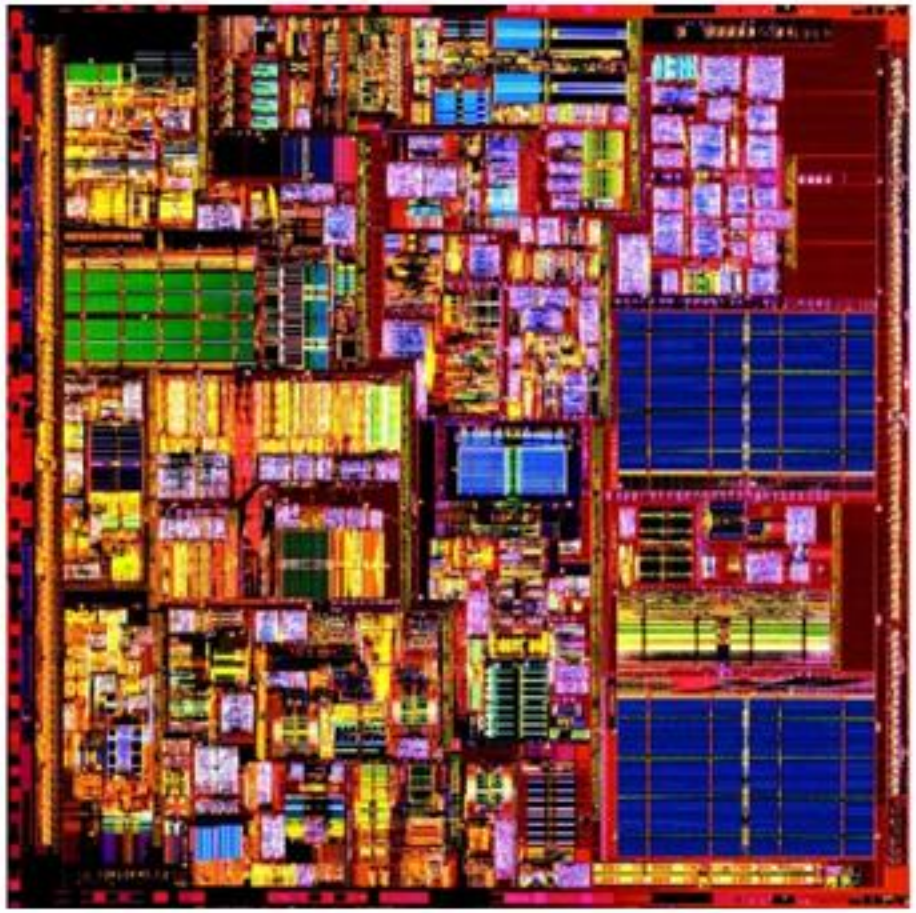
- Rozdíl mezi specifikací a implementací je někdy „neostrý“
- Výhody specifikace:
  - Snadný návrh a psaní
  - Současný návrh architektury a mikrokódu
- Výhody implementace (v externí ROM)
  - Snadná změna, protože se jedná o změnu obsahu paměti
  - Lze emulovat jiné architektury
  - Lze využívat interní 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

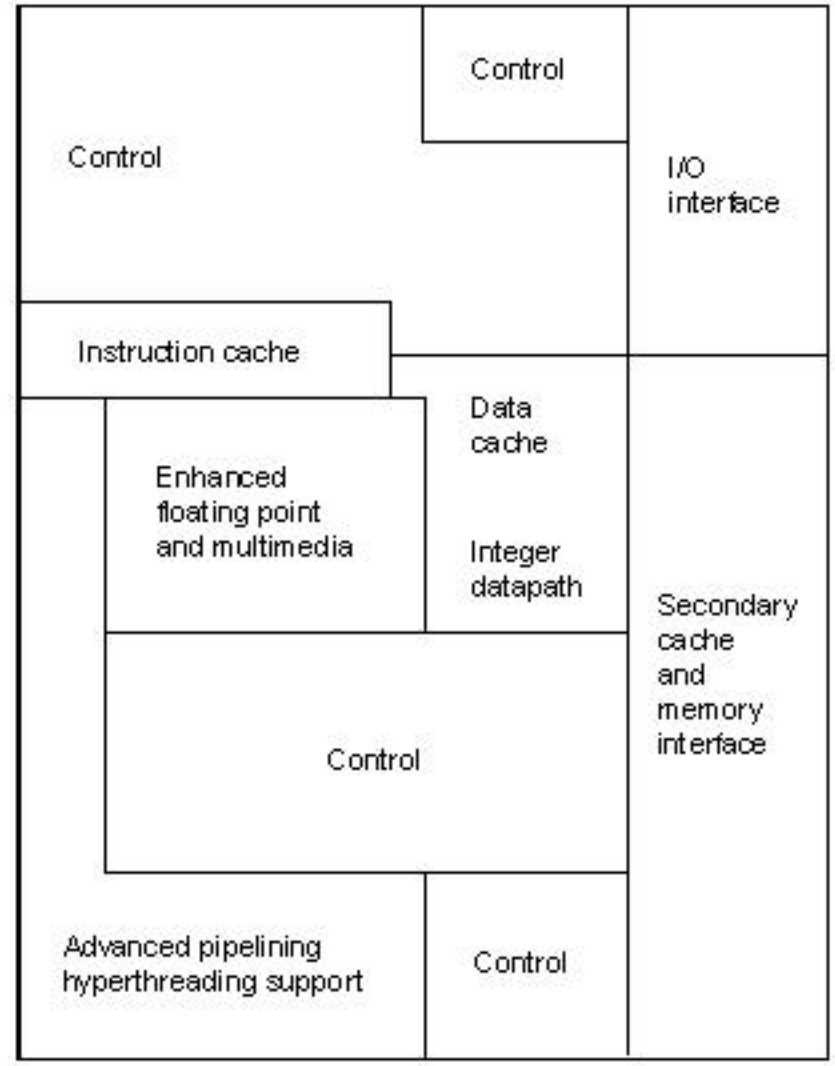
Historický přehled

- 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





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

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

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)

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:

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:

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 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:

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

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

- 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

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:**
  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 výjimek

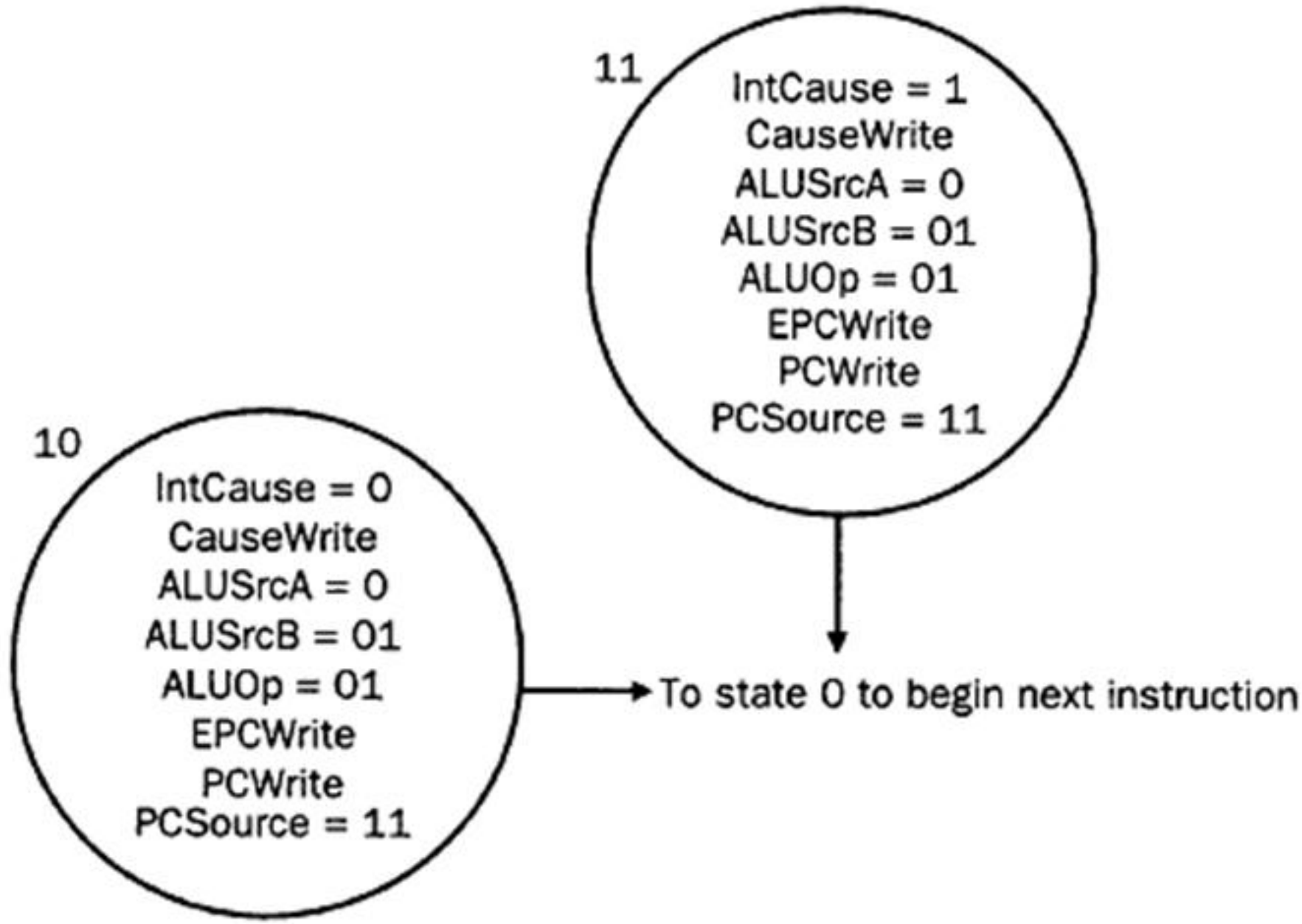
- **Dvě metody:** *EPC/Cause* a *Vektorové interrupty*
- *Vektorové interrupty*:
  1. Každá výjimka má vyhrazenou určitou adresu A_E
  2. Výjimka detekována => A_E je kvůli ošetření zapsána do PC
- *EPC / Příčina:* (MIPS) (EPC …Exception Program Counter)
  1. Výjimka detekována => Adresa instrukce uložena 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

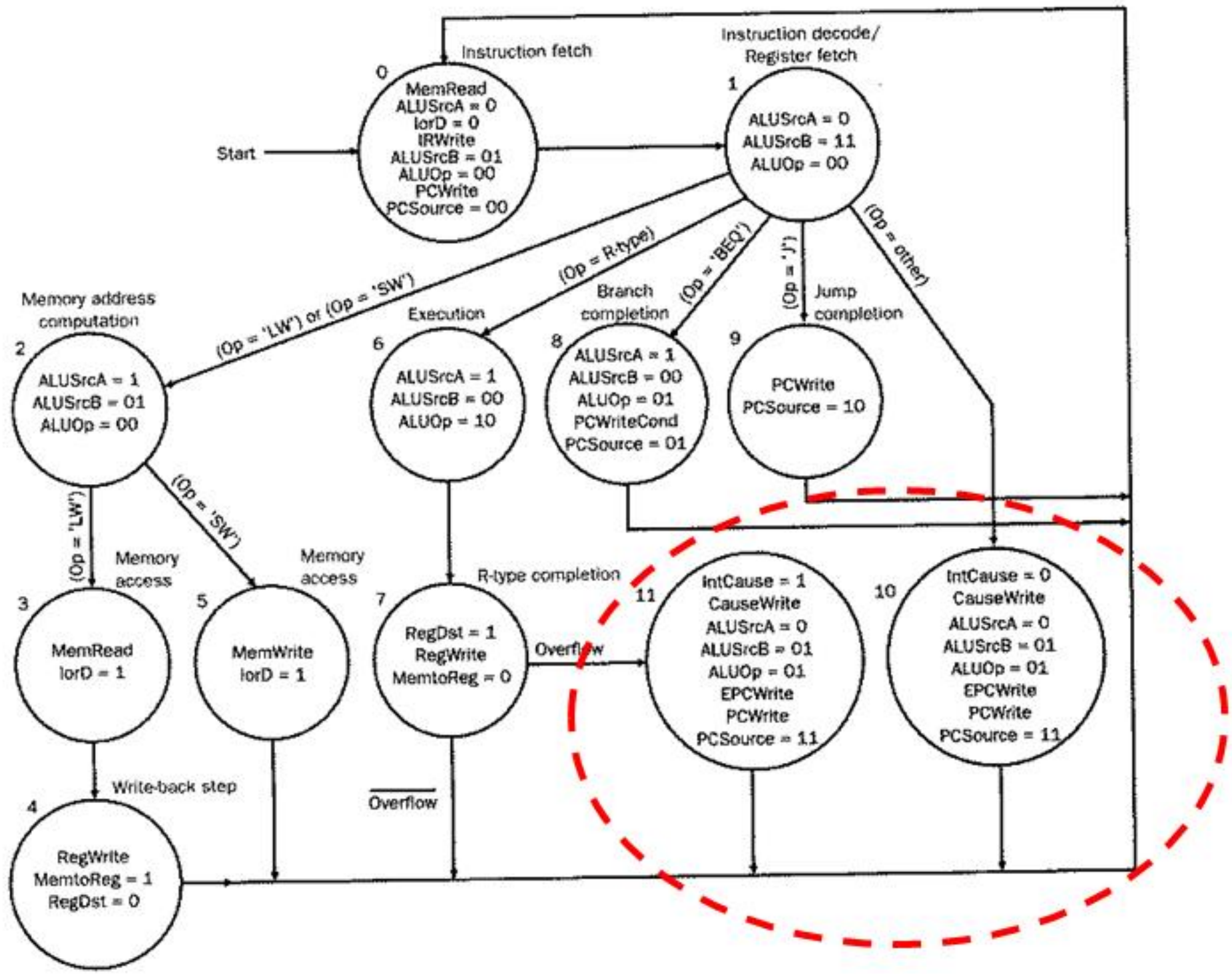
1. Nové registry - **$epc** a **Cause** (32-bitů)
2. Nové řídící signály – EpcWrite, CauseWrite
3. Nové řídící linky – 0 pro nedefinovanou instrukci
   1 pro přetečení
4. Nový Mux signál pro zdroj PC - PCsource = 11_2
   - Staré vstupy PC: PC+4, BTA, JTA
   - Nový vstup: $A_E = C0000000_{16}$ u MIPS

Opakování: detekce přetečení ALU "již instalováno"

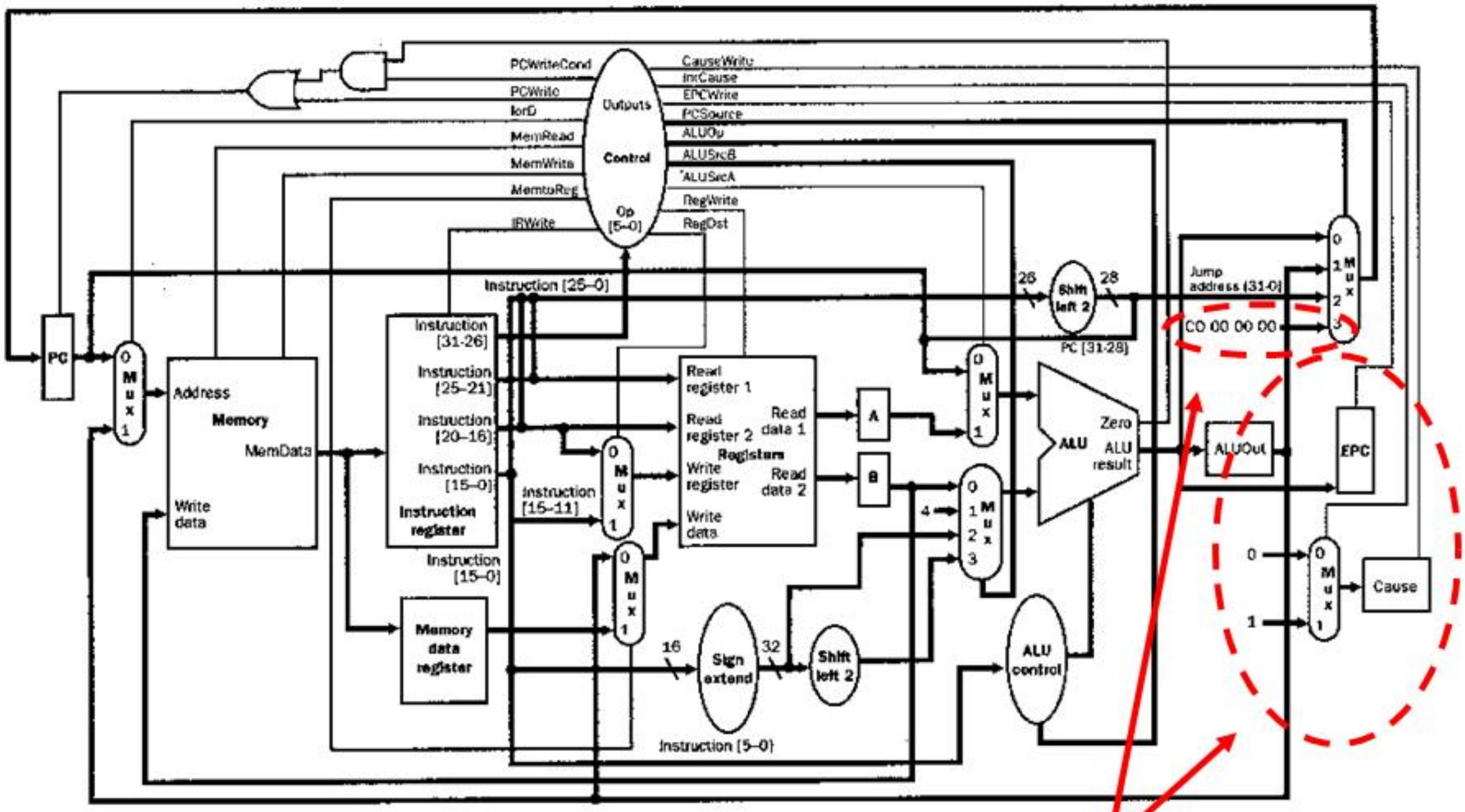
Nové stavy pro ošetření výjimek



Nový FSC se stavy výjimek



Nové uspořádání MIPS



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

Závěr

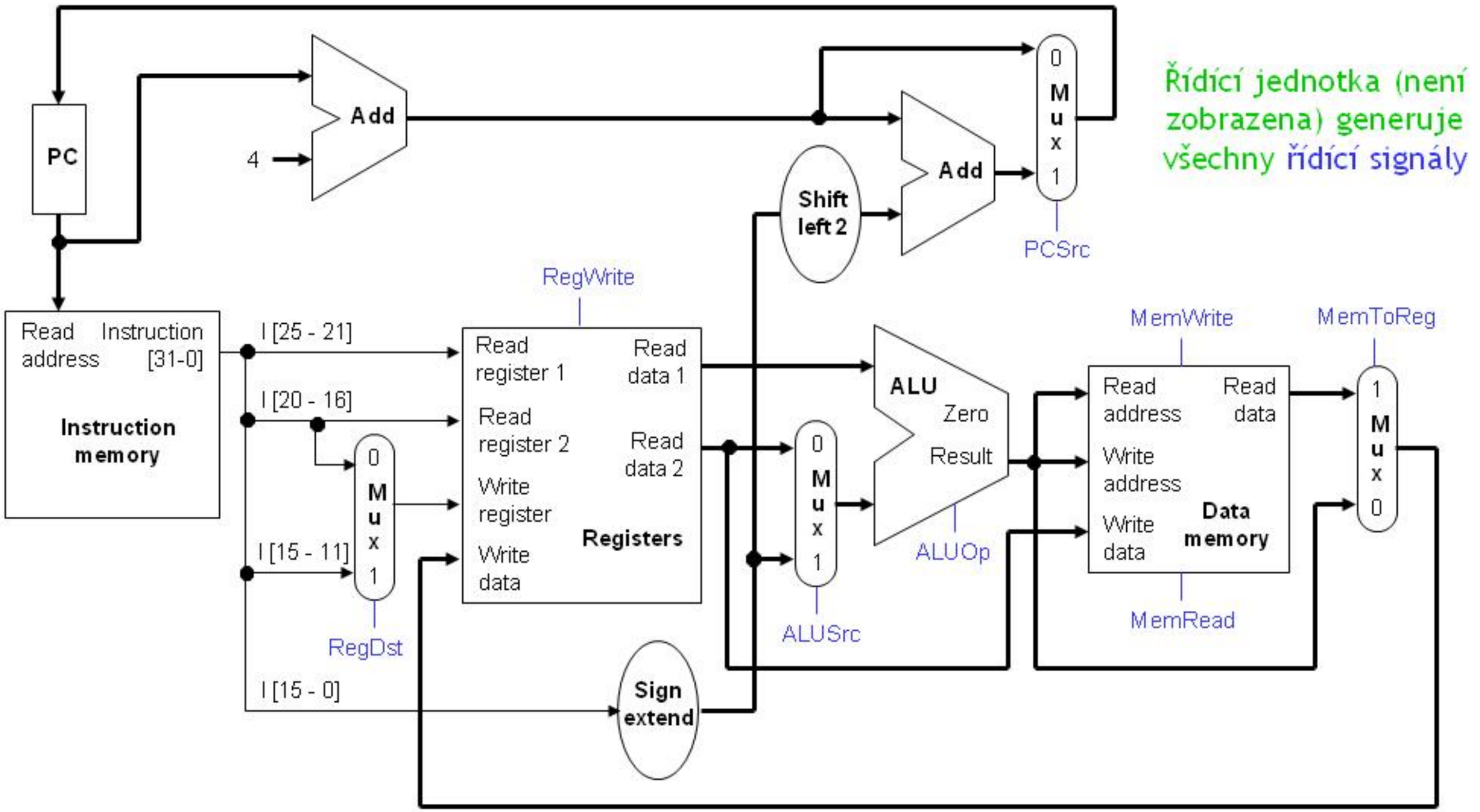
- **Mikroprogramování** – Flexibilnější pro rozlehlé ISA
- **Úkol:** Navrhnout pole a signály konzistentní
- **Omyl:** Mikroprogramy jsou nutně pomalejší
- **Omyl:** Přidávání instrukcí "je zdarma"
- **Výjimky** (interní události) vs. **Interrupty** (externí)
  - *Detekce výjimek* vyžaduje speciální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

Závěr

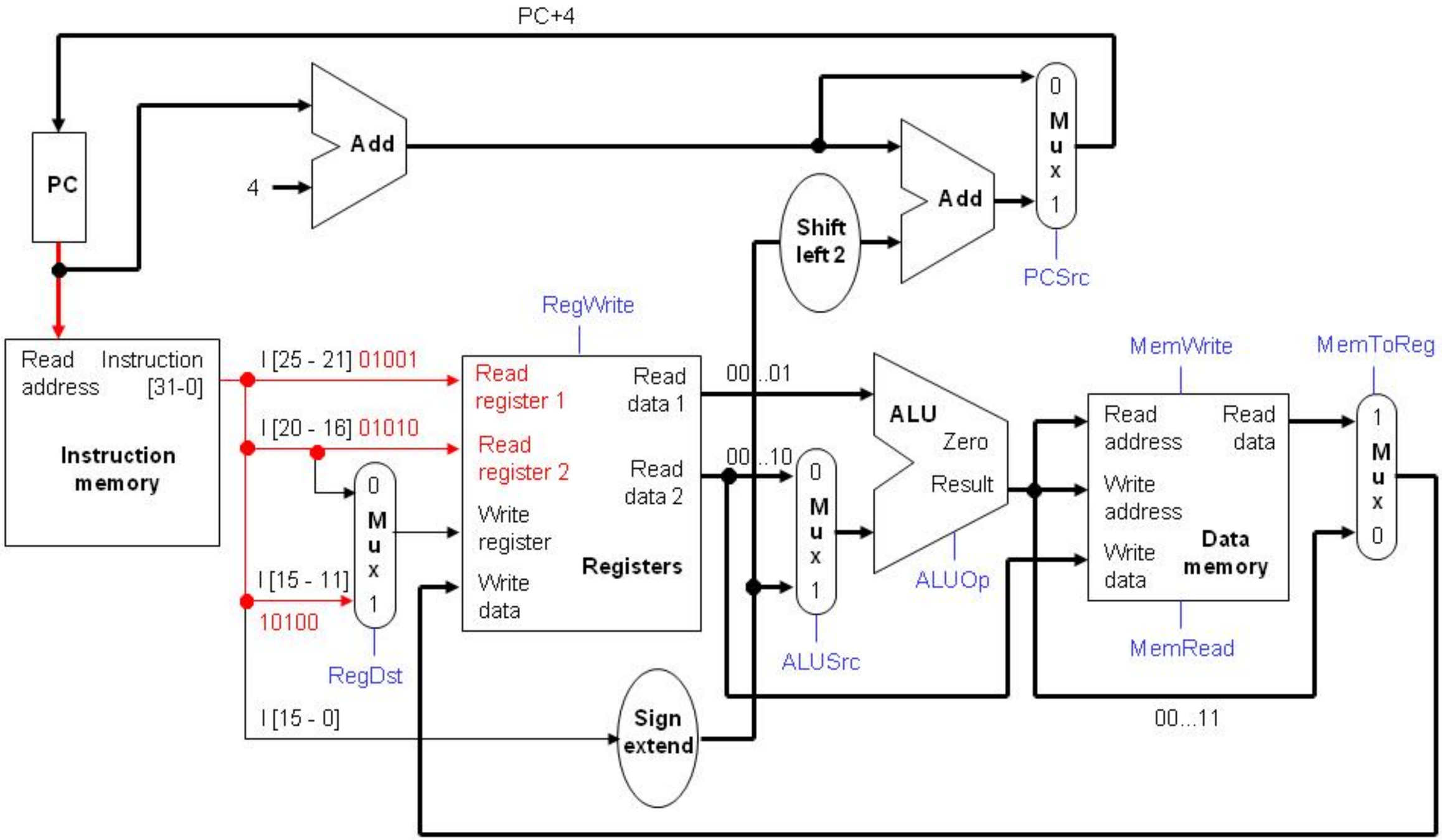
- Jestliže rozumíme instrukcím…
  Můžeme postavit jednoduchý procesor!
- Trvají-li instrukce různou dobu, je výhodnější multicyklové zpracování
- Datové cesty se implementují s využitím:
  - Kombinační logiky pro aritmetiku
  - Paměťové prvky (klop. obvody) pro zapamatování informace
- Implementace řízení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

Od jednocyklové implementace k pipelinningu

Jednocyklová jednotka

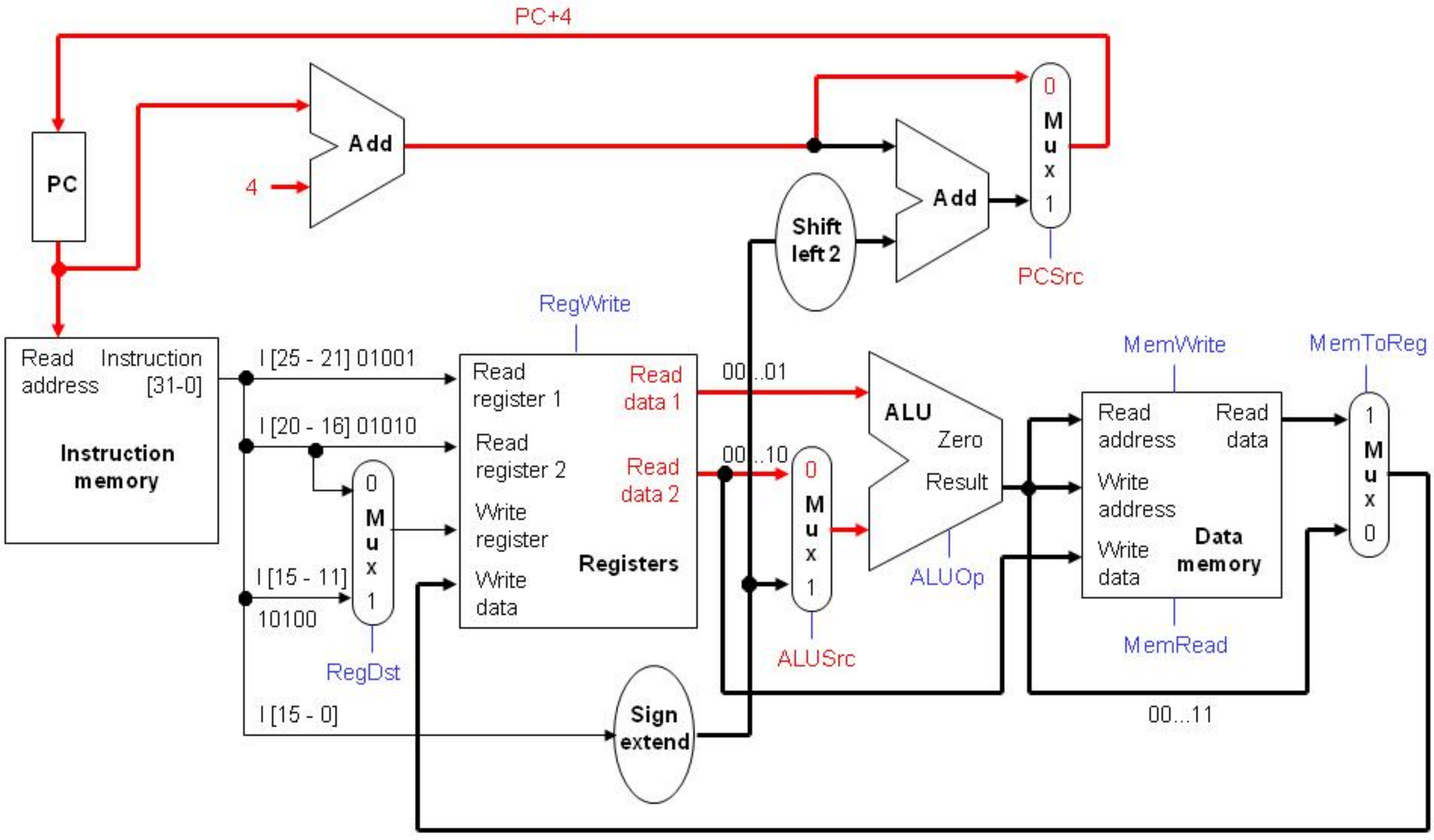


Provádění instrukce add $s4, $t1, $t2



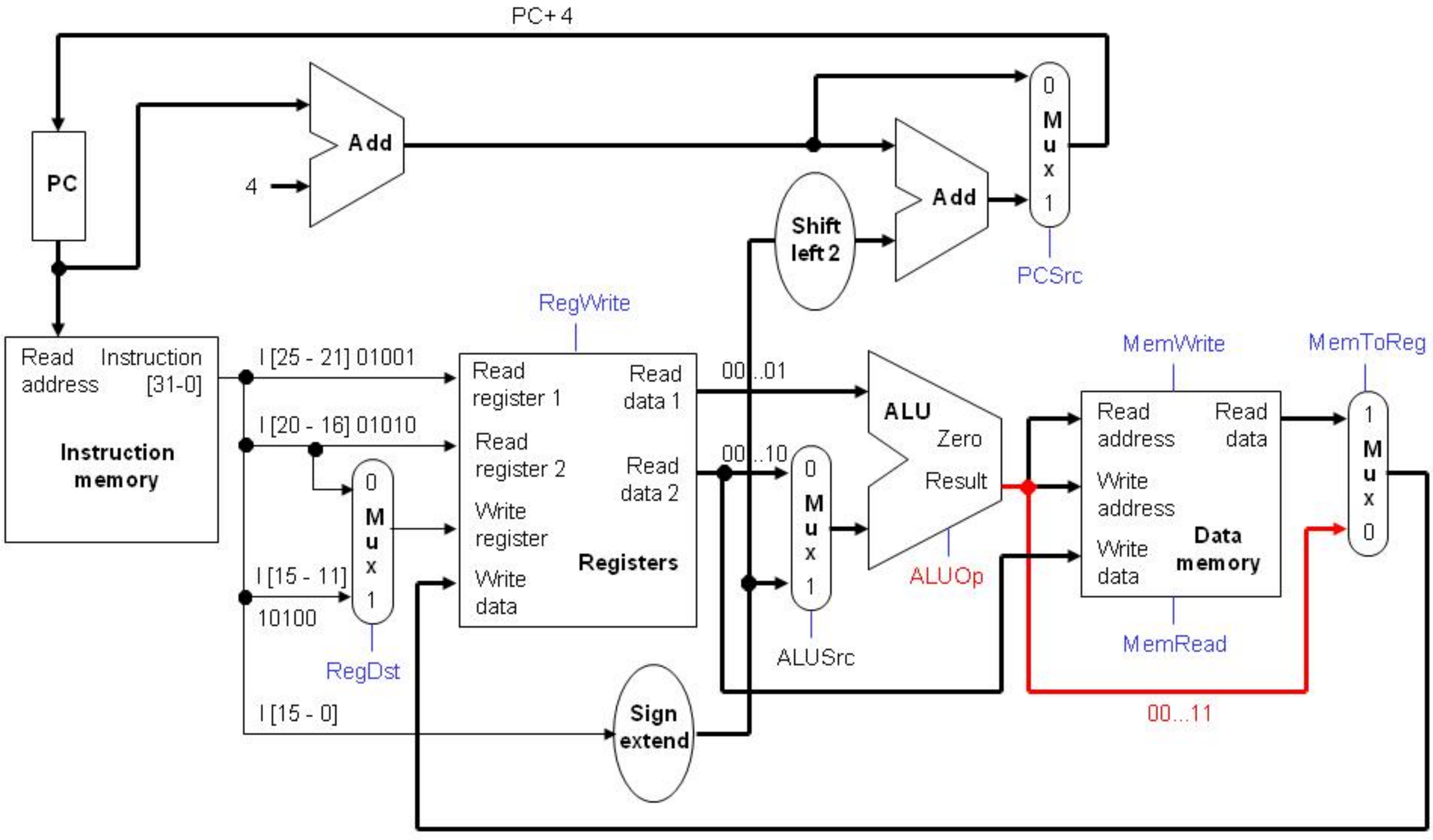
Čtení instrukce z paměti

Provádění instrukce add $s4, $t1, $t2



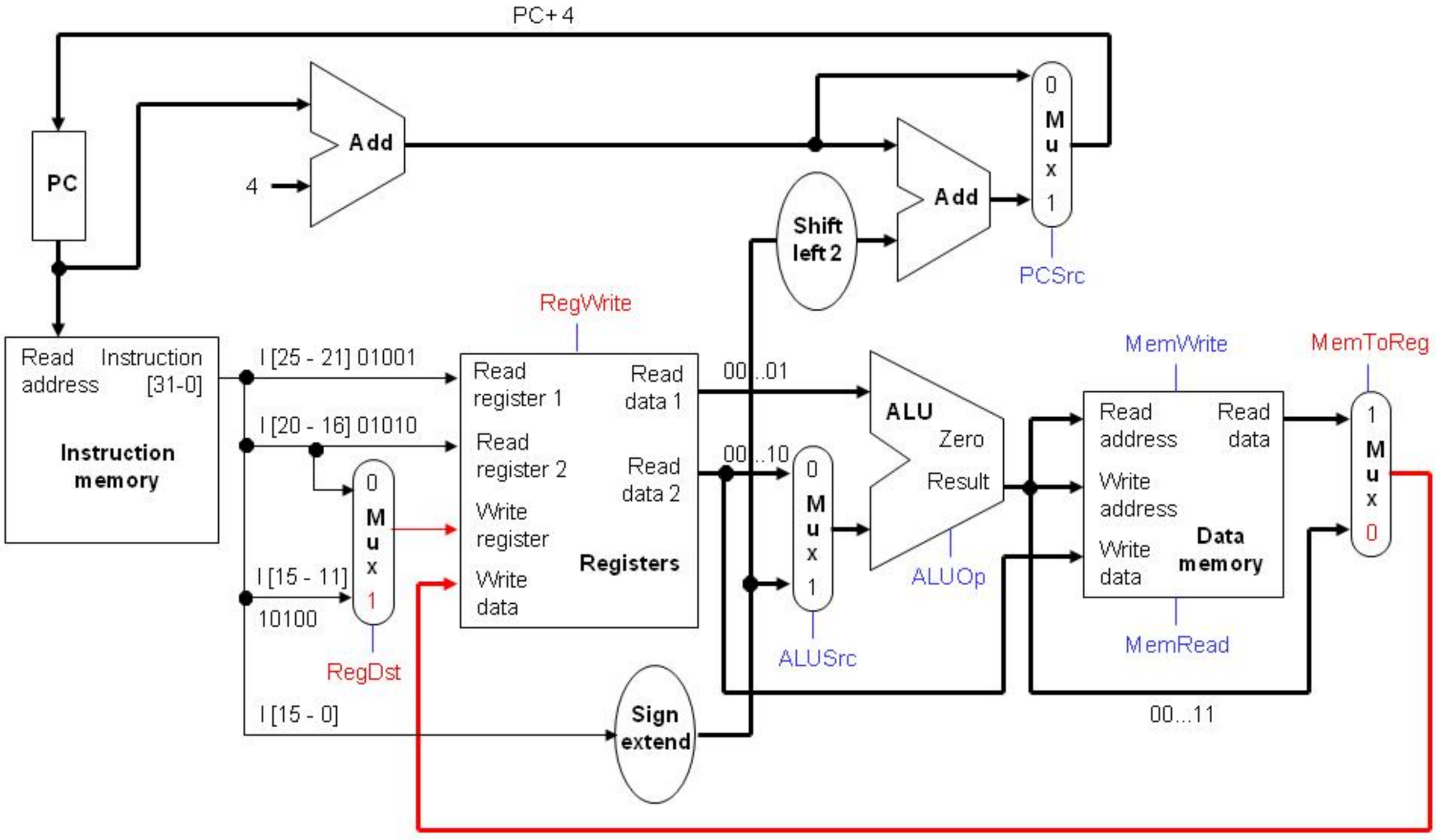
Čtení registrů, inkrement PC + 4, zápis nové hodnoty PC

Provádění instrukce add $s4, $t1, $t2



Provedení součtu v ALU

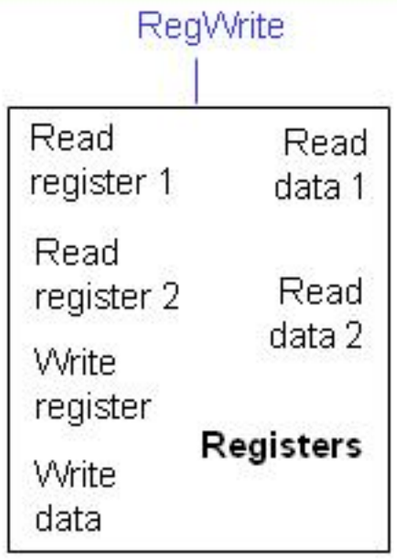
Provádění instrukce add $s4, $t1, $t2

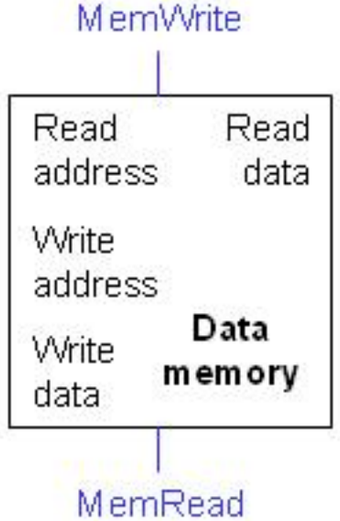


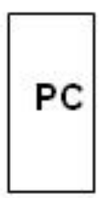
Zápis výsledku do registru

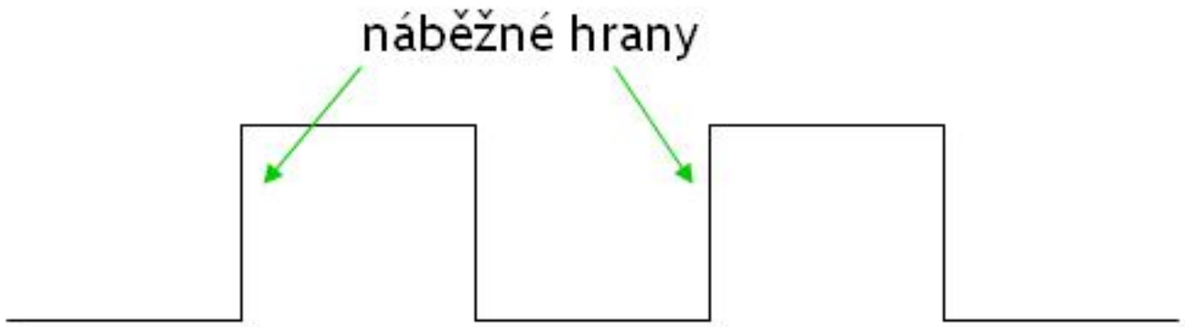
Časování: hrany hodinového signálu

- 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

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

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.

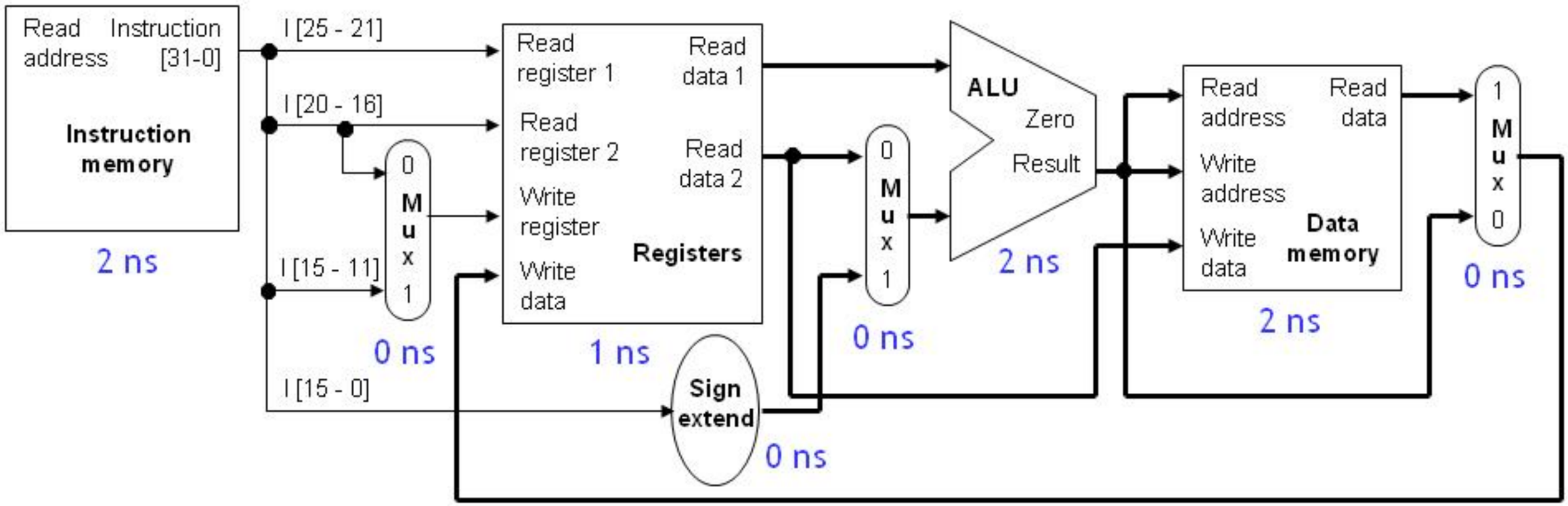
- U jednocyklové verze všechno v bodě 2 se musí odehrát během jednoho hodinového cyklu, před příchodem další náběžné hrany hodin.

Jak dlouho má trvat perioda hodin?

Prvky procesoru

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

čtení instrukční paměti	2ns	8ns
čtení registrového souboru	1ns	
výpočet v ALU	2ns	
přístup do datové paměti	2ns	
zápis do registrového souboru	1ns	

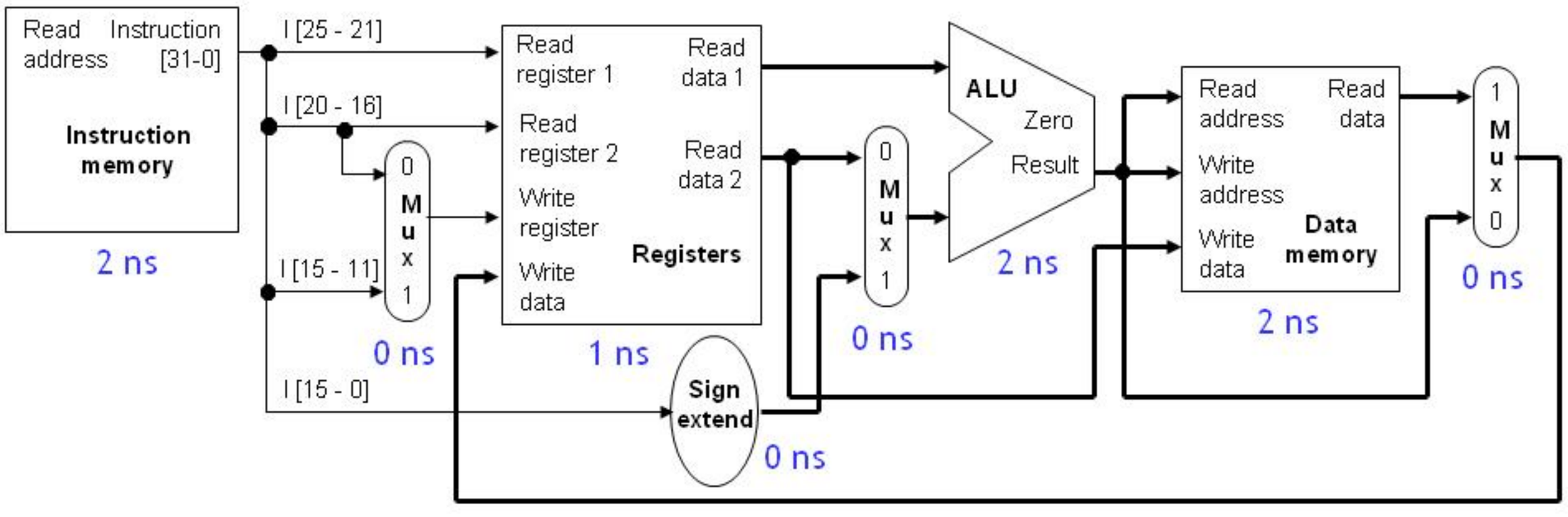


Nejpomalejší instrukce...

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

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

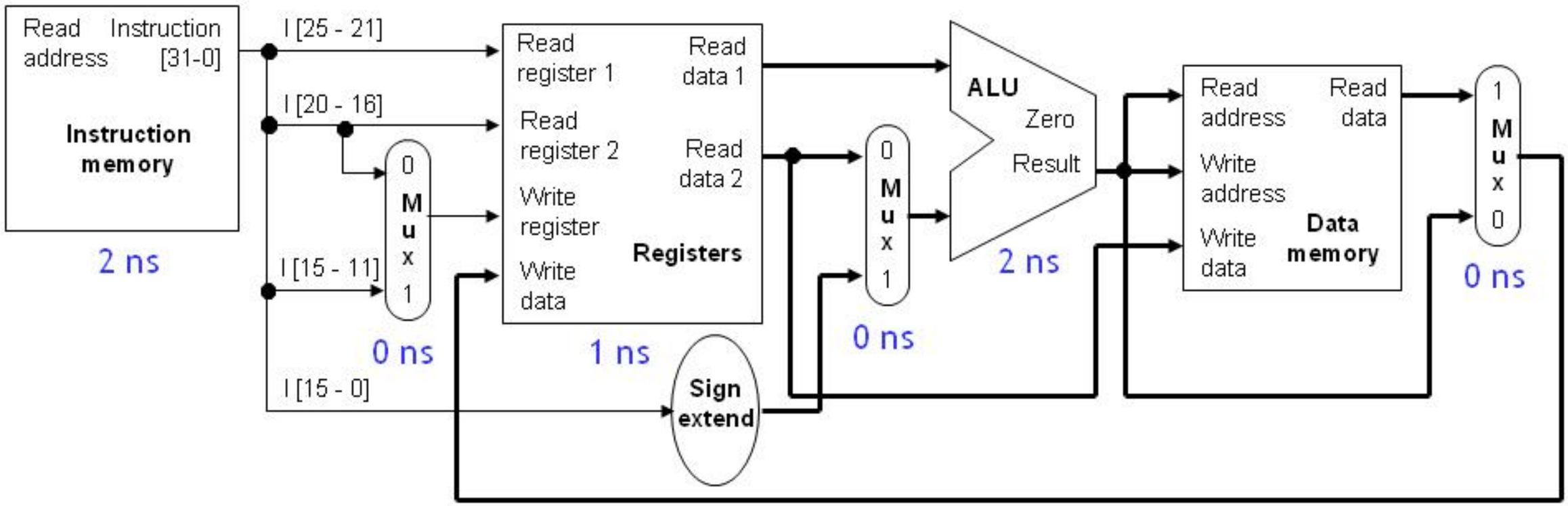
čtení instrukční paměti	2ns	8ns
čtení bázového registru $sp	1ns	
výpočet adresy paměti $sp-4	2ns	
čtení datové paměti	2ns	
uložení dat zpět do $t0	1ns	



…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.

čtení instrukční paměti	2ns	6ns
čtení registrů $t1 a $t2	1ns	
výpočet $t1 + $t2	2ns	
uložení výsledku do $s0	1ns	



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!

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

- 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*

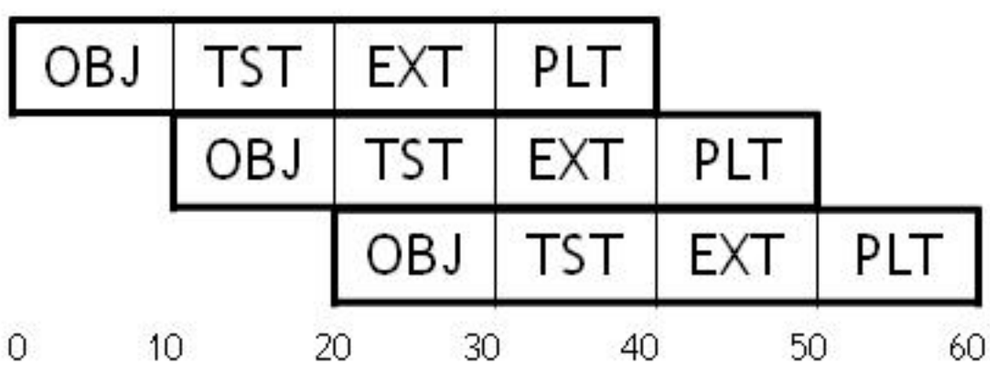
- Příklady pipeliningu: jakýkoliv proces montáže (auta, bagety), prádelna (pračka + sušička – lze provádět v režimu pipeline), atd.

Příklad

- 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:



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“

- 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

- 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

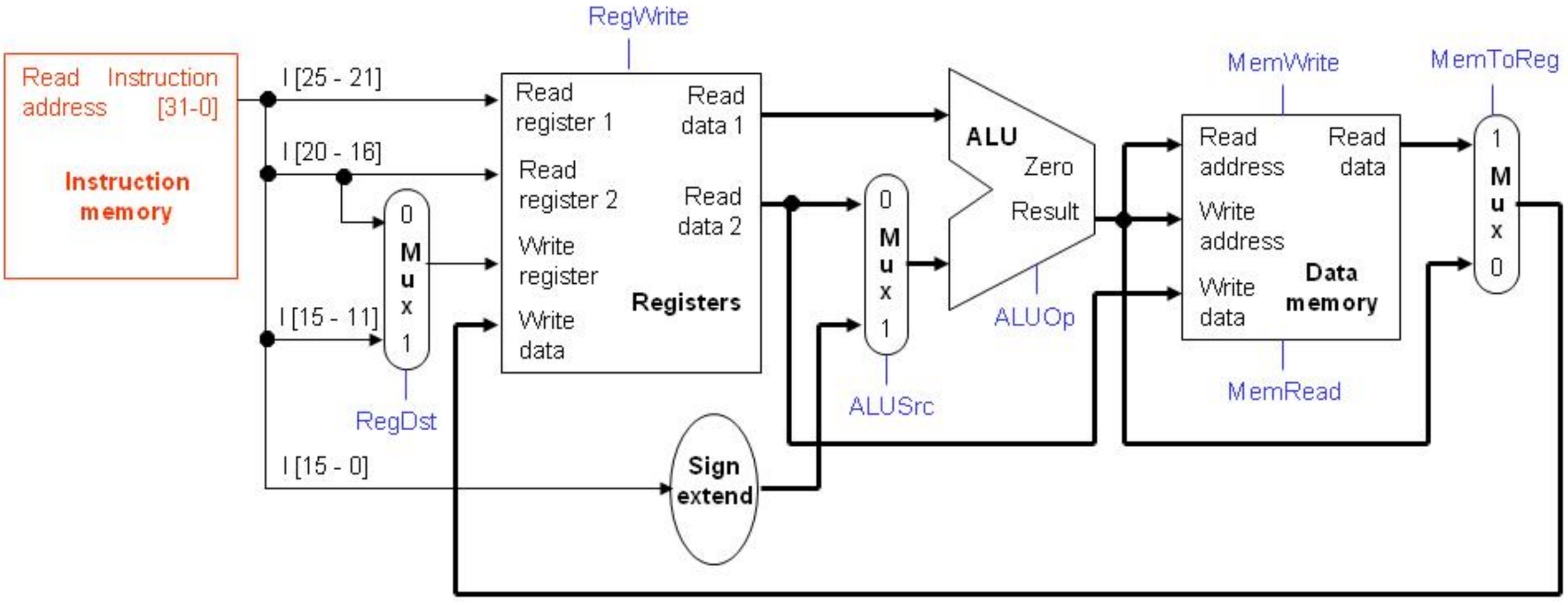
- Všechny instrukce nepotřebují všech 5 stupnů…

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

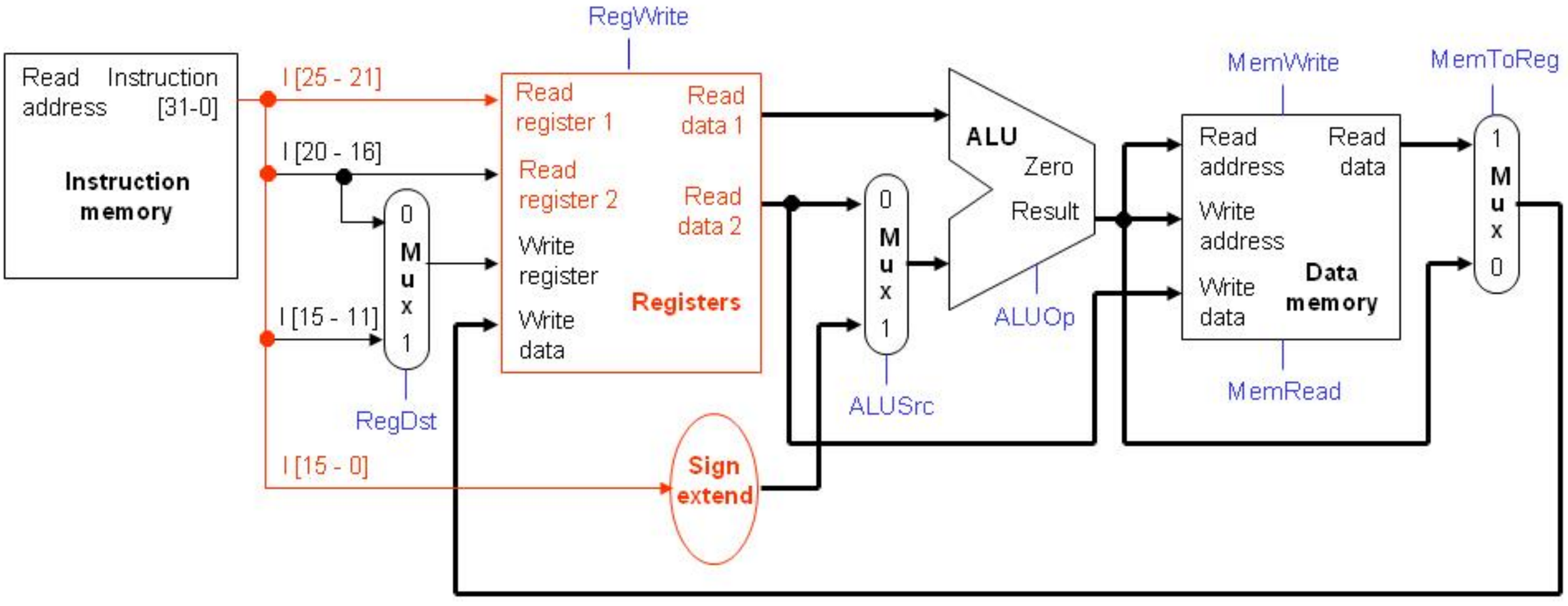
Čtení instrukce (IF)

- Zatímco se provádí IF, zbytek jednotky je nečinný …



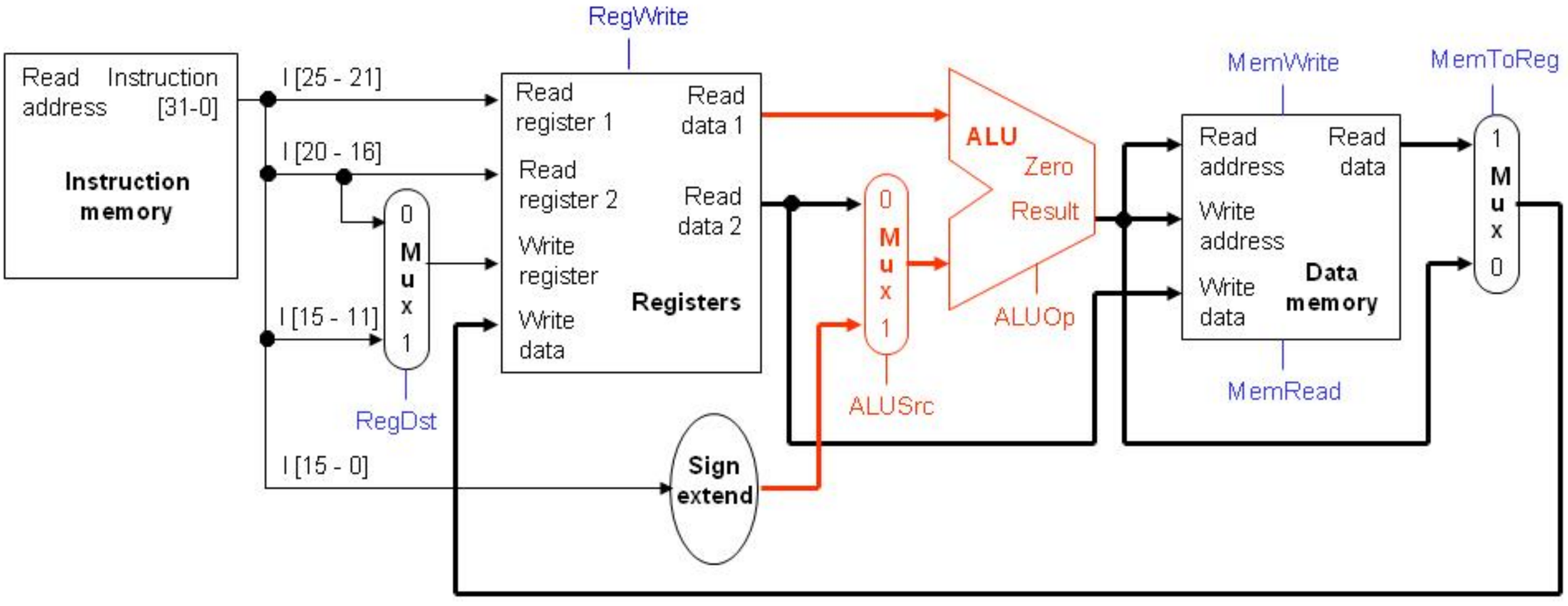
Dekódování instrukce (ID)

- Při provádění ID je část, odpovídající IF nečinná …



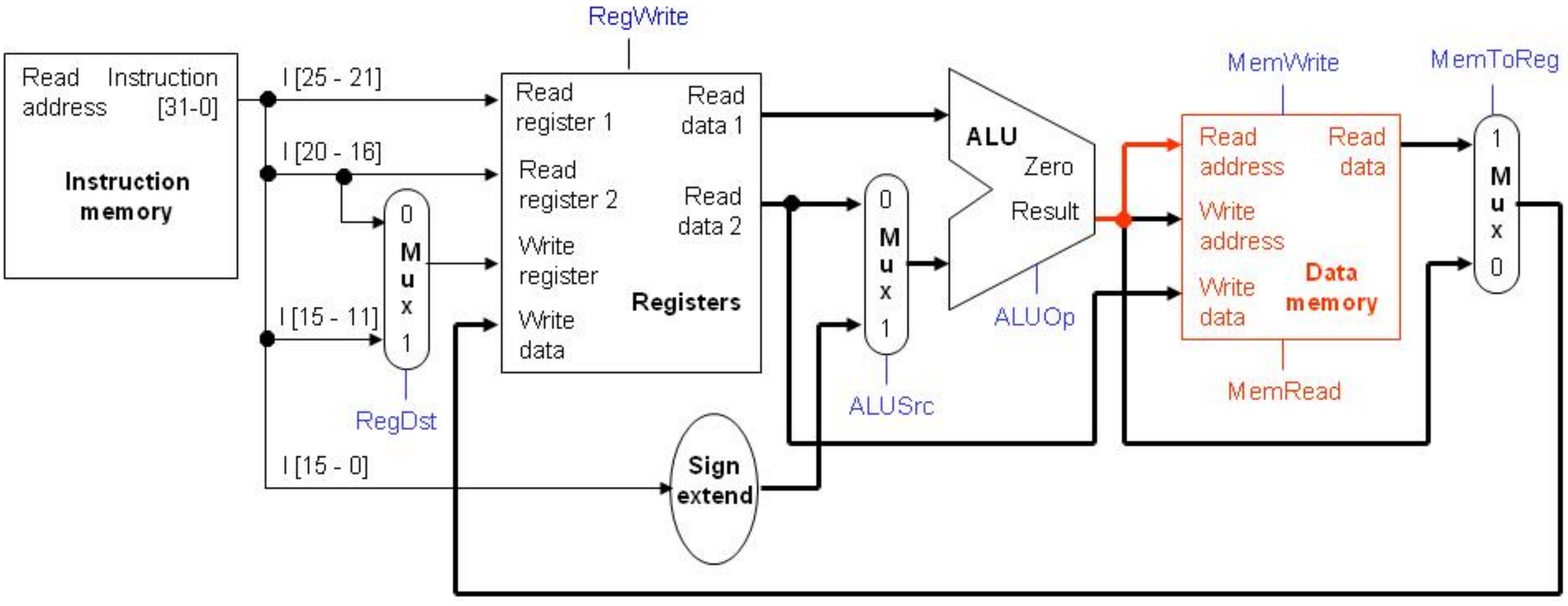
Prováděcí fáze (EX)

- ..totéž platí pro EX ...



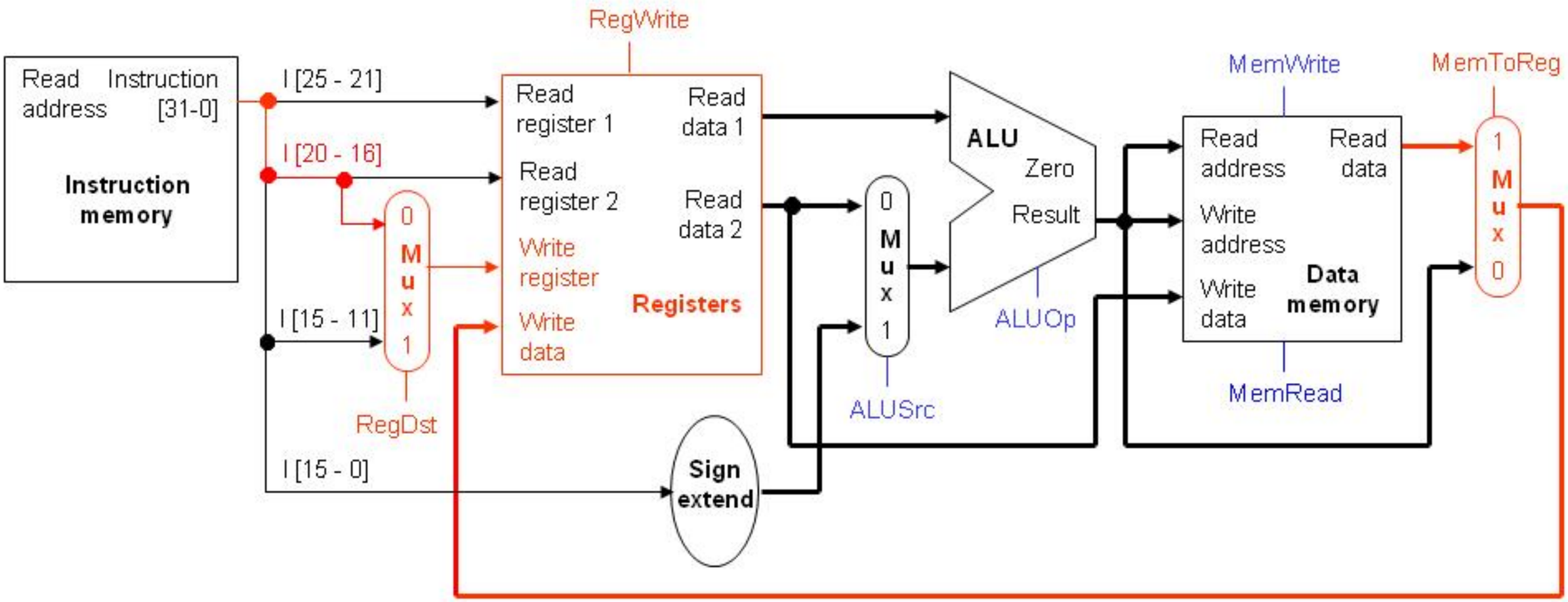
Paměť (MEM)

- … část MEM …



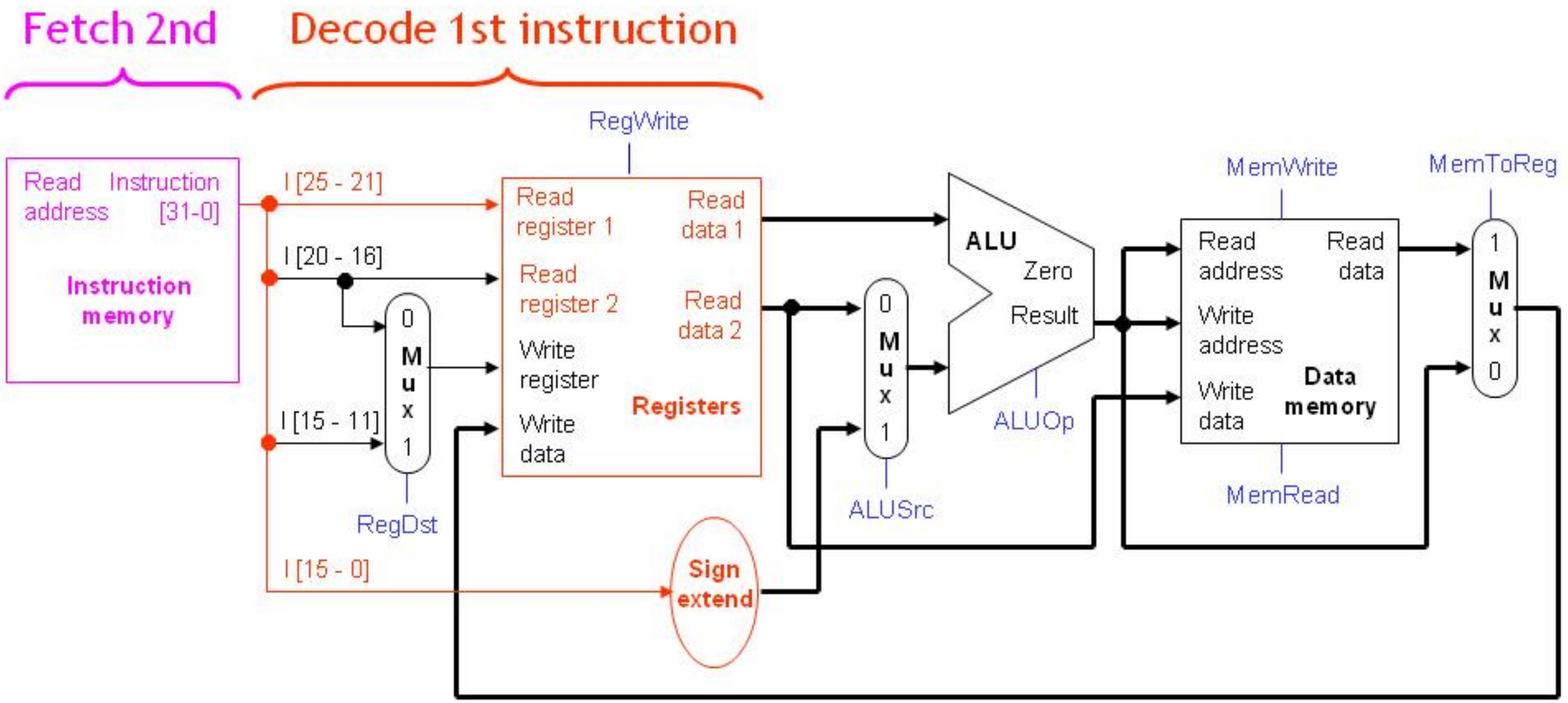
Zápis výsledku (WB)

- … 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.



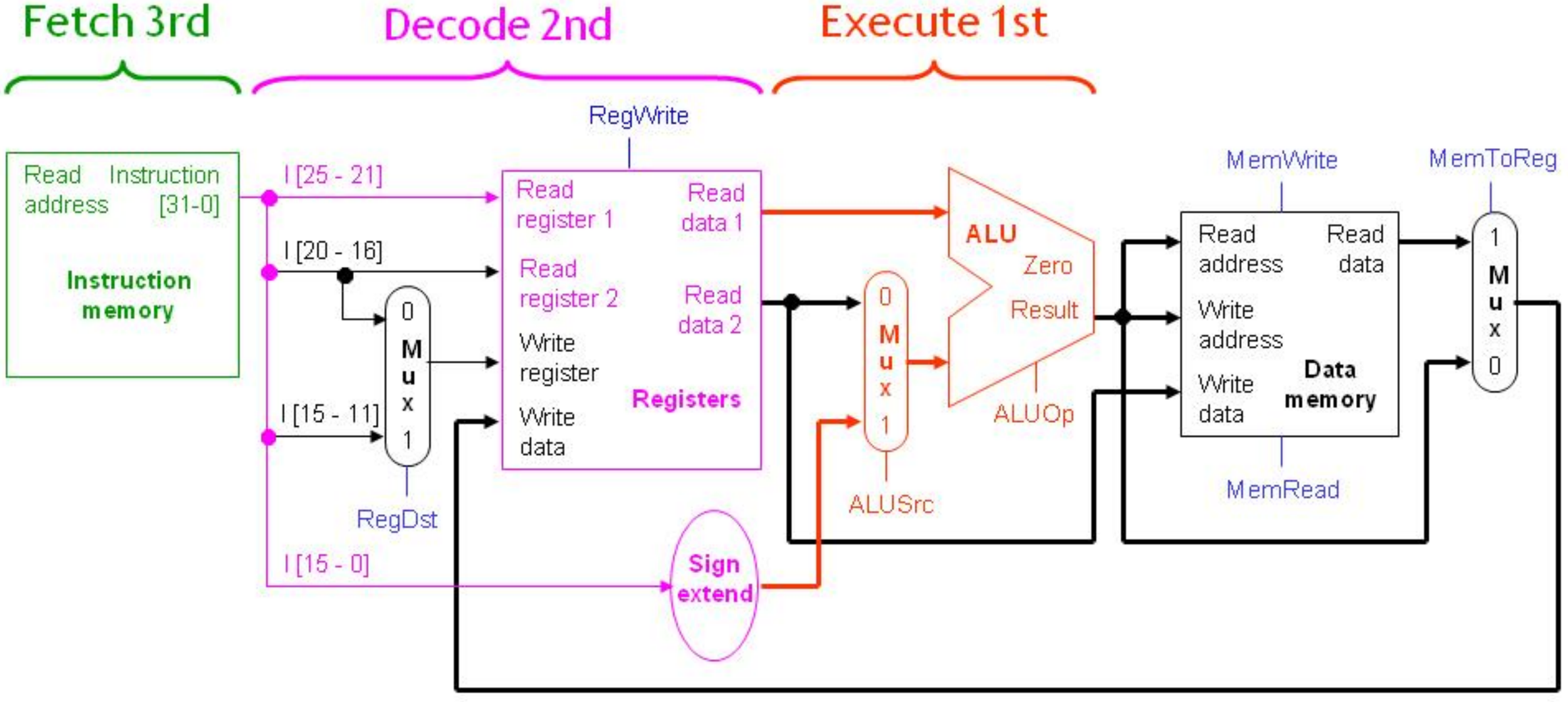
Dekódování a čtení instrukce současně

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



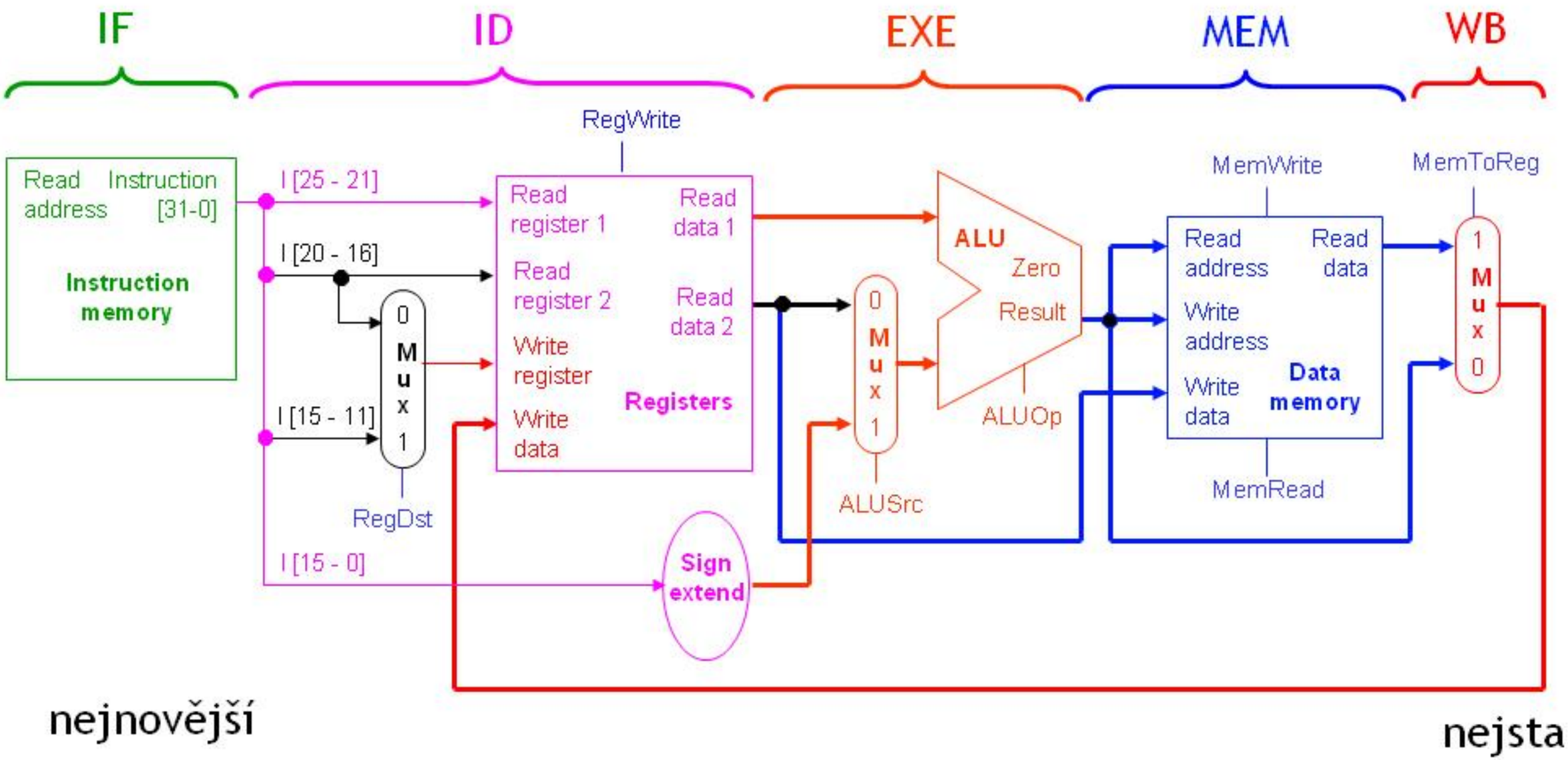
Provádění, dekódování a čtení instrukce

- 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!

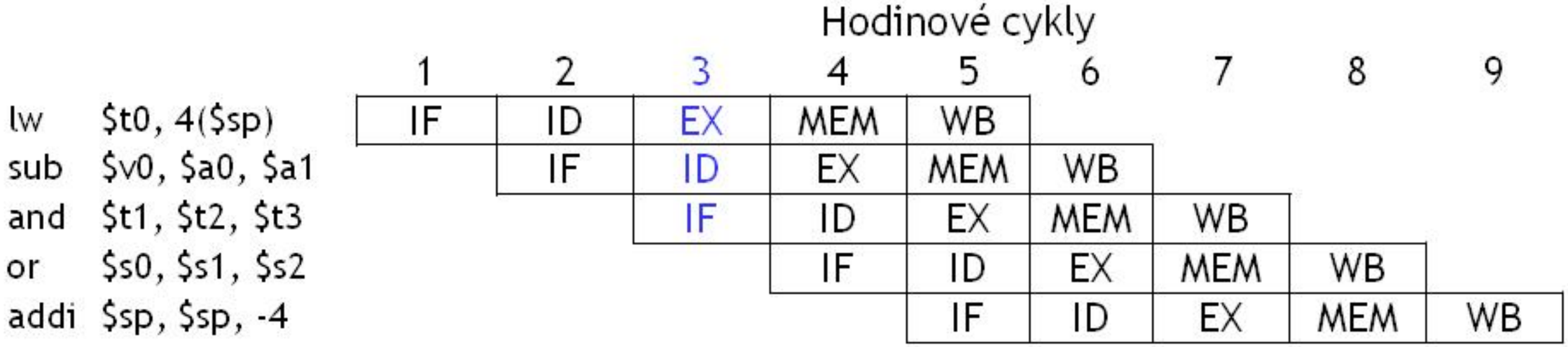


Rozdělení jednotky do 5 stupňů

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

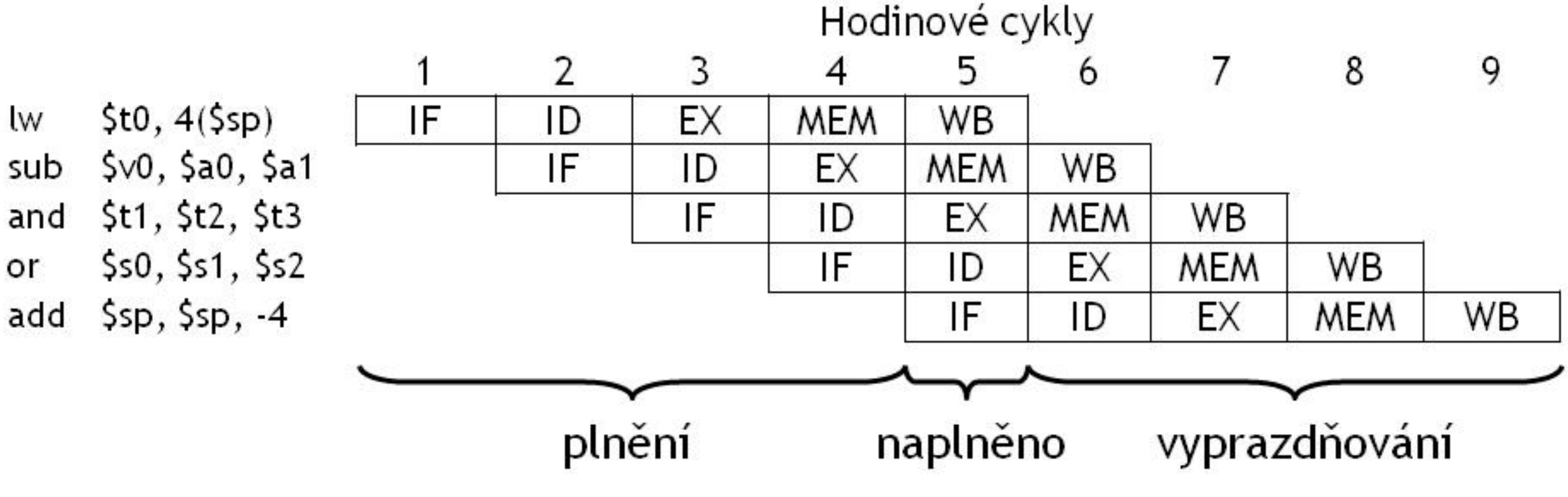


Pipeline diagram



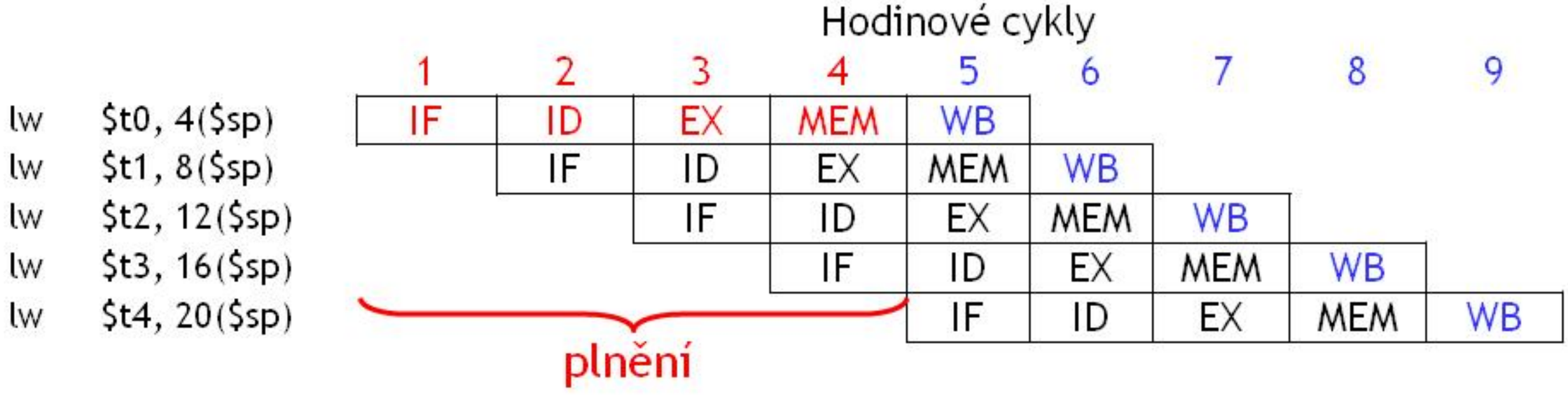
- Pipeline diagram ukazuje provádění série instrukcí
- Posloupnost instrukcí se zobrazuje vertikálně shora dolů
- Hodinové cykly ze zobrazují horizontálně zleva doprava
- Každá instrukce je rozdělena do stupnů
- Názorně ukazuje překrývání instrukcí. Například ve třetím cyklu jsou aktivní tři instrukce.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

Terminologie



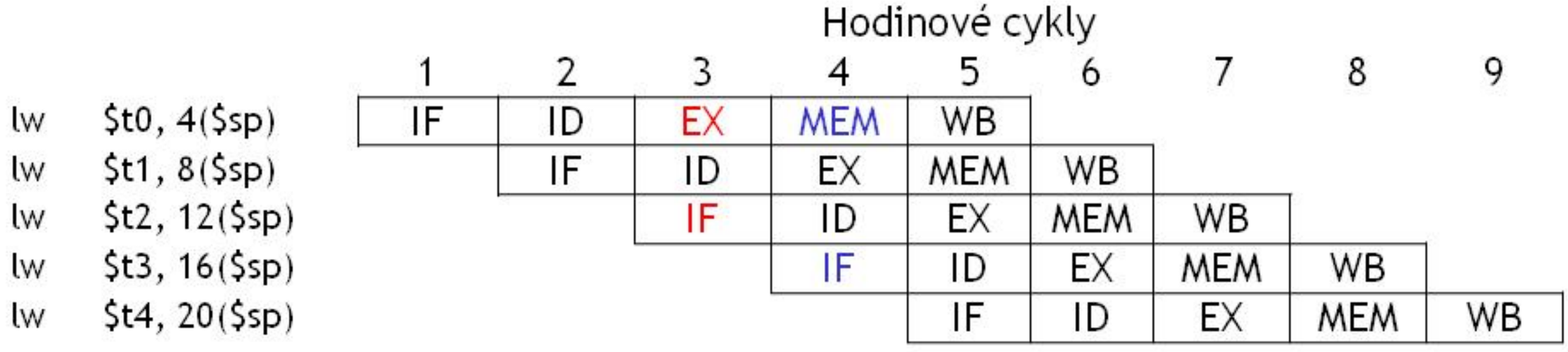
- Hloubka pipeline je počet stupňů, zde pět
- V prvních čtyřech probíhá plnění, jsou zde nevyužité funkční jednotky
- V cyklu 5 je pipeline plná. Probíhá zpracování pěti instrukcí současně, jsou využity všechny hardwarové jednotky
- V cyklech 6-9 se pipeline vyprazdňuje

Výkon pipeline struktury



- 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 ?

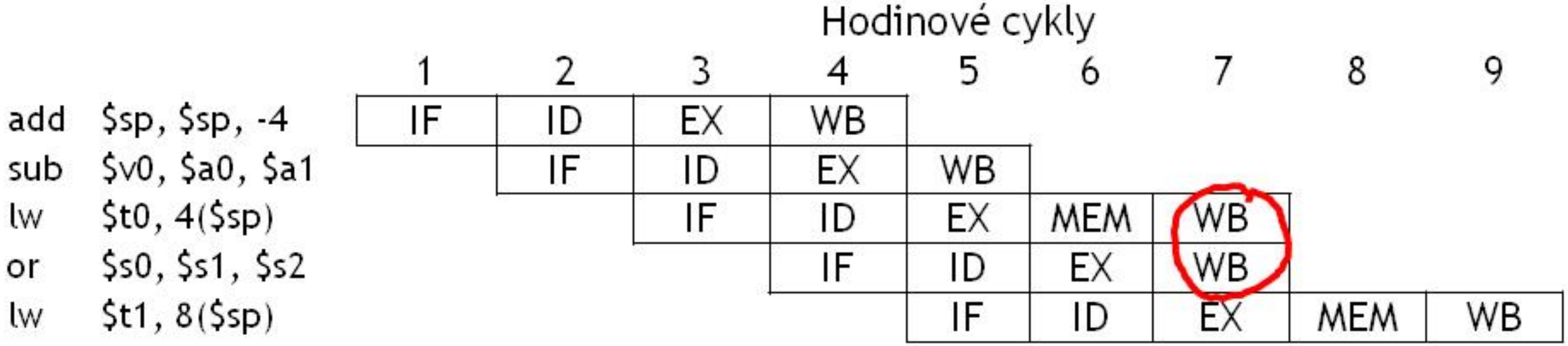
Datové cesty pro pipelinning: Požadavky



- 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 (instruční paměť a datová paměť)

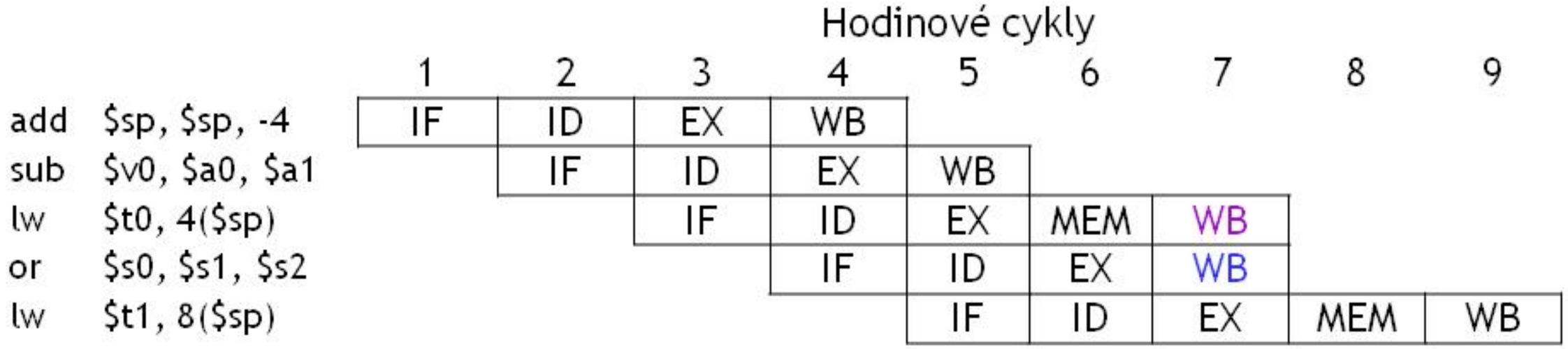
Pipelining u dalších instrukcí

- 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?



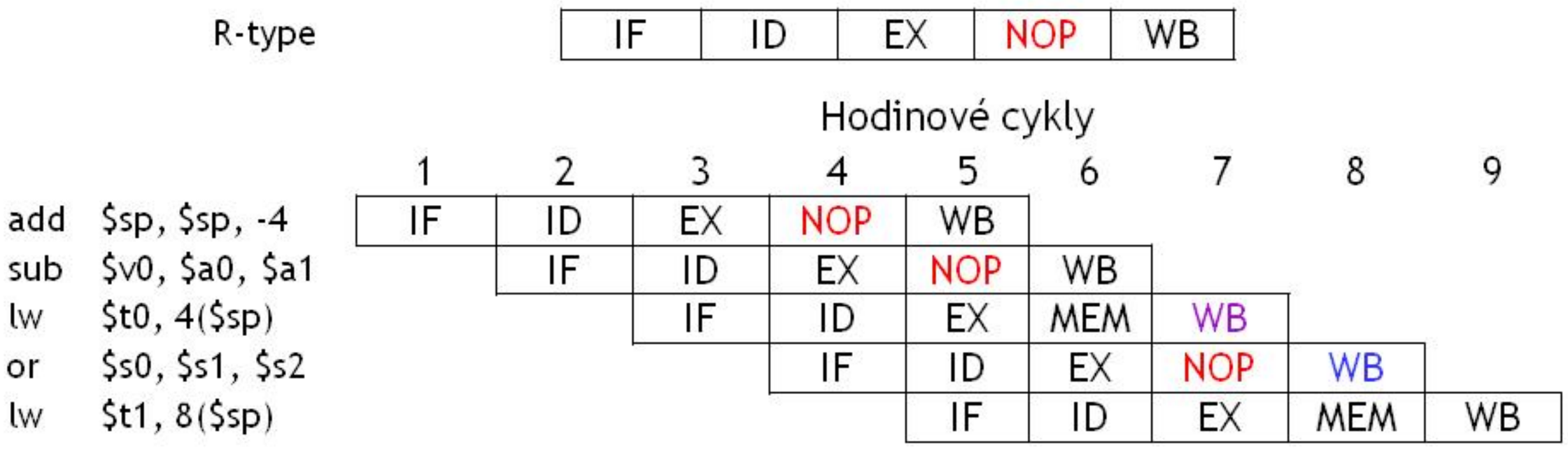
Konstatování

- 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

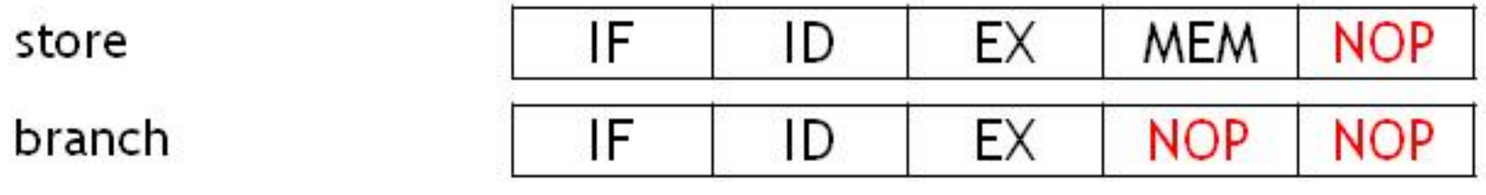


Řešení: Vložení NOP

- 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



- Zápisy a větvení mají NOPy …



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

Úvod do organizace počítače

Pipelining

Provádění a řízení výpočtu ve struktuře pipeline

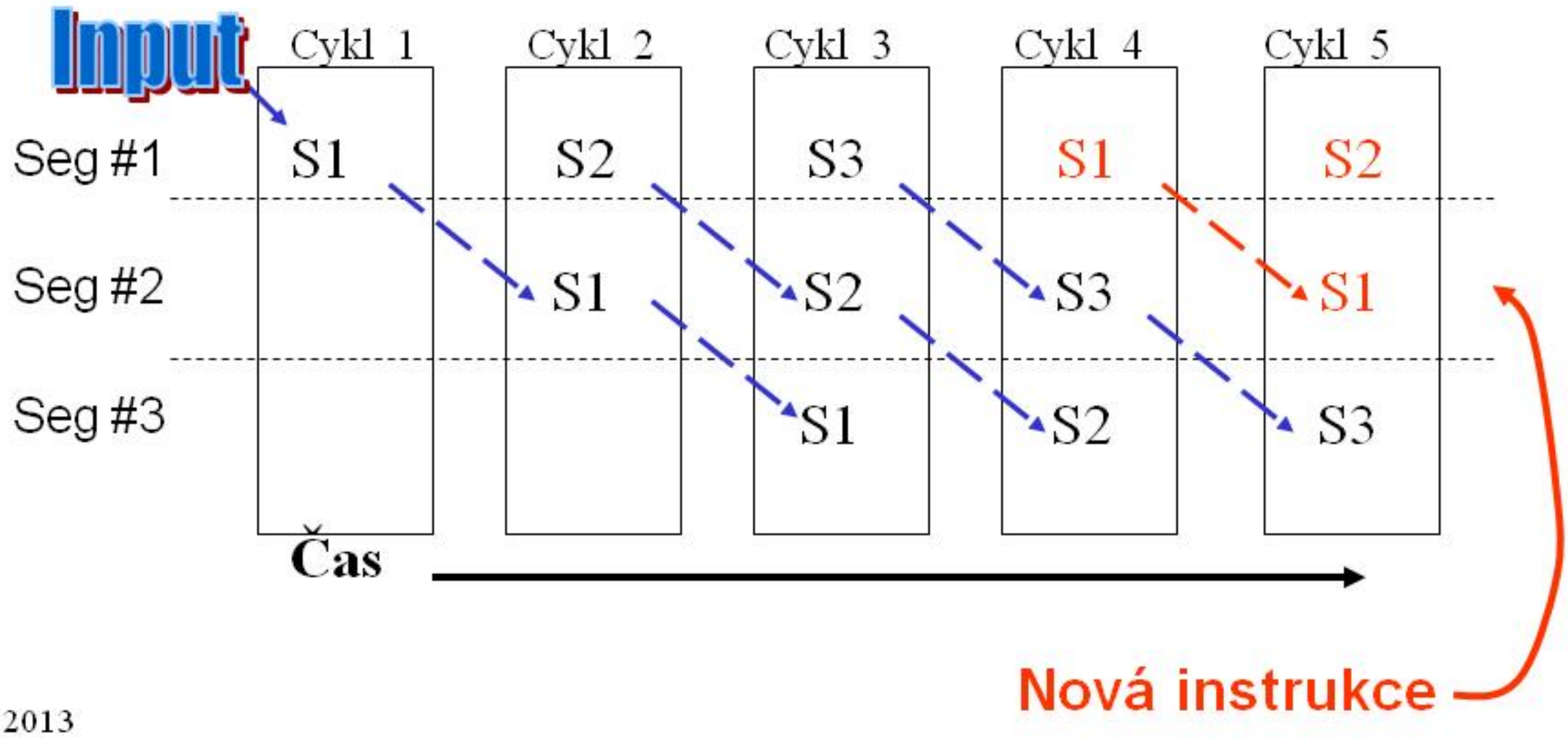
Řízení pipeline a hazardy

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"

Příklad pipeliningu

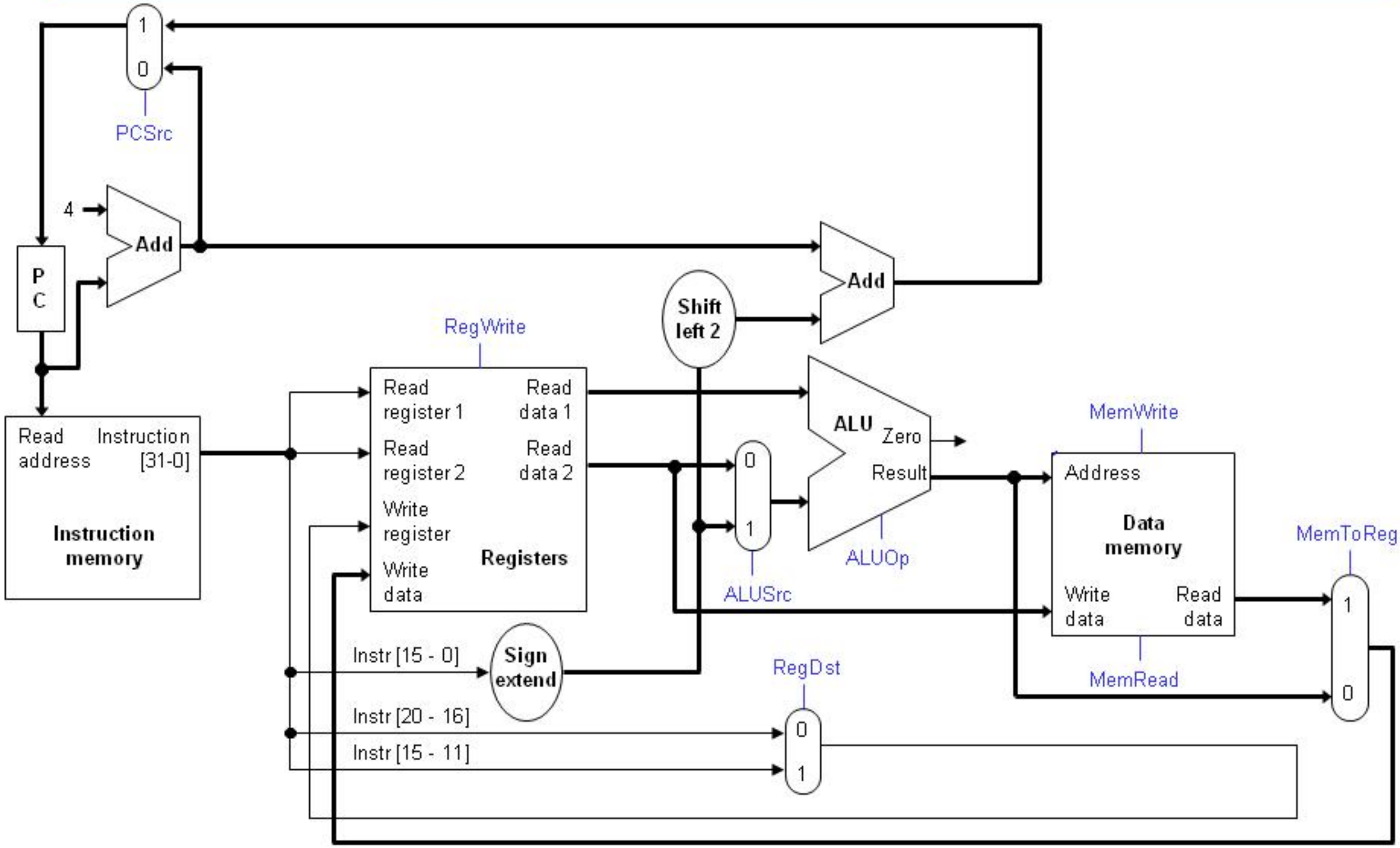
- Předpoklad: Jeden instrukční formát (snadné)
- Předpoklad: Každá instrukce má 3 kroky S1..S3
- Předpoklad: Pipeline má 3 segmenty (jeden/krok)



Návrh ISA pro pipelining

- 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

Opakování: Jednoduché datové cesty



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.
  - Řídící signály jsou stejné.
- Pouze byly provedeny některé kosmetické úpravy tak, aby se zmenšilo schéma.
  - Byla vynechána návěští a zmenšeny multiplexery.
  - Datová paměť má pouze jeden vstup Address. Aktuální operaci paměti určují řídící signály MemRead a MemWrite.
- Byl vytvořen prostor pro vložení *pipeline registrů*.

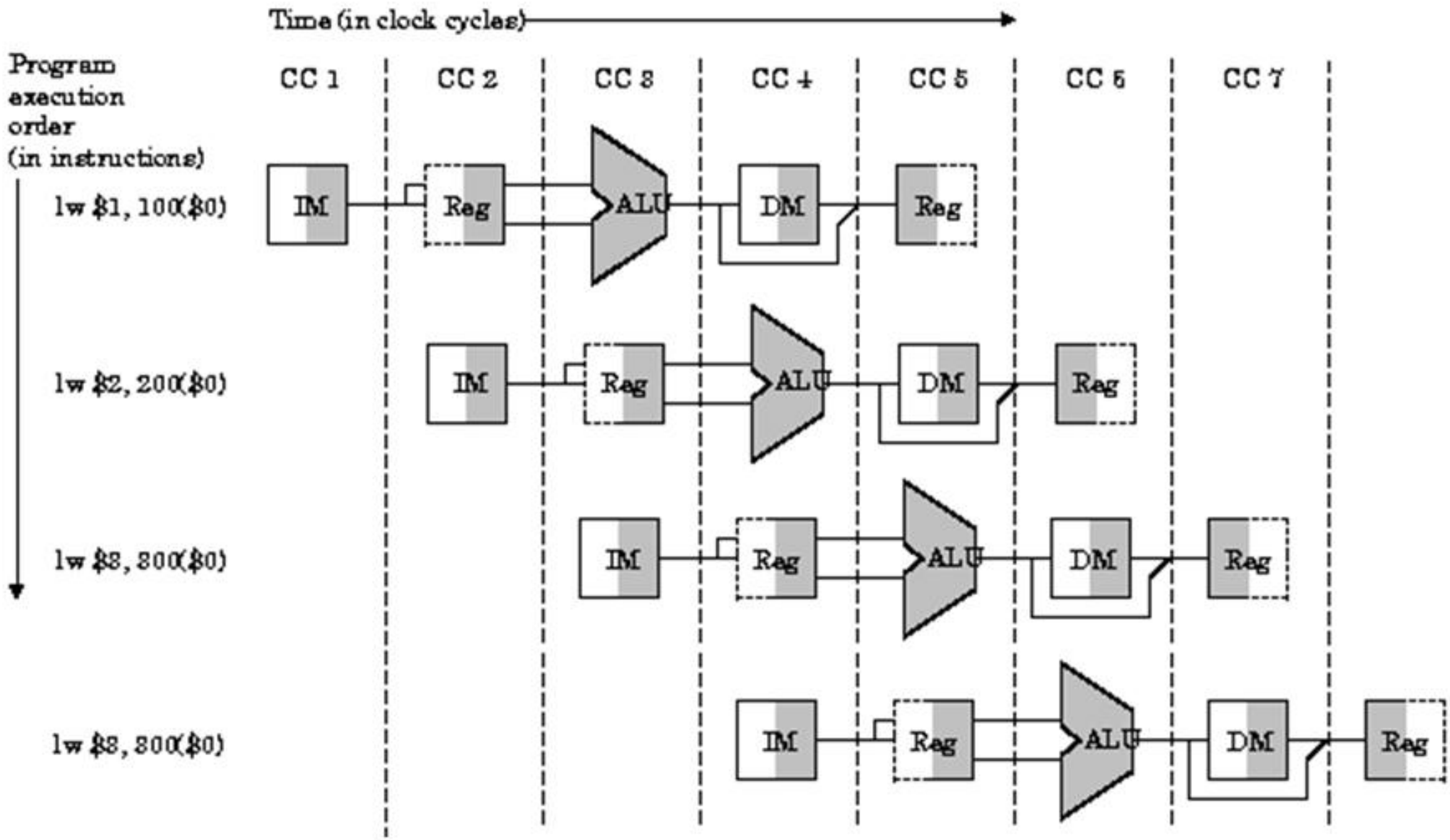
Pipeline registry

- 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.

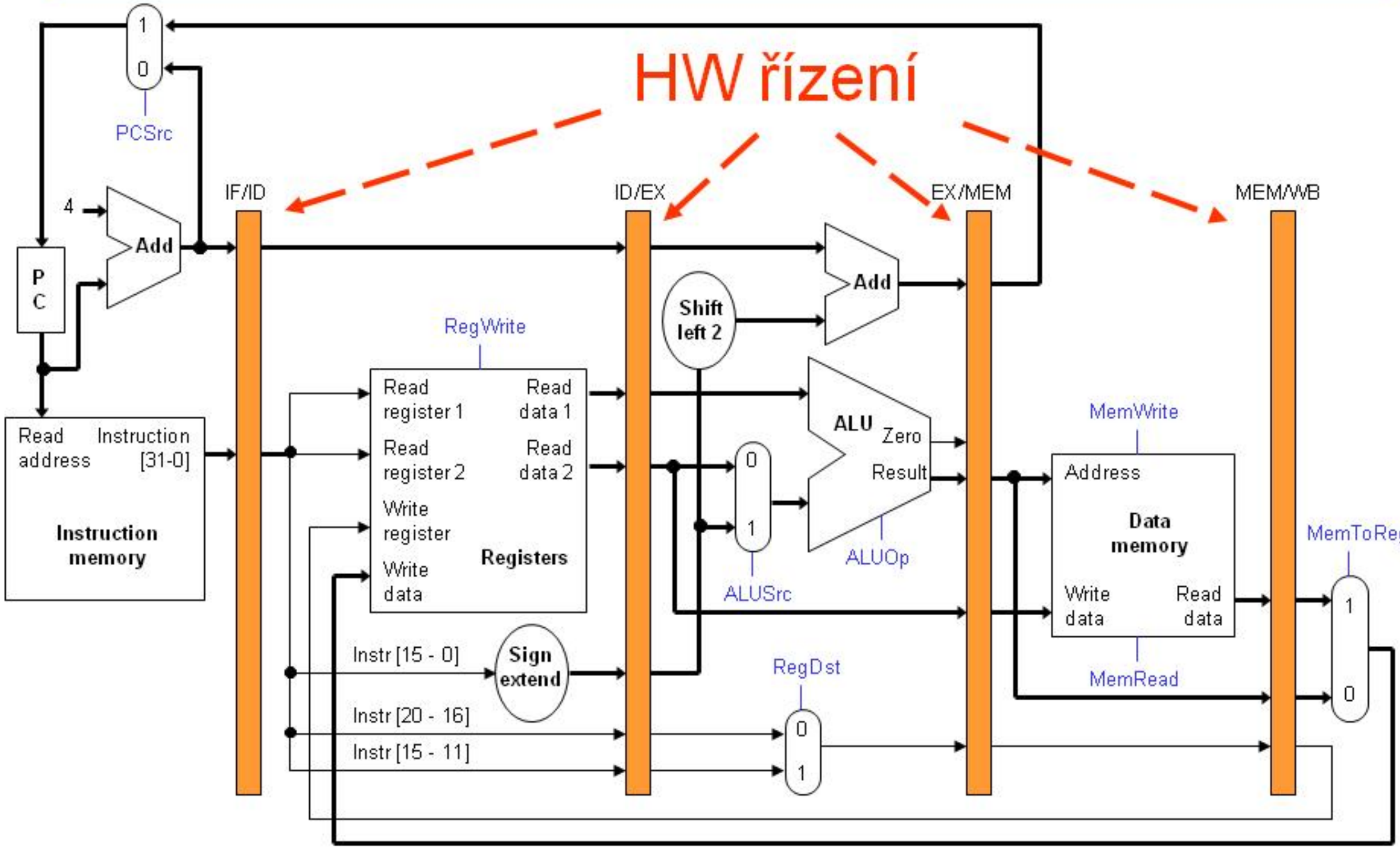
IF/ID ID/EX EX/MEM MEM/WB

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

Výpočetní struktura – uspořádání pipeline



Jednotka - režim pipeline



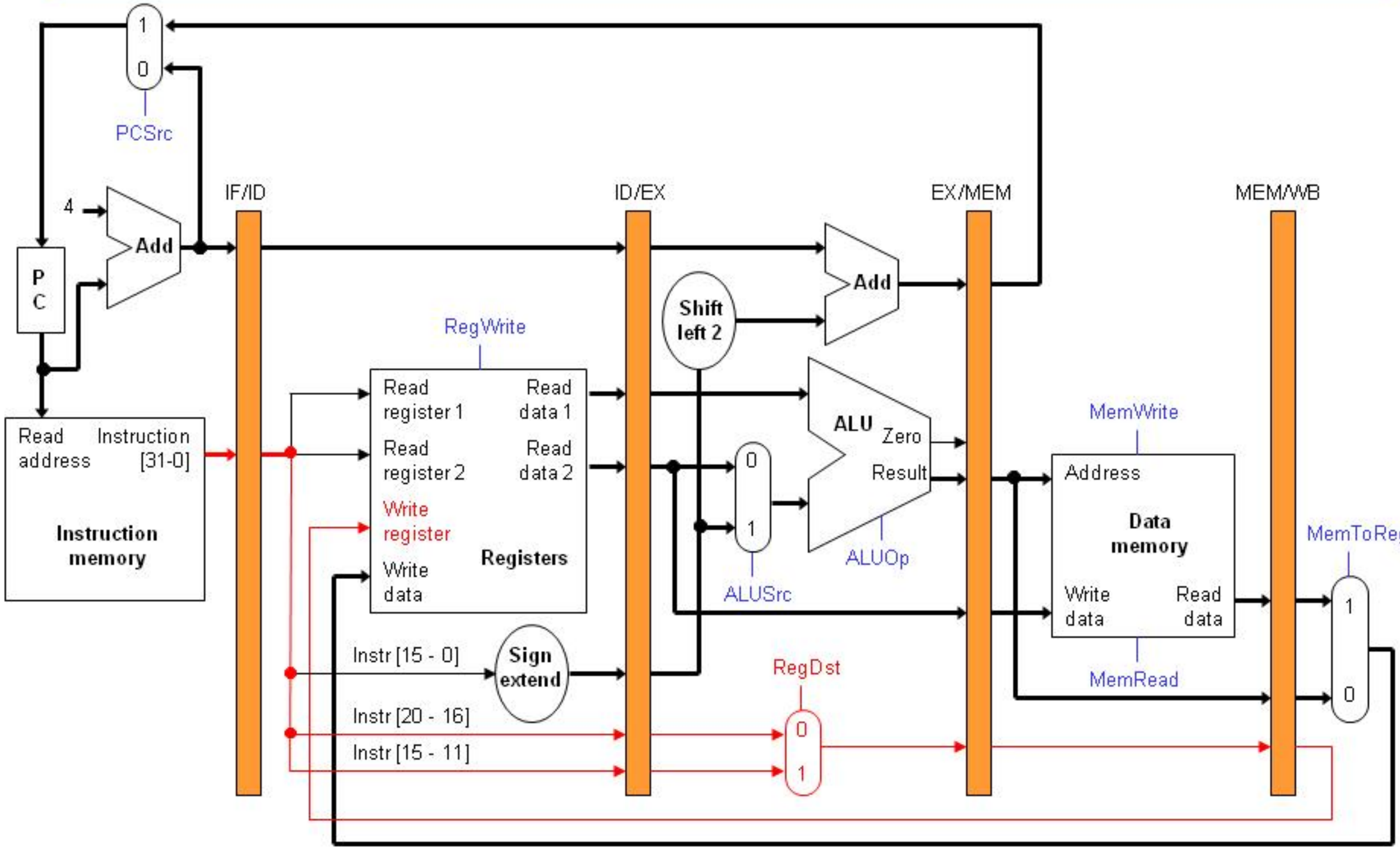
Přesun rozpracovaných dat

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.

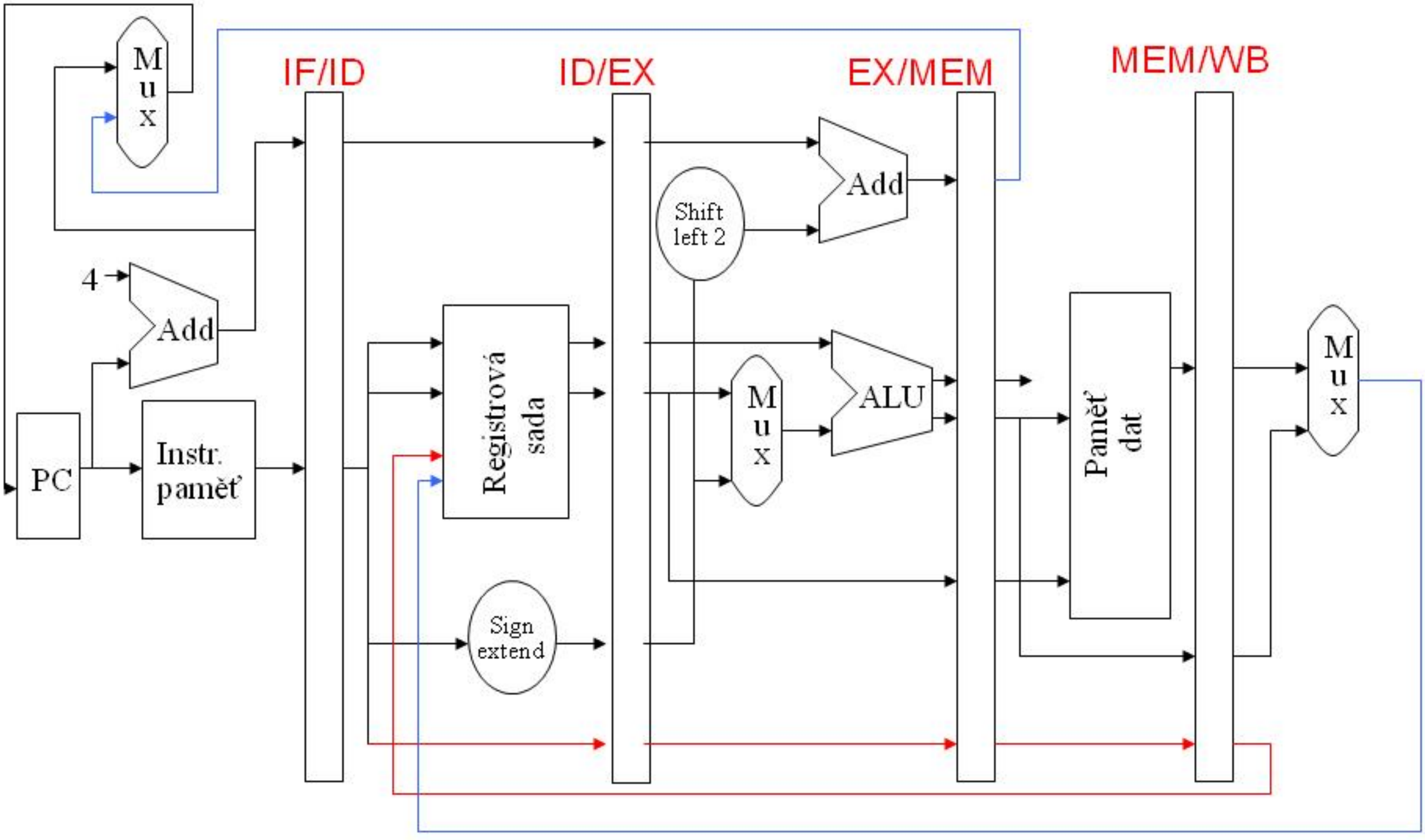
Registr adresy výsledku



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

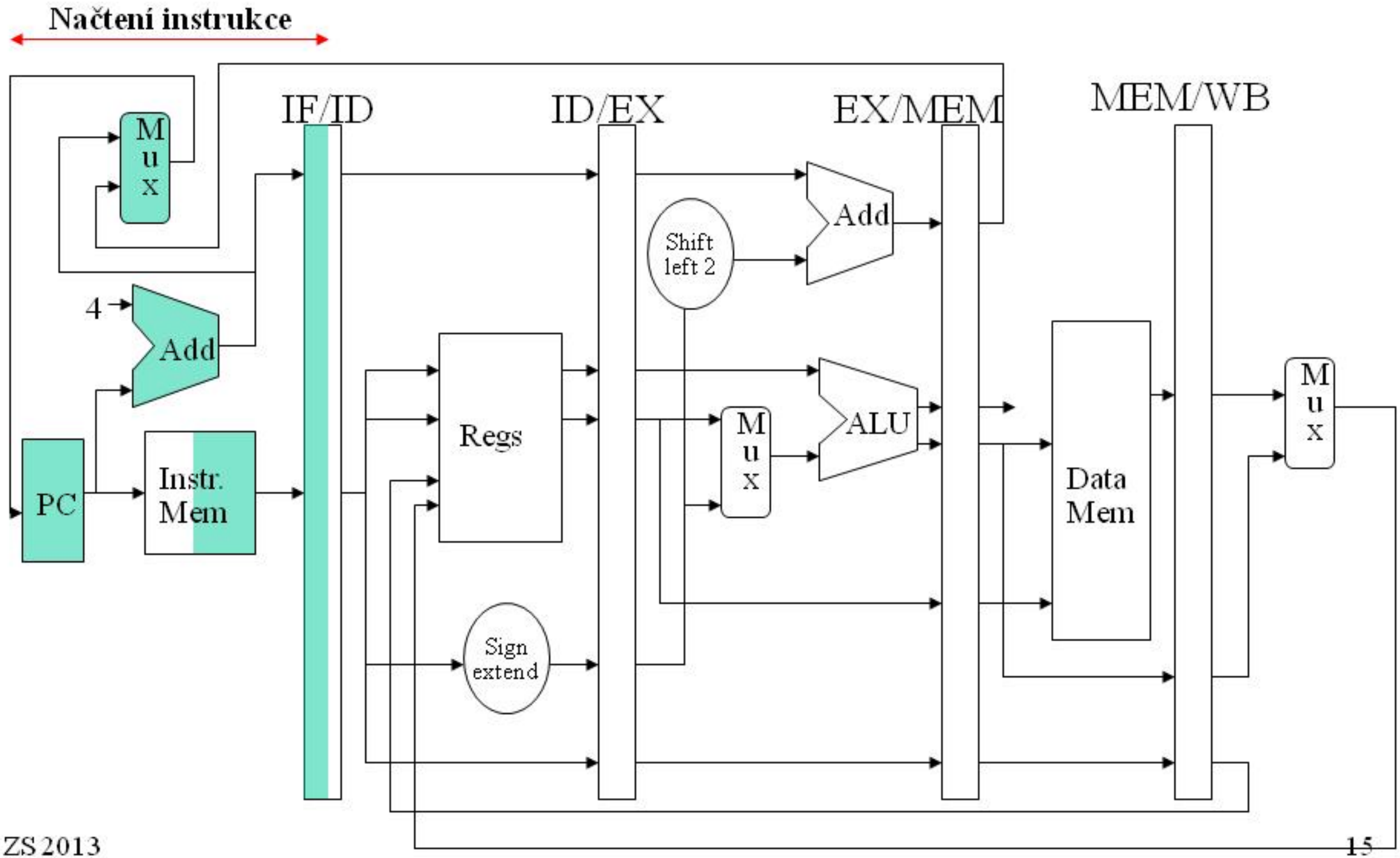


Pipeline u MIPS

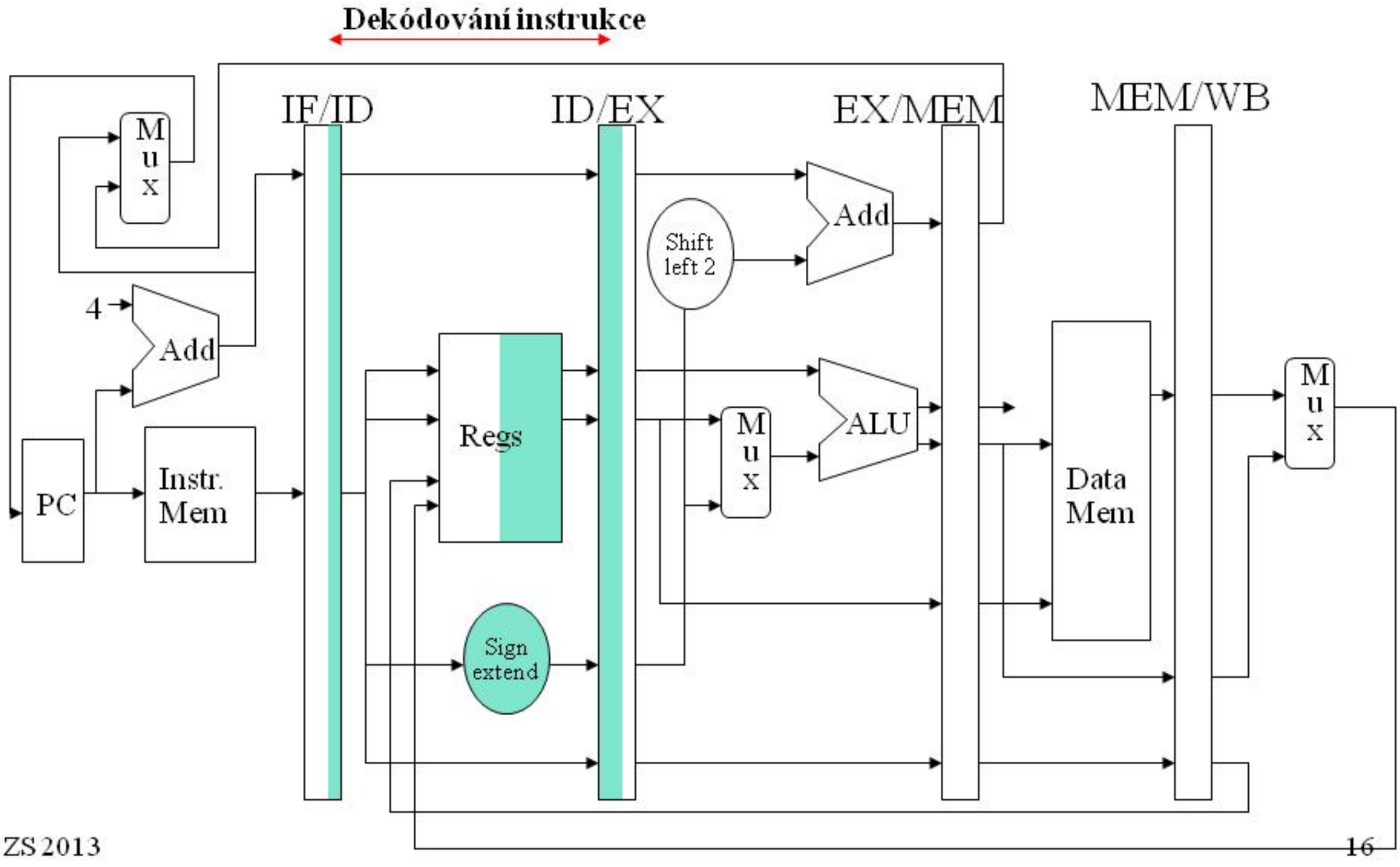
- Podmnožina MIPS
  - *Přístup do paměti*: lw a sw
  - *Aritmetické a logické instrukce*: and, sub, and, or, slt
  - *Instrukce větvení*: beq

- 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

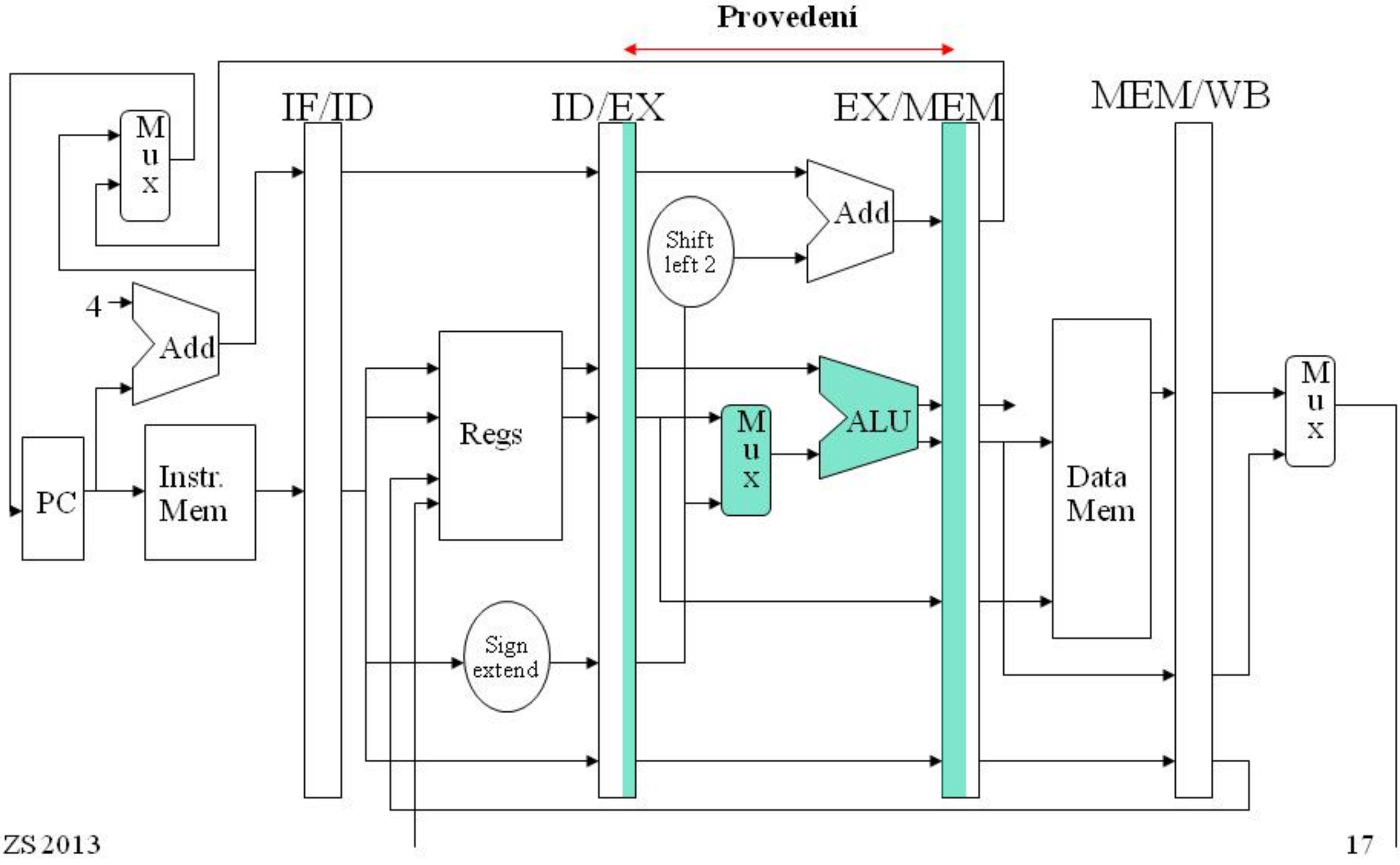
Provedení lw (1/5)



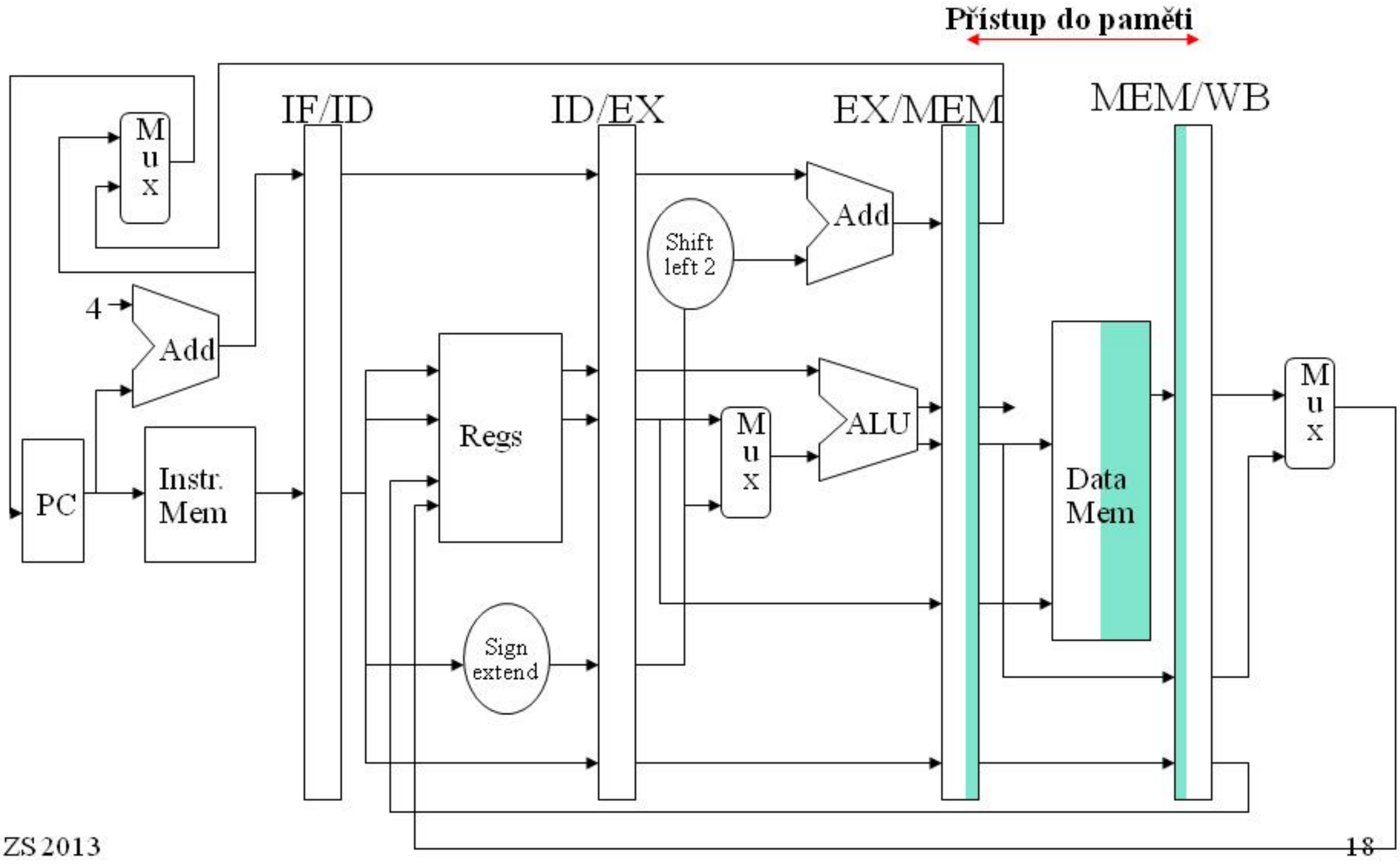
Provedení lw (2/5)



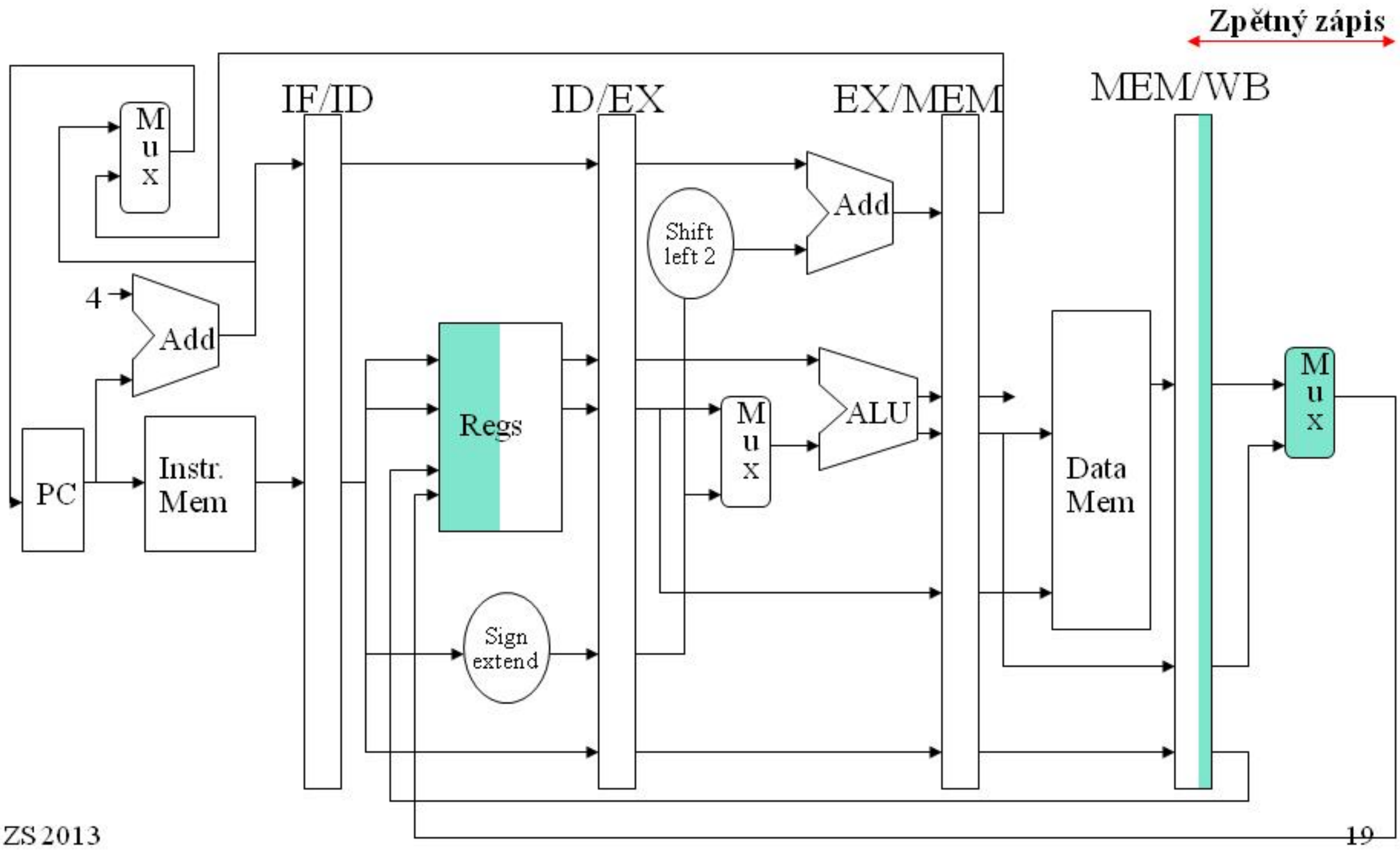
Provedení lw (3/5)



Provedení lw (4/5)



Provedení lw (5/5)



Zdroje použité pro vykonání lw

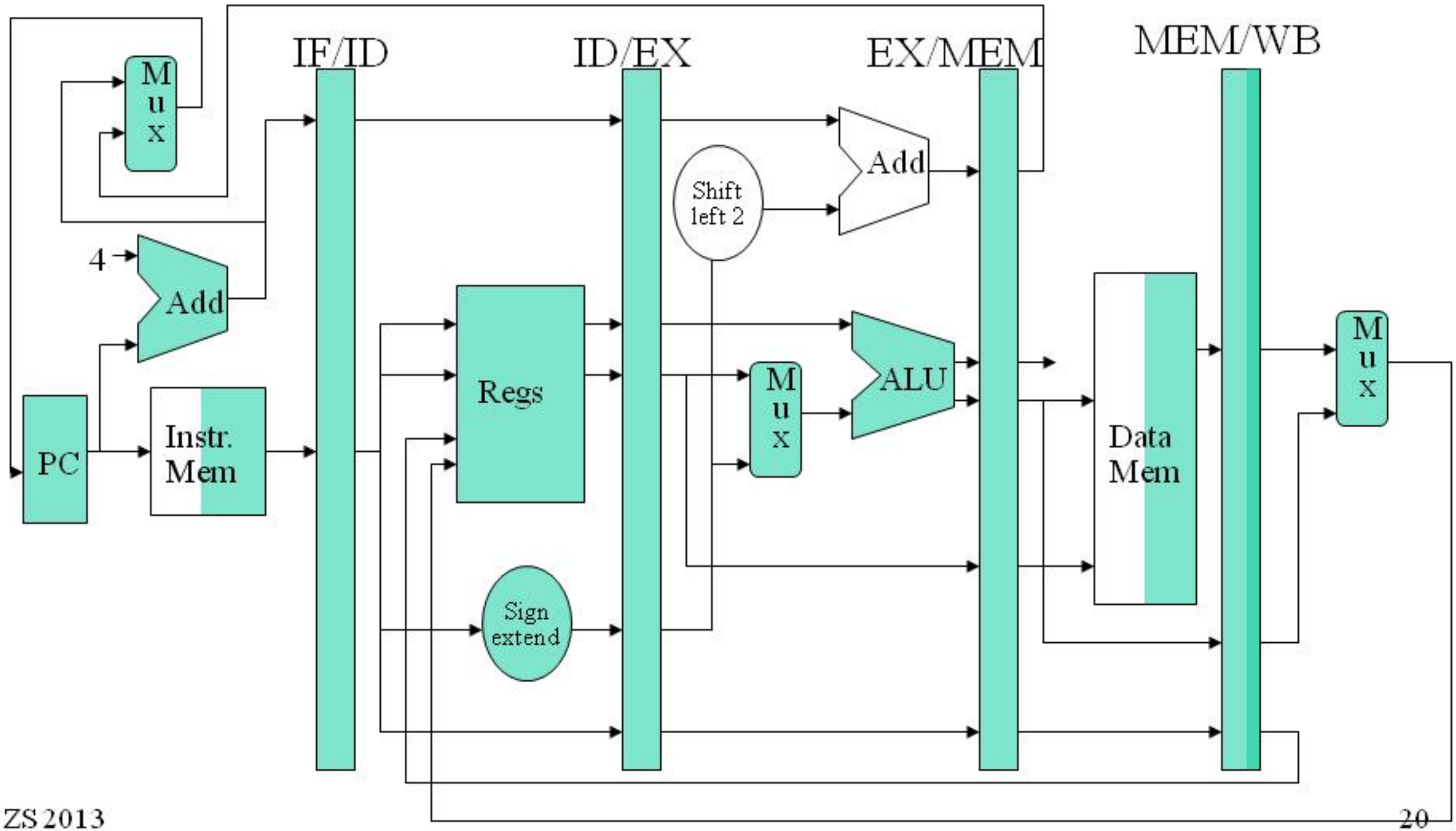
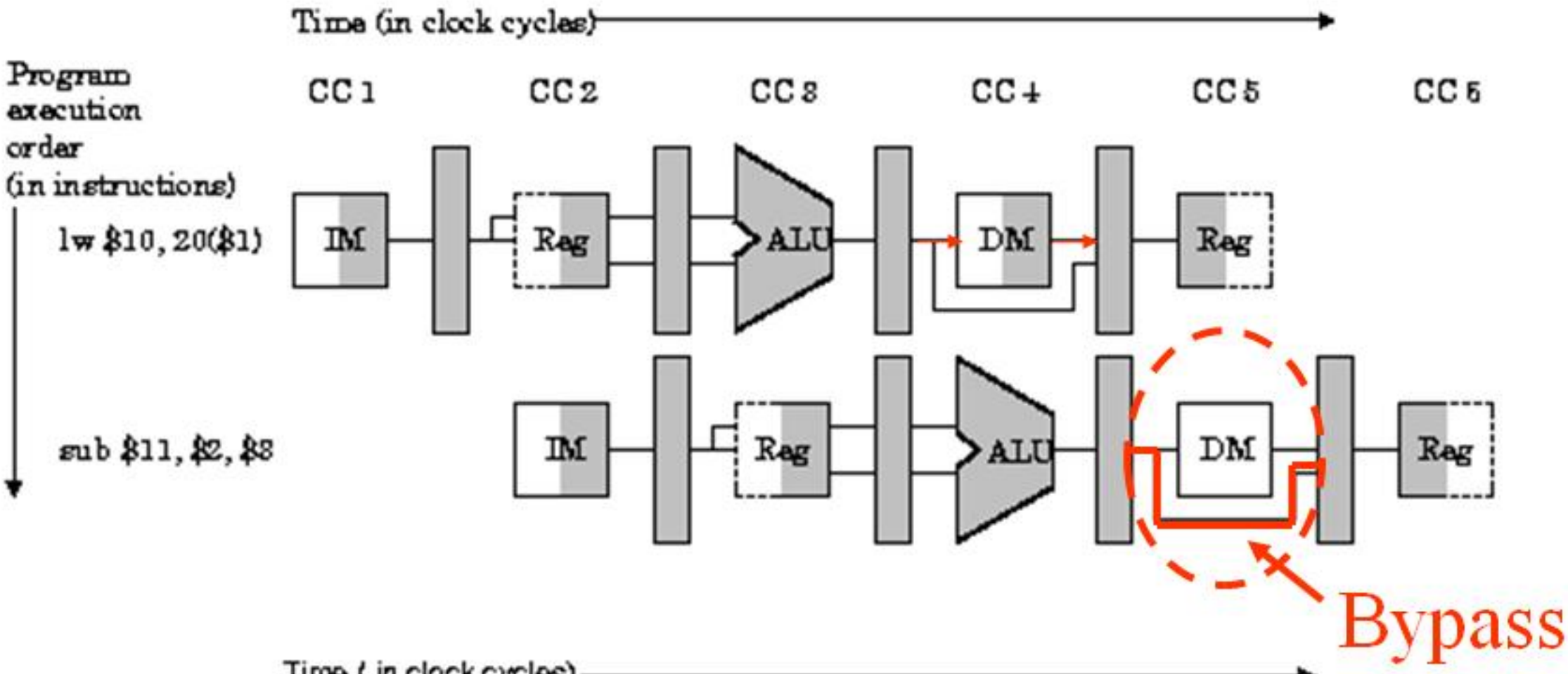


Diagram – časový rozvoj



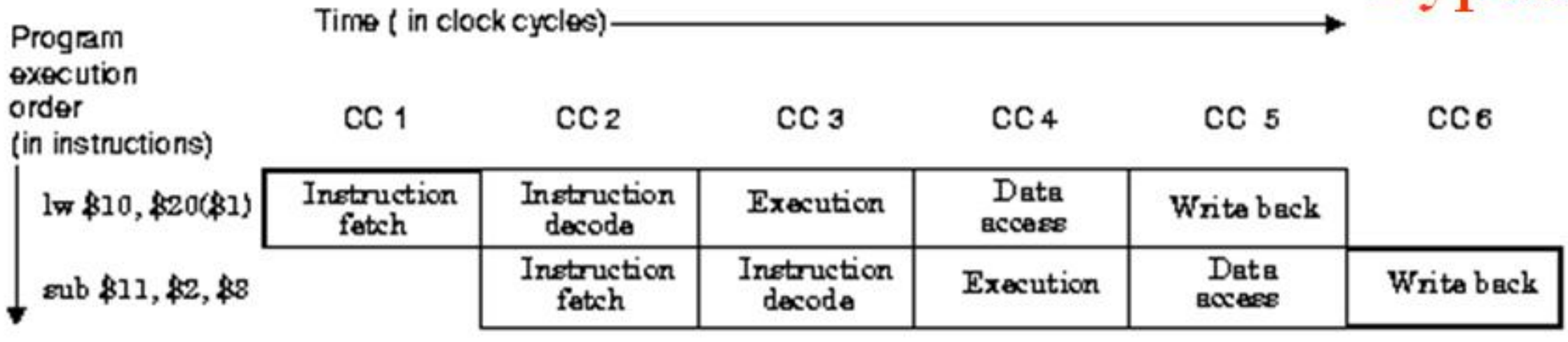


Diagram jednoho cyklu (cykl 1)

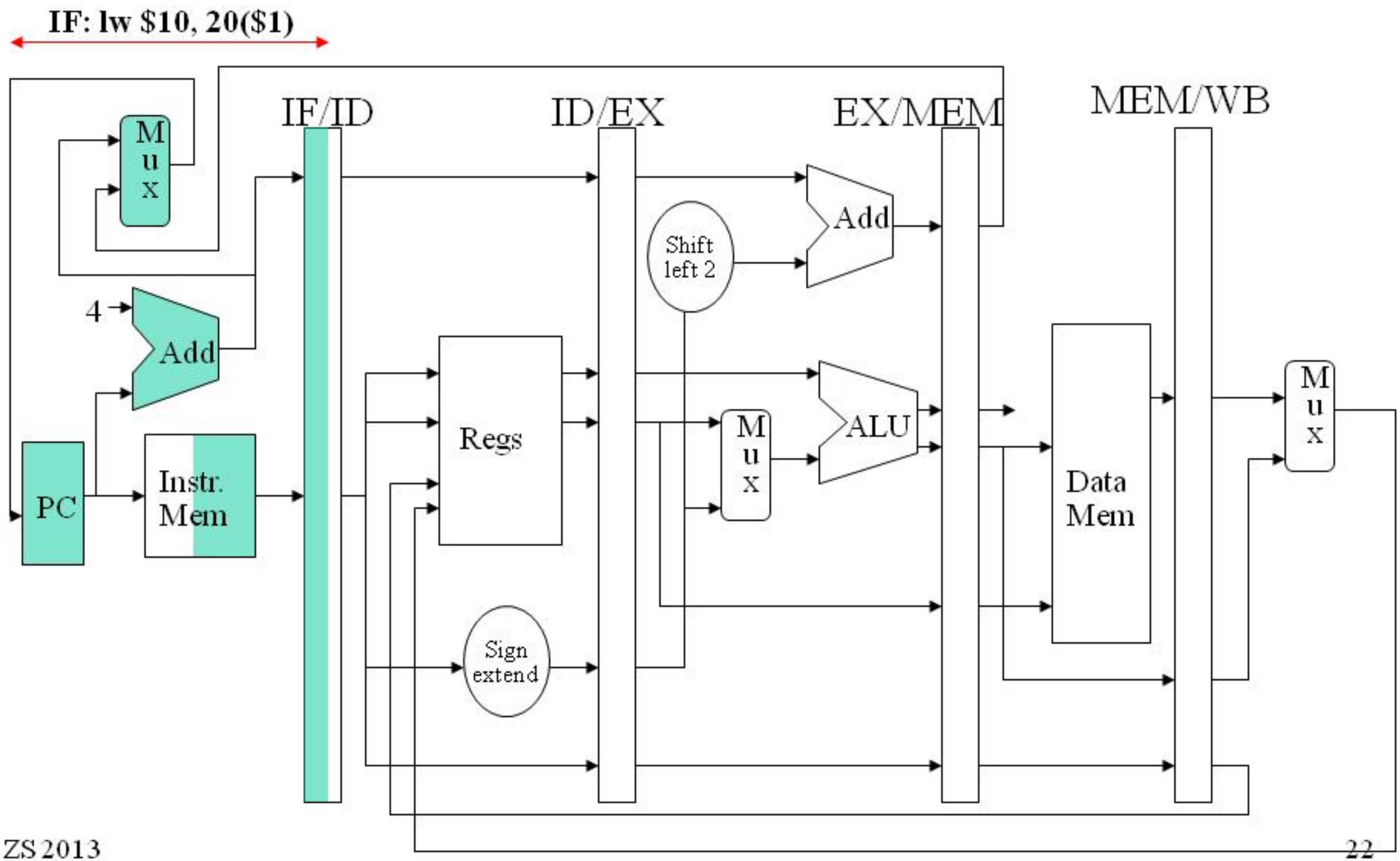


Diagram jednoho cyklu (cykl 2)

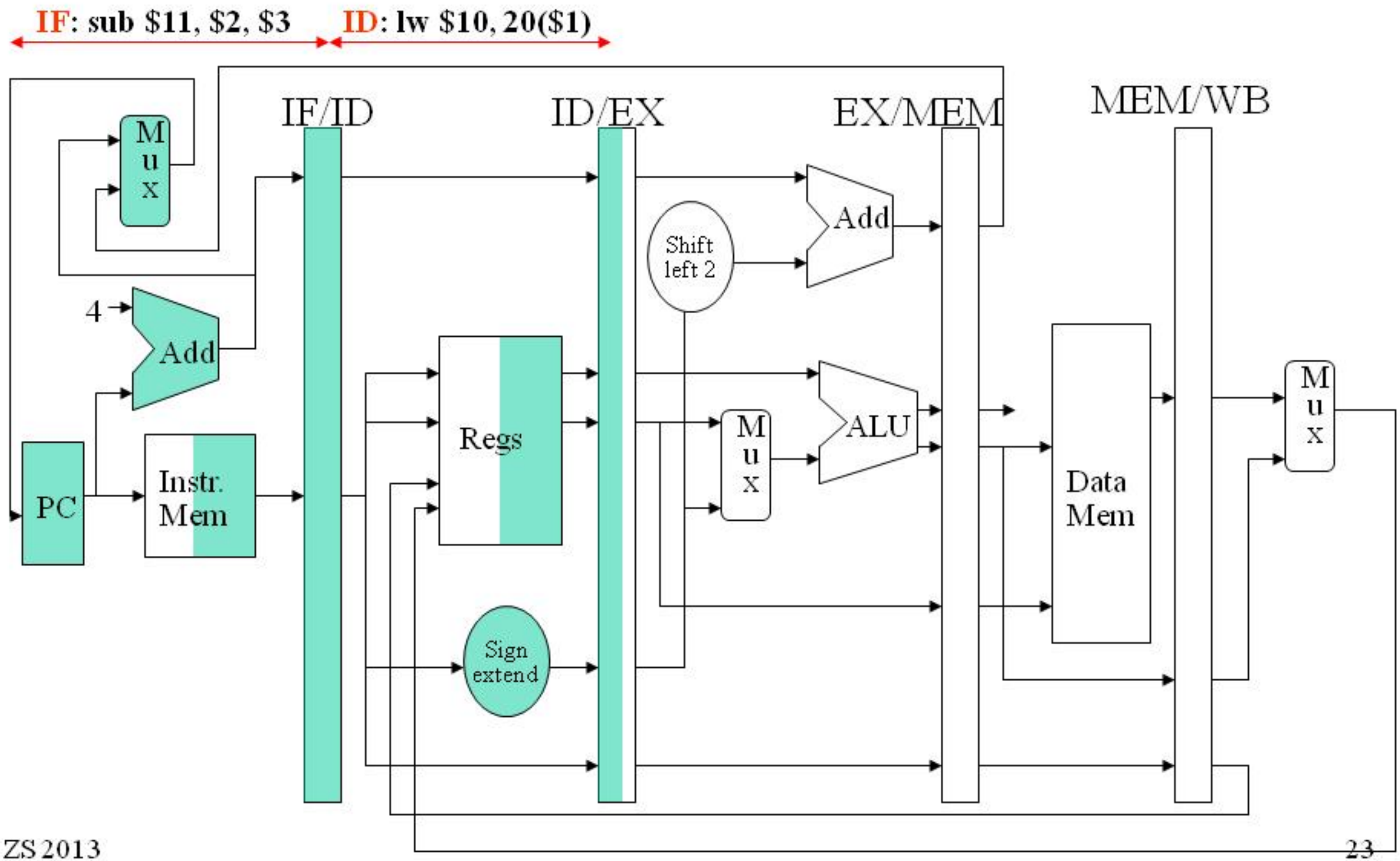


Diagram jednoho cyklu (cykl 3)

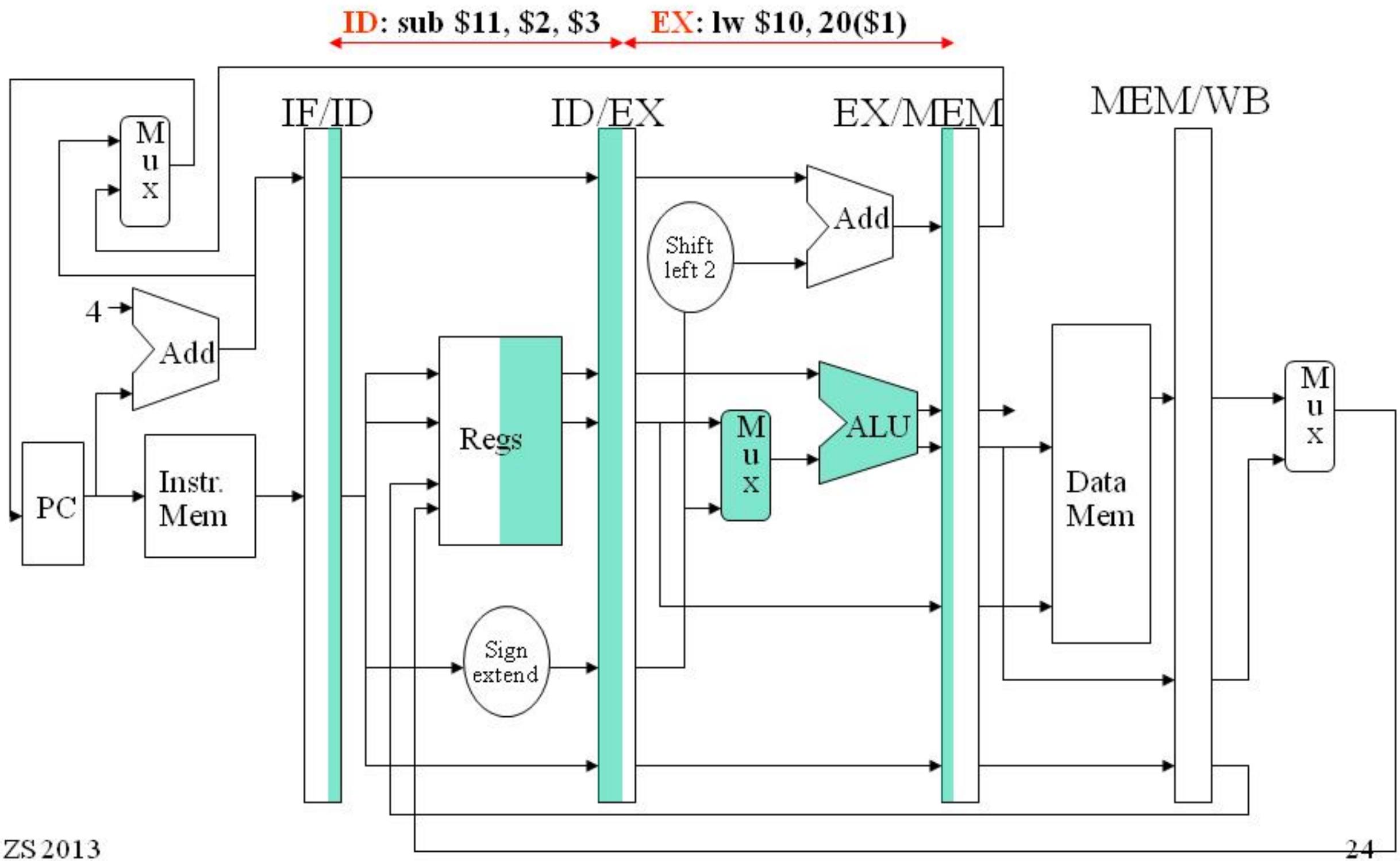


Diagram jednoho cyklu (cykl 4)

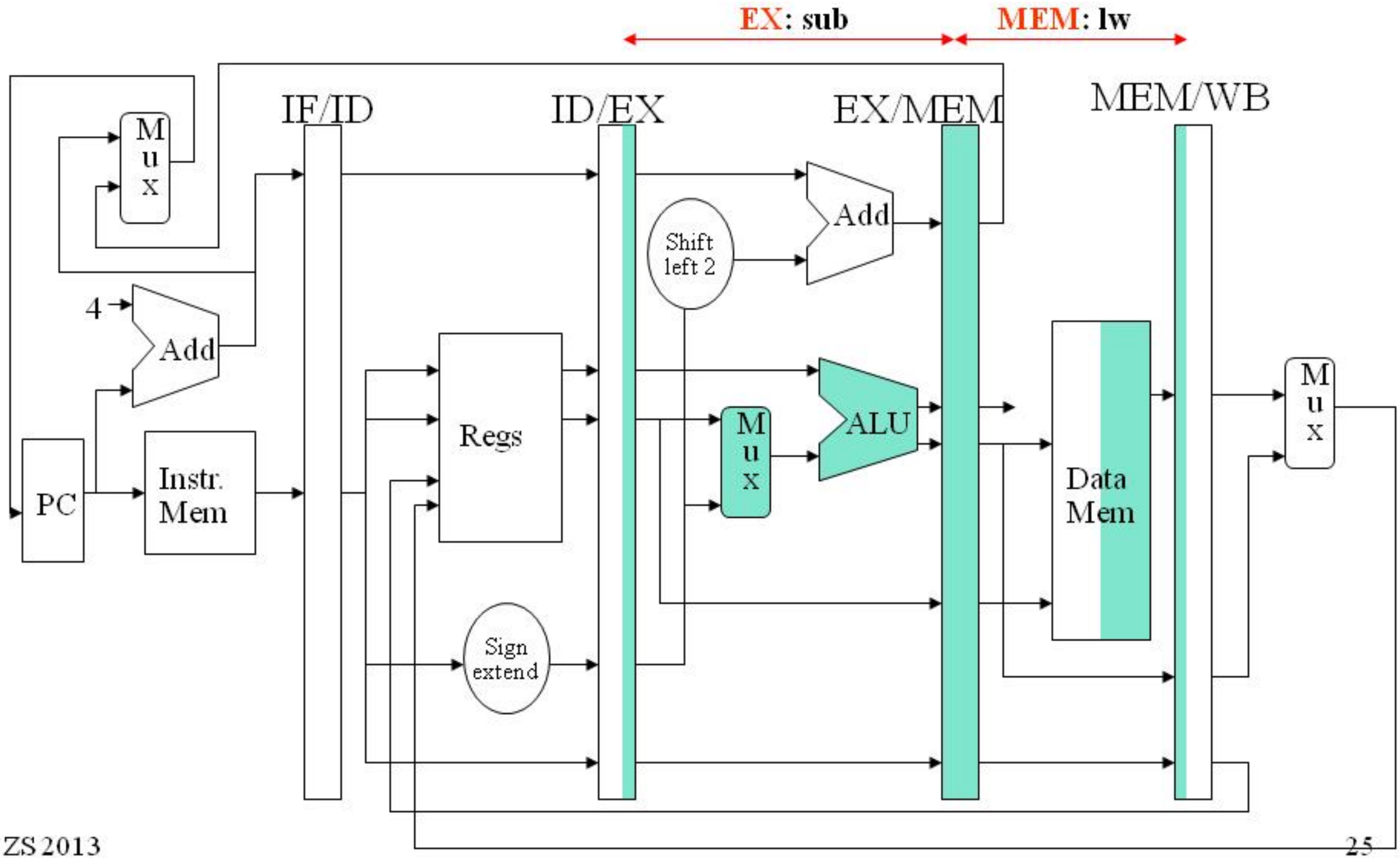


Diagram jednoho cyklu (cykl 5)

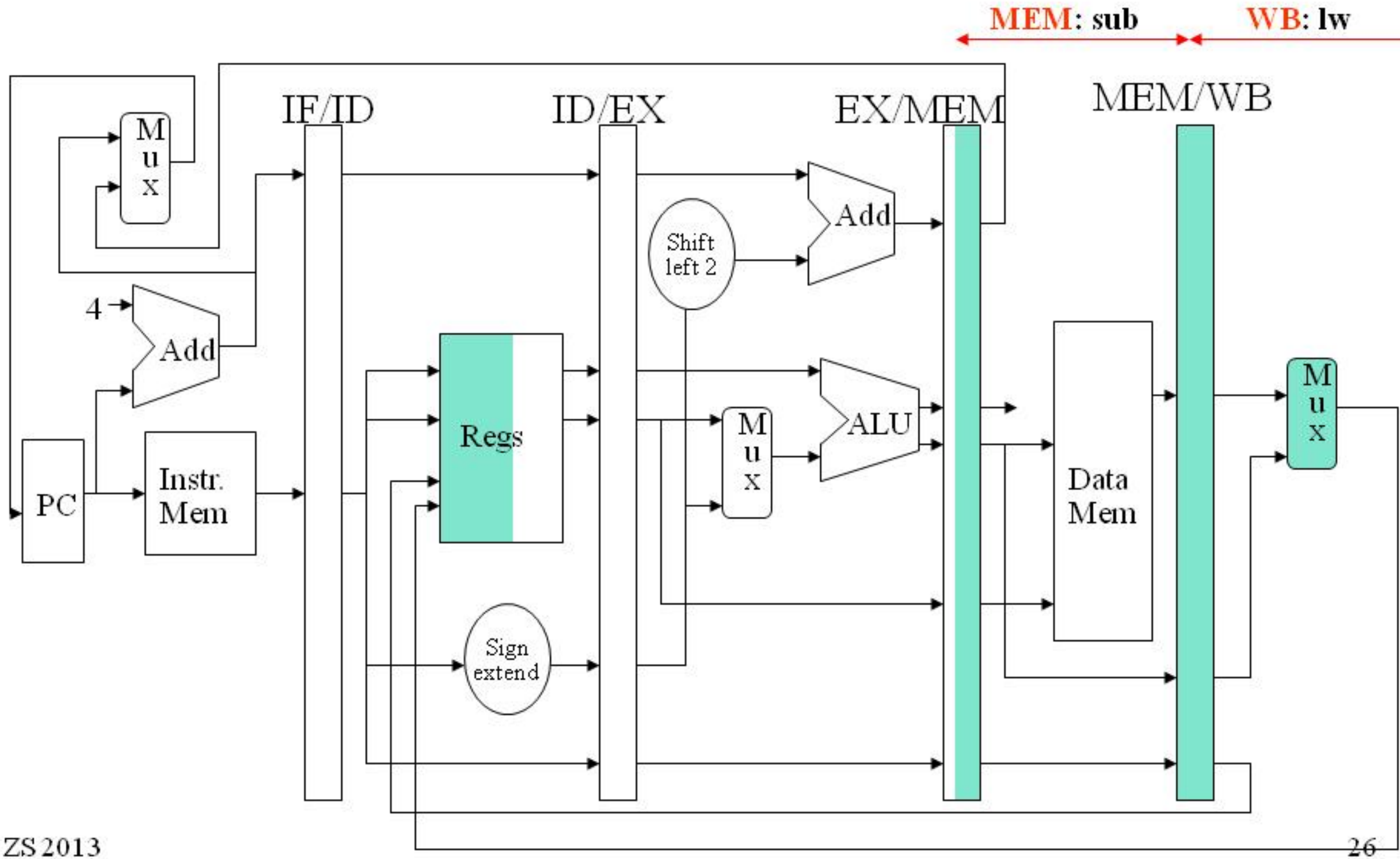
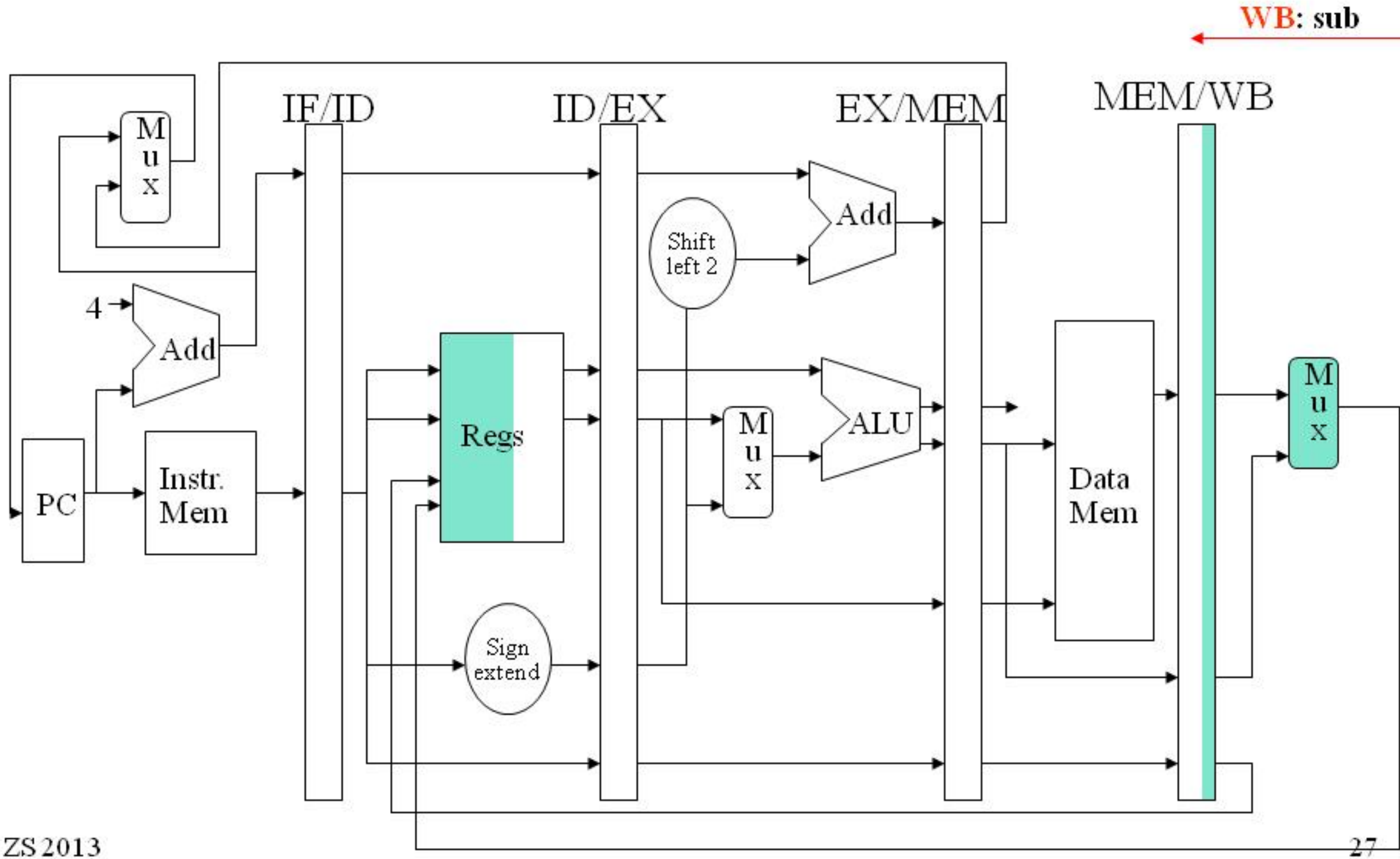
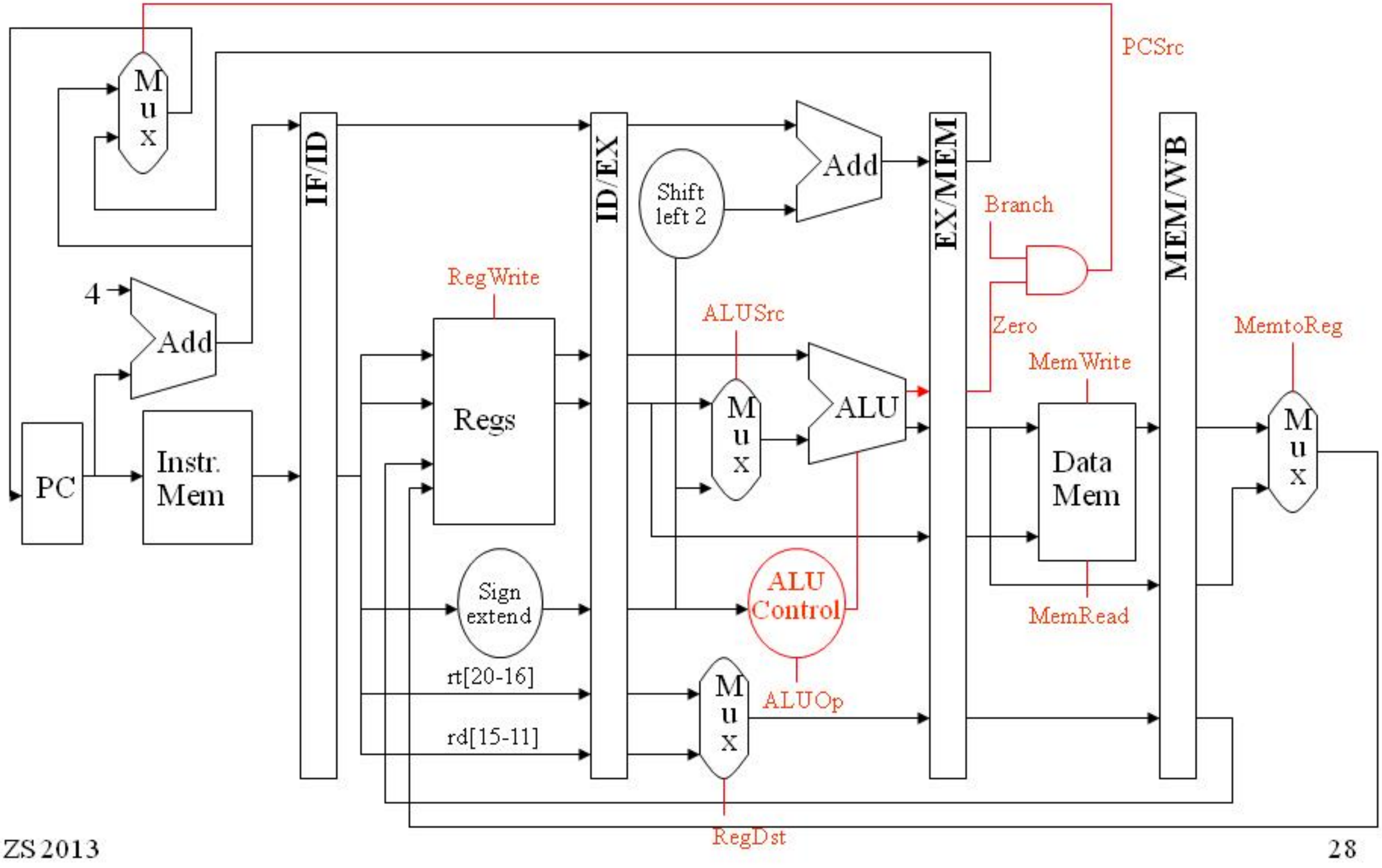


Diagram jednoho cyklu (cykl 6)



Řídící signály



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

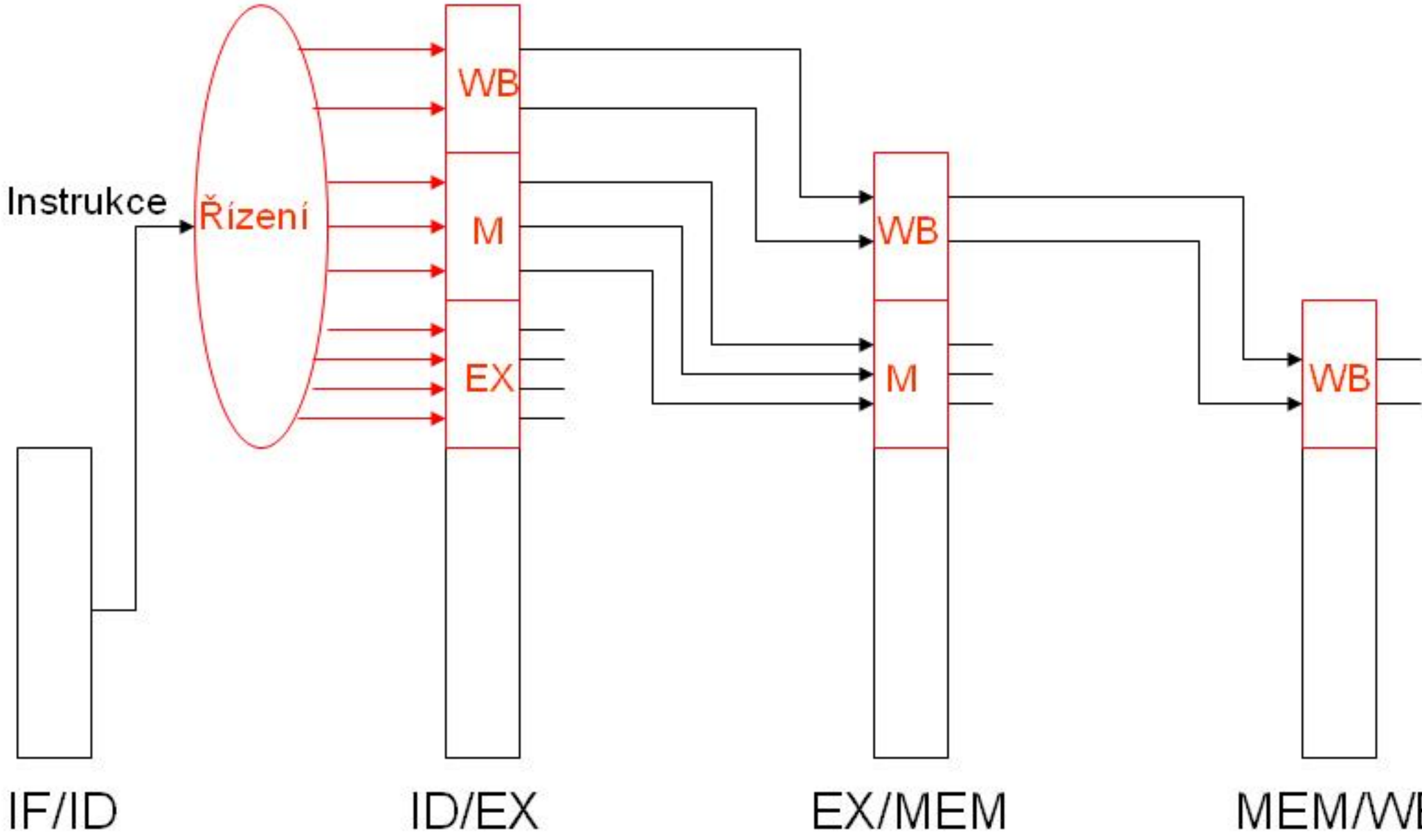
Řídící linky

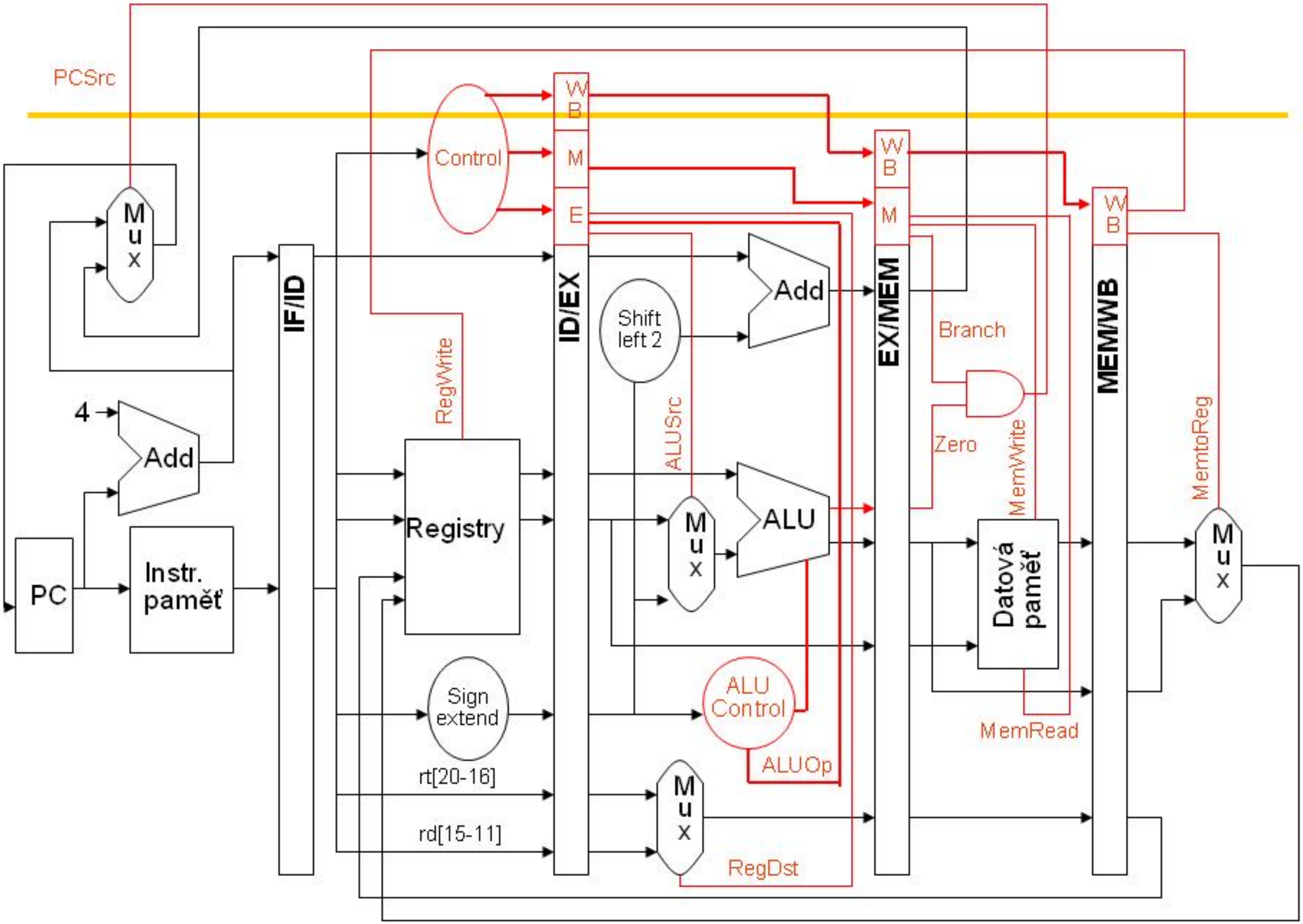
Instrukce	Řídící linky prováděcích stupnů				Řídící linky přístupu do paměti			Řídící linky zpětného zápisu	
	Reg Dst	ALU Op1	ALU Op2	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem 2Reg
R-formát	1	1	0	0	0	0	0	1	0
Lw	0	0	0	1	0	1	0	1	1
Sw	x	0	0	1	0	0	1	0	x
Beq	x	0	1	0	1	0	0	0	x

Implementace řízení

- 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




PCSrc
M
u
x
4
Add
PC
Instr.
paměť
IF/ID
Control
W
B
M
E
ID/EX
RegWrite
Registry
Sign
extend
rt[20-16]
rd[15-11]
Shift
left 2
Add
ALUSrc
M
u
x
ALU
ALU
Control
ALUOp
M
u
x
RegDst
EX/MEM
Branch
Zero
MemWrite
Datová
paměť
MemRead
MEM/WB
MemtoReg
M
u
x

Příklad posloupnosti instrukcí

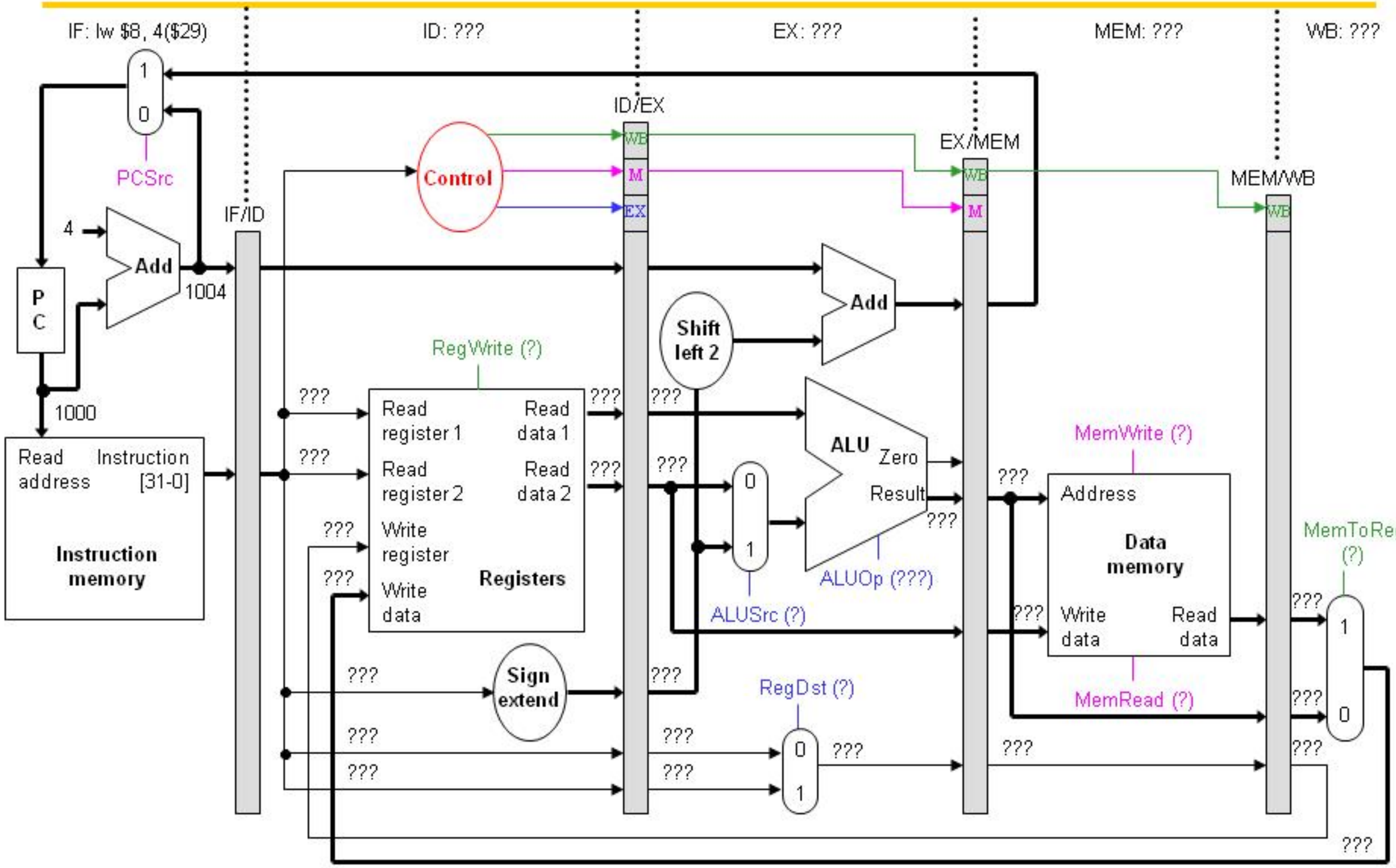
Příklad zpracovávané posloupnosti instrukcí:

Adresy dekadicky

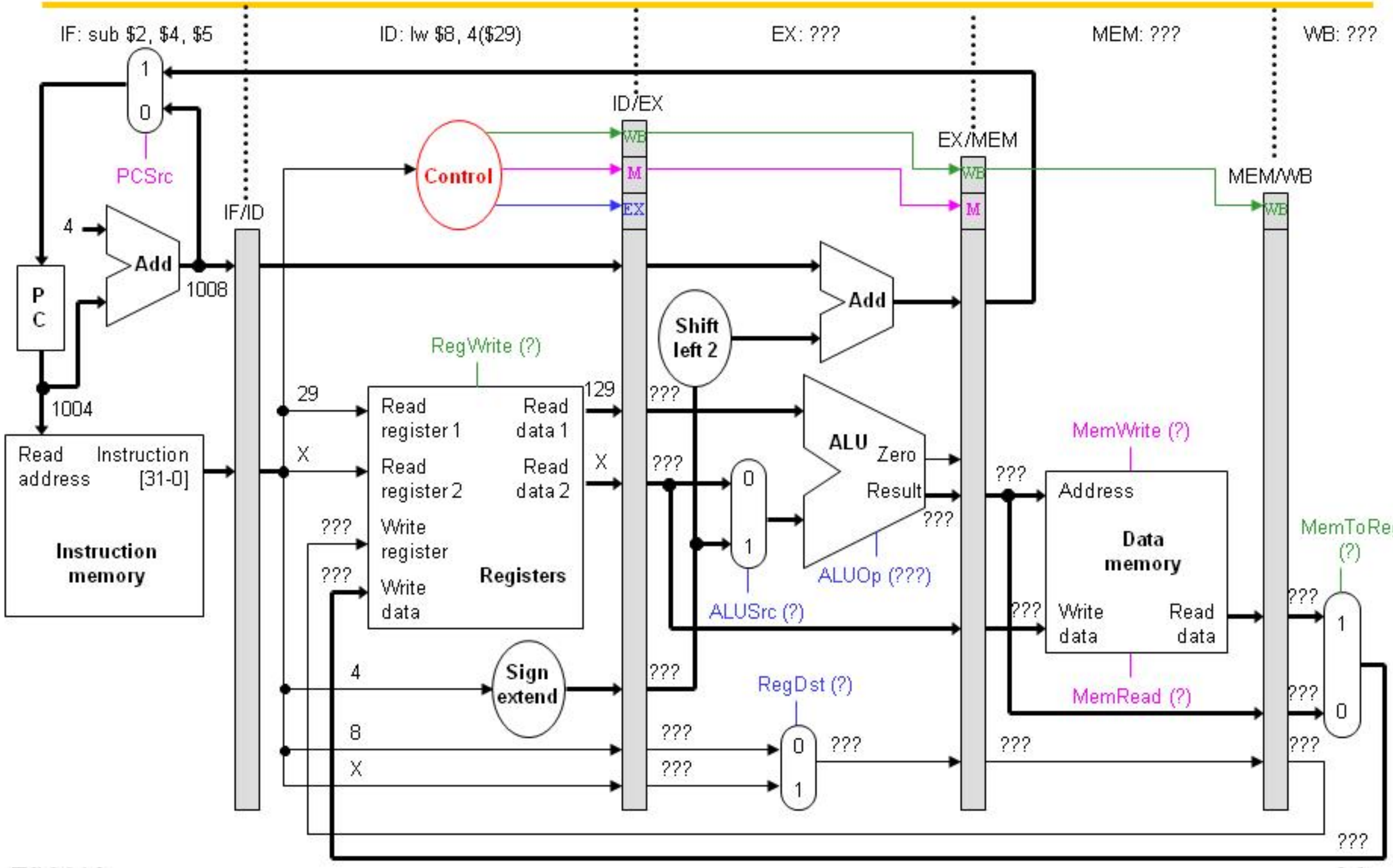
```
1000:   lw    $8, 4($29)
1004:   sub   $2, $4, $5
1008:   and   $9, $10, $11
1012:   or    $16, $17, $18
1016:   add   $13, $14, $0
```

- 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 závisejí na předchozích instrukcích v našem příkladu.

Cykl 1 (plnění)

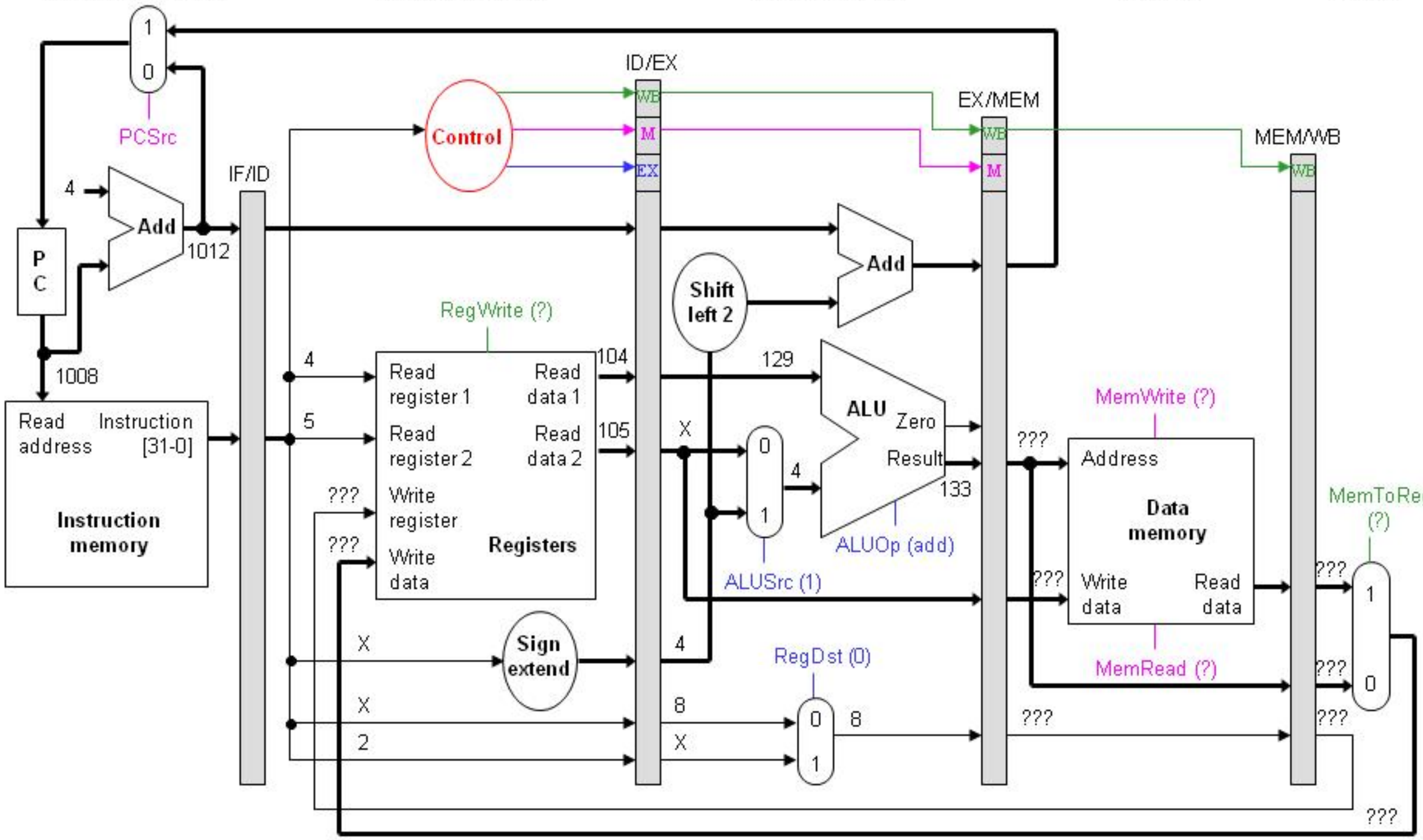


Cykl 2


IF: sub $2, $4, $5
ID: lw $8, 4($29)
EX: ???
MEM: ???
WB: ???
1
0
PCSrc
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
IF/ID
4
Add
1008
P
C
1004
Read address
Instruction [31-0]
Instruction memory
RegWrite (?)
29
X
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
129
X
???
???
???
Shift left 2
Add
ALU
Zero
Result
ALUOp (???)
ALUSrc (?)
RegDst (?)
4
8
X
Sign extend
MemWrite (?)
MemRead (?)
Address
Data memory
Write data
Read data
MemToReg (?)

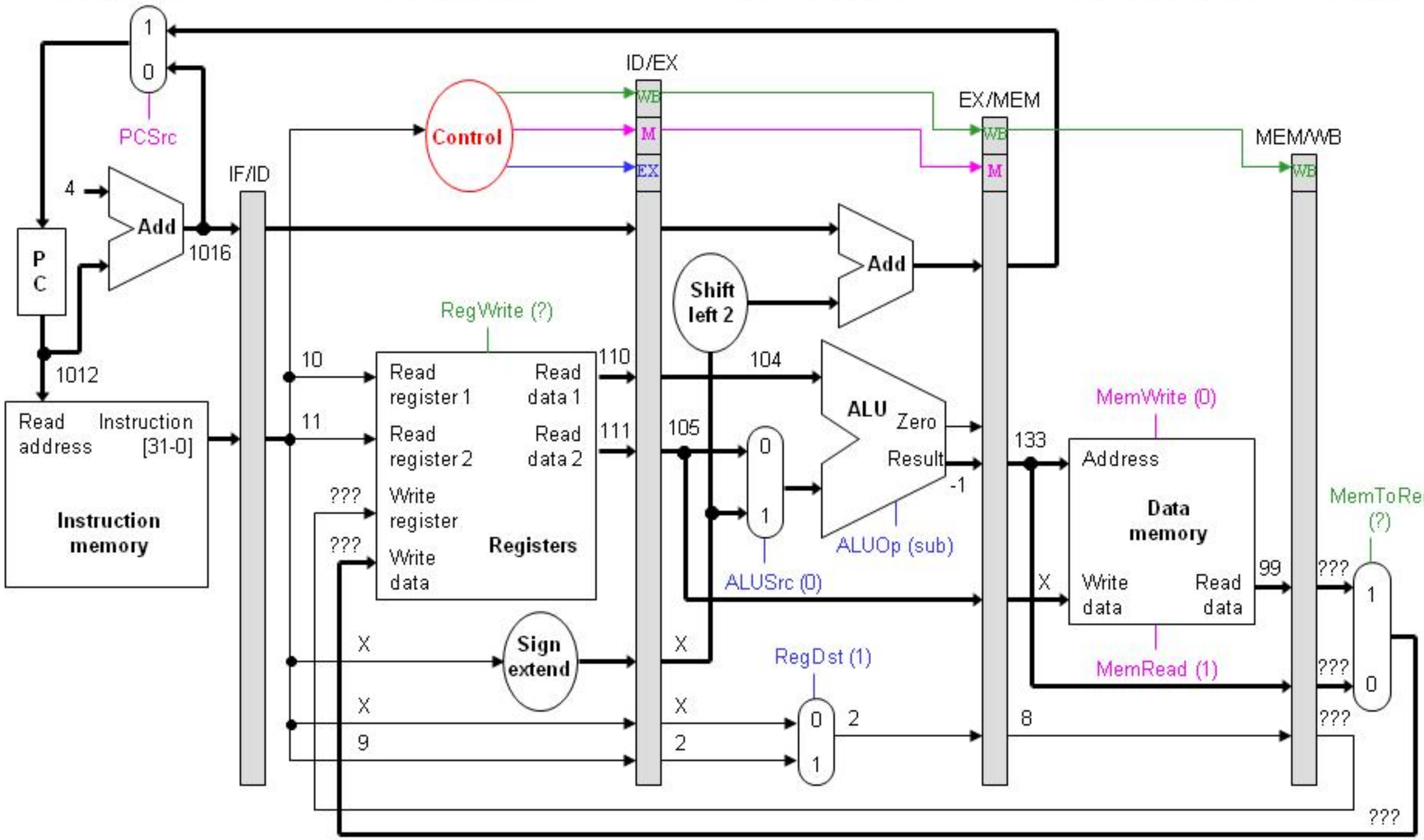
Cykl 3

IF: and $9, $10, $11 | ID: sub $2, $4, $5 | EX: lw $8, 4($29) | MEM: ??? | WB: ???

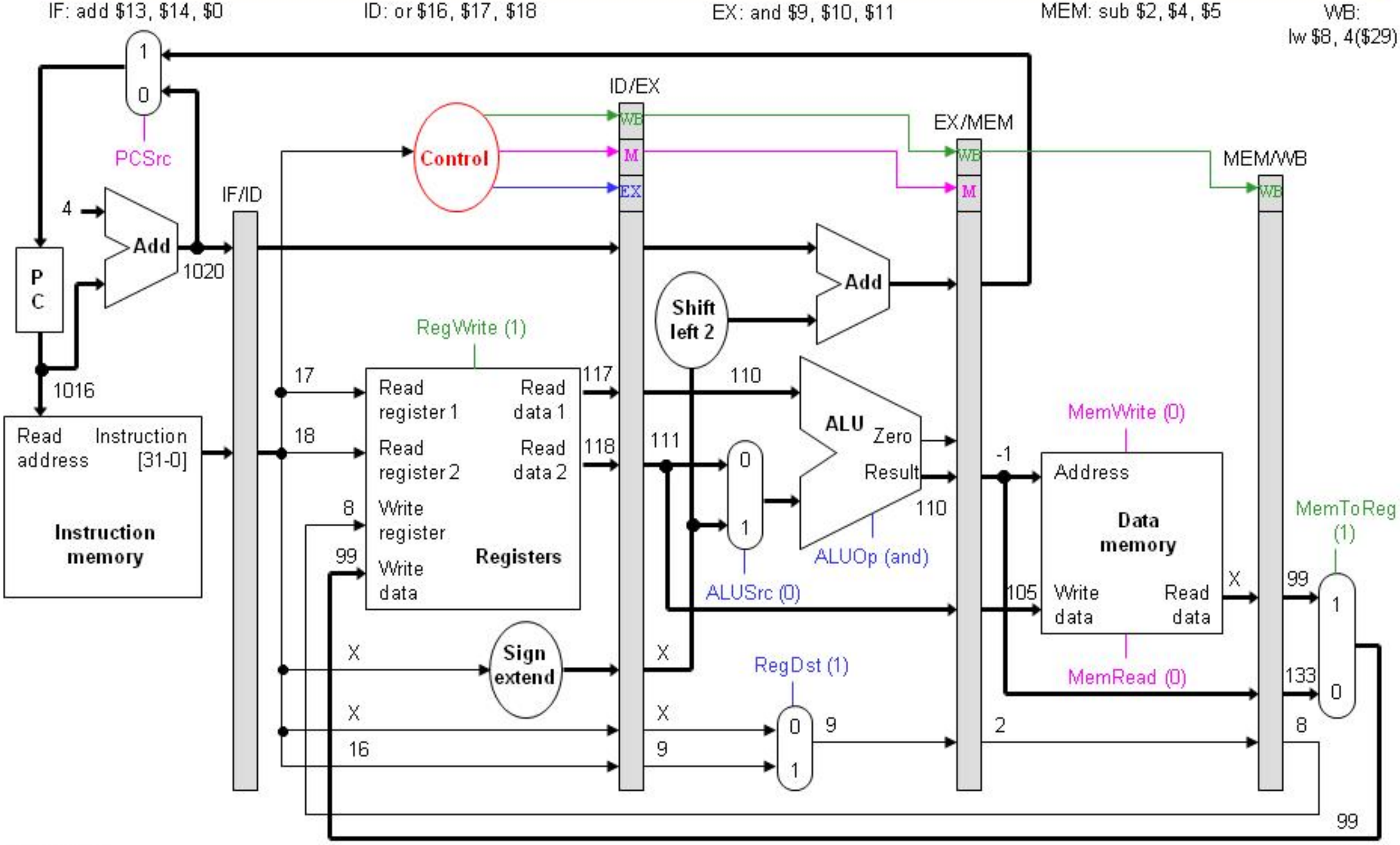


Cykl 4

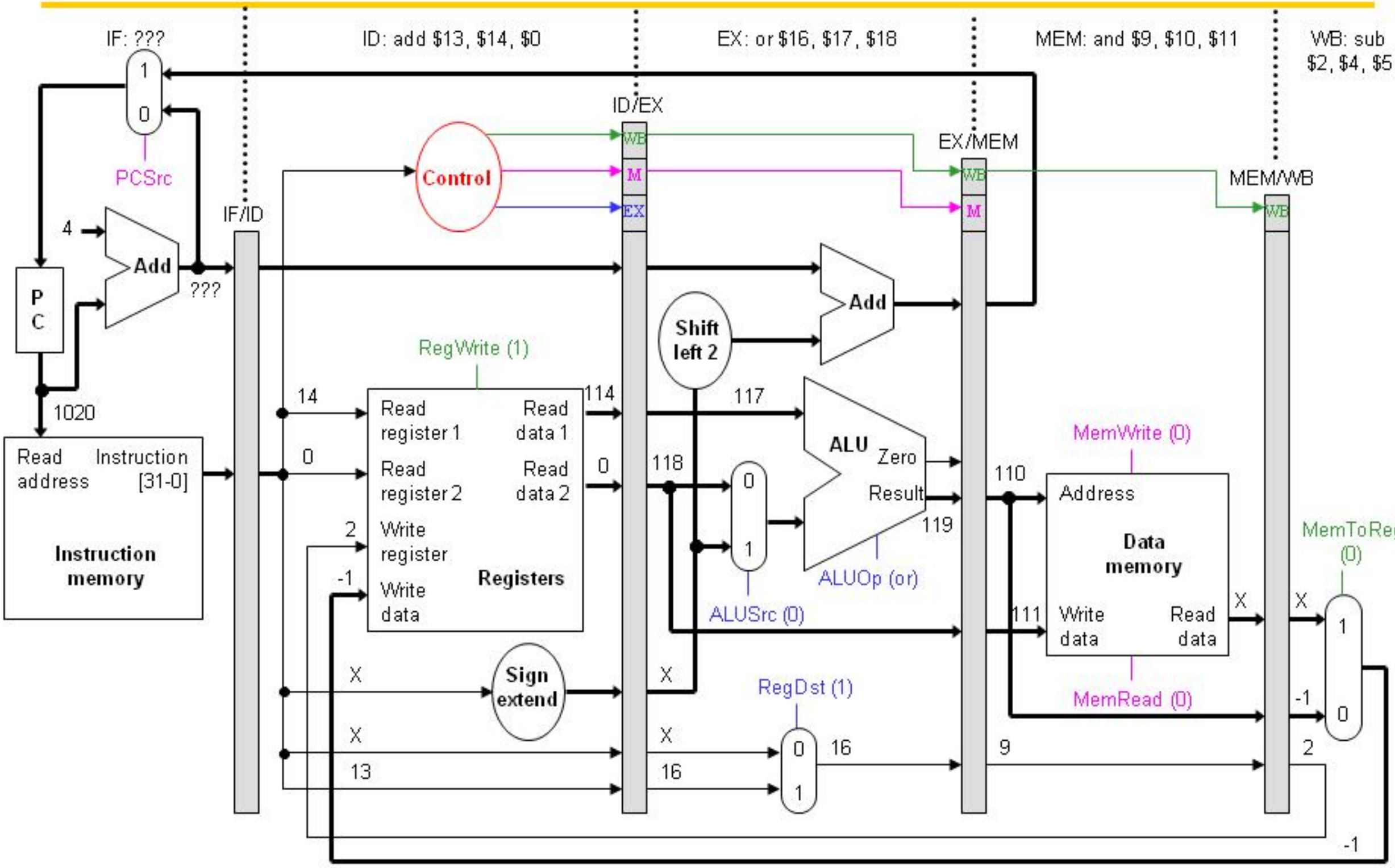
IF: or $16, $17, $18 | ID: and $9, $10, $11 | EX: sub $2, $4, $5 | MEM: lw $8, 4($29) | WB: ???



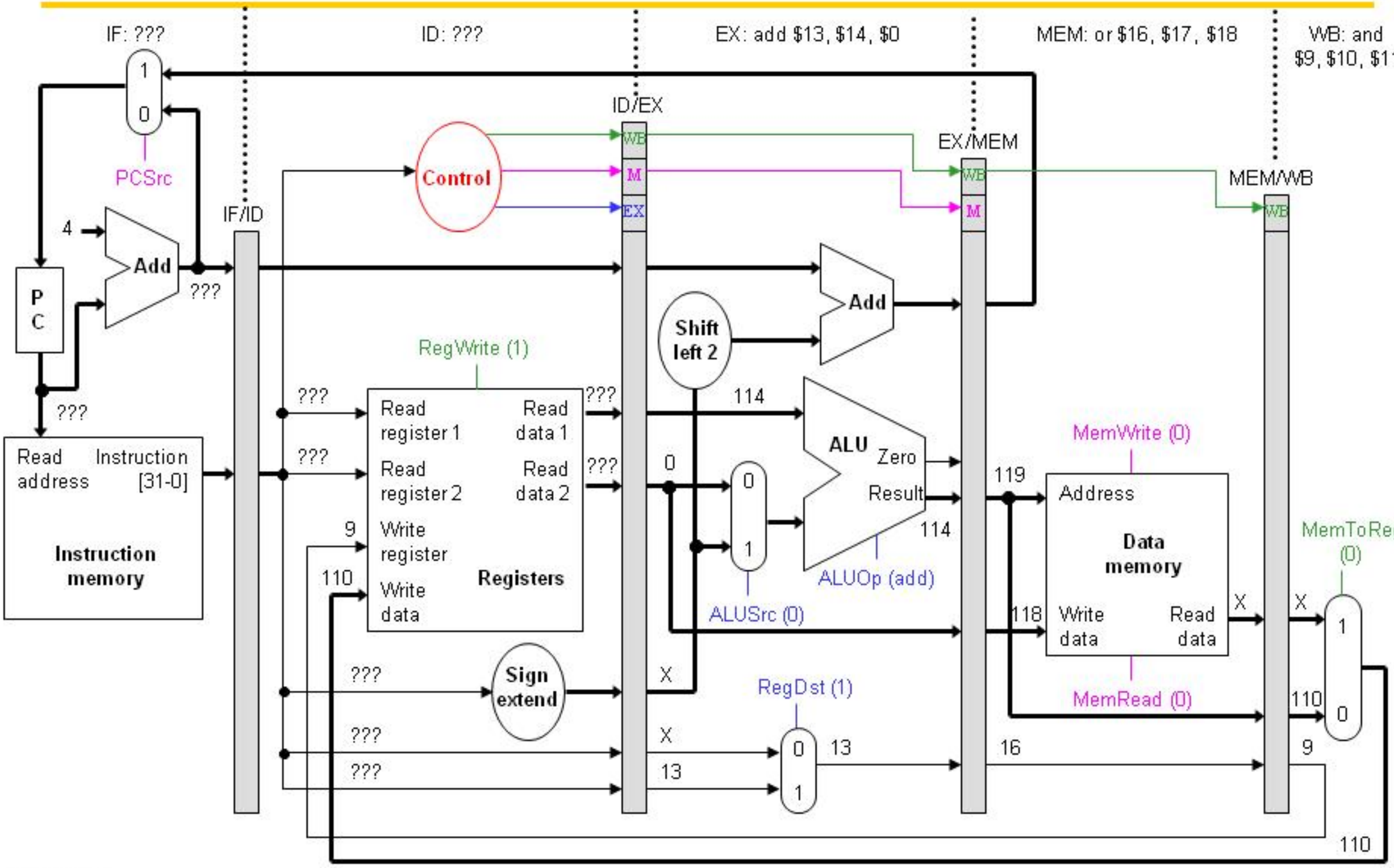
Cykl 5 (zaplněno)



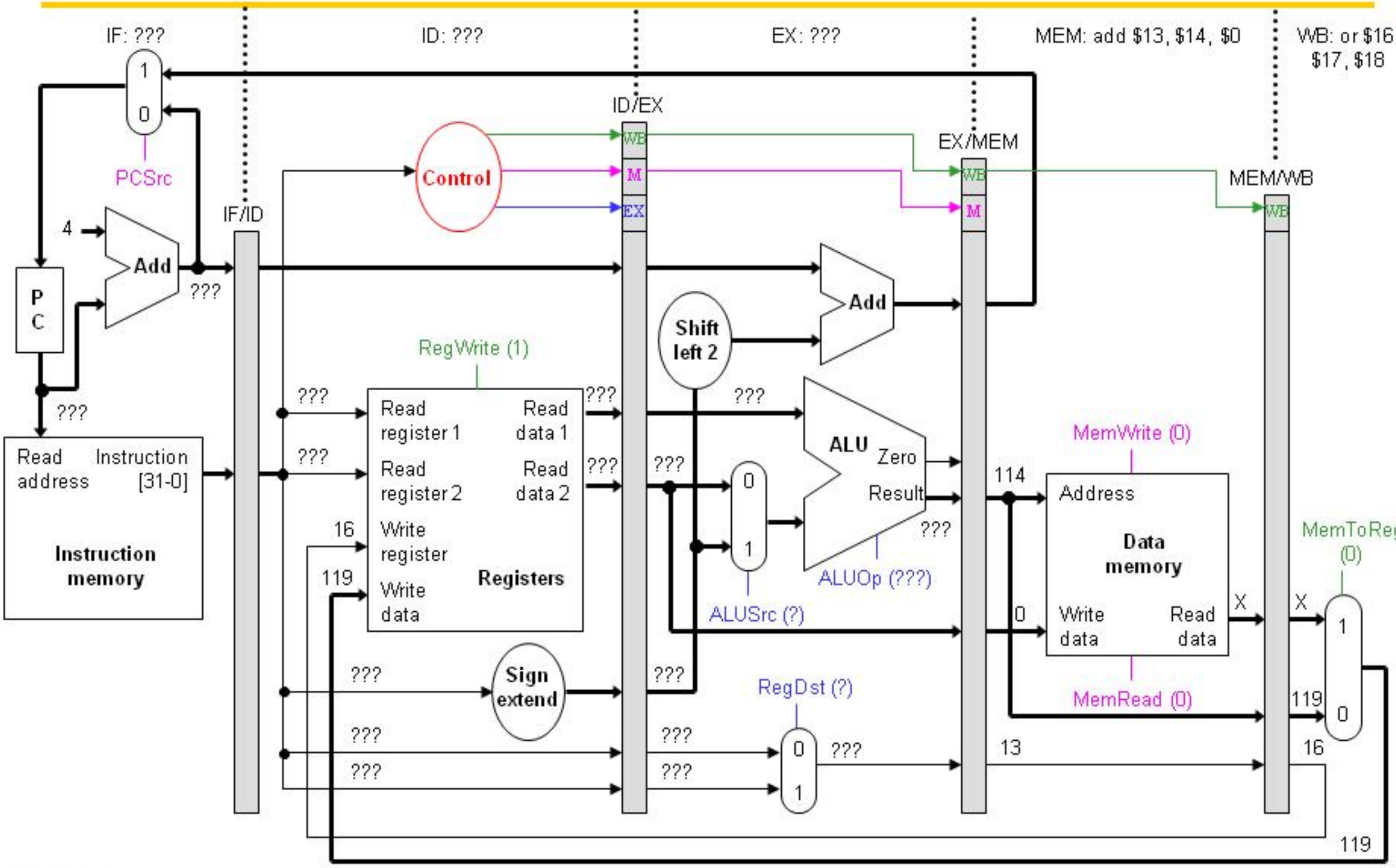
Cykl 6 (vyprazdňování)



Cykl 7


IF: ???
ID: ???
EX: add $13, $14, $0
MEM: or $16, $17, $18
WB: and $9, $10, $11
1
0
PCSrc
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
IF/ID
4
Add
???
P C
???
Read address
Instruction [31-0]
Instruction memory
RegWrite (1)
???
???
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
9
110
???
???
???
Sign extend
Shift left 2
Add
114
0
0
1
ALUSrc (0)
ALU
Zero
Result
114
ALUOp (add)
X
X
X
RegDst (1)
0
1
13
13
MemWrite (0)
119
118
Address
Data memory
Write data
Read data
MemRead (0)
X
MemToReg (0)
X
1
0
110
16
9
110

Cykl 8


IF: ???
ID: ???
EX: ???
MEM: add $13, $14, $0
WB: or $16, $17, $18
1
0
PCSrc
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
4
Add
???
P C
IF/ID
Shift left 2
Add
RegWrite (1)
???
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
119
Read address
Instruction [31-0]
Instruction memory
ALU
Zero
Result
ALUOp (???)
ALUSrc (?)
MemWrite (0)
114
Address
Data memory
Write data
Read data
MemRead (0)
MemToReg (0)
X
Sign extend
RegDst (?)
13
119
ZS 2013
42

Cykl 9

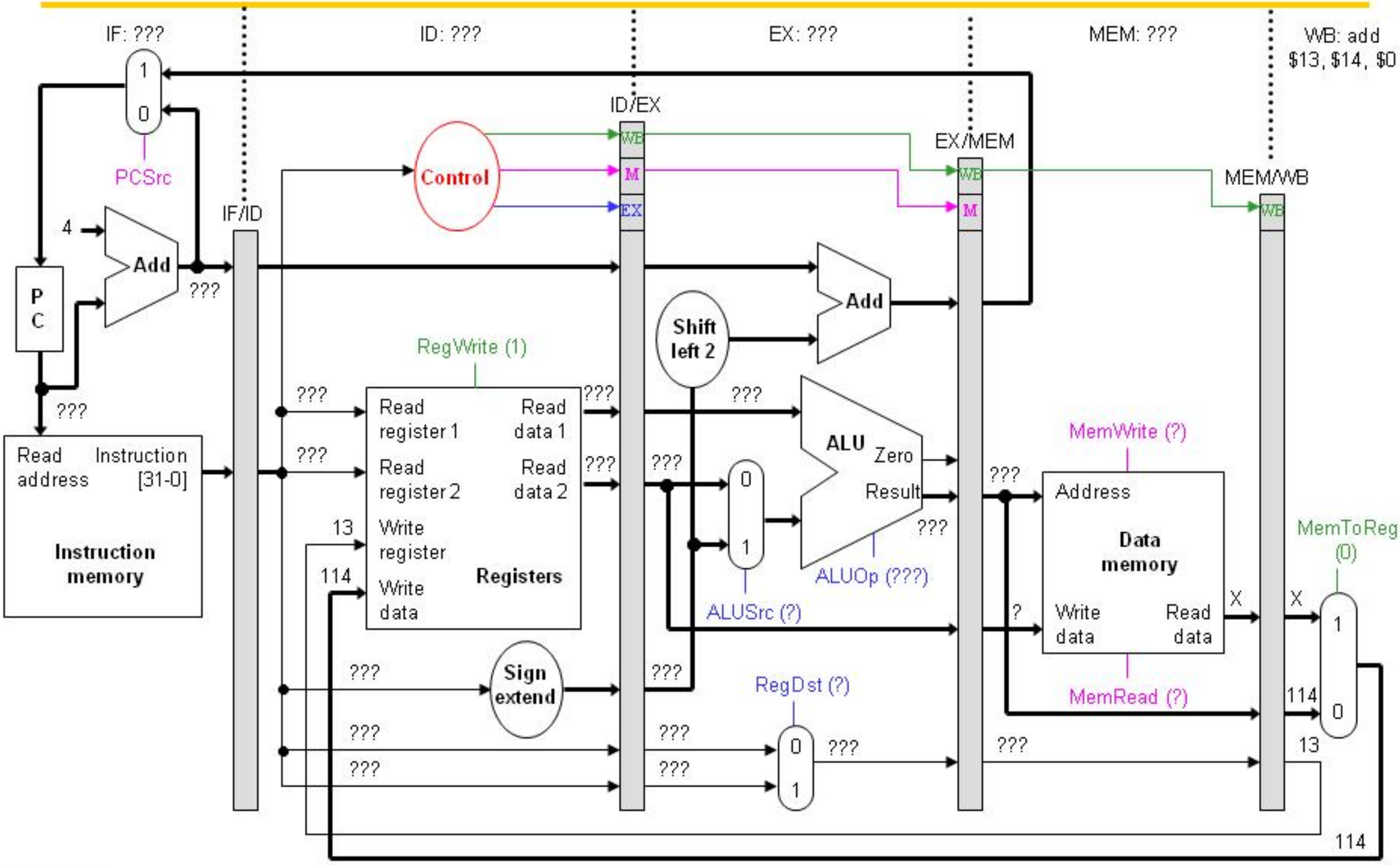
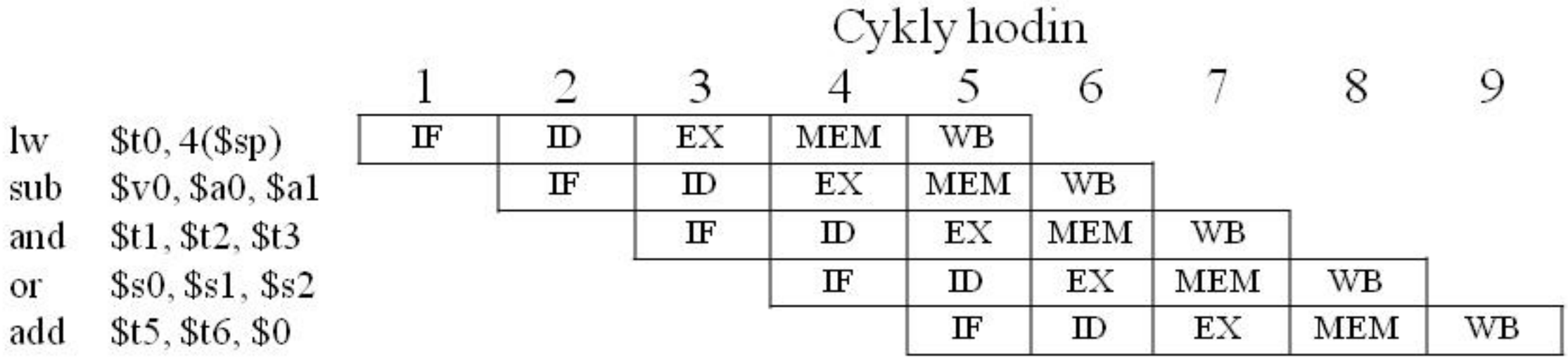

IF: ???
ID: ???
EX: ???
MEM: ???
WB: add $13, $14, $0
1
0
PCSrc
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
IF/ID
4
Add
???
P C
???
Read address
Instruction [31-0]
Instruction memory
RegWrite (1)
Shift left 2
Add
???
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
13
114
ALU
Zero
Result
ALUOp (???)
ALUSrc (?)
MemWrite (?)
Address
Data memory
Write data
Read data
MemRead (?)
MemToReg (0)
X
?
Sign extend
RegDst (?)
114

Diagram překrývání



Porovnejte devět posledních snímků s diagramem nahoře.

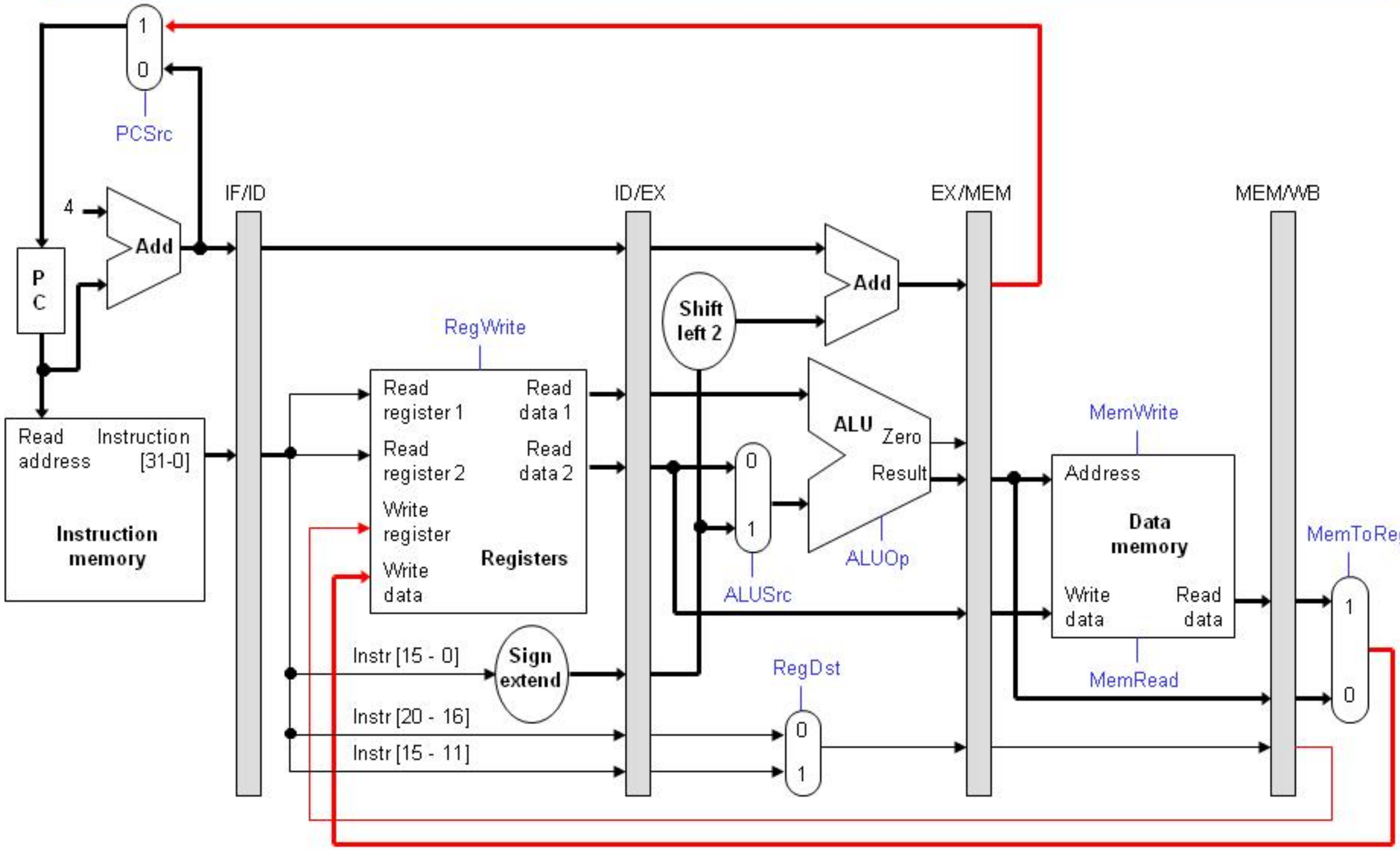
- Lze sledovat překrývání instrukcí.
- Každá funkční jednotka je v každém cyklu využita *jinou* instrukcí.
- Pipeline registry zachytí řízení a data generované v předchozím cyklu pro pozdější použití.
- Po zaplnění pipeline v cyklu 5 jsou všechny jednotky využity. To je ideální situace a také důvod,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

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* vlastním provedením ve stupni EX.

Vše jde zleva doprava s výjimkou …



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
```

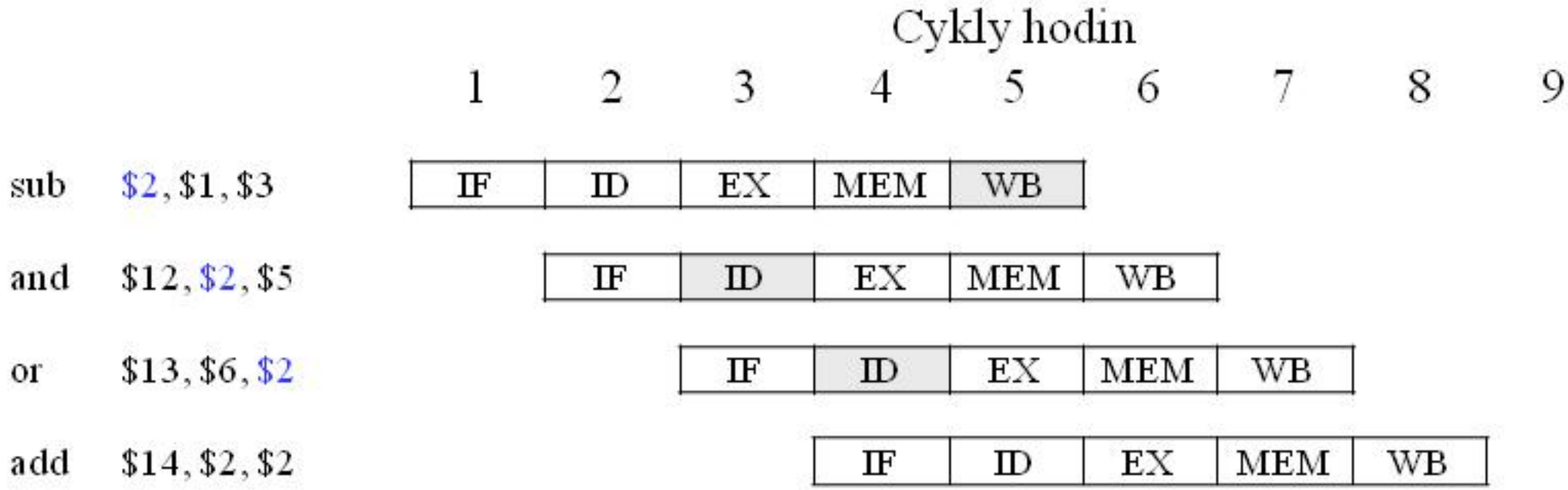
- Instrukce v příkladu jsou nezávislé.
  - Každá instrukce čte a modifikuje úplně jiné registry.
  - Taková posloupnost se snadno ošetřuje.
- Většina posloupností instrukcí v reálných programech ale nejsou nezávislé!

Příklad se závislostmi

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

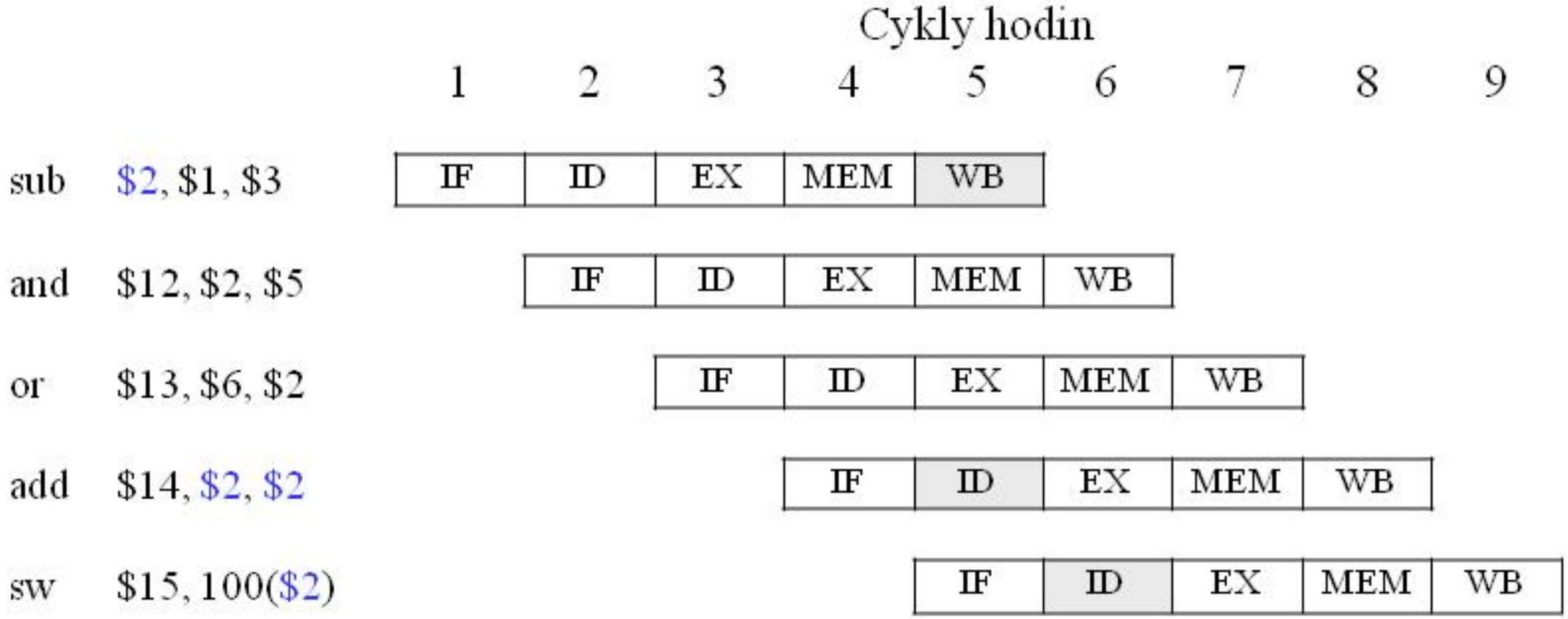
- Mezi instrukcemi je několik závislostí.
  - Prvá instrukce SUB ukládá hodnotu do $2.
  - Tento registr je pak použit jako zdroj v dalších instrukcích.
- Pro jednocyklovou jednotku by to nepředstavovalo problém.
  - Každá instrukce je zakončena před započetím další instrukce.
  - To zajistí, že instrukce 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



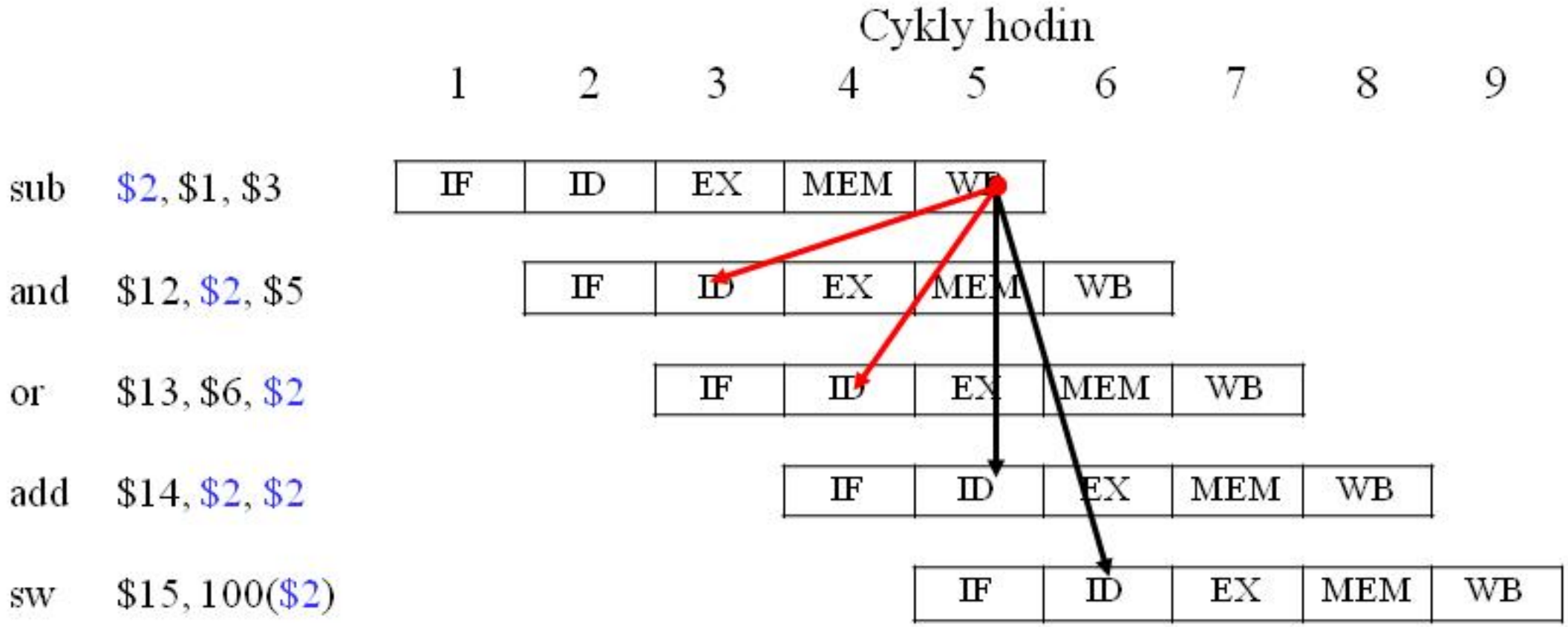
- Instrukce SUB zapisuje do registru $2 až v cyklu 5. To způsobí dva datové hazardy v naší současné jednotce, pracující v režimu pipeline.
  - AND čte registr $2 v cyklu 3. Protože SUB dosud nezapsala do $2, bude přečtena *stará* hodnota $2, nikoliv nová.
  - Podobně instrukce OR používá registr $2 v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

Problémy nevznikají



- 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 nečiní problém vůbec, protože čte $2 až po ukončení SUB.

Závislosti



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

Hazardy - souhrn (1/2)

- **Datové hazardy**
  - *Závislosti*: Instrukce je závislá na výsledku předchozí instrukce, která je stále v pipeline

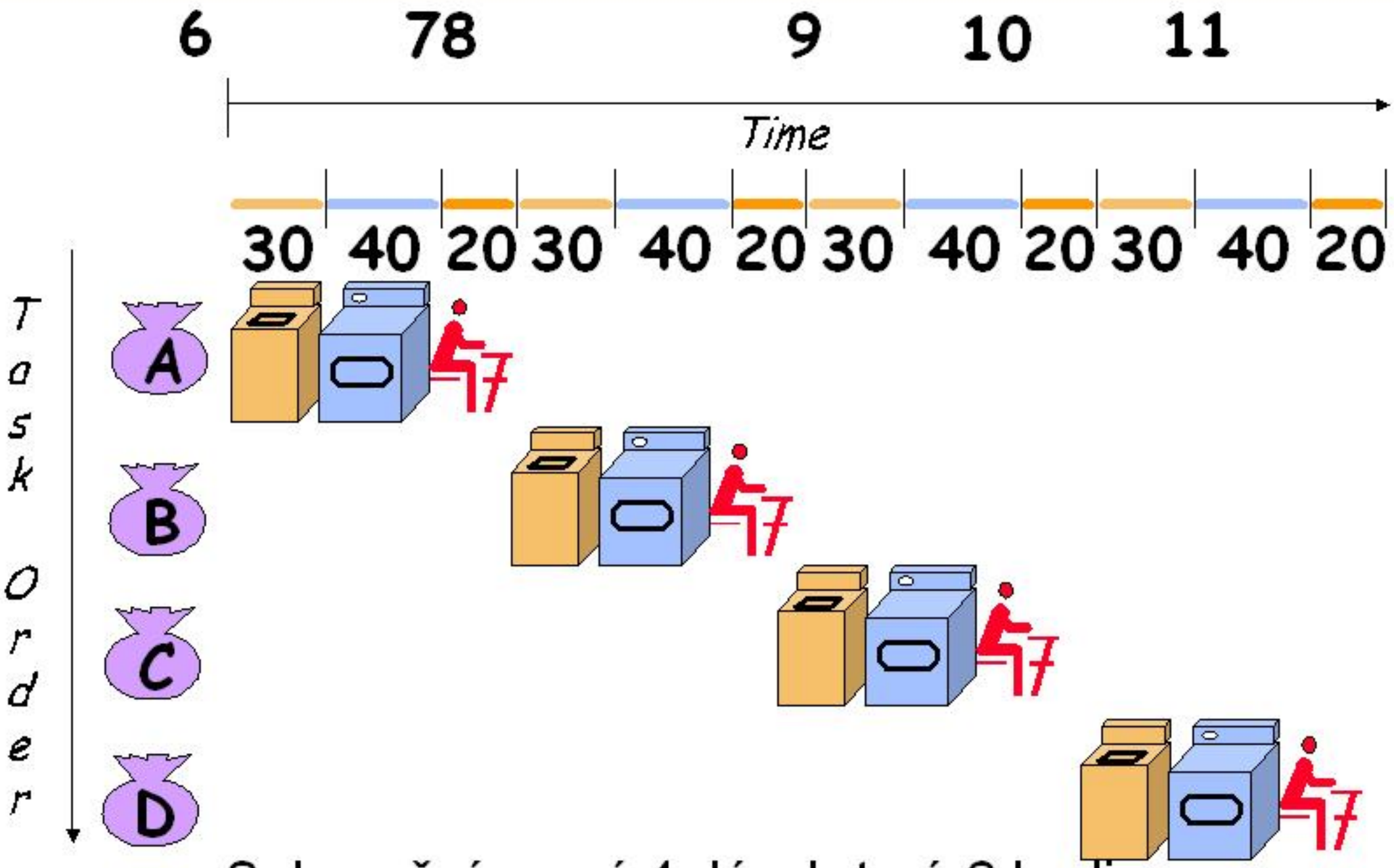
```
Add  $s0, $t0, $t1
Sub  $t2, $s0, $t3
```

  - *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 (2/2)

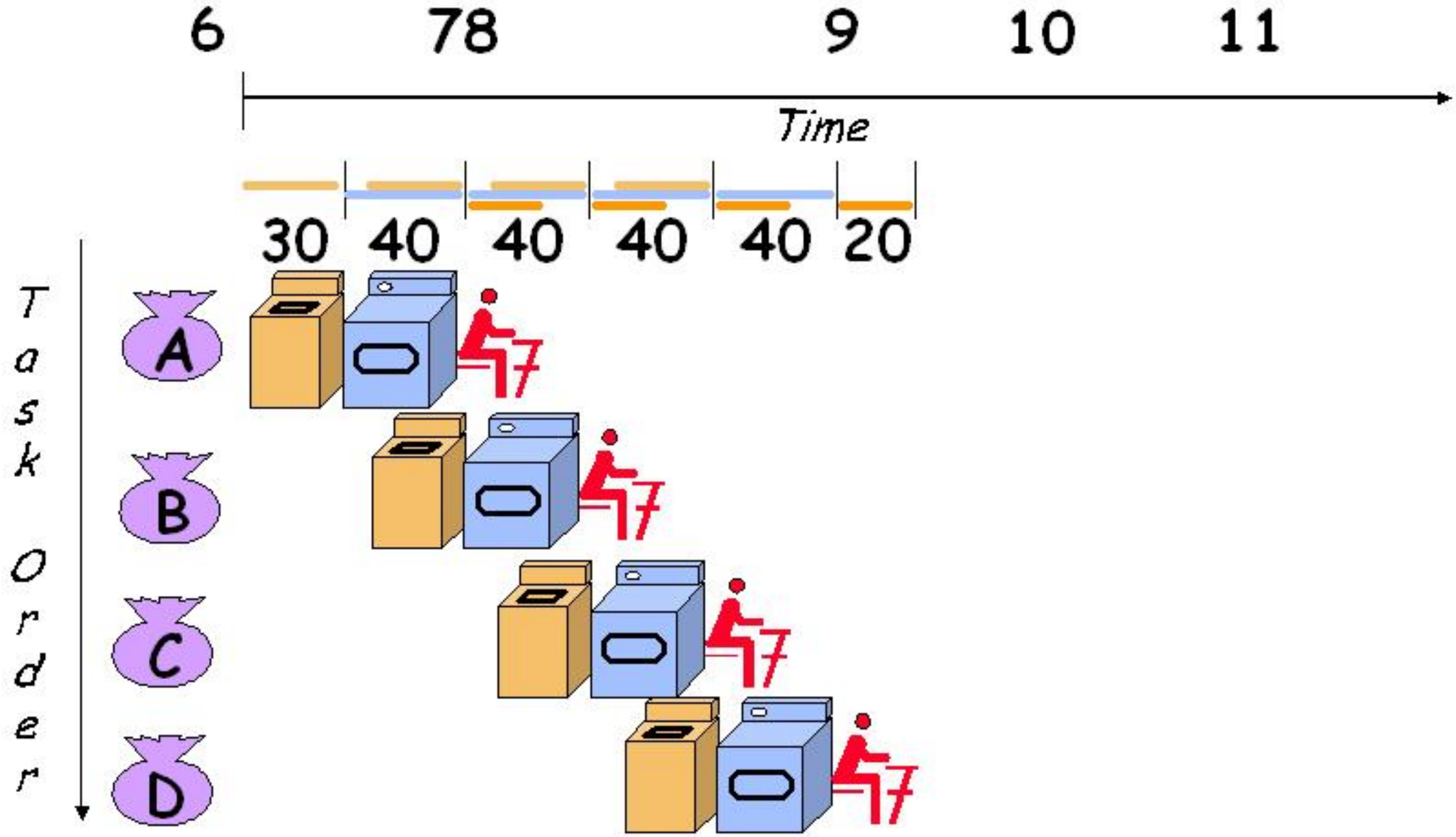
- **Strukturní**
  - Různé instrukce se snaží používat současně stejné funkční jednotky (např. paměť, registrovou sadu)
  - Řešení: duplikovat hardware
- **Řídící** (větvení)
  - Cílová adresa je známa až na konci třetího cyklu => POZASTAVENÍ
  - Řešení
    - Predikce (statická a dynamická): Smyčky
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

Sekvenční prádelna



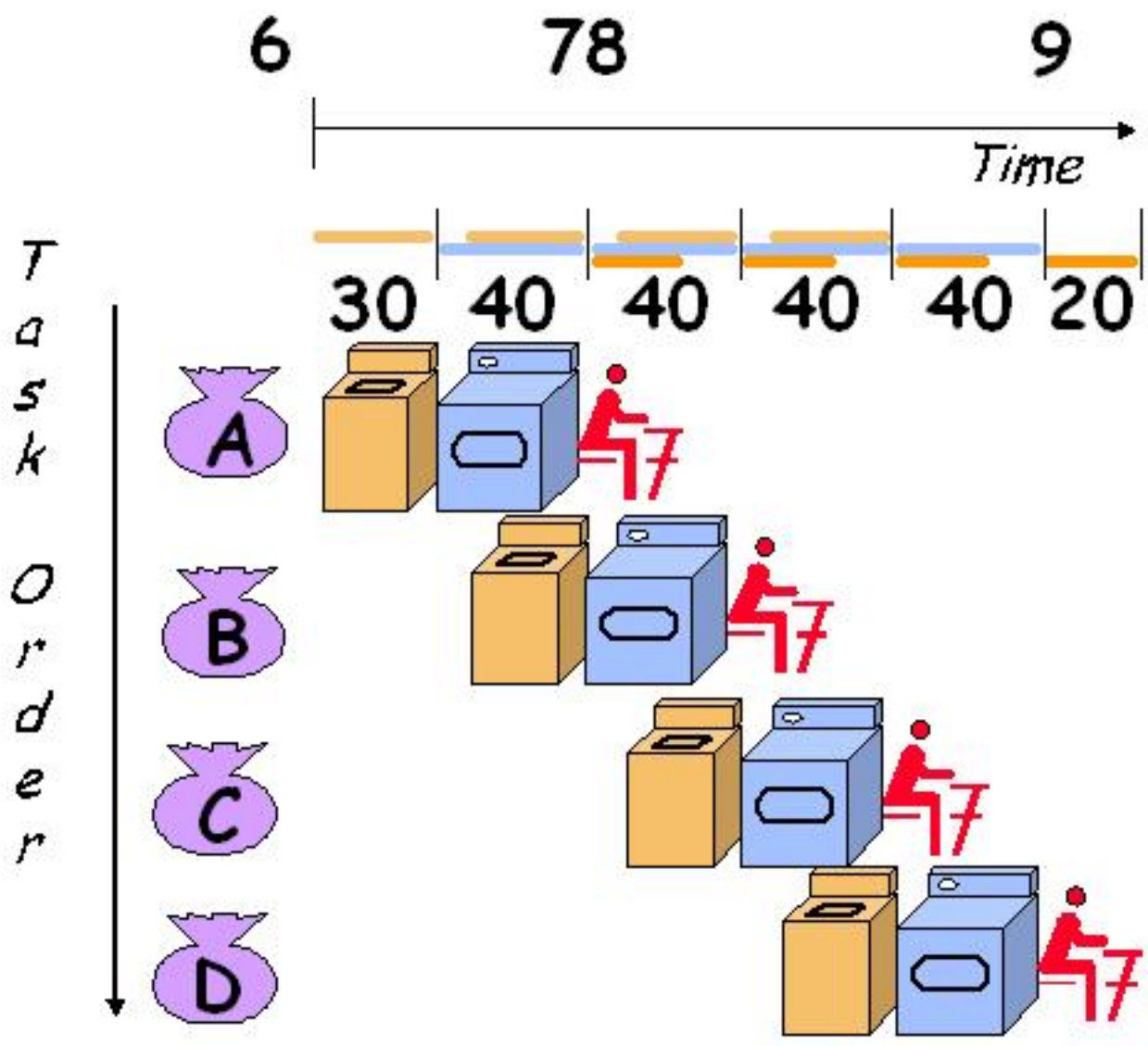
- Sekvenční praní 4 dávek trvá 6 hodin
- Jak dlouho by prali, kdyby znali pipelining?

Prádelna v režimu pipeline



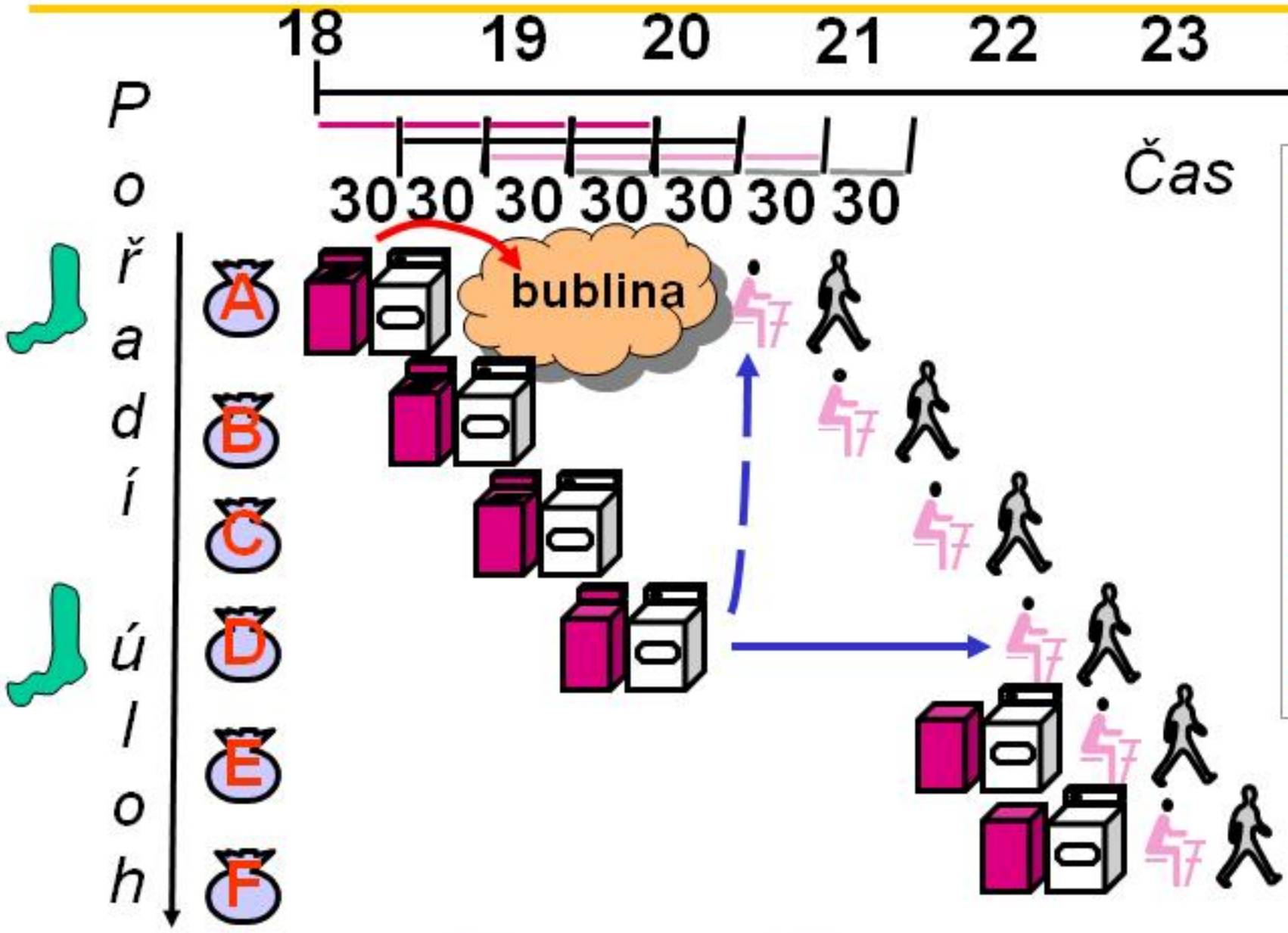
- Praní v režimu pipeline trvá 3.5 hodiny pro 4 dávky

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)

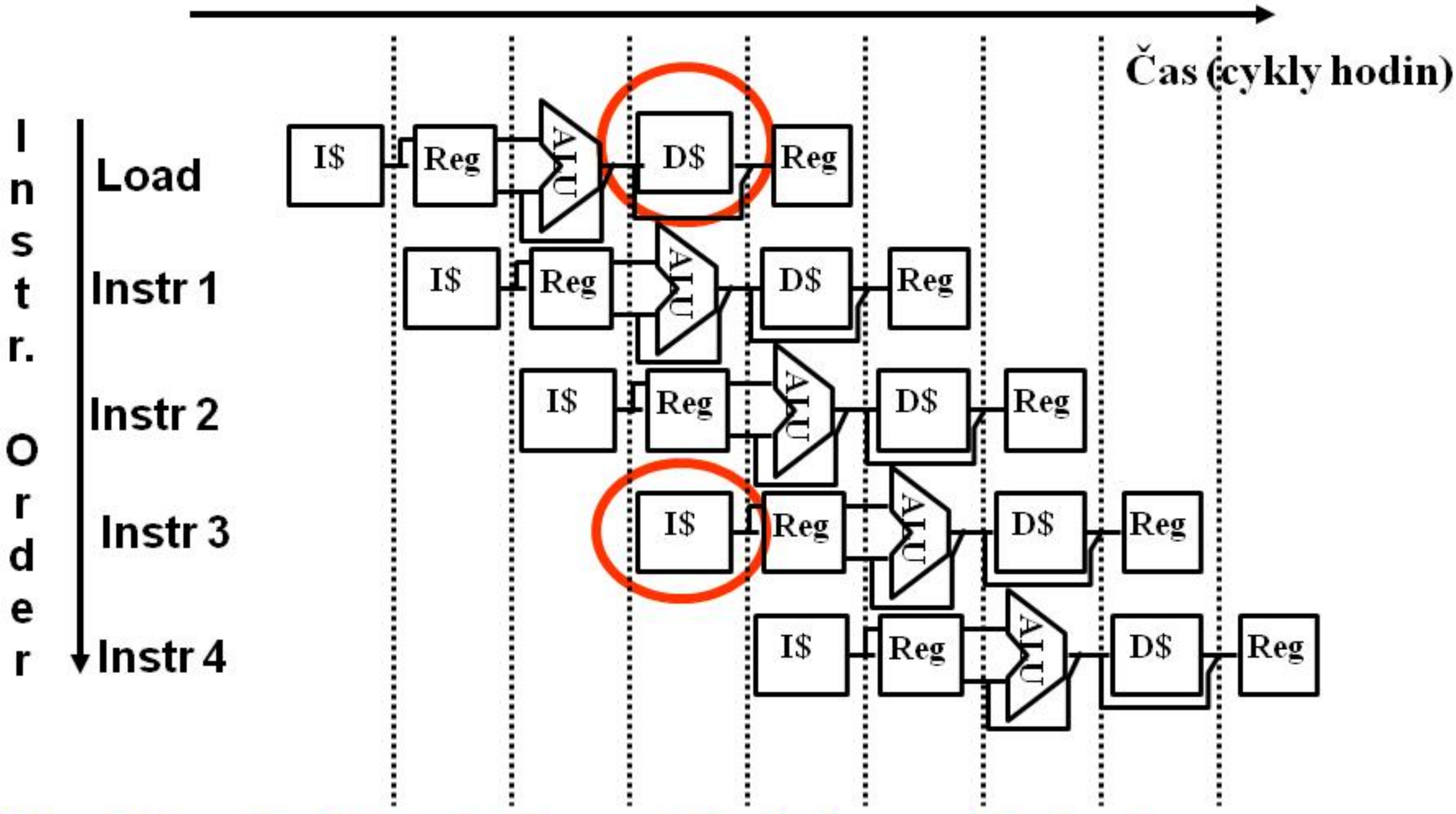


Balík A: Řadič vloží 90 min. bublinu do pipeline mezi sušení a balení (hotovo v 21)

Balík D: Nelze dokončit do 22:30 (pouze jedno balící zařízení k dispozici)

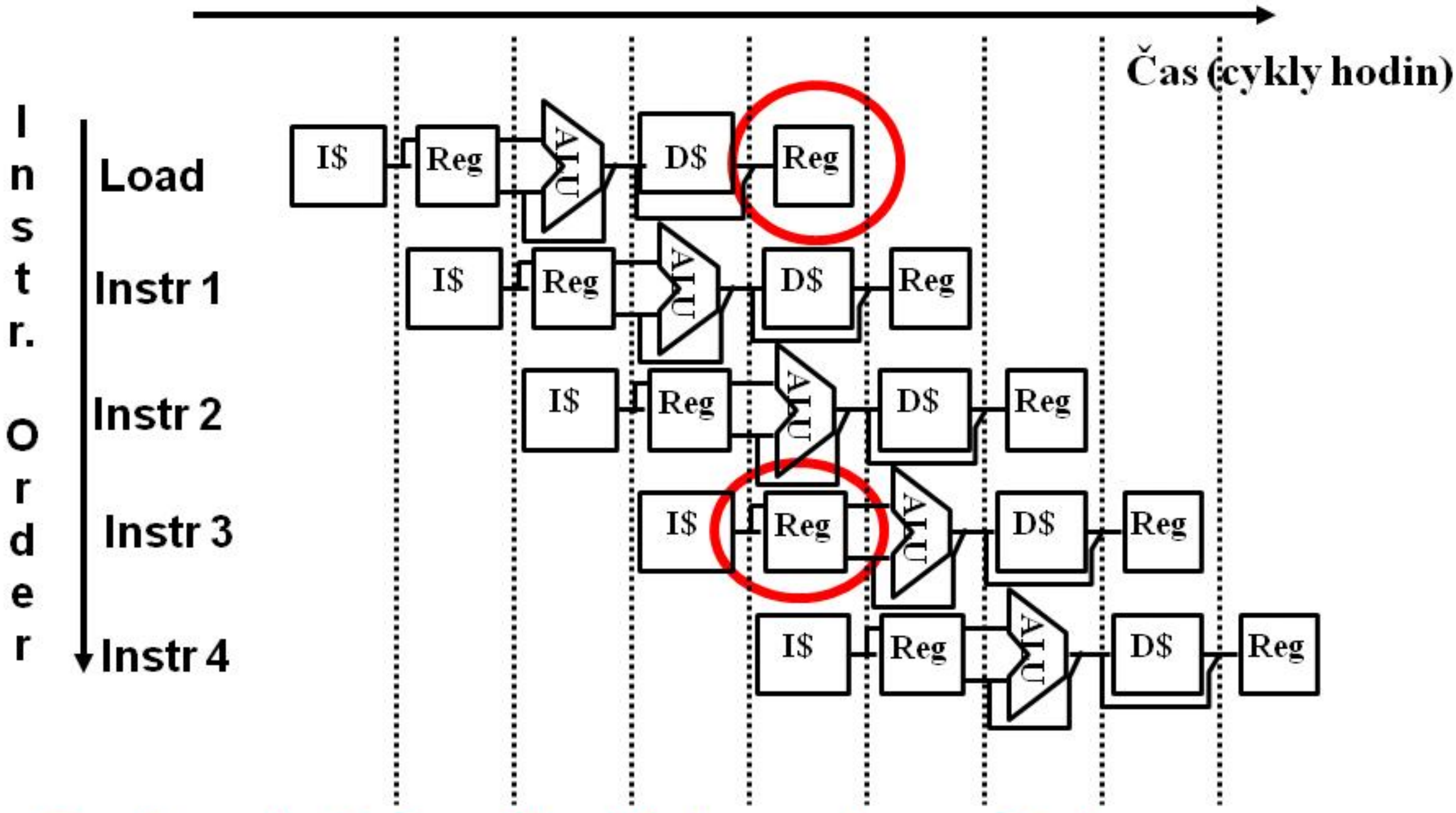
- Případ **zelených ponožek**: jedna v A druhá v D
- D závisí na A → pozastavení dokud je balička obsazena

Strukturní hazard 1: Jediná paměť



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

Strukturní hazard 2: Registrová sada



Současný zápis a čtení do/ze sady registrů

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: 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)

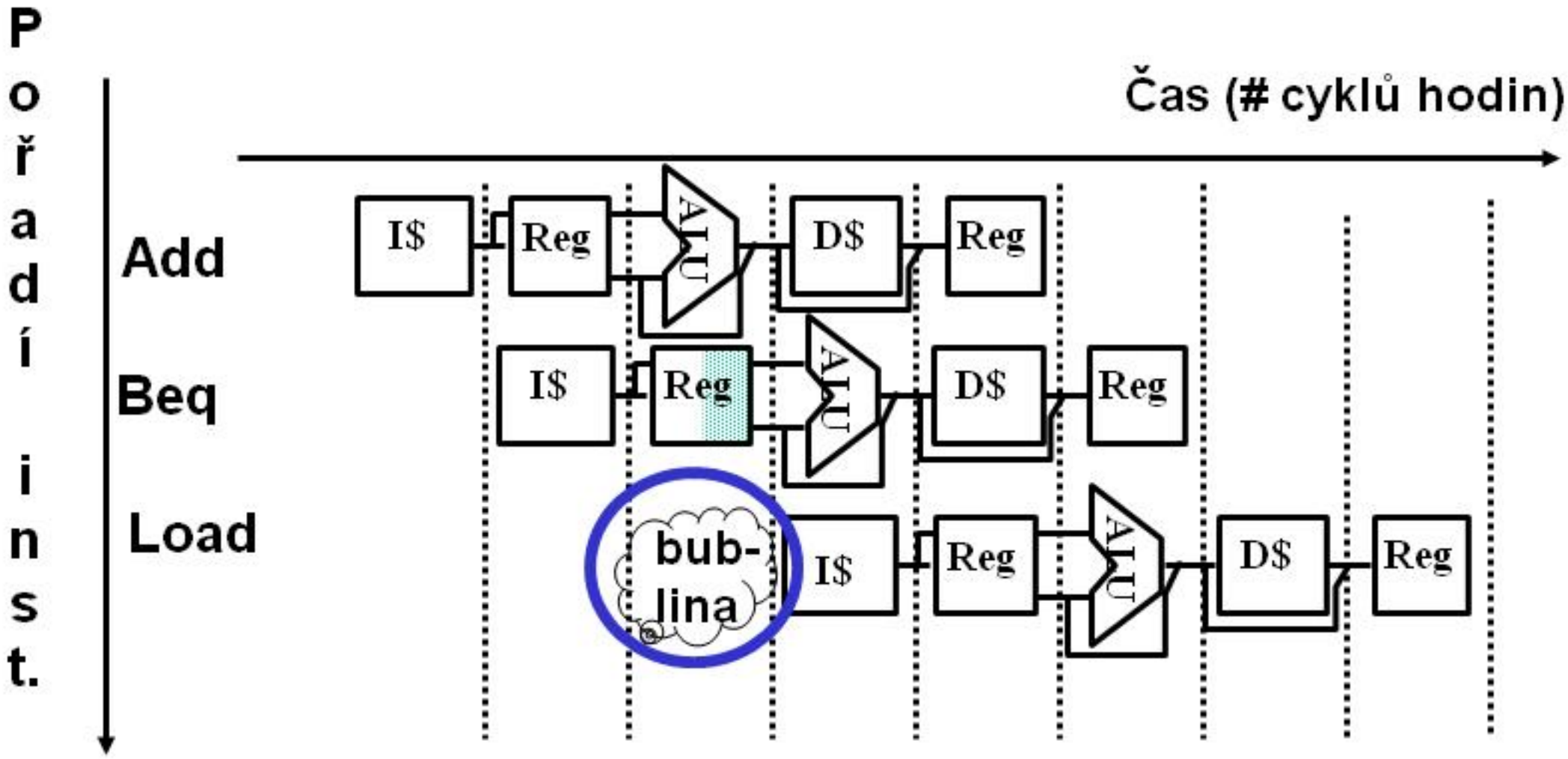
Hazard řízení : Instrukce větvení (2/2)

- **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

- 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: 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

Nezpožděná větvení

```
or    $8, $9 ,$10

add $1  ,$2,$3

sub $4, $5,$6

beq $1, $4, Exit

xor $10, $1,$11

...
Exit:
```

Zpožděná větvení

```
add $1  ,$2,$3

sub $4, $5,$6

beq $1, $4, Exit

or    $8, $9 ,$10

xor $10, $1,$11

...
Exit:
```

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

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**

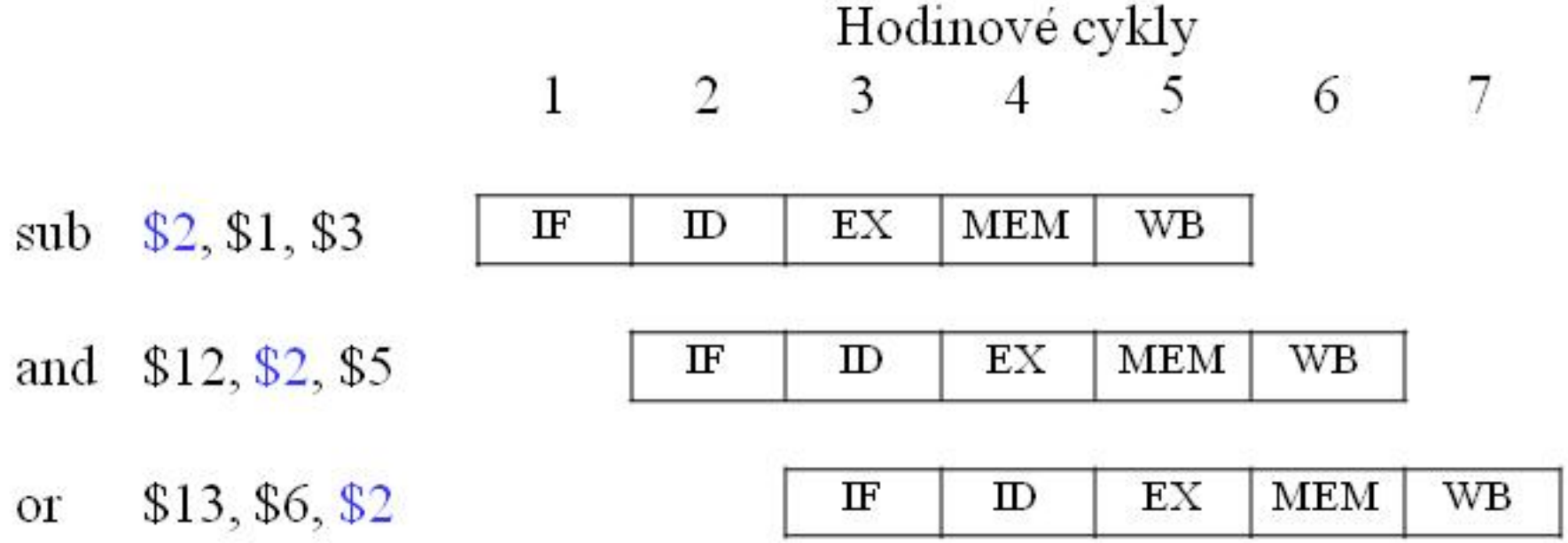
Závěr (3/3)

- 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

Pipelining - detaily

Detailní pohled na pipeline

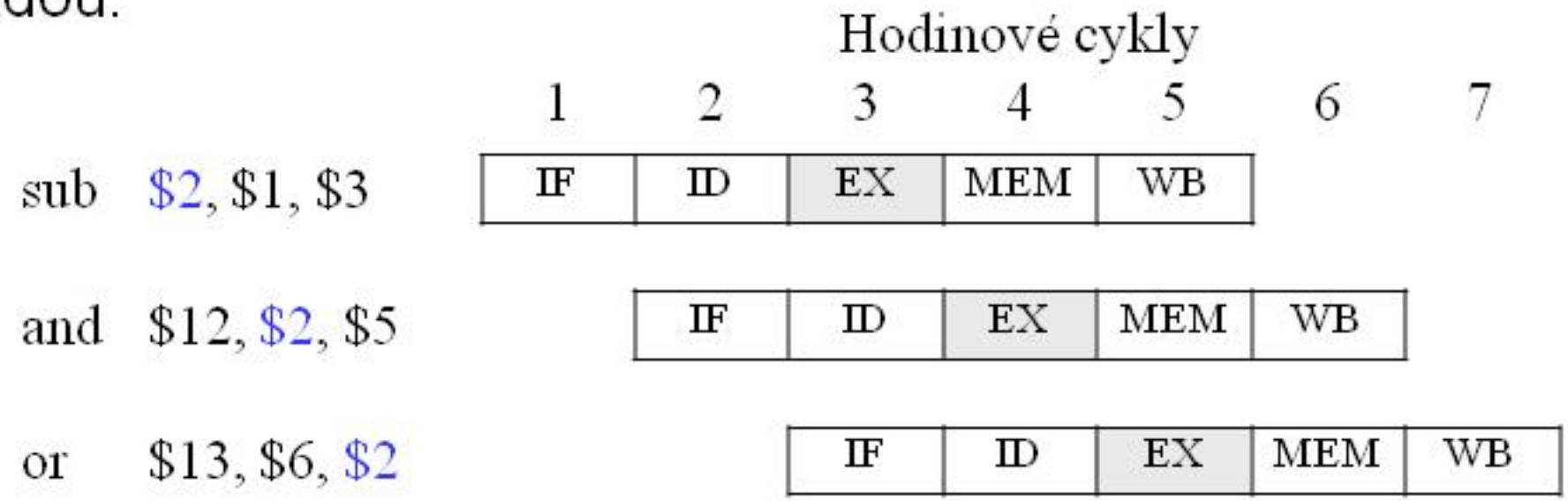
- Budeme se zabývat problémy, které způsobují datové hazardy v procesoru s pipeliningem a jak je eliminovat metodou, která se nazývá forwarding.



- 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?

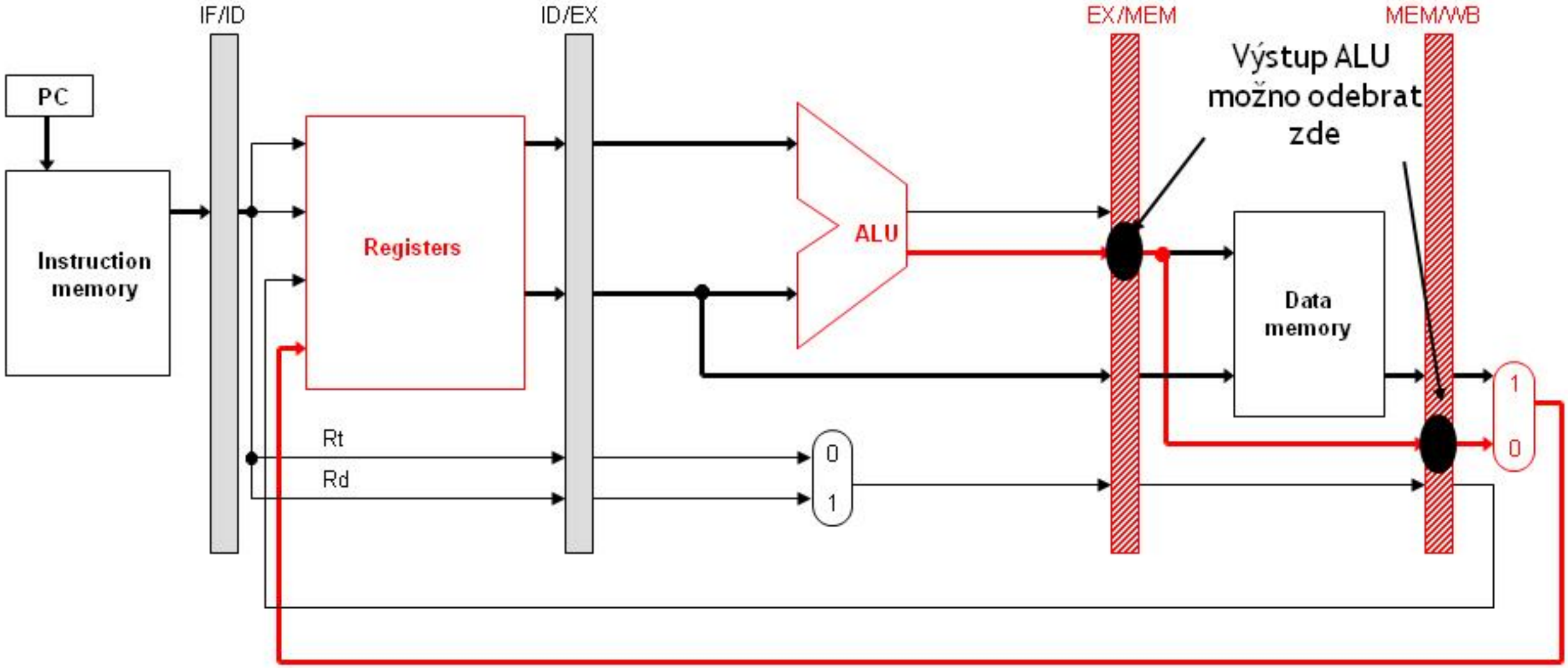
„Bypass“ registrového souboru

- 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.



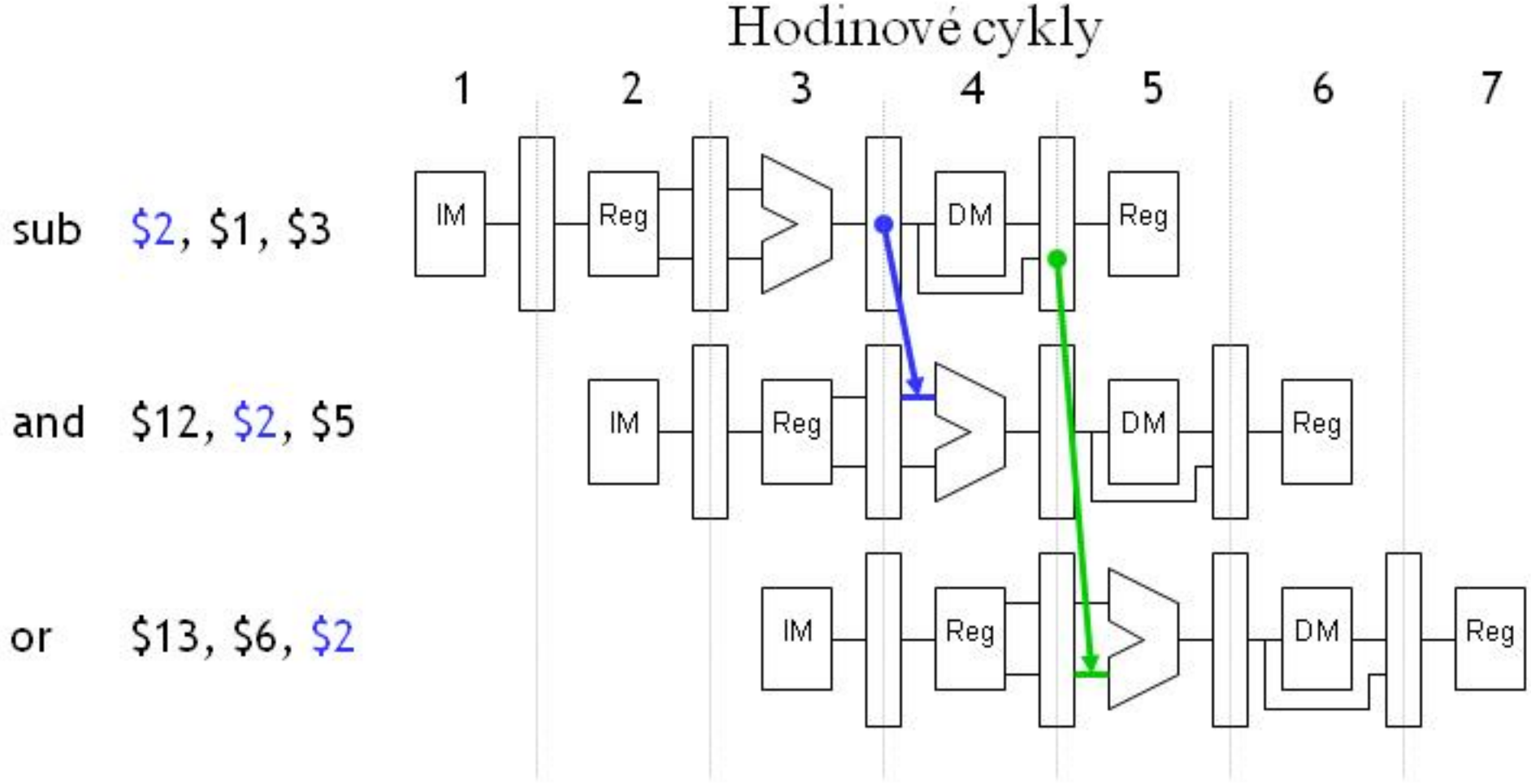
Kde odebrat výsledek ALU

- 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

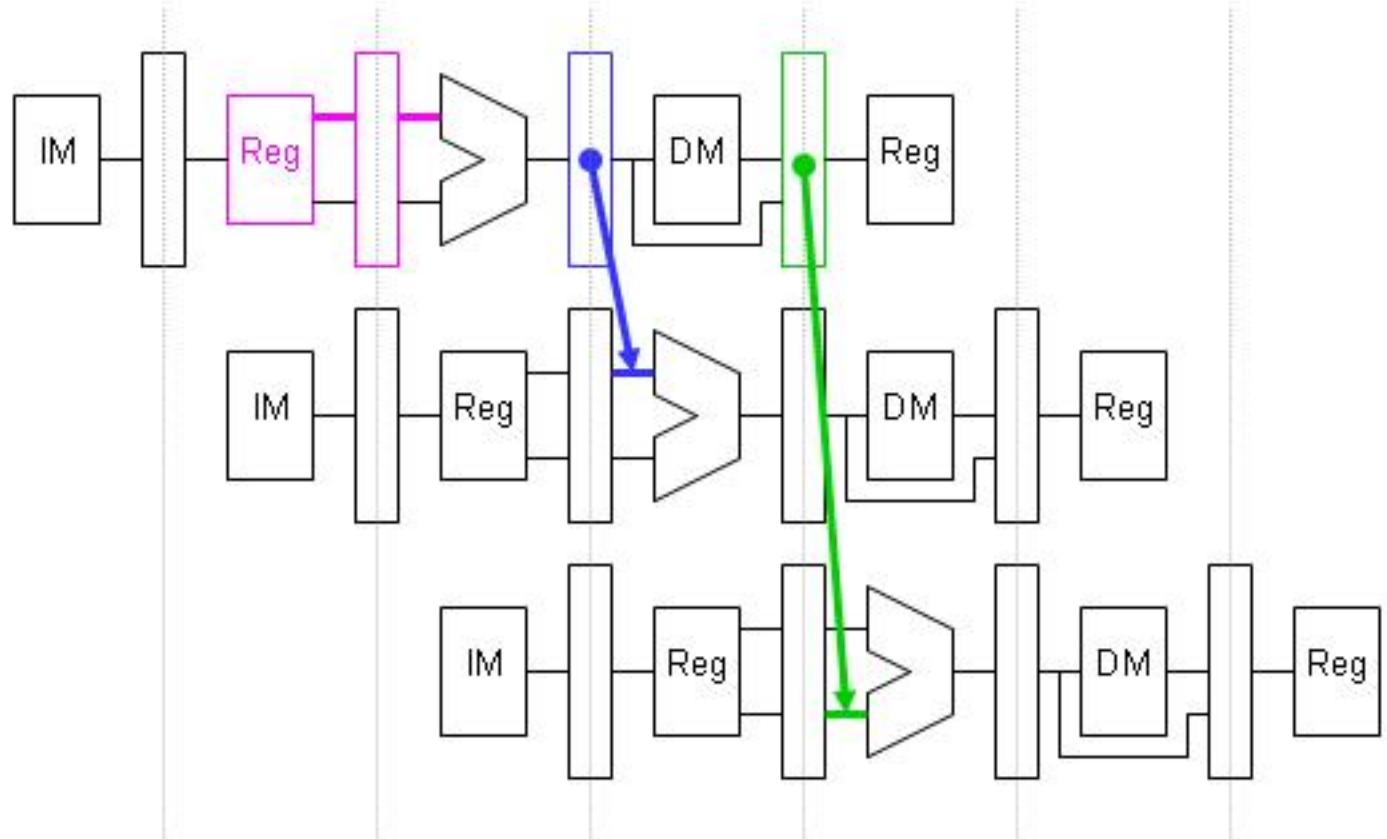
- Protože pipeline registry již obsahují výsledek ALU, přesuneme jej přímo (forward) do následující instrukce, abychom zabránili datovým hazardům.
  - V hodinovém cyklu 4 dostane instrukce AND hodnotu $1 - $3 z pipeline registru EX/MEM, použitém při odečtení (sub).
  - Potom v cyklu 5 instrukce OR dostane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



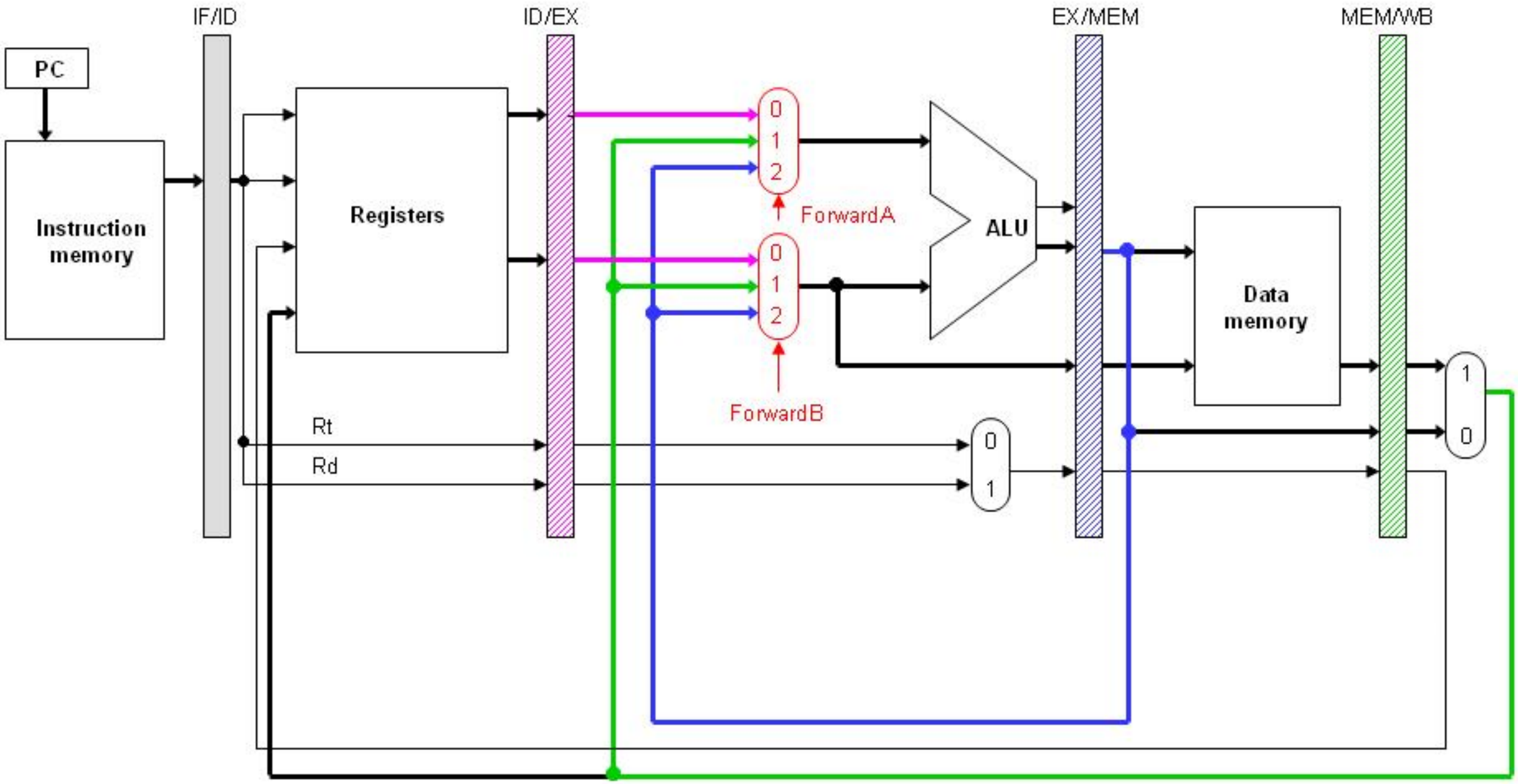
Návrh hardware pro forwarding

- 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.

```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
```



Zjednodušené datové cesty s multiplexery pro forwarding

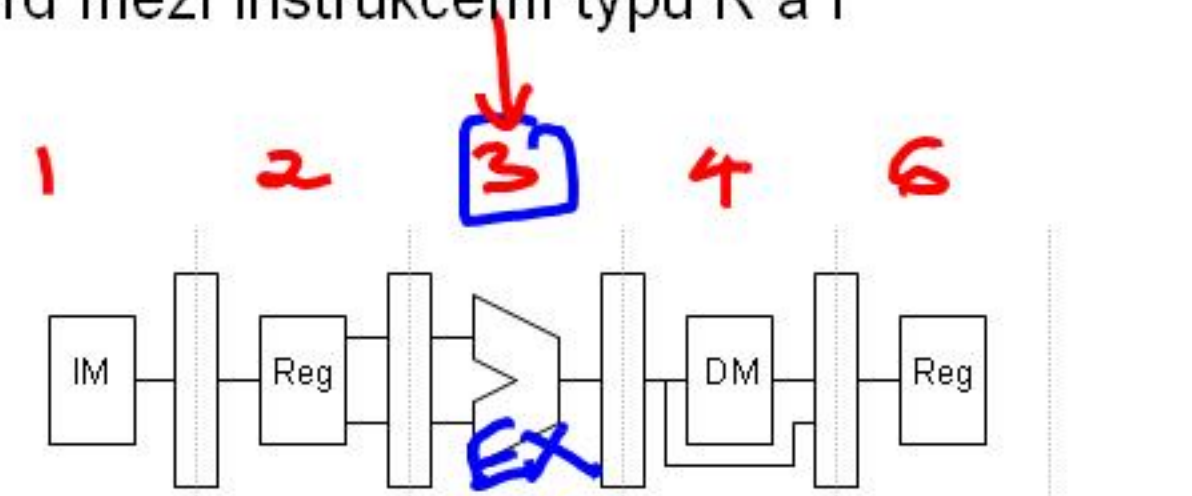


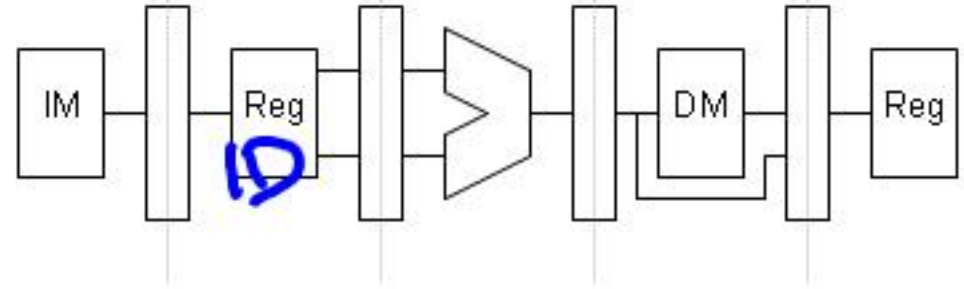
Detekce hazardů EX/MEM

- Sekce: detekuje hazard mezi instrukcemi typu R a I

sub $2, $1, $3

ori $12, $2, 5





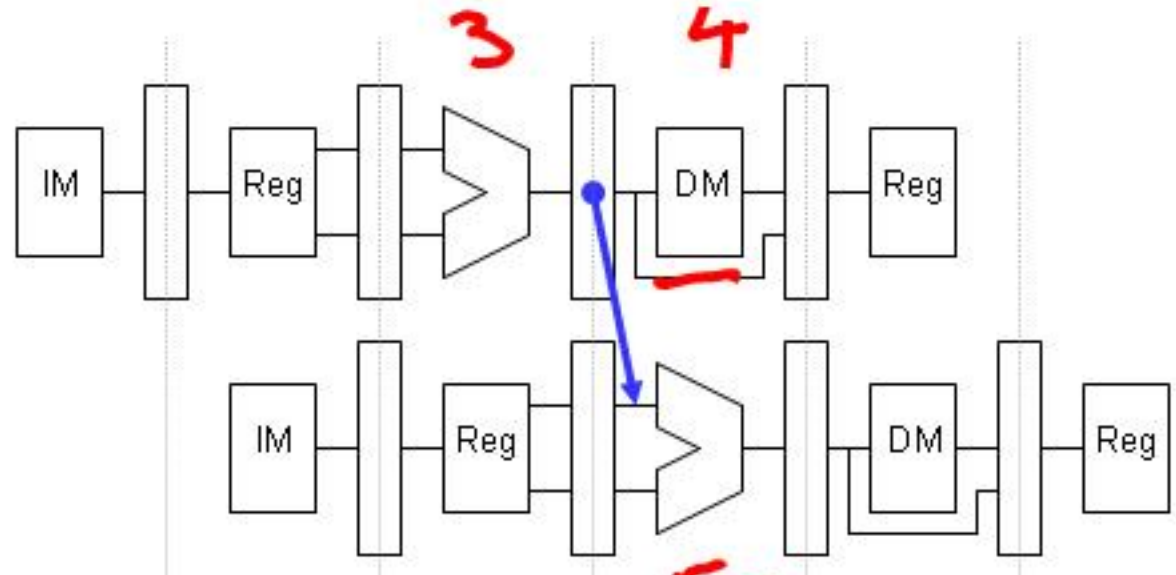
- Nemůže detekovat hazard do cyklu 3: `sub` je ve stupni EX, `ori` je v ID
- Vzniká hazard, protože: ID/EX.RegisterRd = IF/ID.RegisterRs
- V terminologii MP 5 (Java): `instr[EX].rd() == instr[ID].rs()`

Eliminace datových hazardů EX/MEM

- 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.

sub $2, $1, $3

and $12, $2, $5



- 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

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

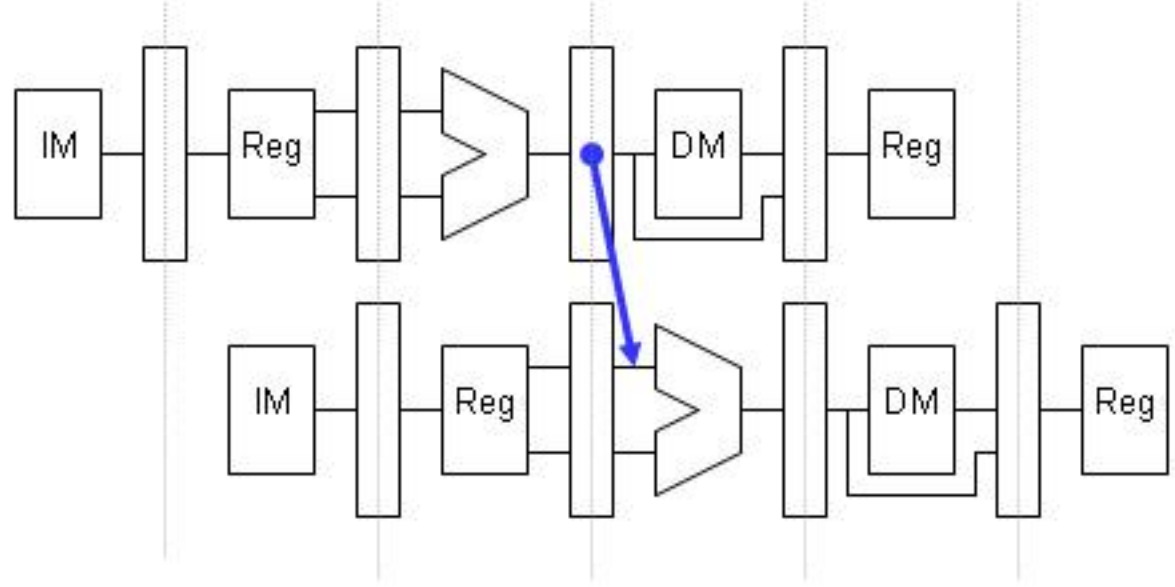
if (EX/MEM.RegWrite = 1
    and EX/MEM.RegisterRd = ID/EX.RegisterRs)
then ForwardA = 2

- 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



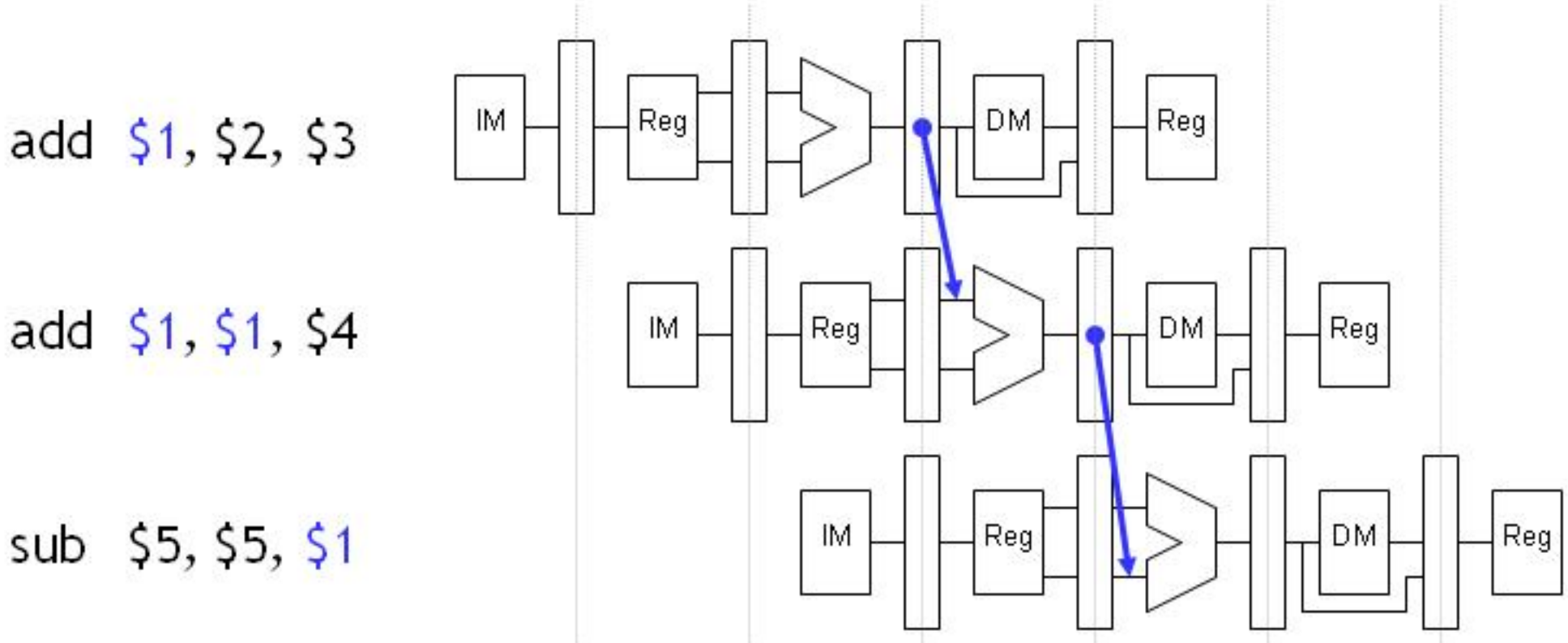
Datové hazardy MEM/WB

- 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.

```
add   $1, $2, $3
add   $1, $1, $4
sub   $5, $5, $1
```

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

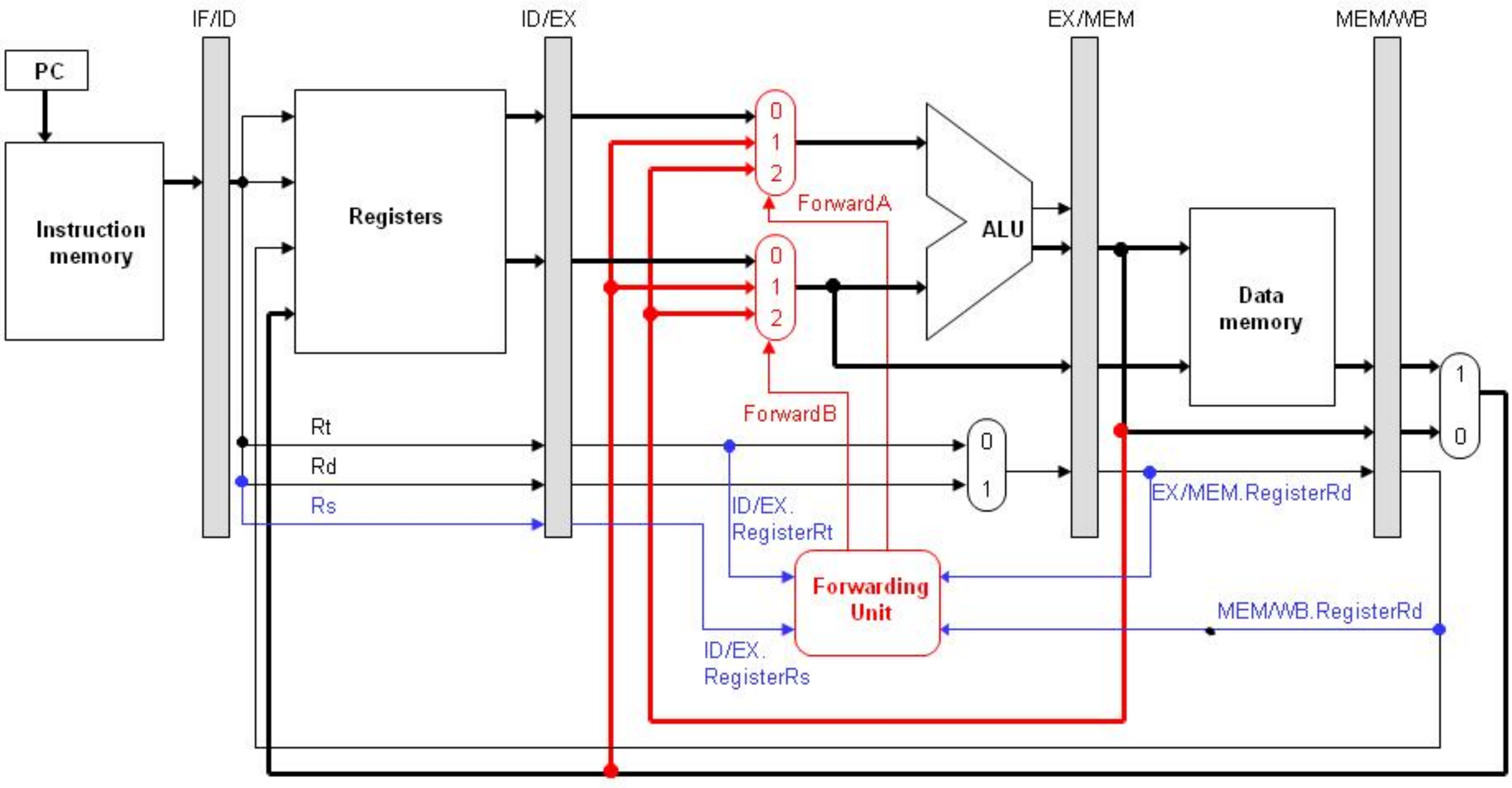
- Rovnice pro detekci a ošetření MEM/WB hazard prvého operandu ALU.

  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



Jednotka pro forwarding

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

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

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

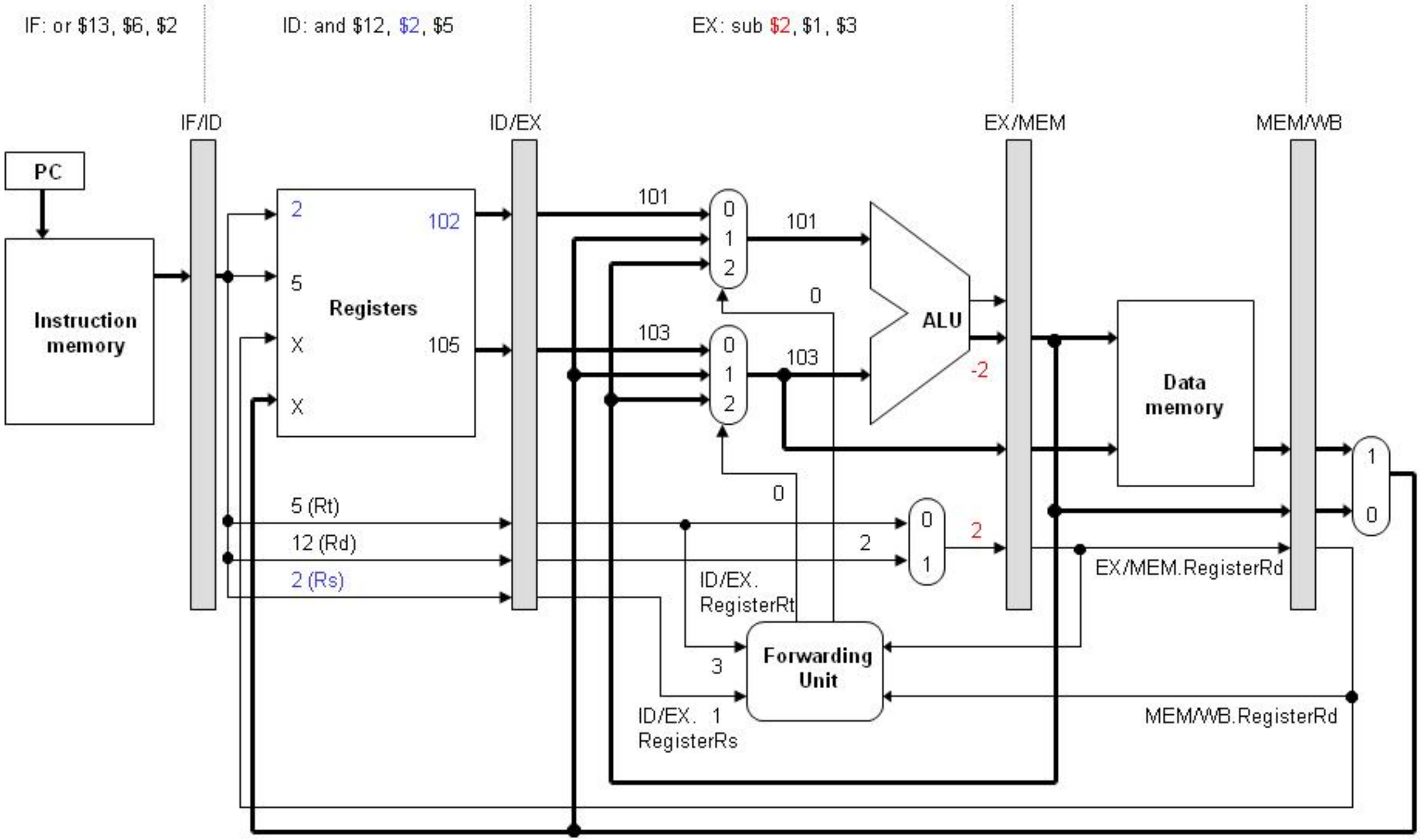
Příklad

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

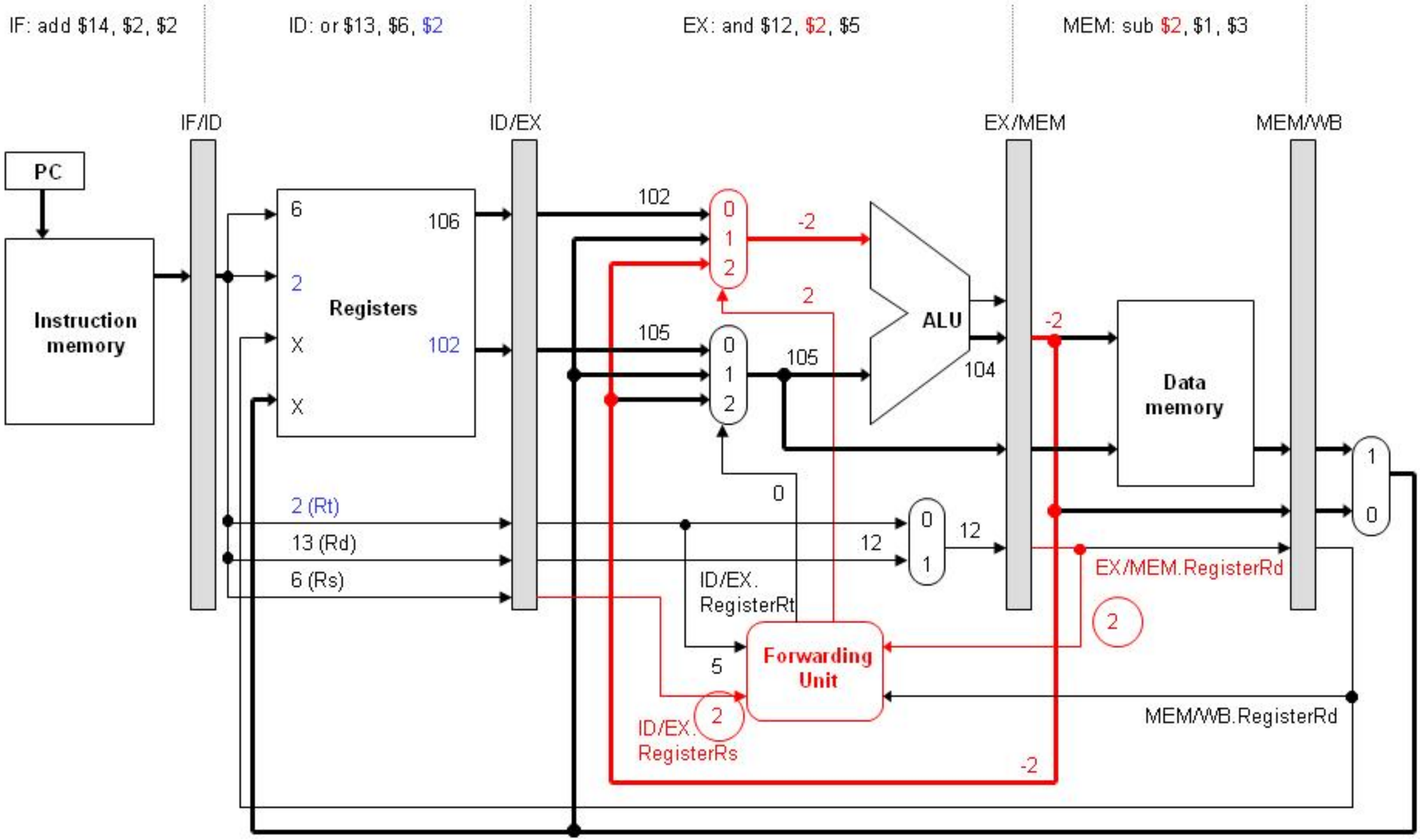
- Předpokládejme, že každý registr obsahuje svoje číslo plus 100.
  - Po provedení prvé instrukce $2 by měl obsahovat −2 (101 − 103).
  - Ostatní instrukce použijí −2 jako jeden z operandů.

- Příklad bude popsán stručně.
  - Předpokládejme, že není třeba žádný forwarding, vyjma registr $2.
  - Přeskočíme první dva cykly, protože jsou shodné jako předtím.

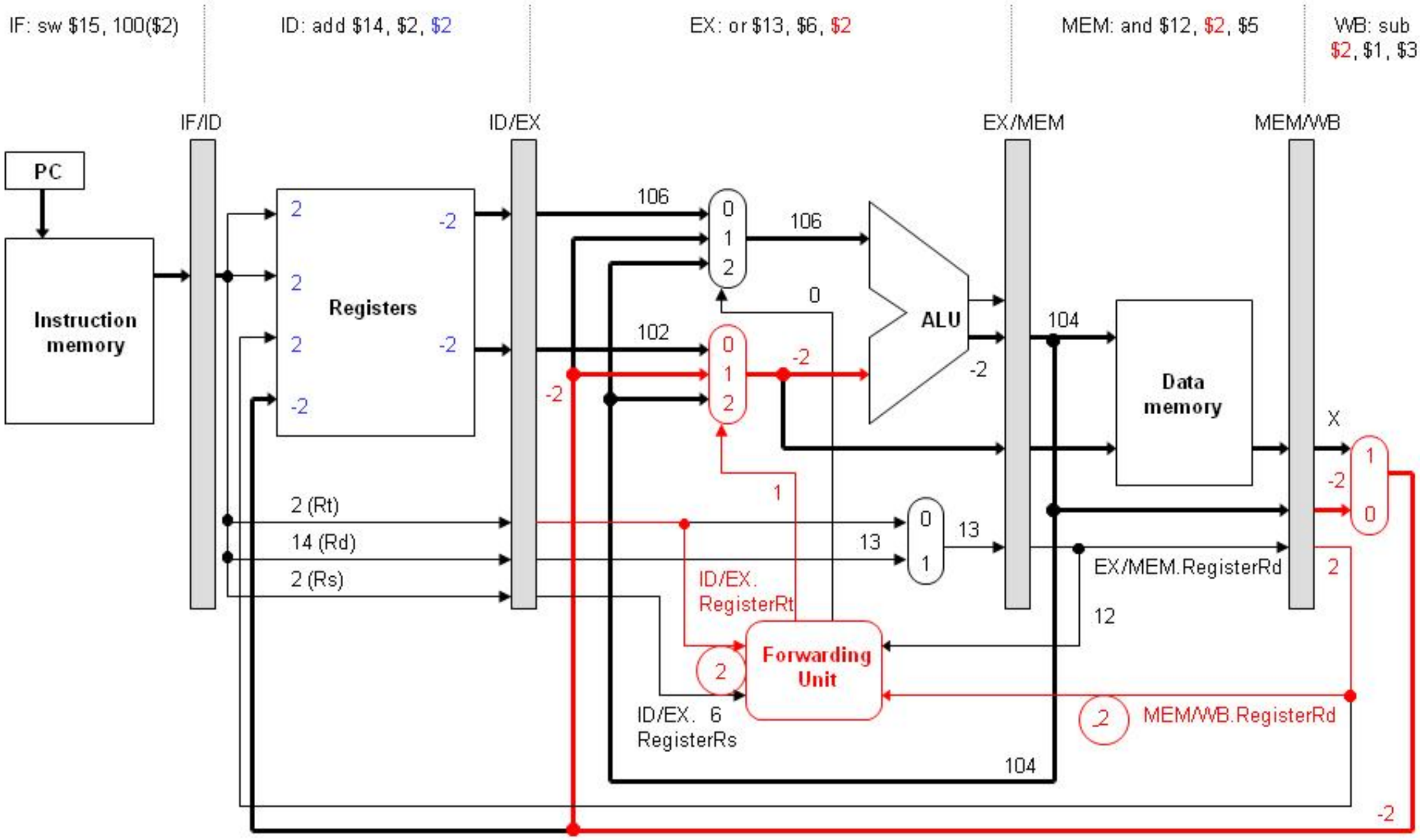
Hodinový cykl 3



Hodinový cykl 4: odběr $2 z EX/MEM



Hodinový cykl 5: odběr $2 z MEM/WB



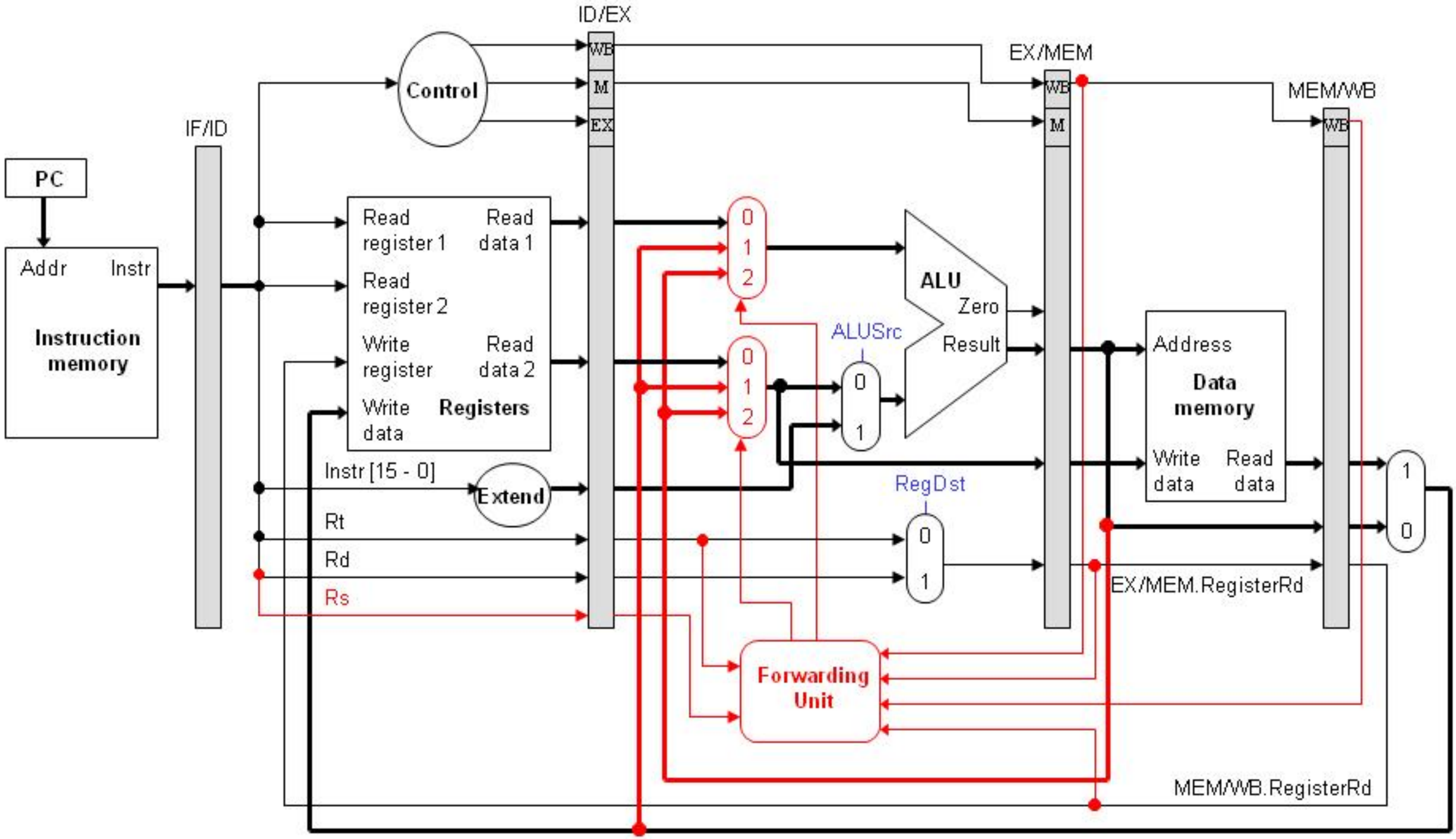
Vzniká řada hazardů

- 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.

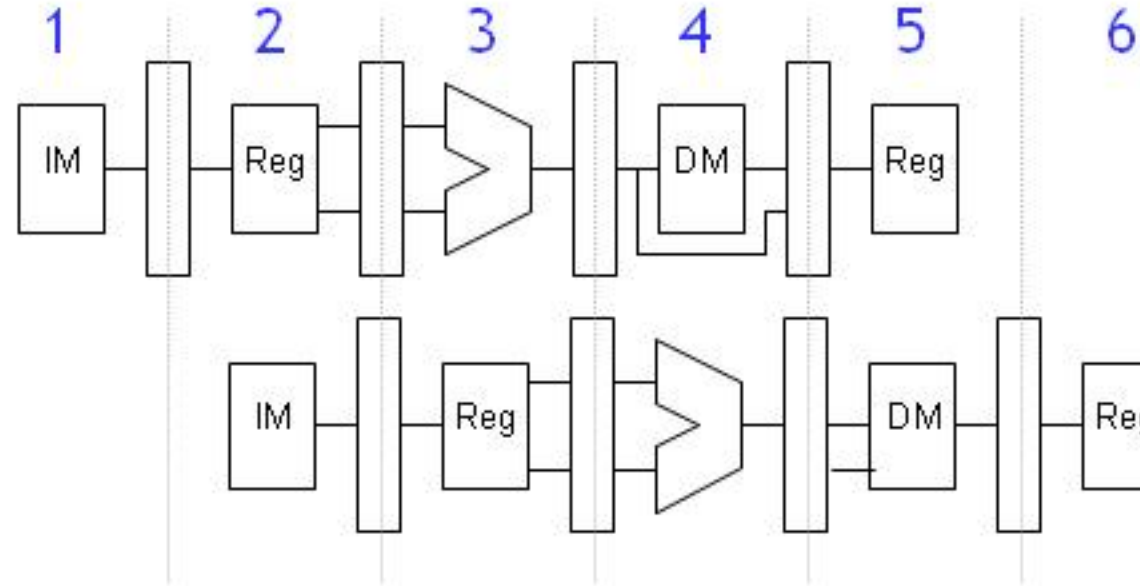
Úplné datové cesty s pipeliningem



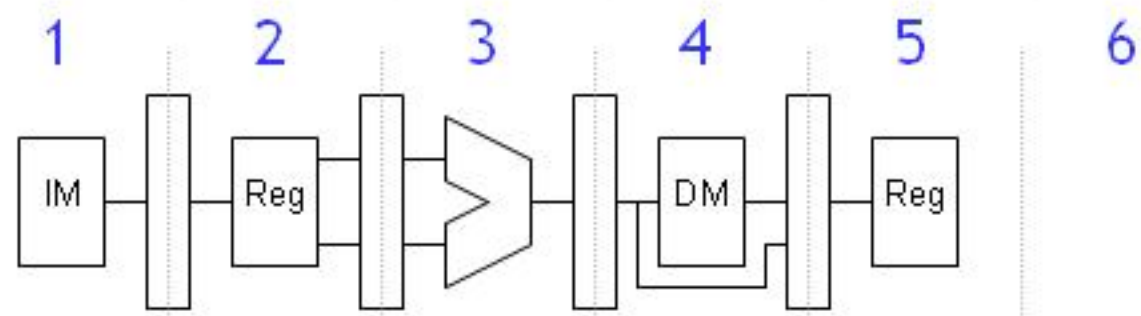
Operace zápisu?

- Dva "jednoduché" případy:

add $1, $2, $3

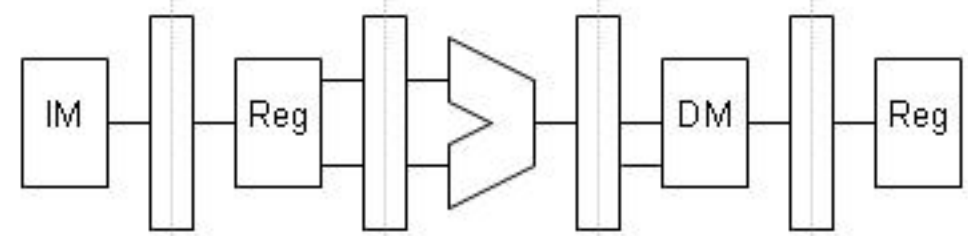


sw $4, 0($1)

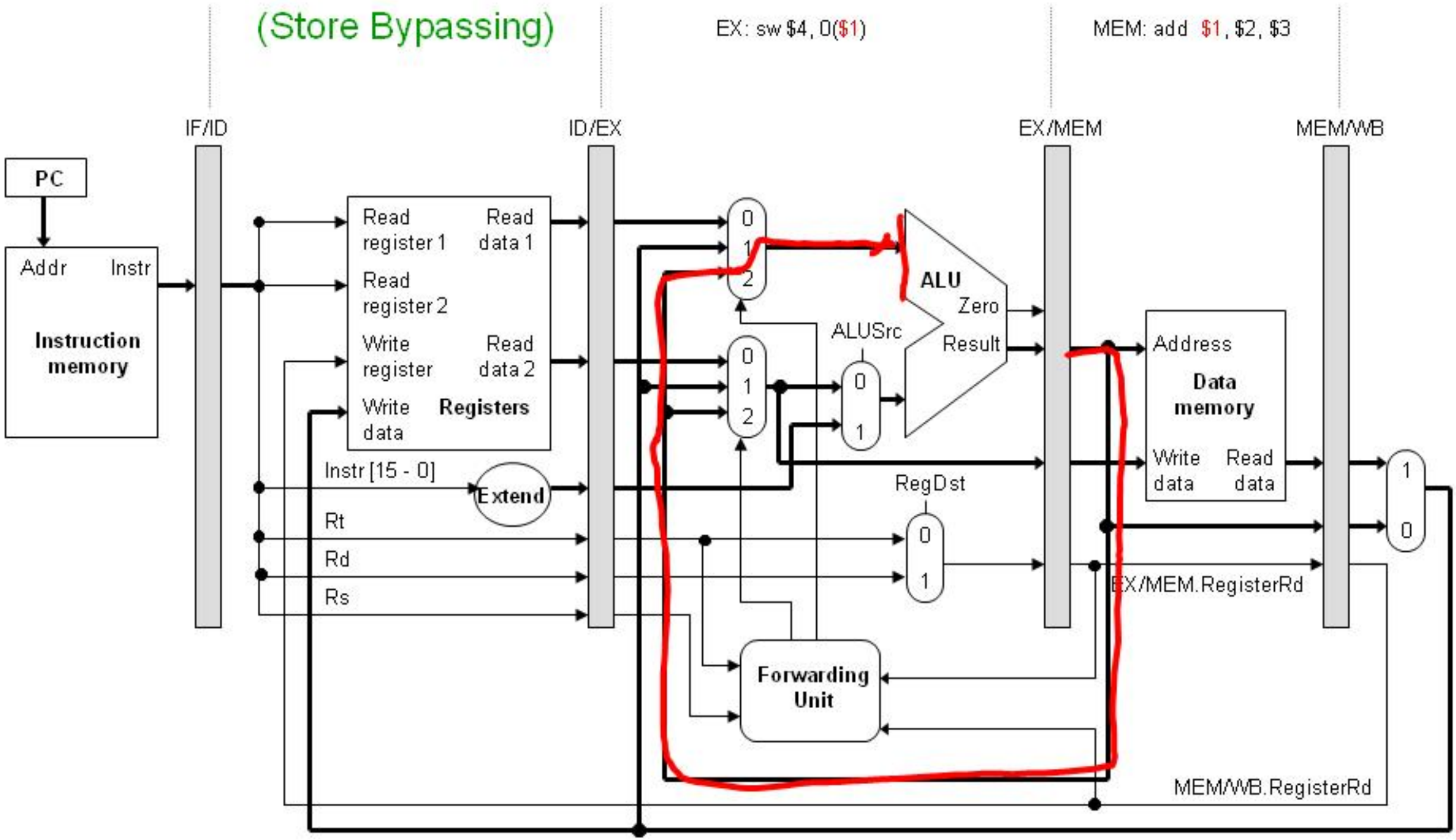


add $1, $2, $3

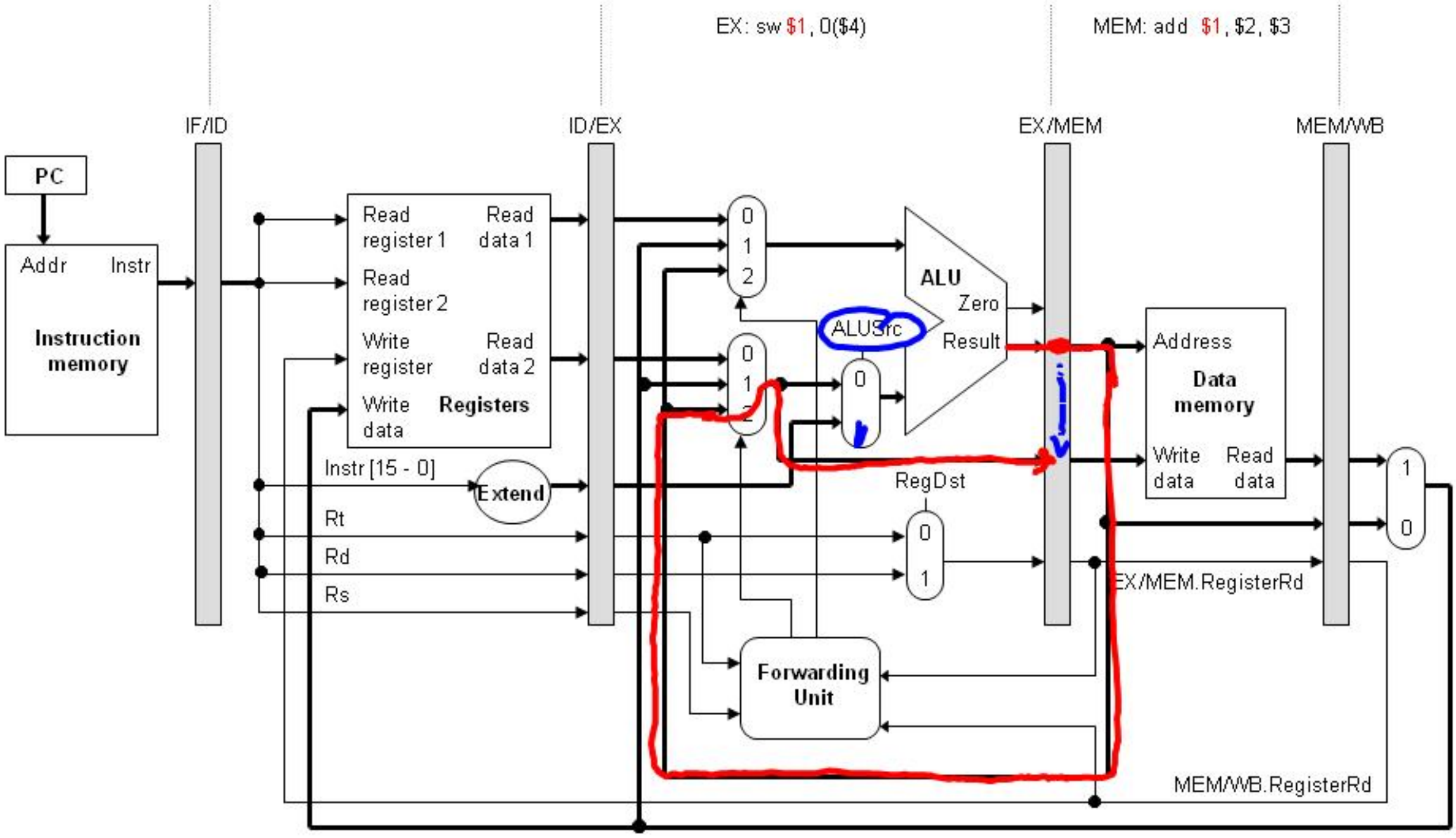
sw $1, 0($4)



Bypass při zápisu: Verze 1



Bypass při zápisu: Verze 2

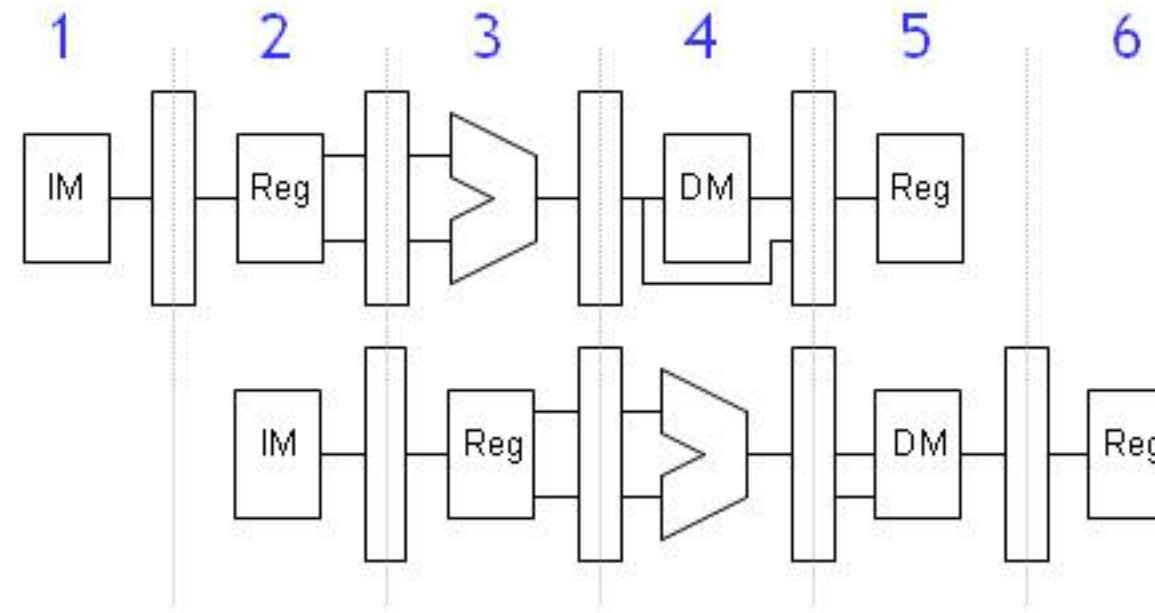


Vlastní zápis?

- Komplikovanější případ:

lw $1, 0($2)

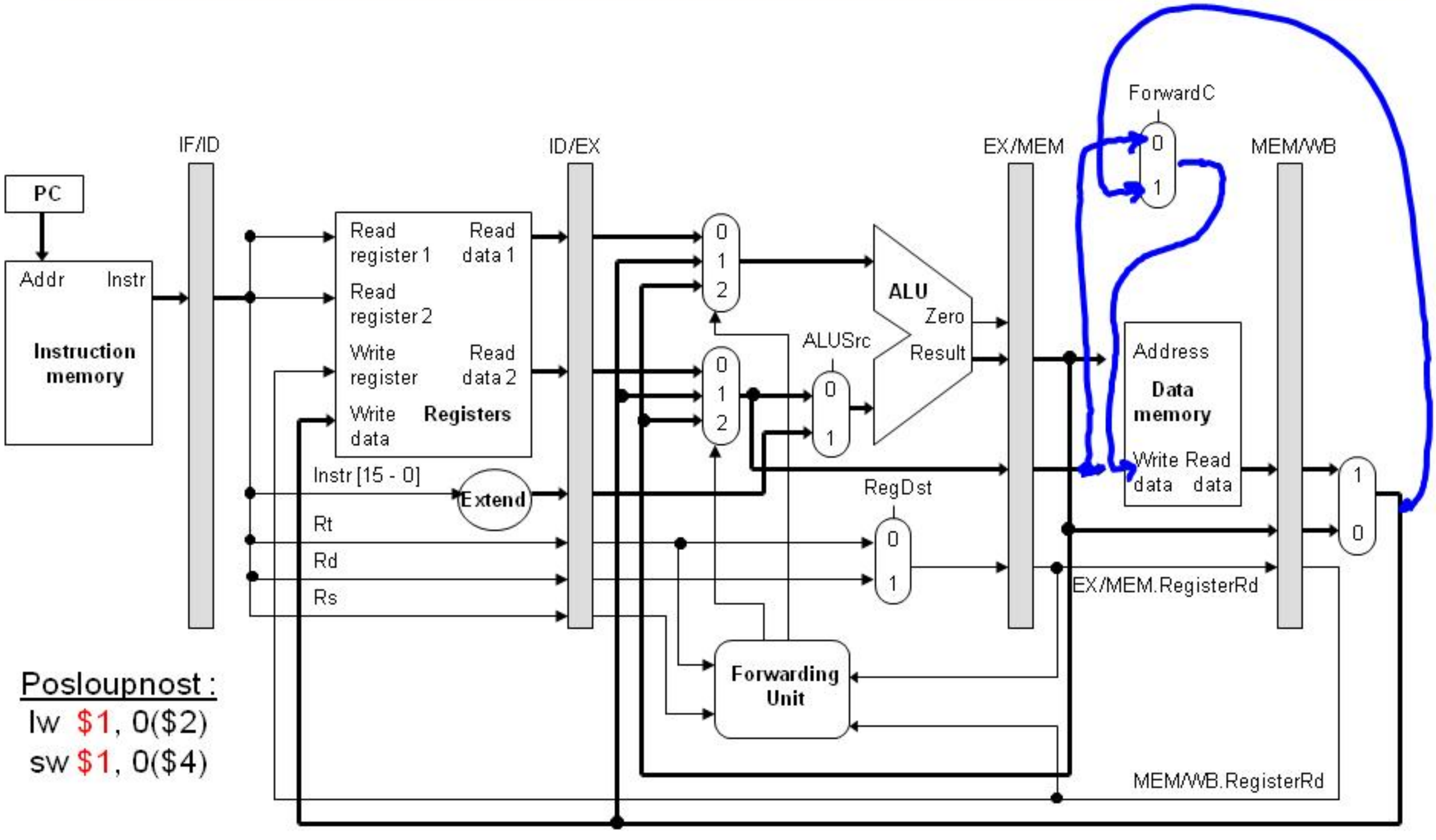
sw $1, 0($4)



- 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/Store bypass



Posloupnost :

lw $1, 0($2)

sw $1, 0($4)

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.

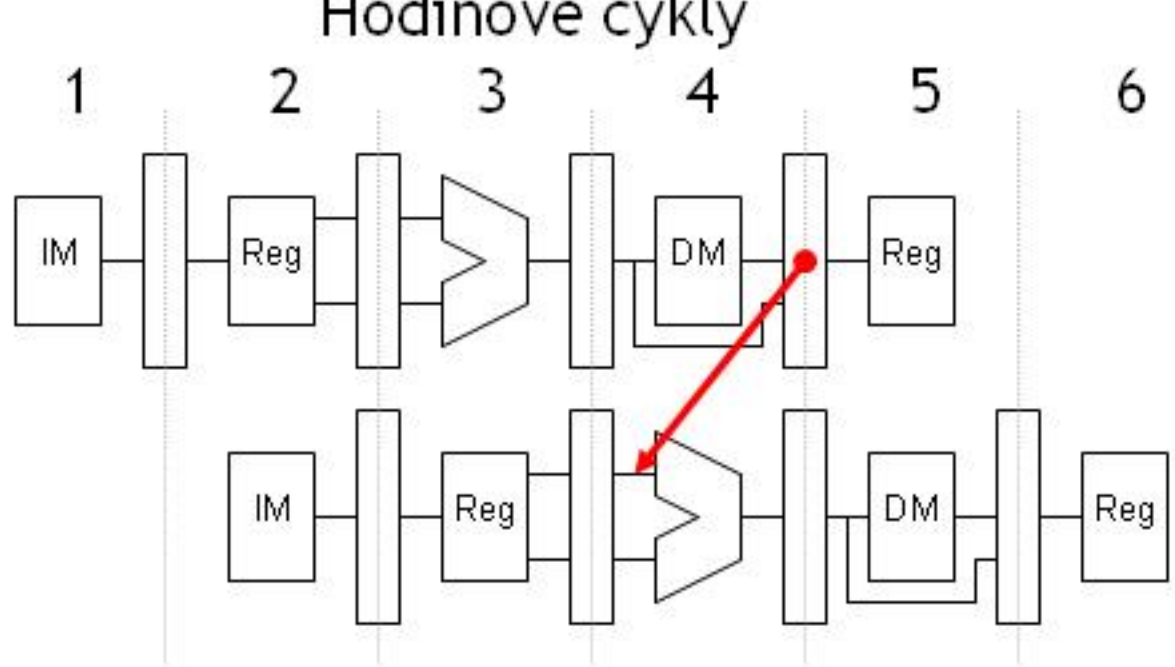
- Sekce 6.4 učebnice obsahuje některé doplňující materiály, které zde nebyly uvedeny.
  - Rovnice pro detekci hazardu testuje, zda zdrojový registr není $0, který nikdy nemůže být modifikován.
  - Je uveden složitější příklad forwardingu.

Problém se čtením

- 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.

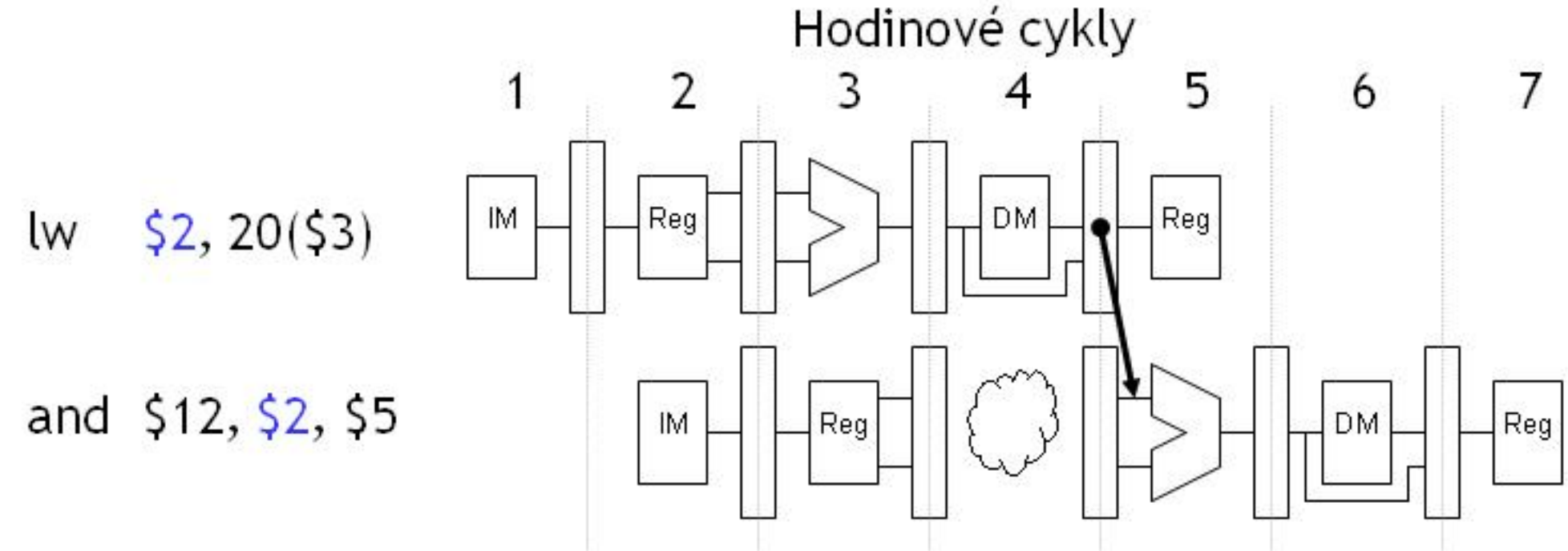
lw $2, 20($3)

and $12, $2, $5



Pozastavení

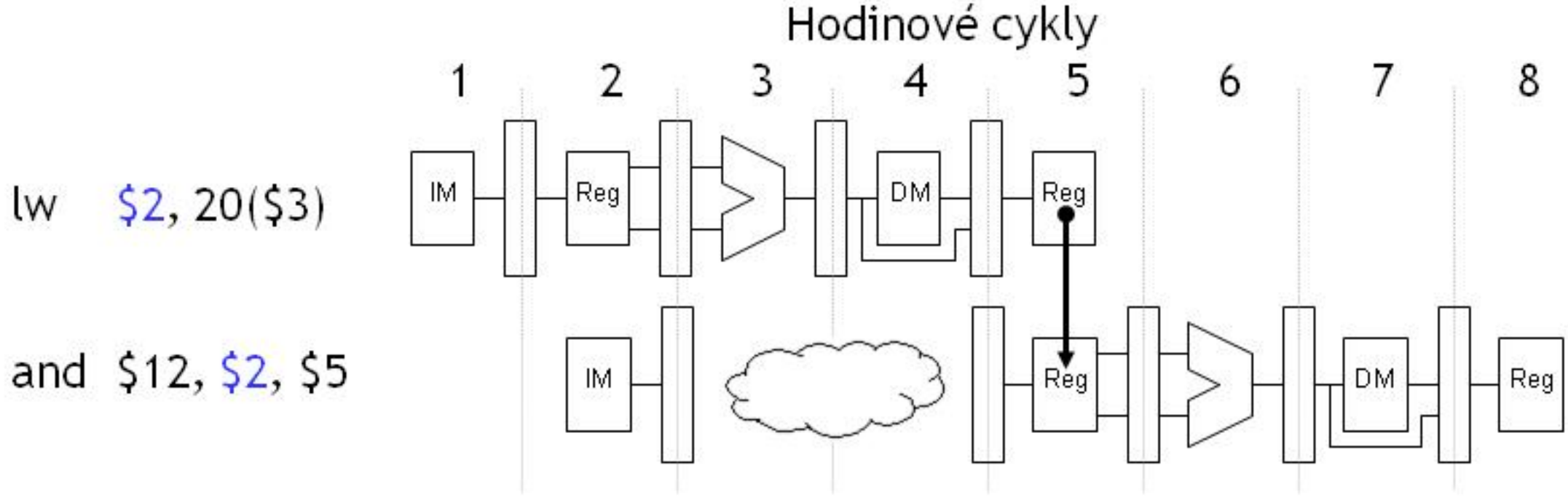
- Nejjednodušším řešením je zastavit pipeline (stall).
- Zpozdíme instrukci AND zařazením zpoždění v délce 1 cyklu do pipeline. Toto zpoždění se nazývá bublina.



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

Pozastavení a „forwarding“

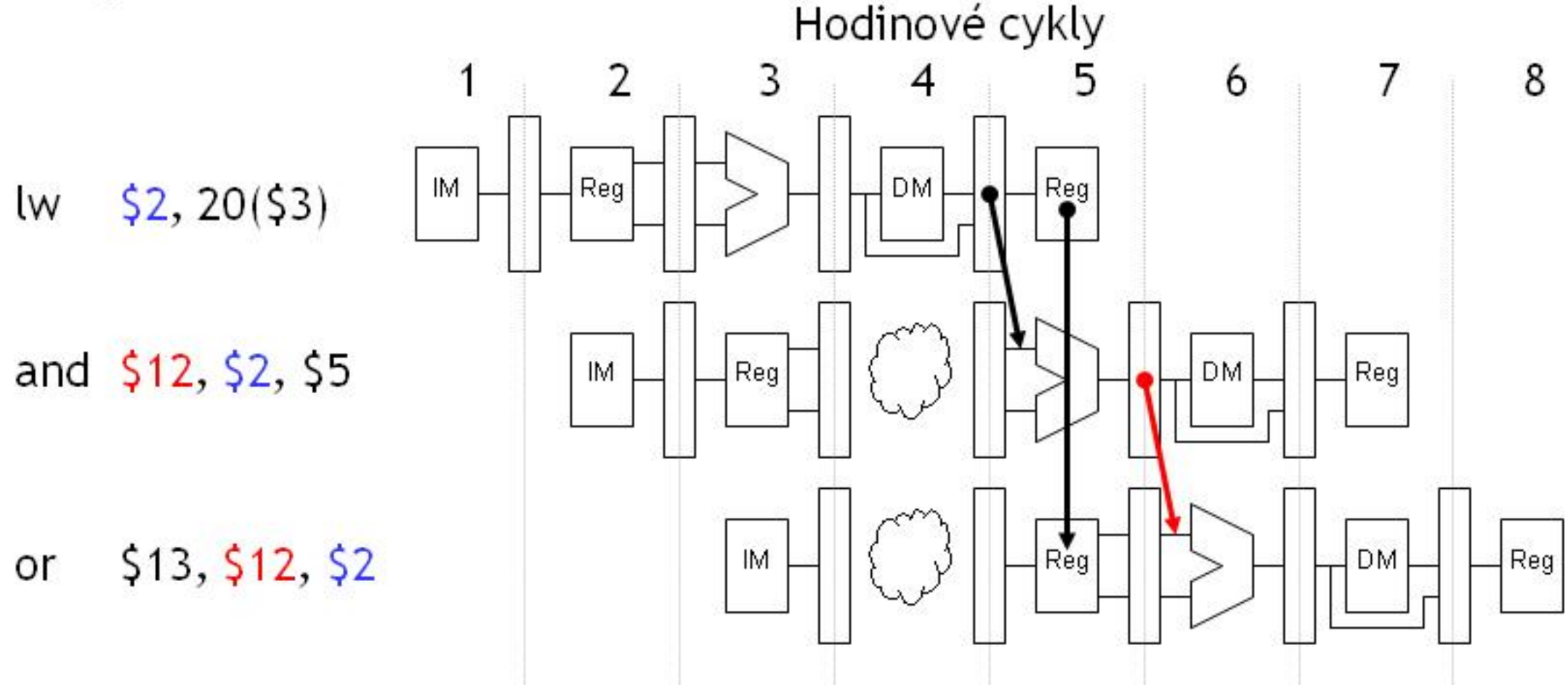
- Bez forwardingu bychom byli nuceni pozastavit na *dva* cykly a čekat na provedení zpětného zápisu instrukce LW.



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

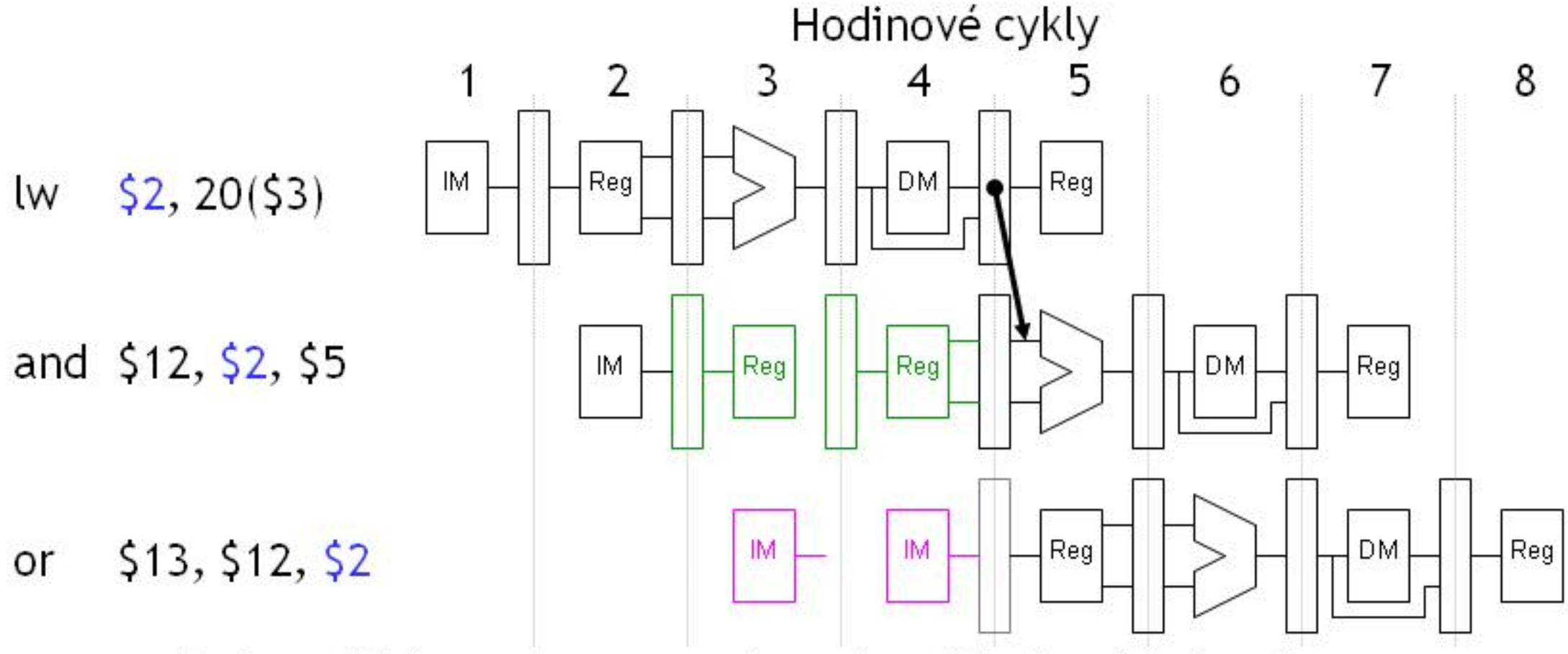
Pozastavení zdrží celou pipeline

- 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.



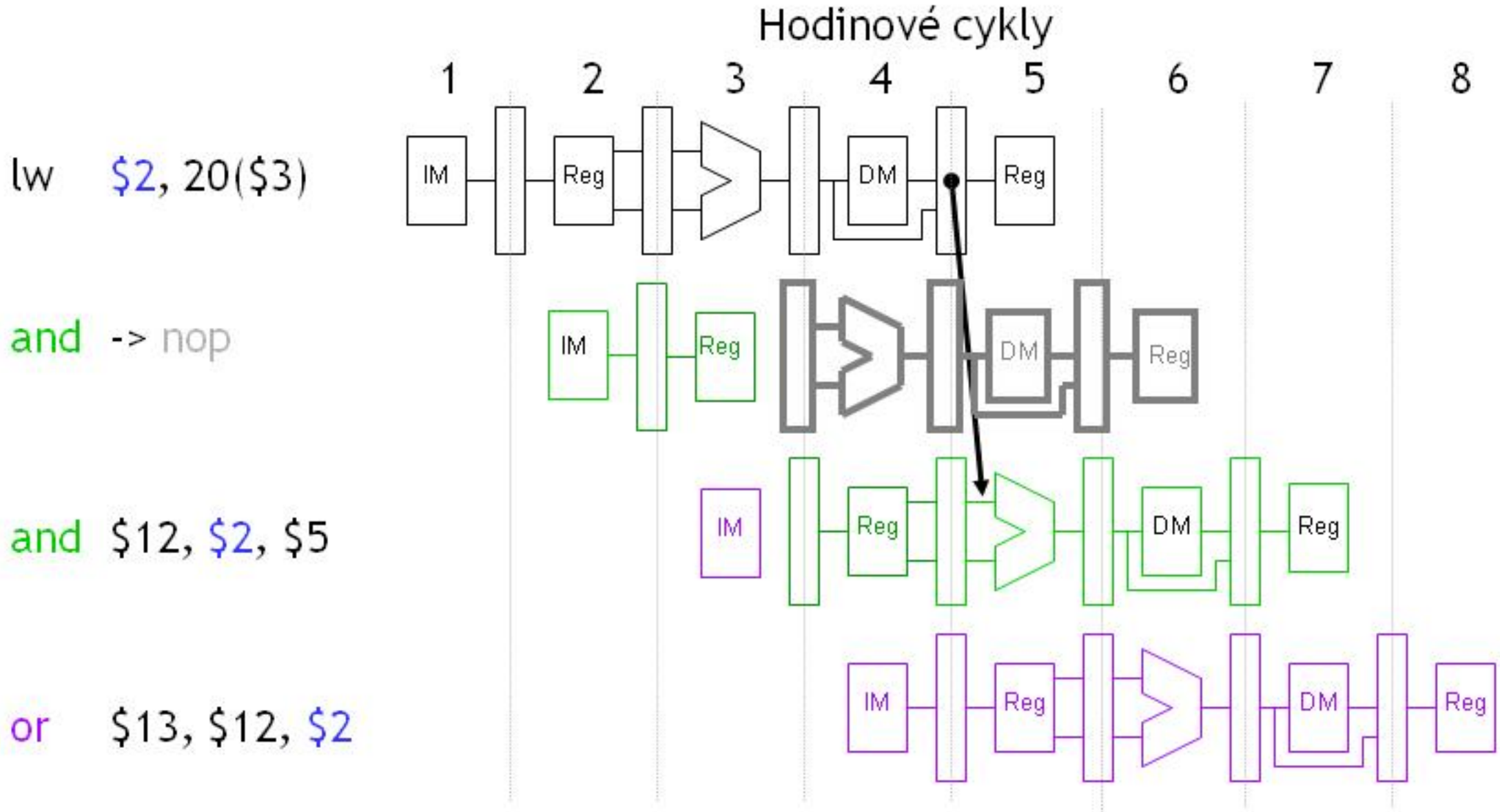
Co s registry EX, MEM, WB ?

- Co 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 v zhledem k pozastavení využity a proto řídící signály EX, MEM a WB jsou neaktivní („0“).

Pozastavení = konverze na Nop



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

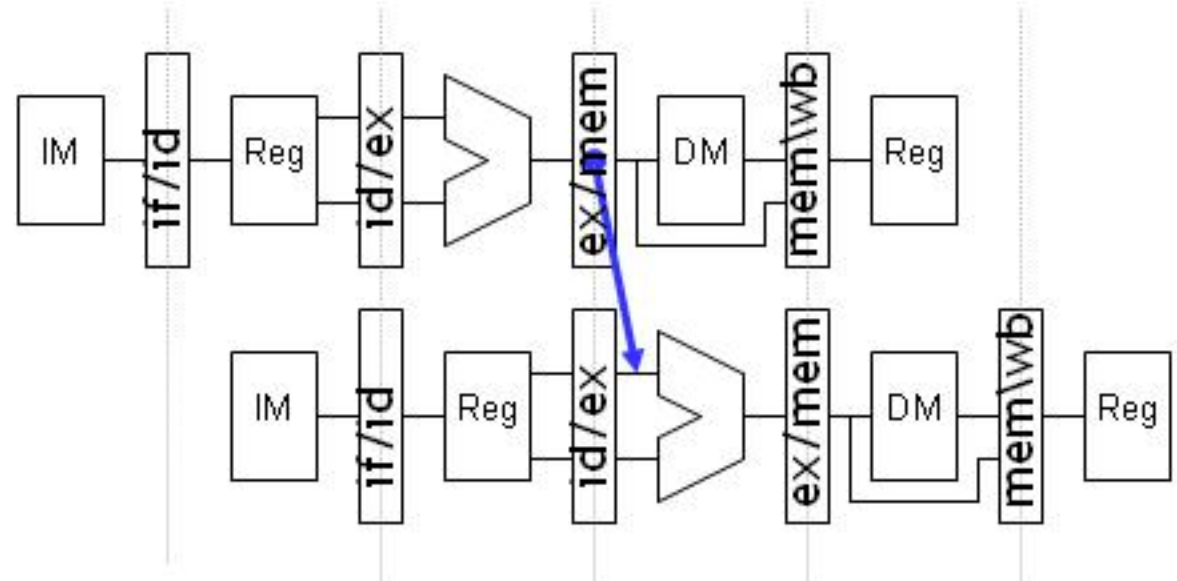
Detekce pozastavení

- 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

sub $2, $1, $3

and $12, $2, $5

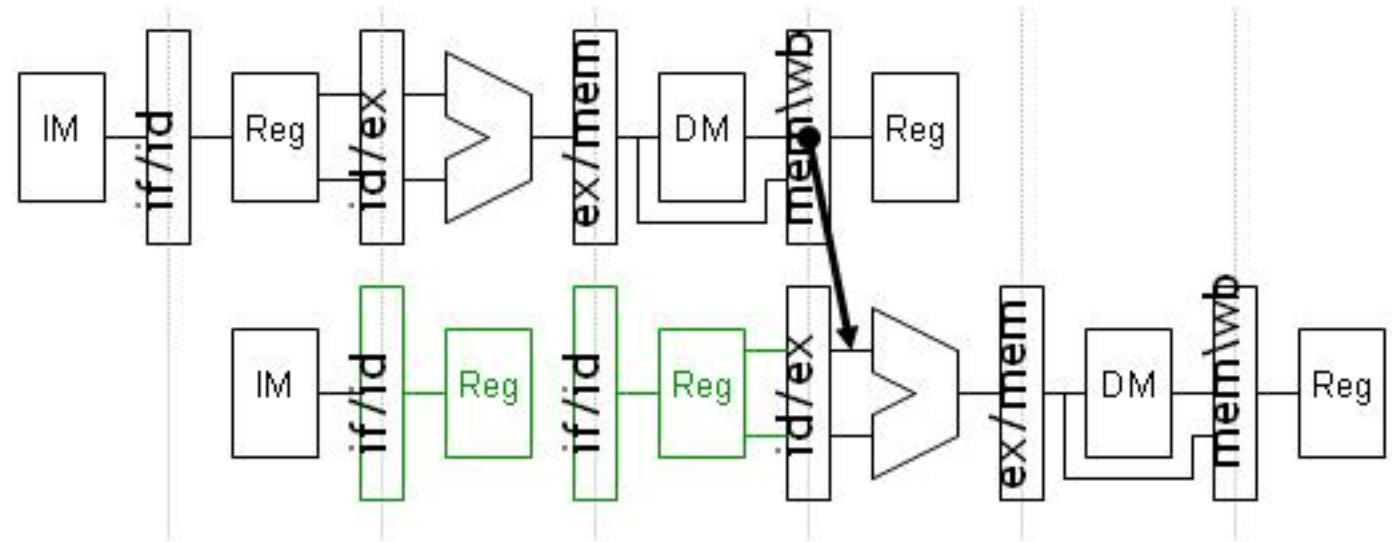


Detekce pozastavení, pokračování

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

lw $2, 20($3)

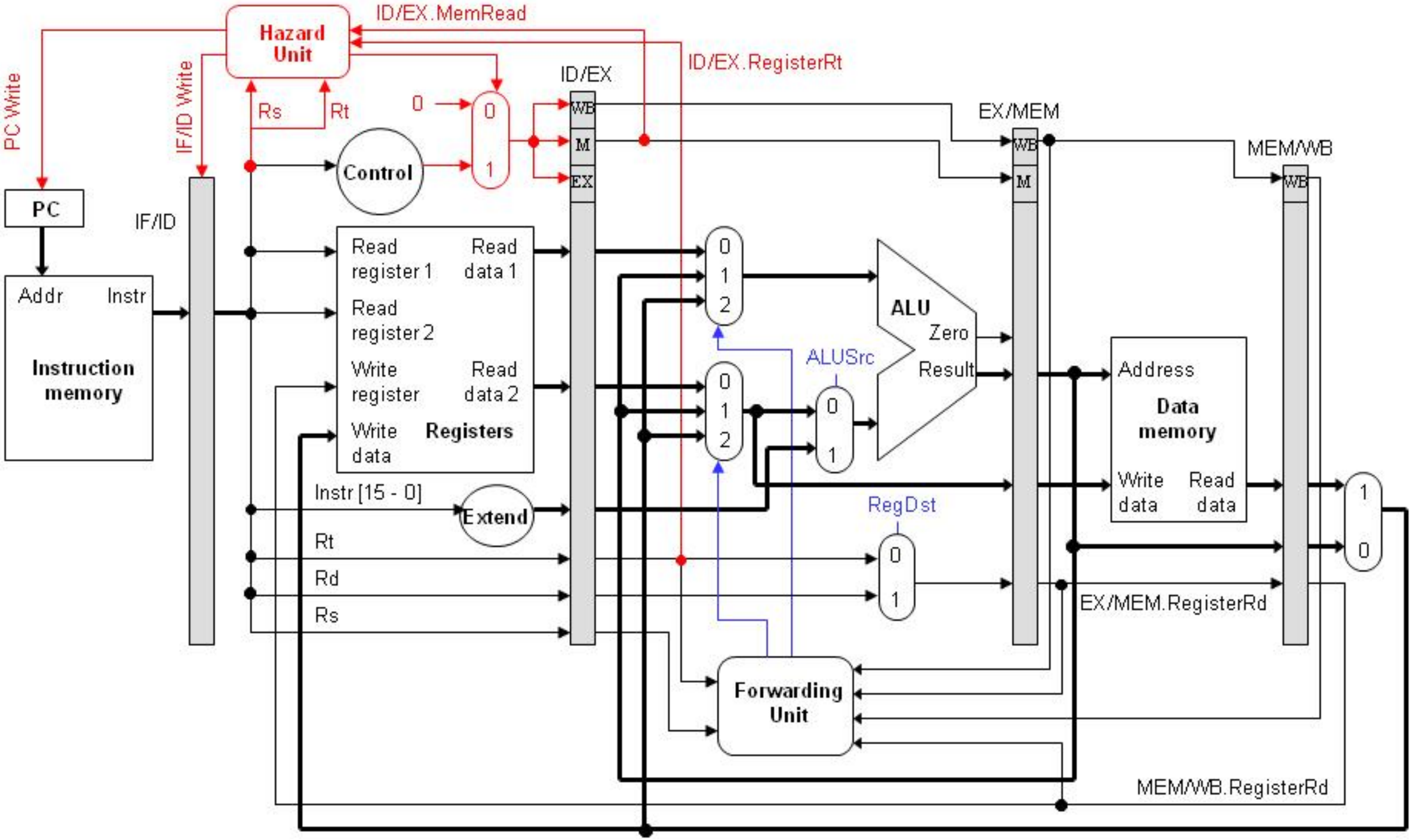
and $12, $2, $5



- 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



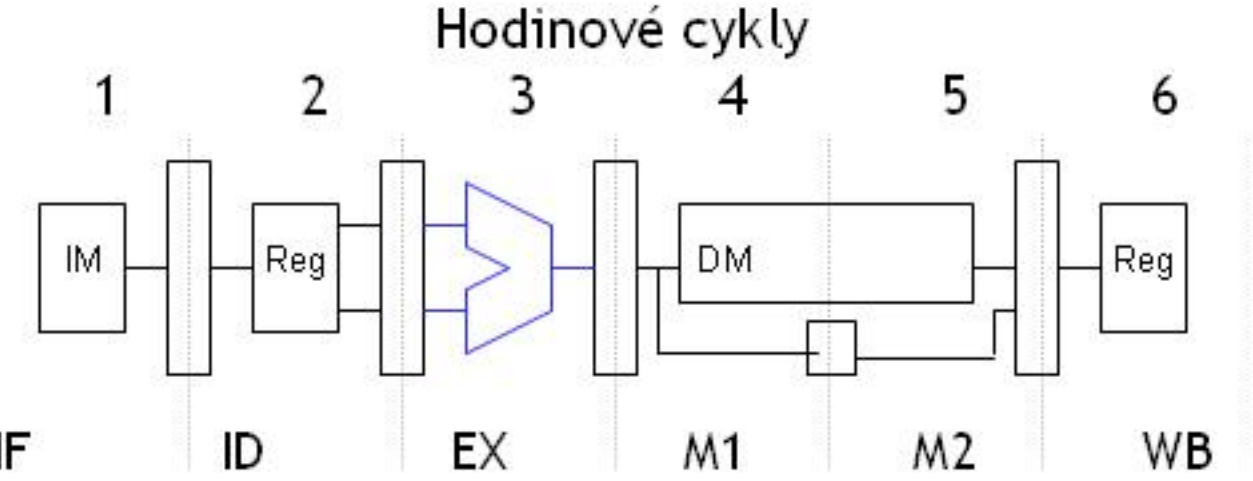
Jednotka pro detekci hazardů

- 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 řídící 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 řídících signálech pro aktuální stupeň EX a příští stupeň MEM/WB v případě, že nastane „stall“.

Zobecnění forwardingu/pozastavení

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



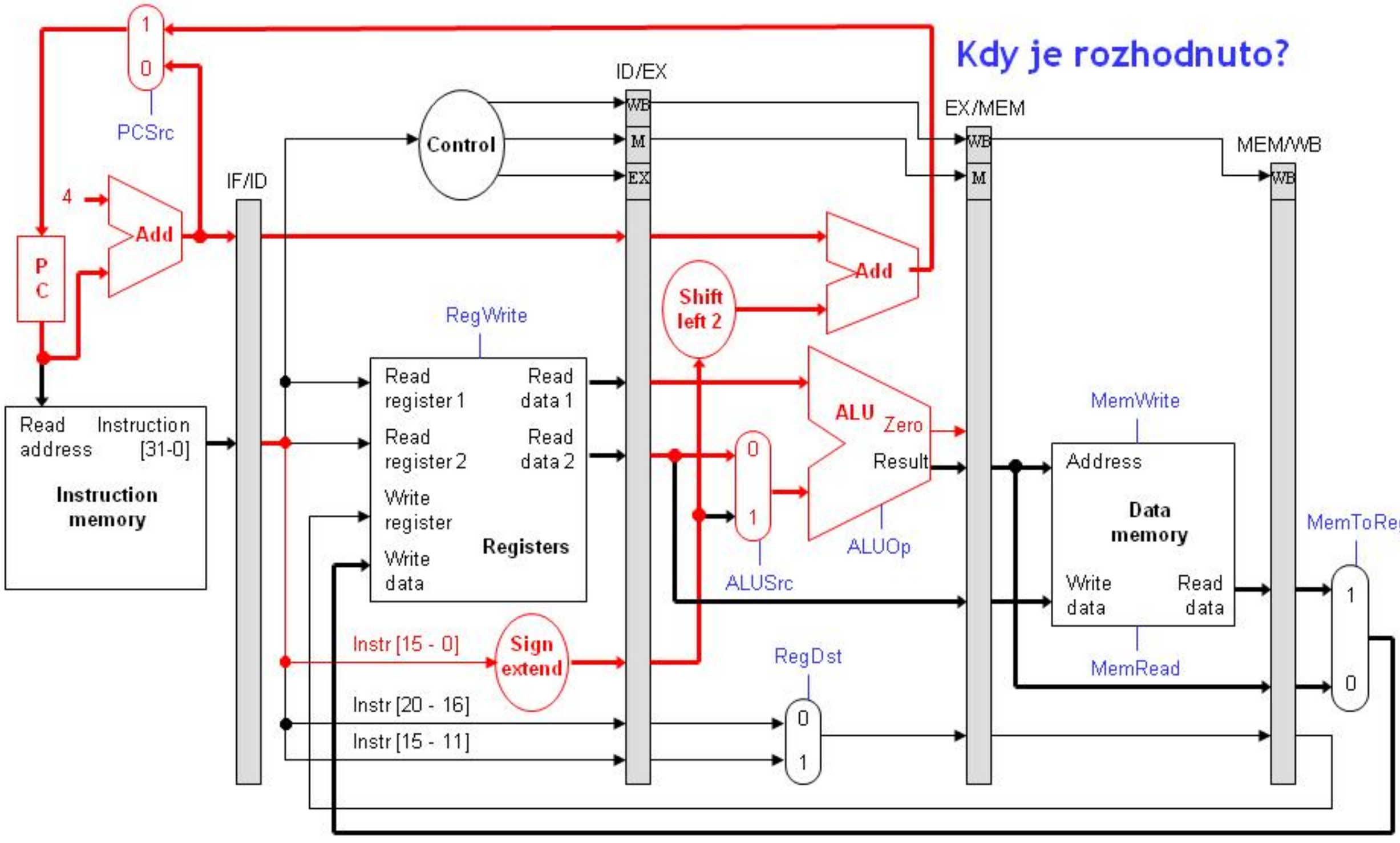
- Kolik vstupů pro bypass musí mít multiplexery ve stupni EX? (Odpověď: 4)
- Které instrukce v příkladu vyžadují pozastavení and/or bypass?

```
lw    $3, 0($1)
add $7, $8, $9

add $5, $7, $3
```

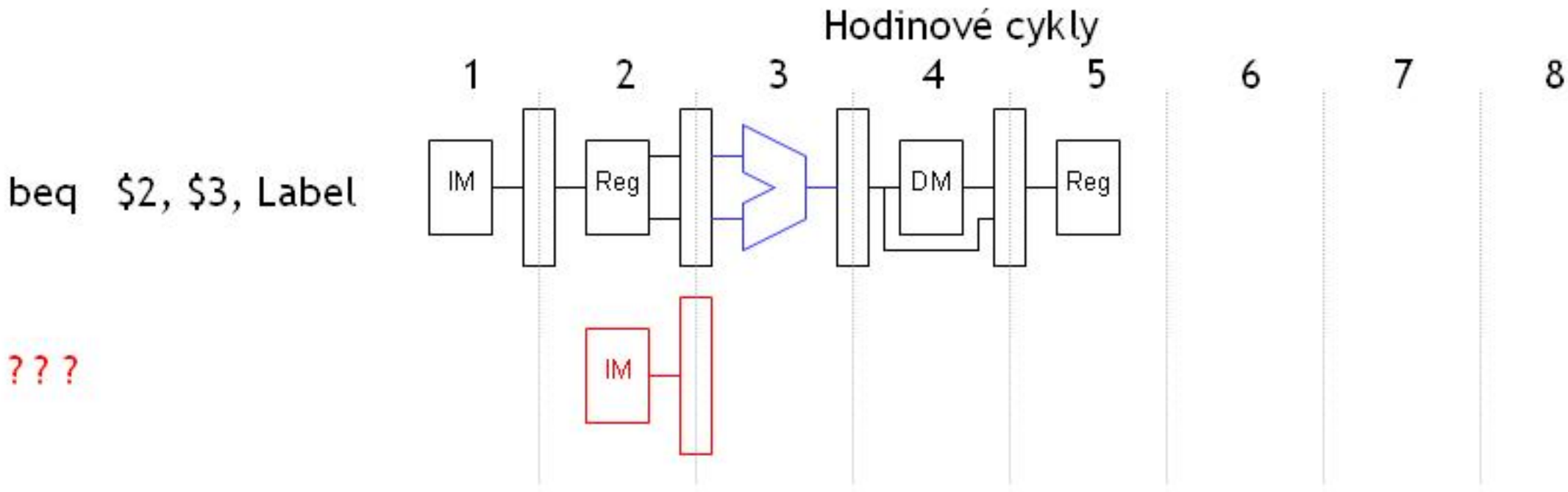
IF	ID	EX	M1	M2	WB				
	IF	ID	EX	M1	M2	WB			
		IF	ID						
				ID	EX	M1	M2	WB	

Větvení v původní jednotce s pipeliningem



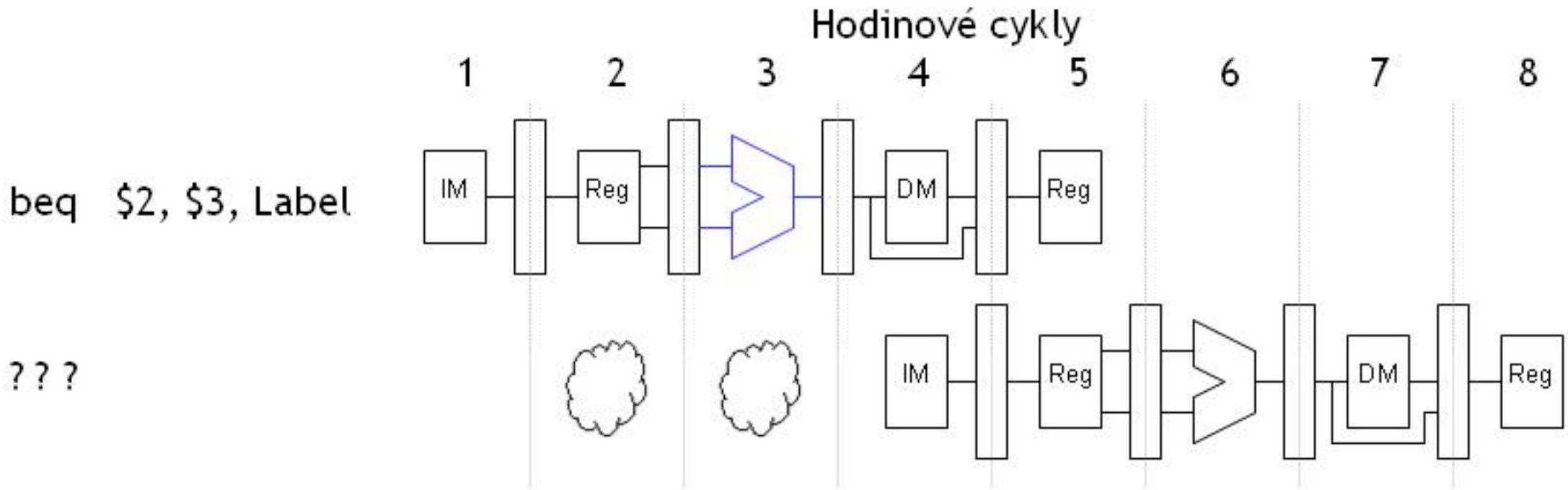
Podmíněné skoky - větvení

- 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 řídicí hazard.



Pozastavení je jen jedno řešení

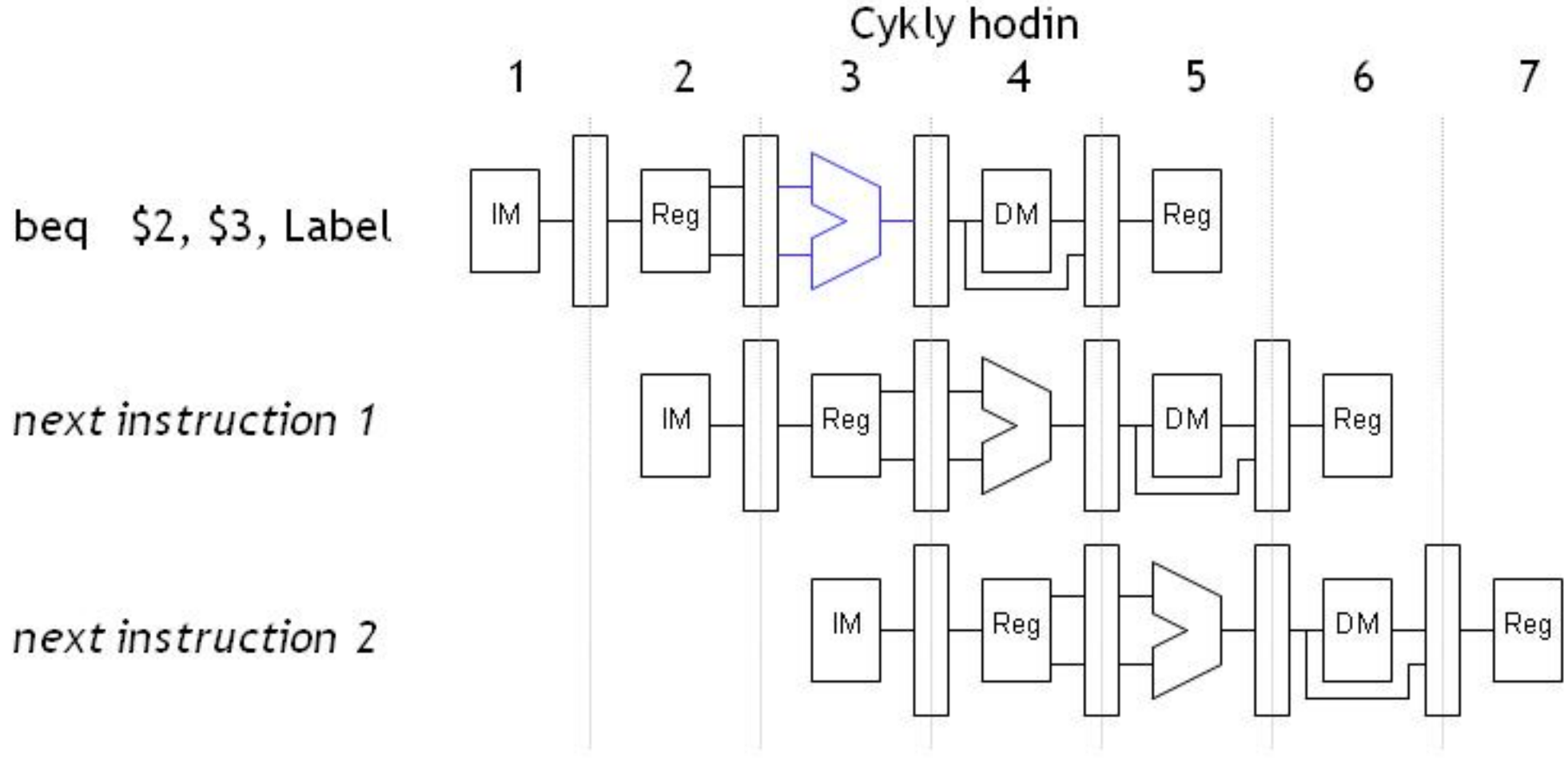
- Přesto, pozastavením lze situaci vždy řešit.



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

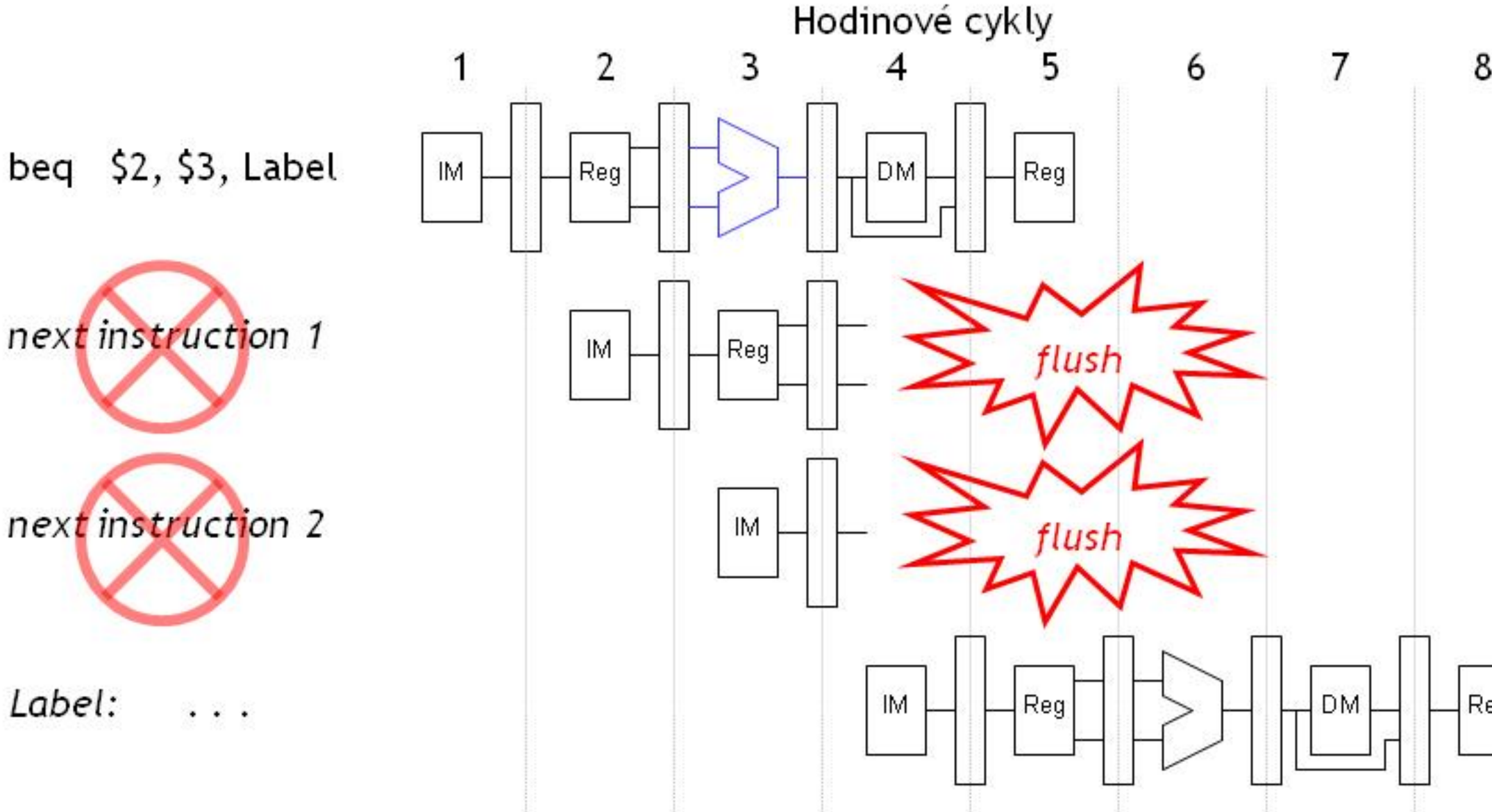
Predikce skoků

- 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 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).



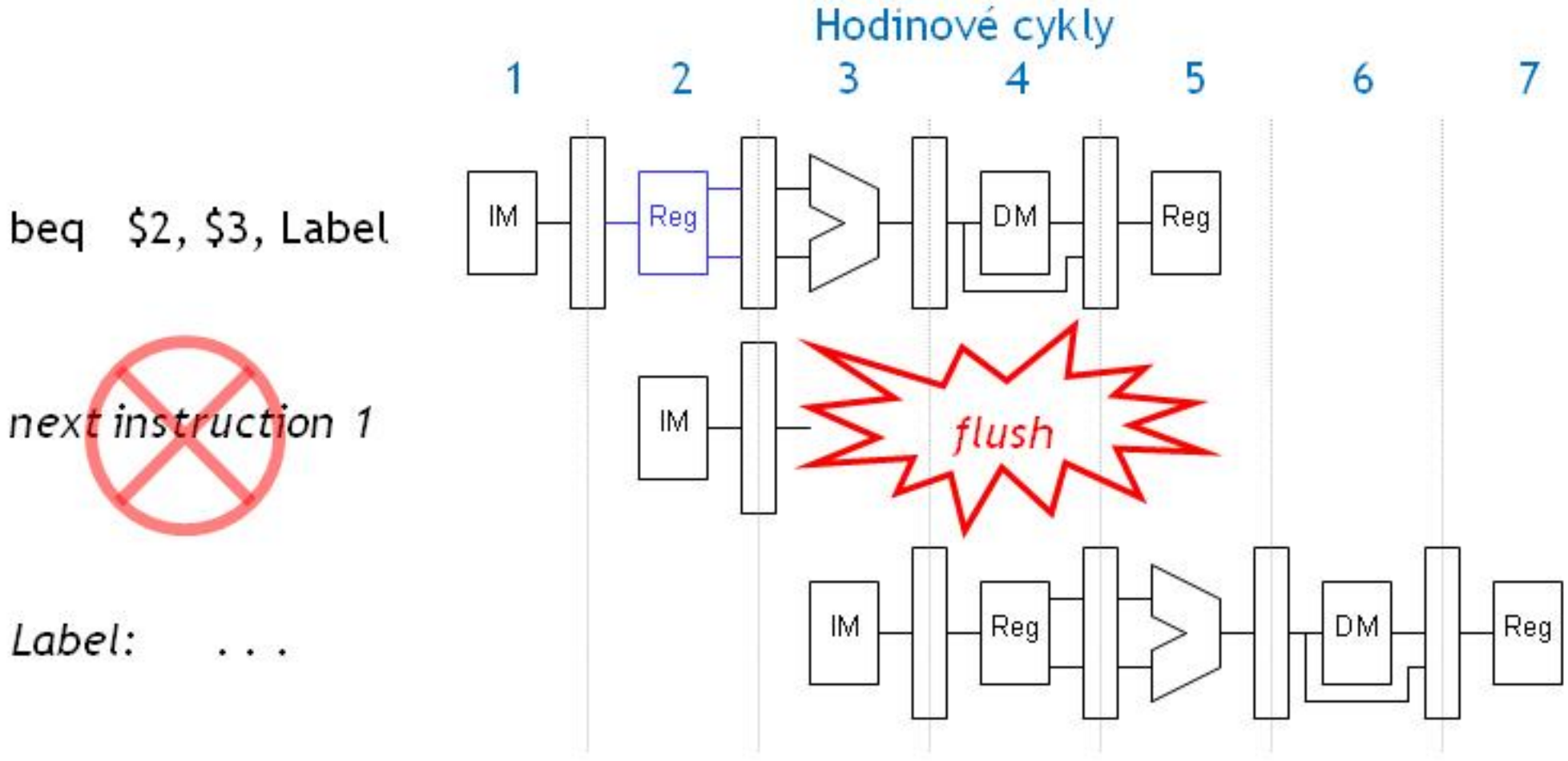
Zisk a ztráta výkonu

- Celkově se predikce skoků vyplácí.
  - Špatná predikce skoku znamená, že se ztratí dva hodinové cykly.
  - Je-li predikce správná, pak ztráta nevznikne. Toto řešení je lepší než pozastavení (stall), při kterém se ztrácí dva cykly při každé instrukci podmíněného skoku.
- Všechny moderní CPU používají predikci skoků.
  - Přesná predikce je důležitá pro optimální výkon.
  - Většina CPU predikuje skoky dynamicky – za běhu se vytváří statistika daného skoku a podle toho se rozhoduje.
- Struktura pipeline má také velký vliv na predikci skoků.
  - Dlouhá pipeline vyžaduje „zahození" většího počtu instrukcí při nezdařené predikci. To vede na ztrátu výkonu.
  - Také je třeba zajistit, aby „zahazované" instrukce mezitím nezpůsobily modifikaci registrů nebo paměti.

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)

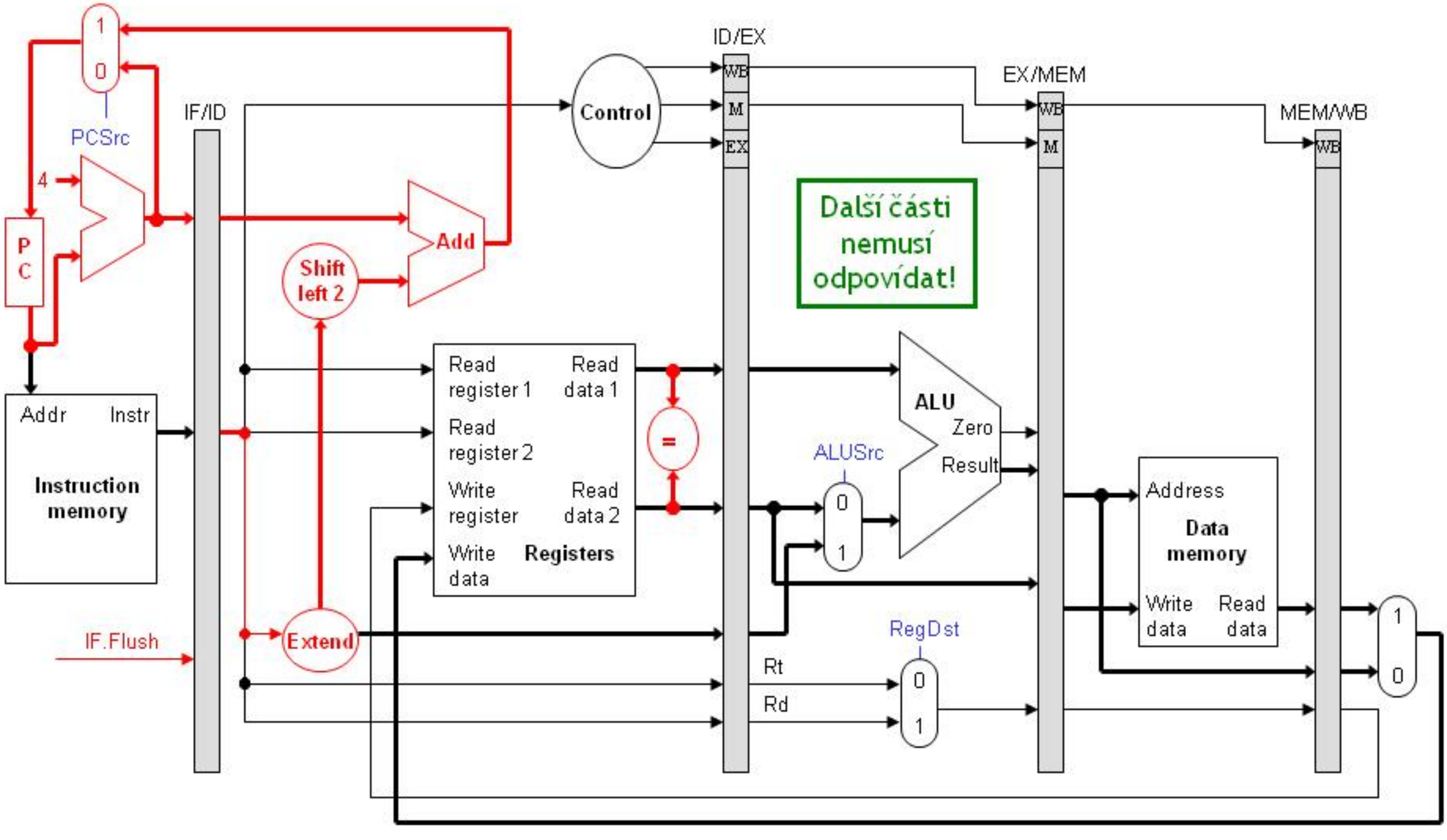
- Musíme „odhodit“ jednu instrukci (v jejím stupni IF), jestliže předchozí instrukce je BEQ a její zdrojové registry jsou shodné.

- „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.

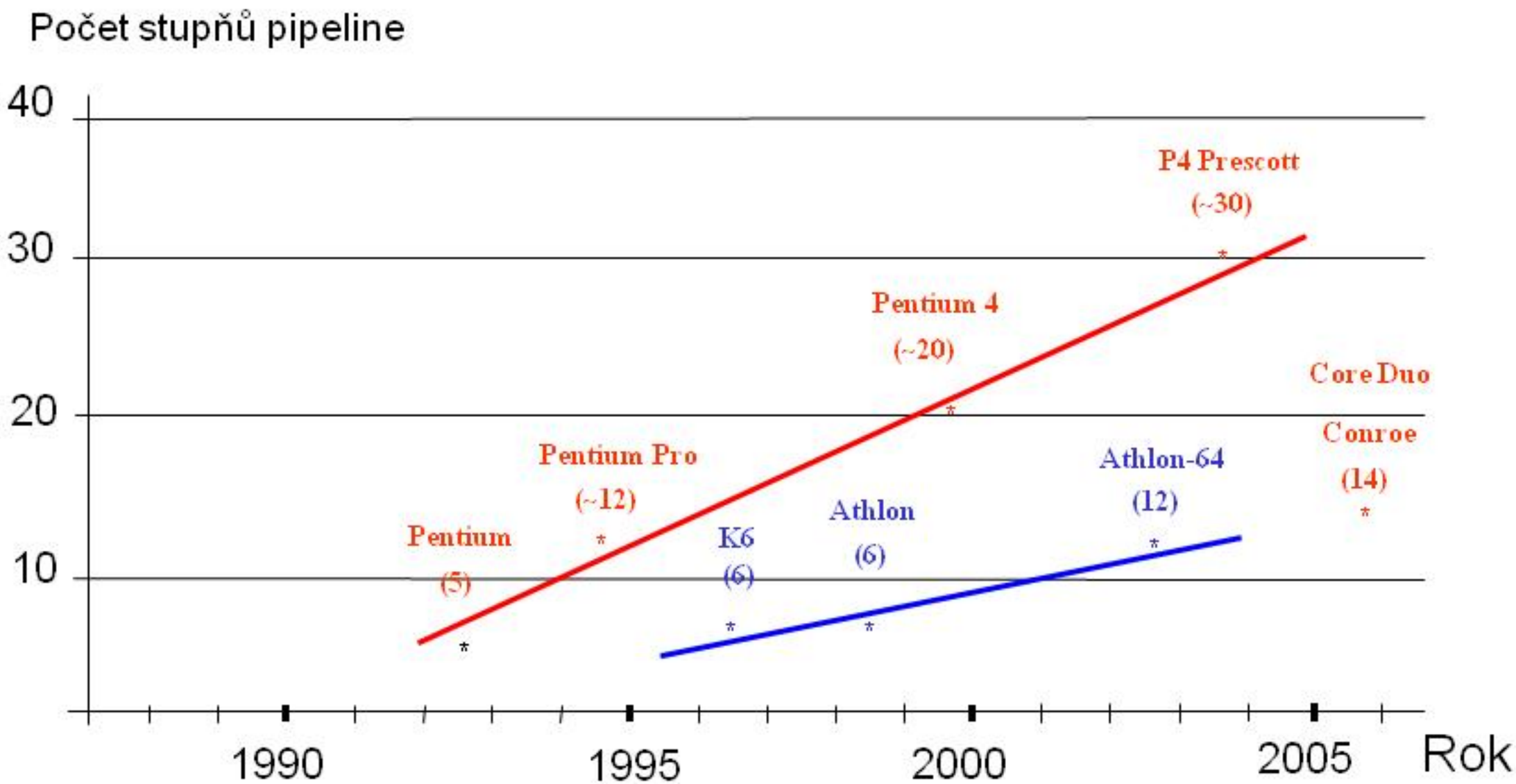
- Tím je zařazena bublina do pipeline, která reprezentuje zpoždění jednoho cyklu při skokové instrukci.

- Řídící signál IF.Flush, uvedený na dalším snímku implementuje tuto myšlenku, ale diagram neobsahuje žádné další podrobnosti.

Větvení bez „forwardingu“ a „load stalls“



Časový vývoj hloubky pipeline



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, pamětí) 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.

Úvod do organizace počítače

Hierarchie paměťového systému

Fyzická a virtuální paměť, cache

Přehled pamětí

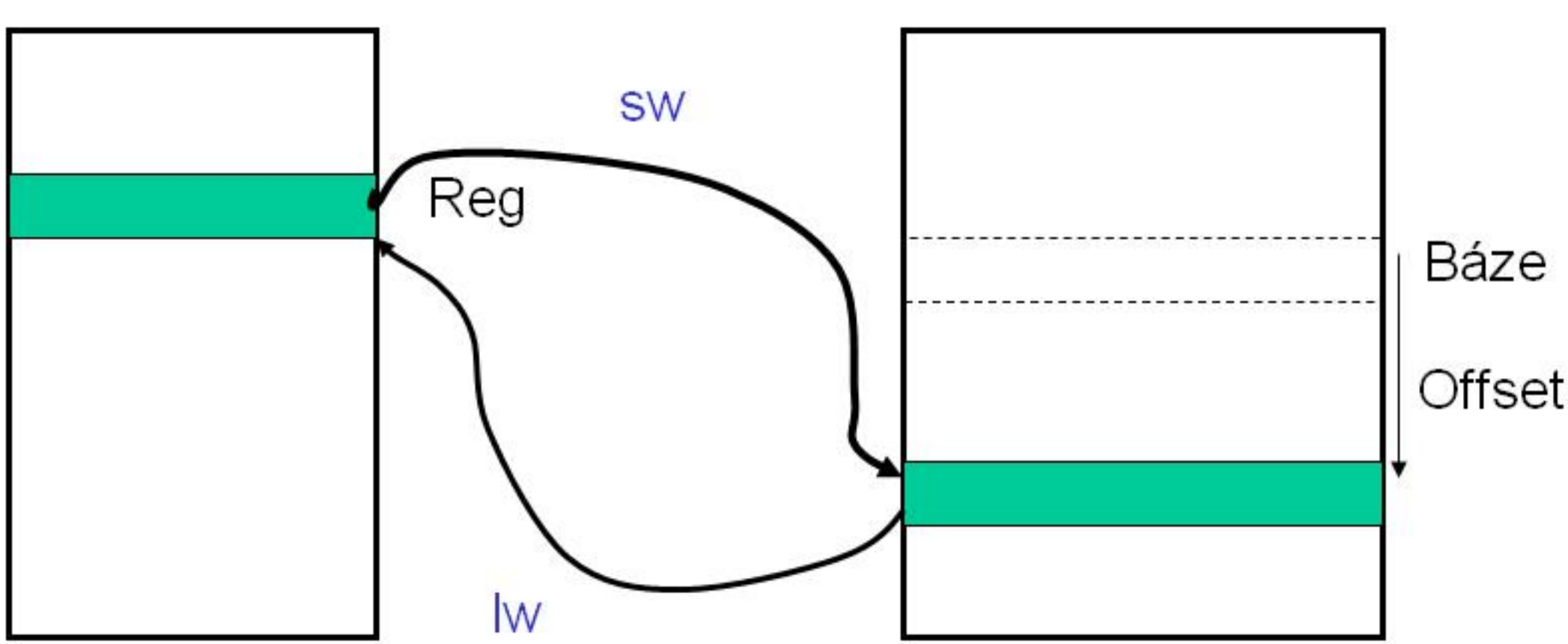
- Uložení dat a instrukcí
- Model - lineární pole 32-bitových slov
- MIPS: 32-bitová adresa => 2^{32} slov
- Doba odezvy je pro každé slovo stejná
- Paměť lze číst/zapsat v 1 cyklu ☺
- Typy pamětí
  - Fyzická: *Instalována v počítači*
  - Virtuální: Disková paměť, která se chová jako hlavní paměť
  -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

Instrukce pro práci s pamětí

- lw reg, offset(base)
- sw reg, offset(base)

Load => přenos paměť do registru

Store => přenos registr do 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

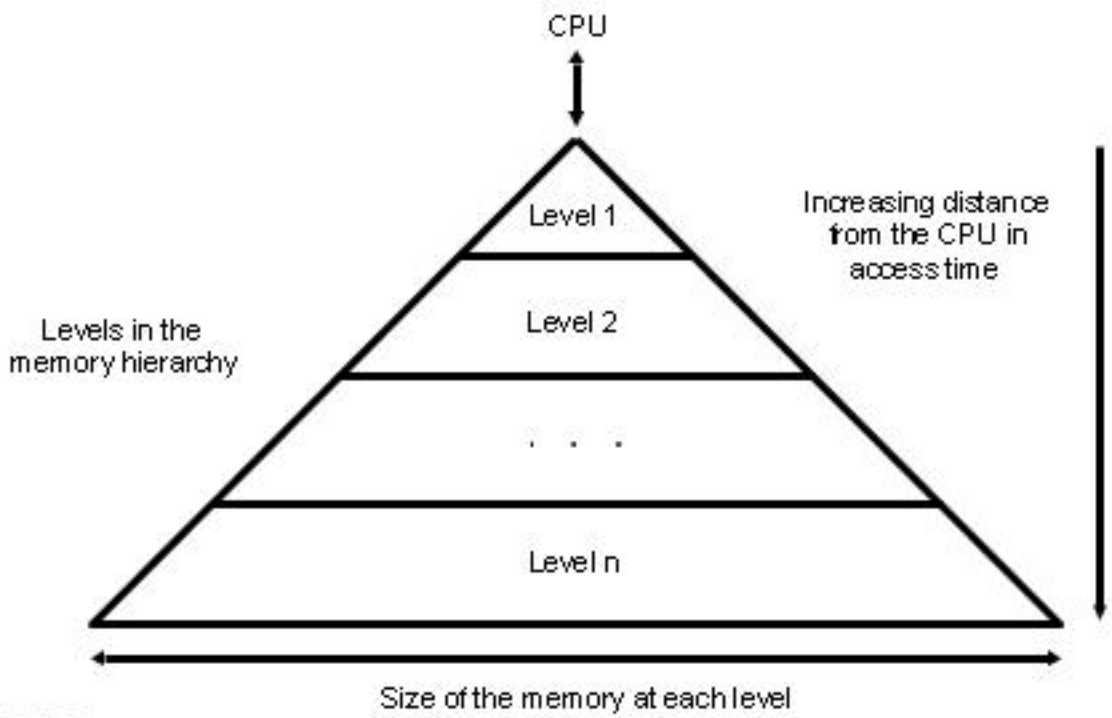
- Tento příklad znázorňuje *princip lokality* – programy přistupují v daném krátkém časovém intervalu k relativně malé části adresního prostoru (stejně jako vy jste přinesli jen malou část knih).
- Existují dva typy lokality …
  - *Časová lokalita* – je-li položka referencována, má tendenci být referencována 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

- 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

- 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

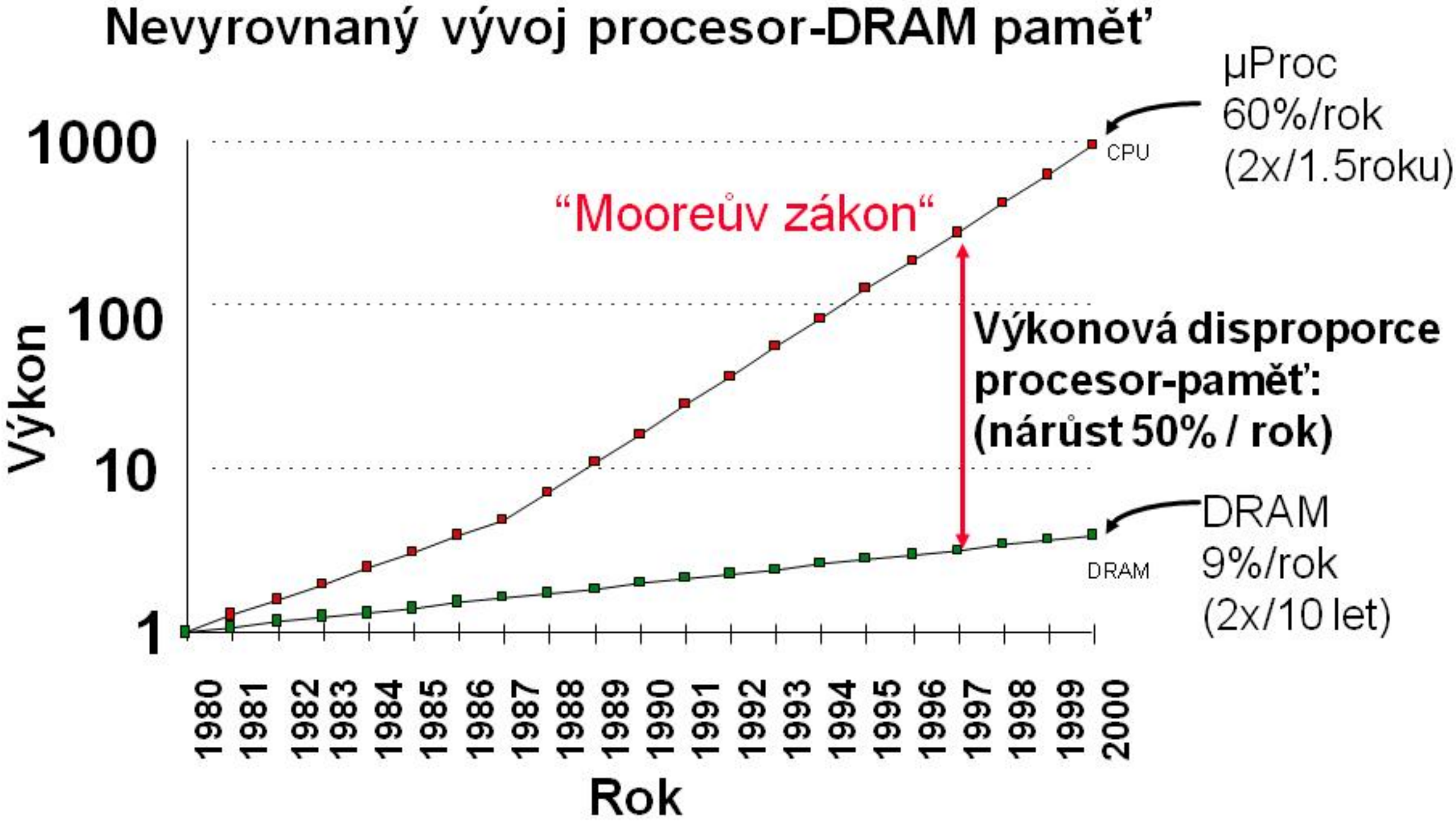
	Kapacita	**Rychlost (latence)**
Logika:	2x za 3 roky	2x za 3 roky
DRAM:	4x za 3 roky	2x za 10 let
Disk:	4x za 3 roky	2x za 10 let

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?

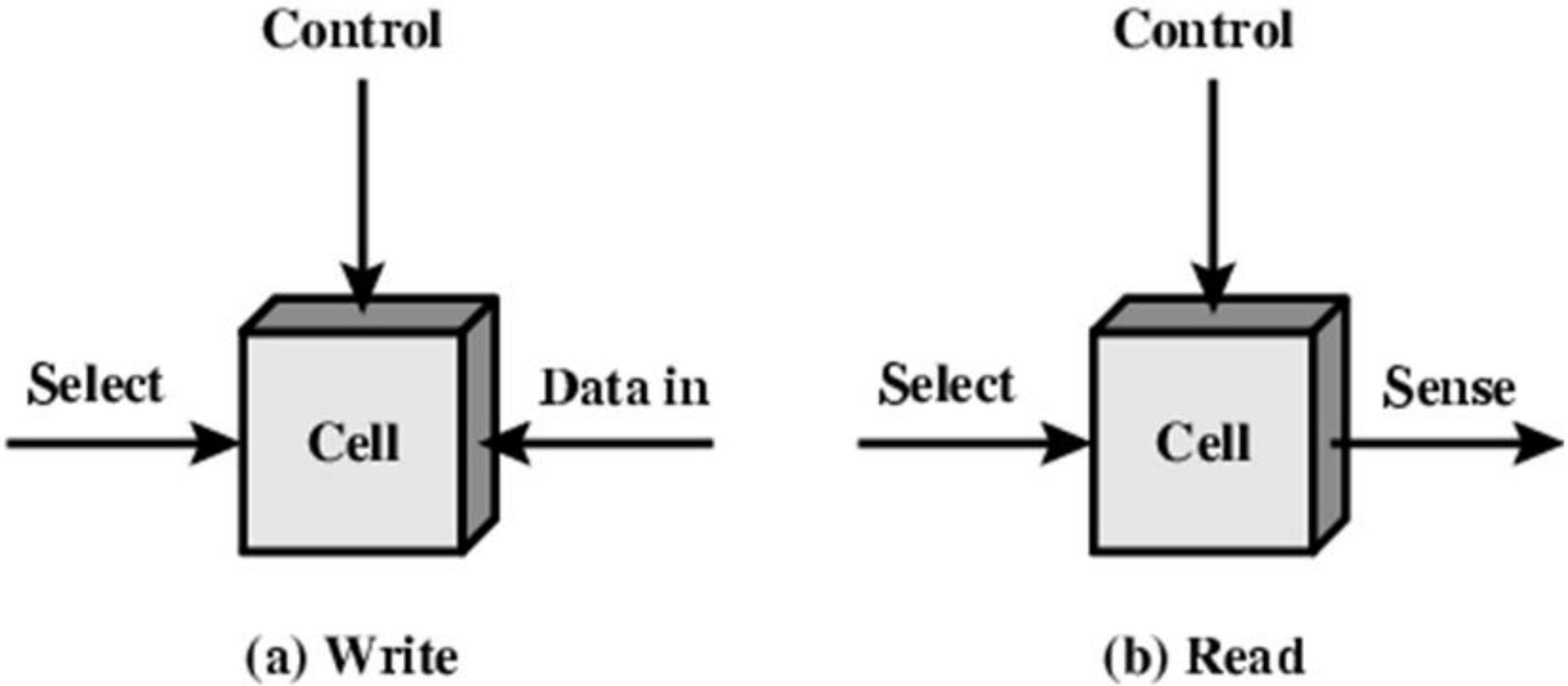


Paměť

- 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 paměti



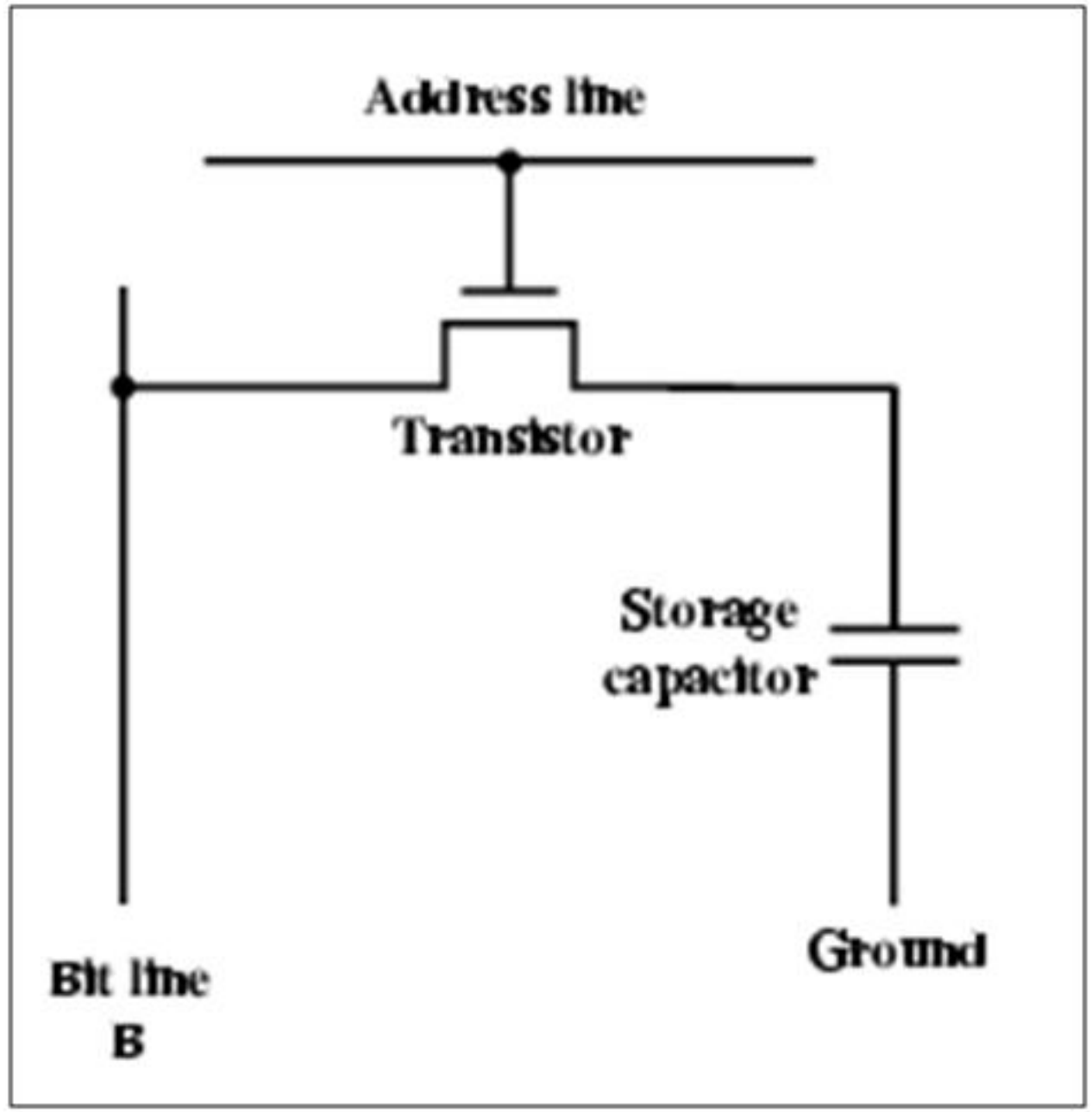
(a) Write

(b) Read

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 dynamické RAM



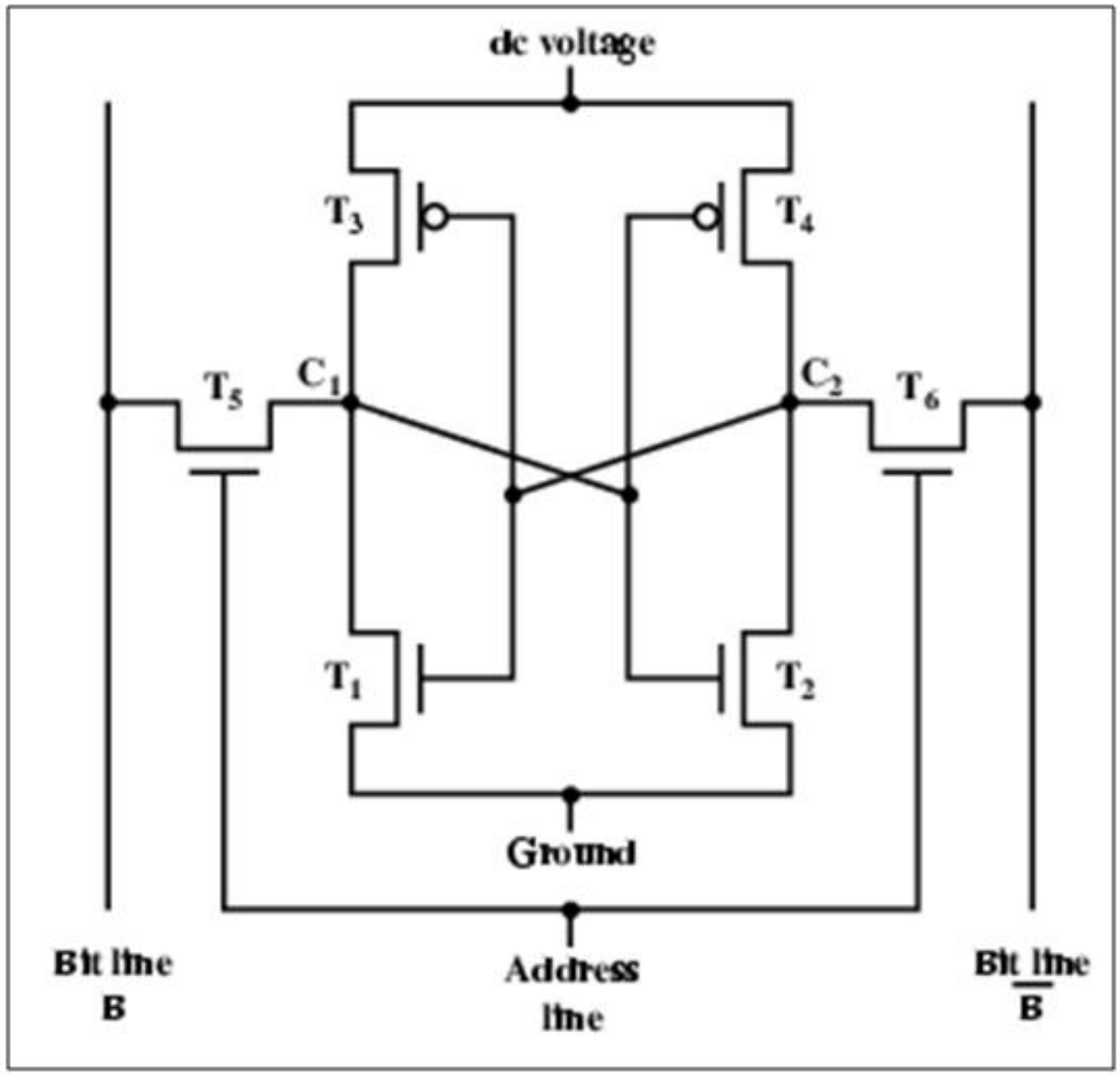
Operace DRAM

- 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

- 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 buňky statické RAM



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 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

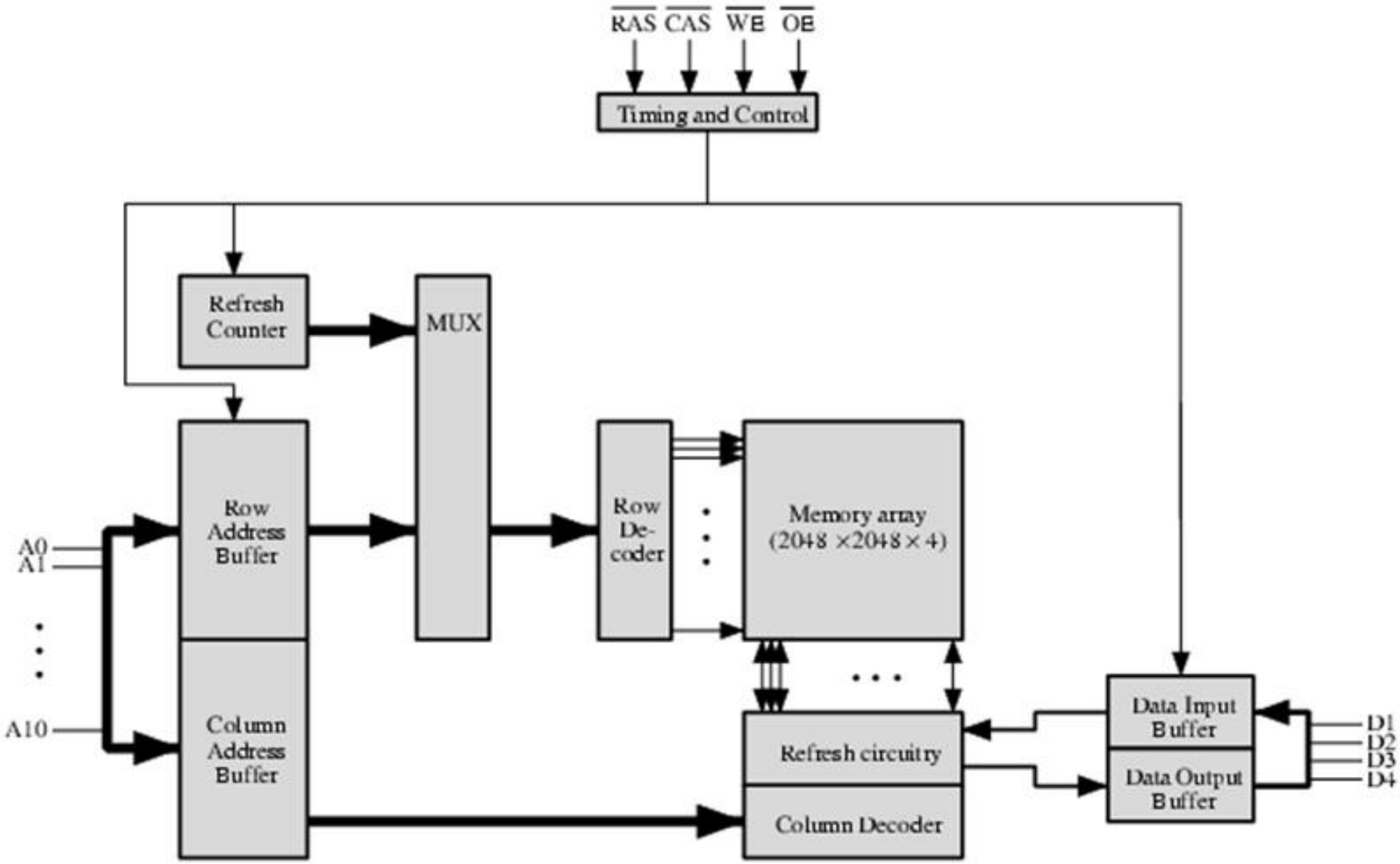
Detaily organizace

- 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

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

Typická 16 Mb DRAM (4M x 4)



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?

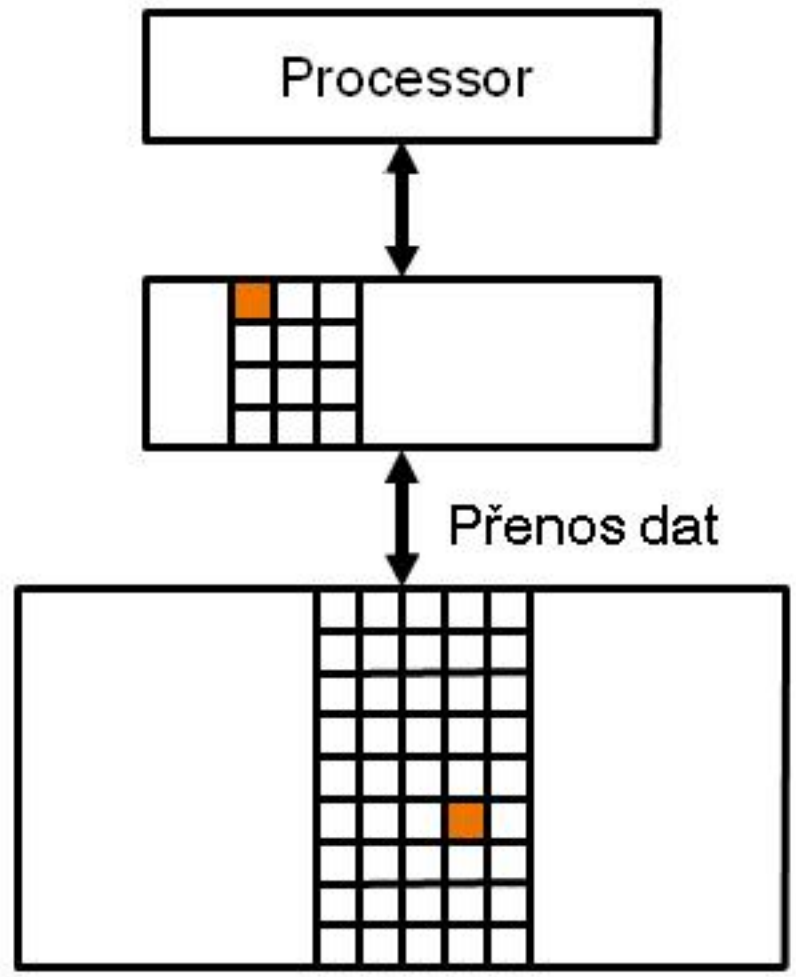
- Vzhledem k rozdílu v ceně a rychlosti je výhodné vybudovat hierarchii úrovní s rychlejšími pamětmi blíže k procesoru a s levnějšími pamětmi na nižších úrovních.
- Cílem tohoto uspořádání je prezentovat uživateli/programátorovi paměťový systém s kapacitou odpovídající levné technologii, ale s rychlostí,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

Organizace hierarchie

- 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

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

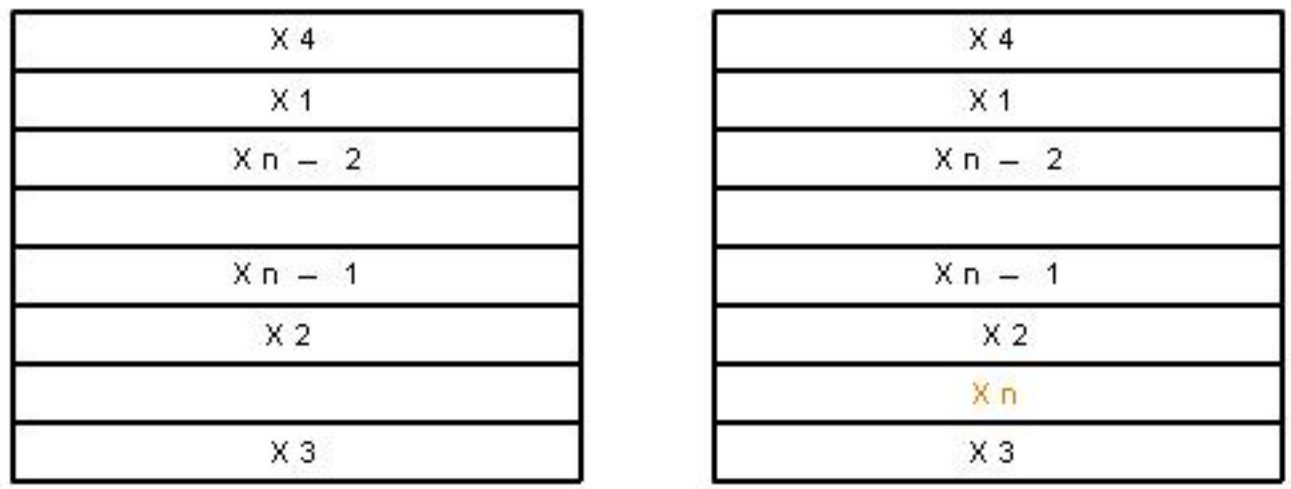


Cache paměť - pojmy

- **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

- 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

b. Po referenci Xn

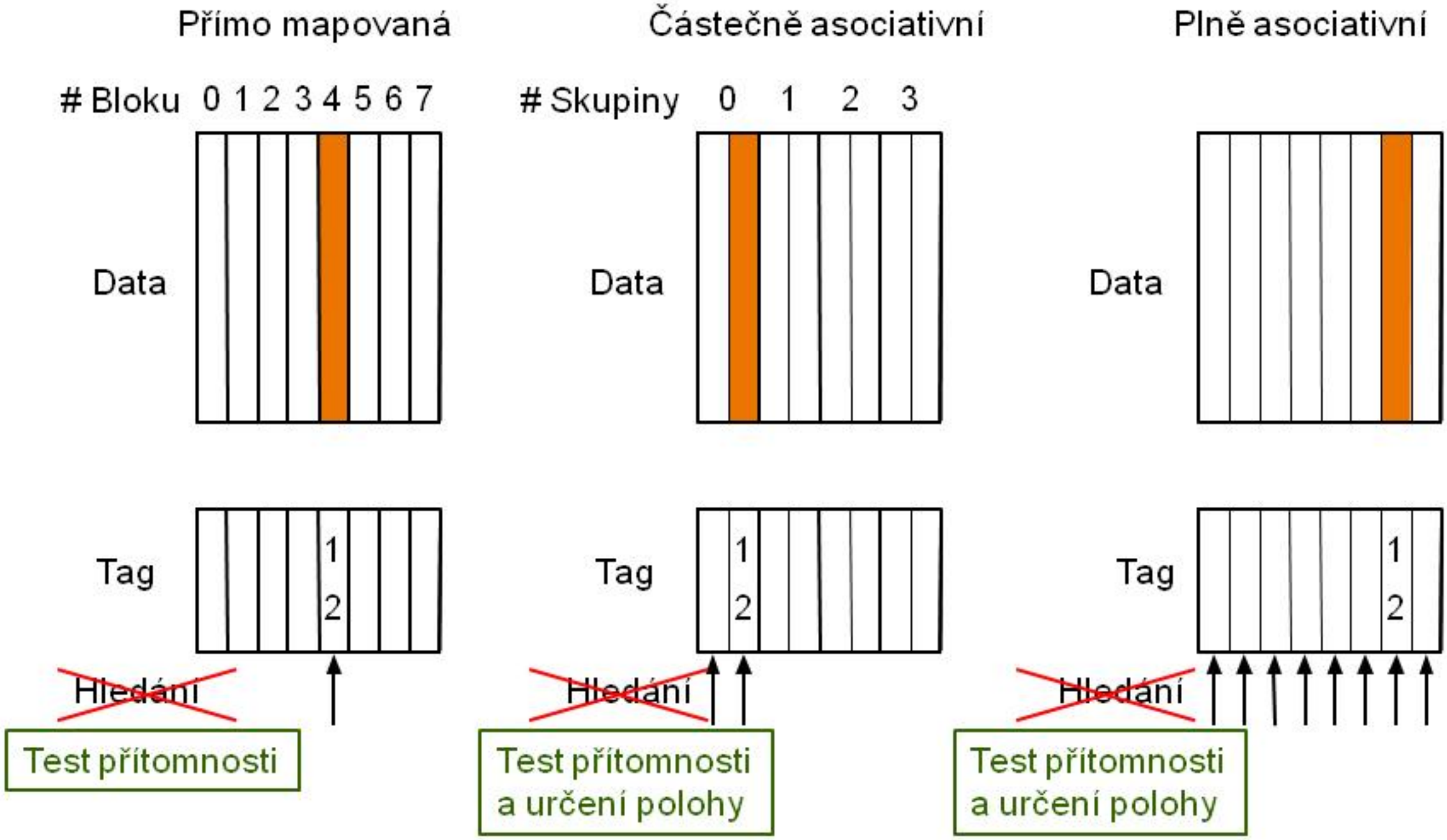
Příklad trochu podrobněji

- 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 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

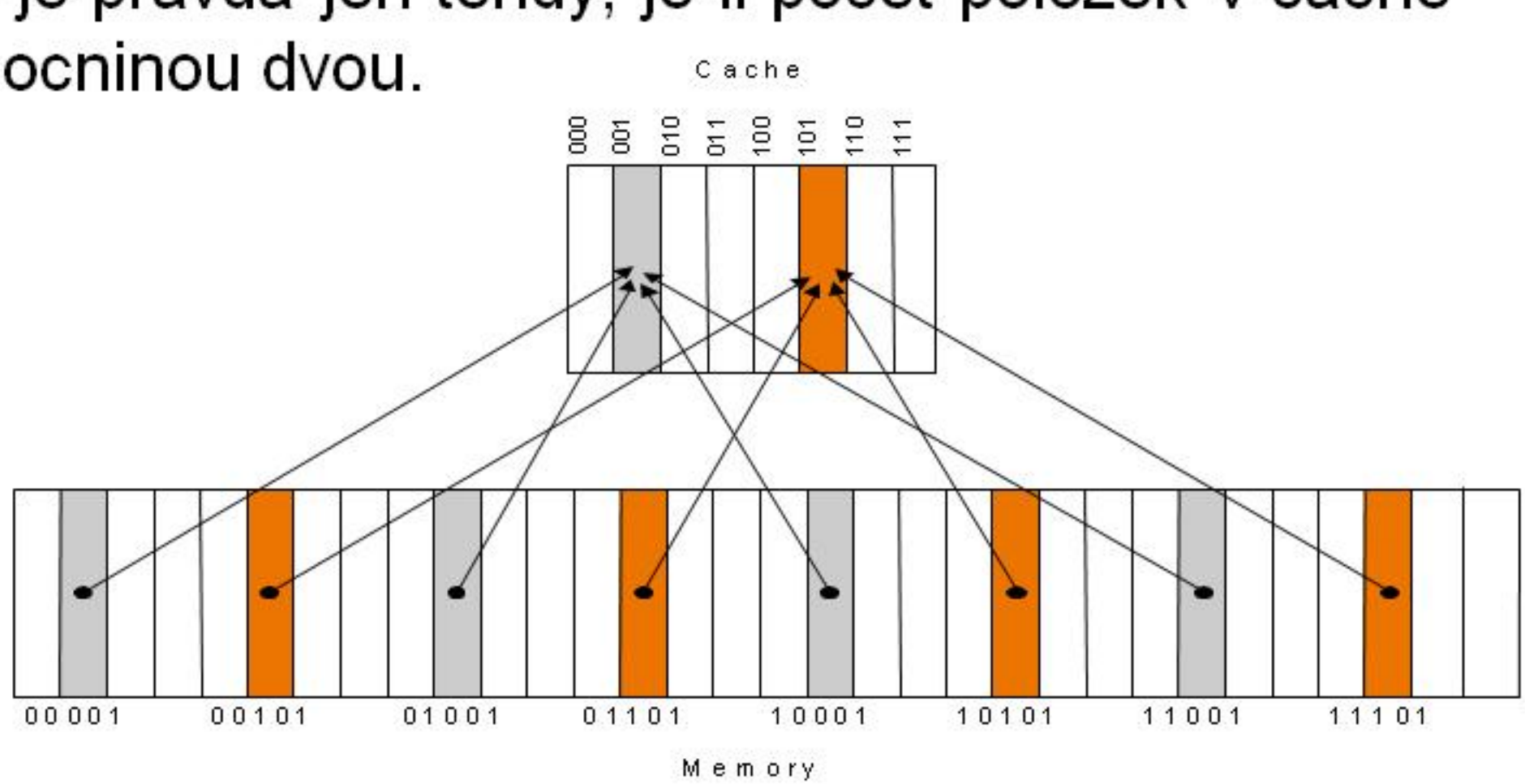
Tři strategie organizace cache



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“

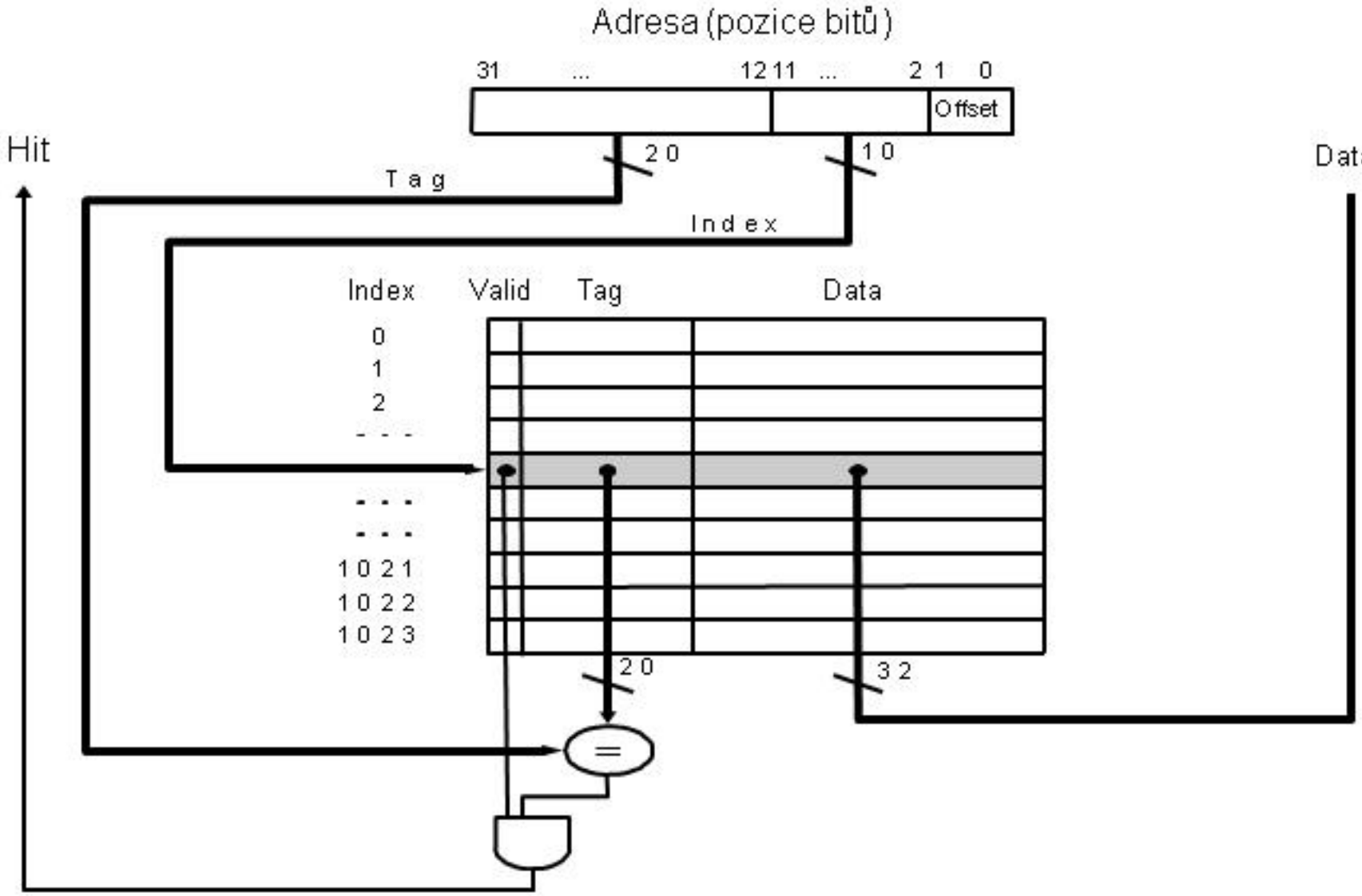
- To znamená, že každý blok hlavní paměti se mapuje na jeden jediný blok cache, ale na jeden vstupní bod cache se mapuje větší množství bloků hlavní paměti.
  - Jak můžeme určit, že právě požadovaný blok je umístěný v cache?
- Zajistíme to doplněním souboru *tagů do* cache. Ke každému bloku v cache jednoduše 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

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



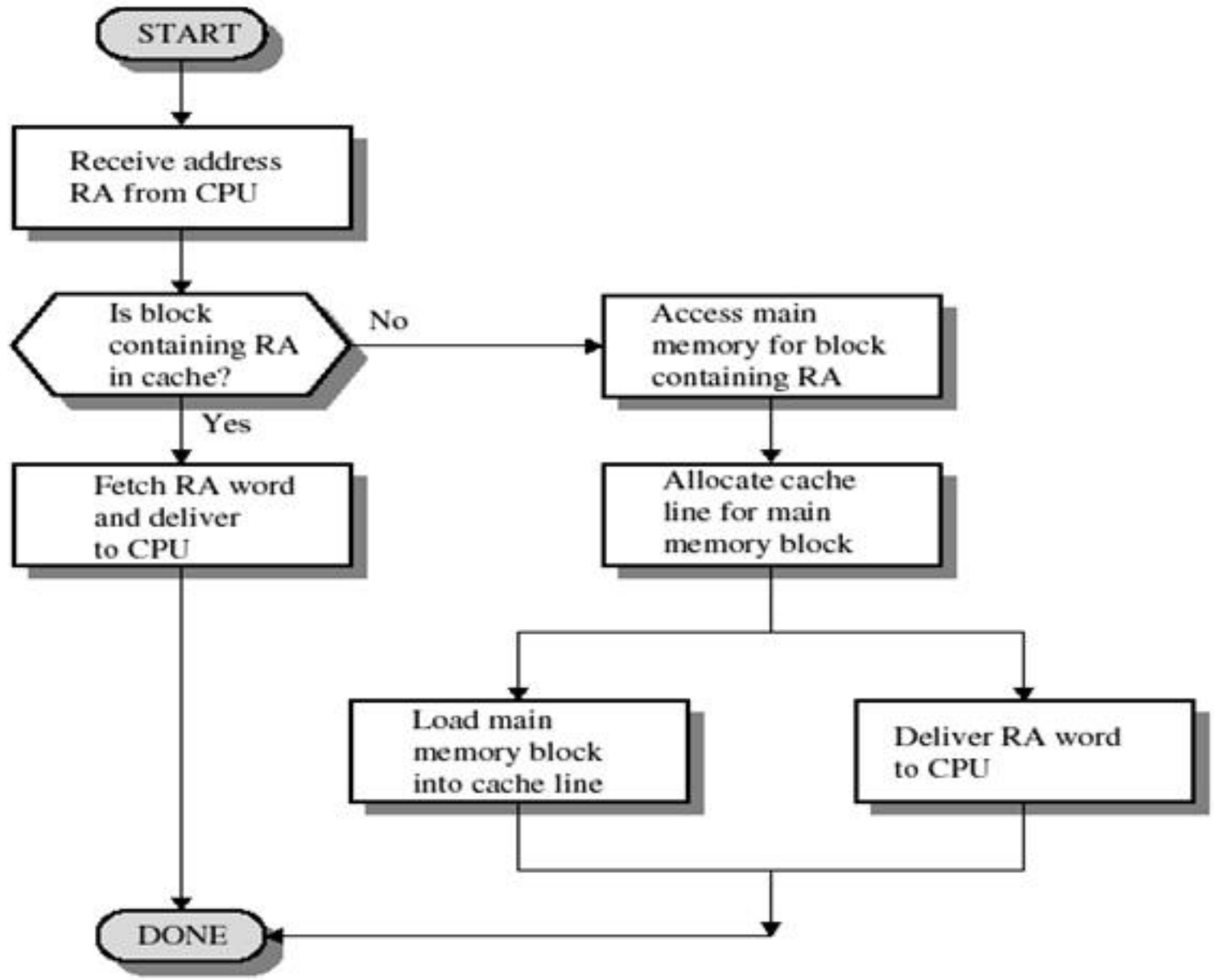
Čtení z cache

- 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).

Cache – operace čtení



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)

- 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).

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

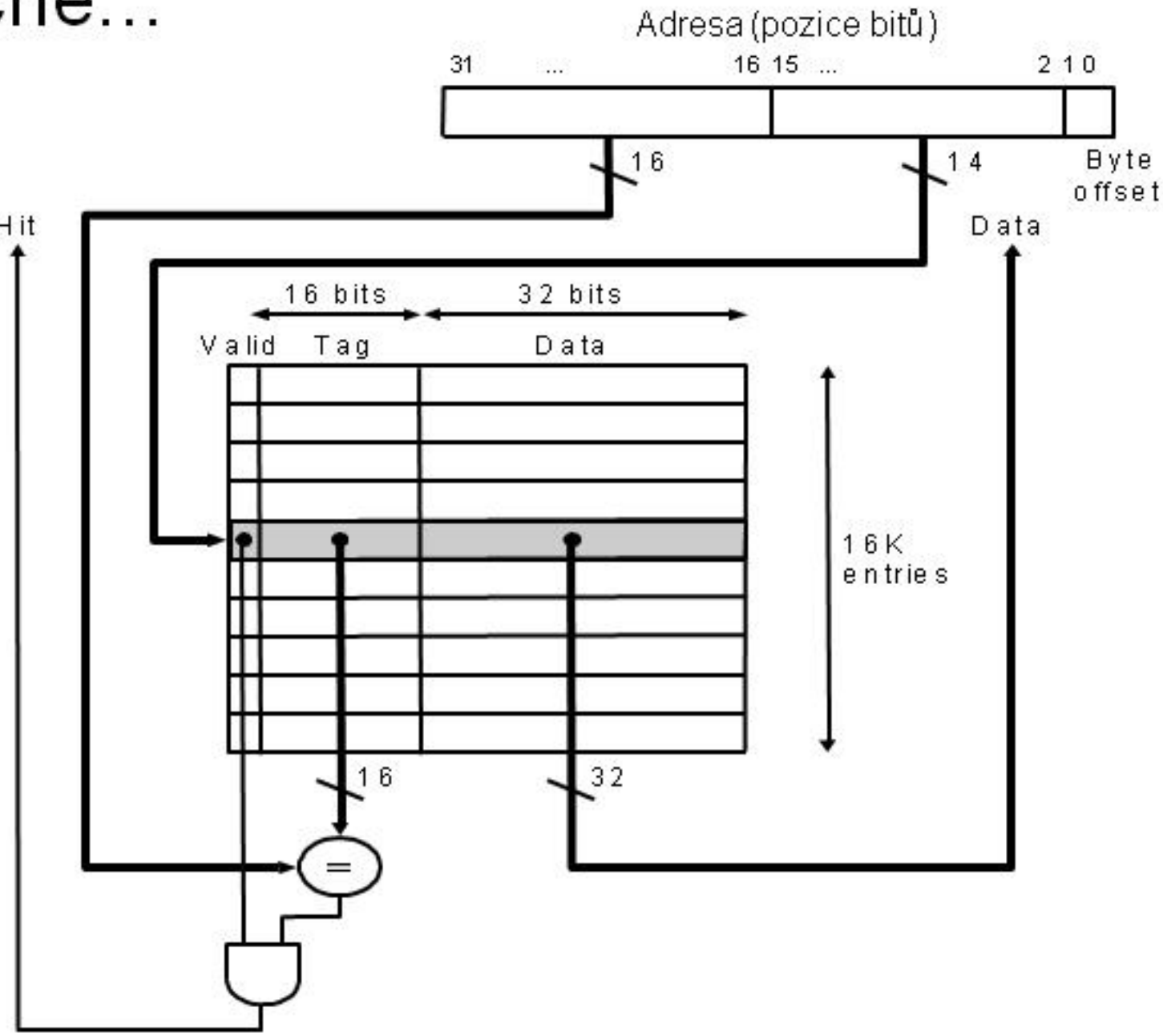
Index	Valid	Tag	Data
000	N		
001	N		
010	Y	10	Memory[10010]
011	N		
100	N		
101	N		
110	Y	10	Memory[10110]
111	N		

Reálné cache

- 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

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



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

- 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 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

- 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?

- 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.

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

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:

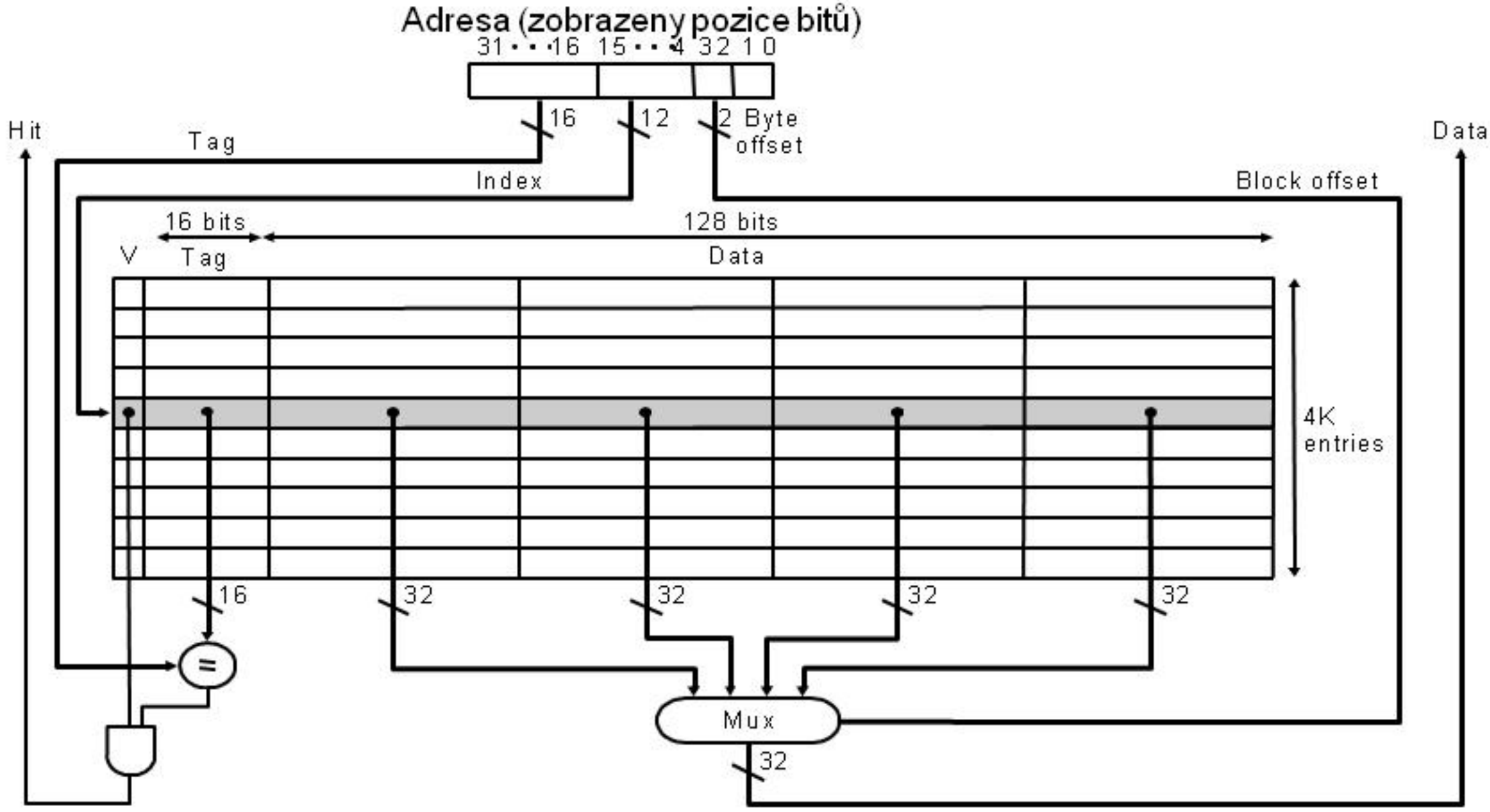
- Zápis dat *pouze* do bloku cache. Zápis do paměti pouze při výměně bloků
- **Výhoda :** Zápisy omezeny pouze rychlostí zápisu cache
- **Výhoda :** Podporovány zápisy více slov najednou, efektivní je pouze zápis celého bloku do hlavní paměti najednou

Prostorová lokalita

- 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

- Struktura cache, která využívá prostorové lokality…



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.

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).

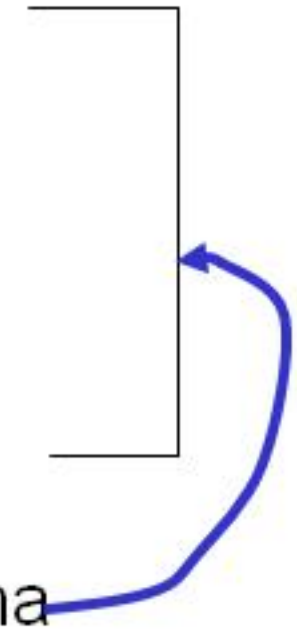
Cena za více slov v bloku

- 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.

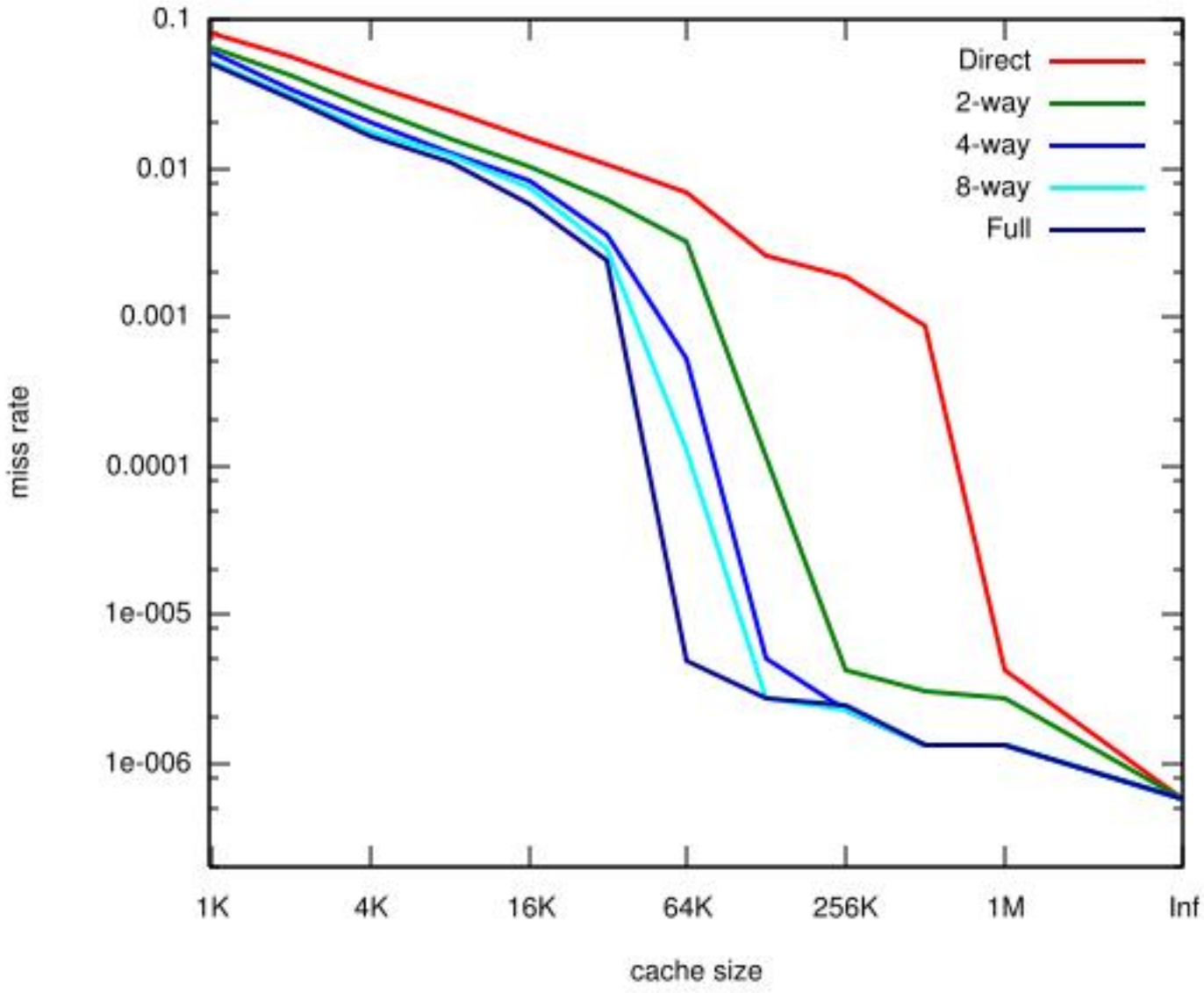
Případ – data v cache nenalezena

***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“ závisí na organizaci



Typy výpadků:

- compulsory miss

  (např. počáteční plnění)

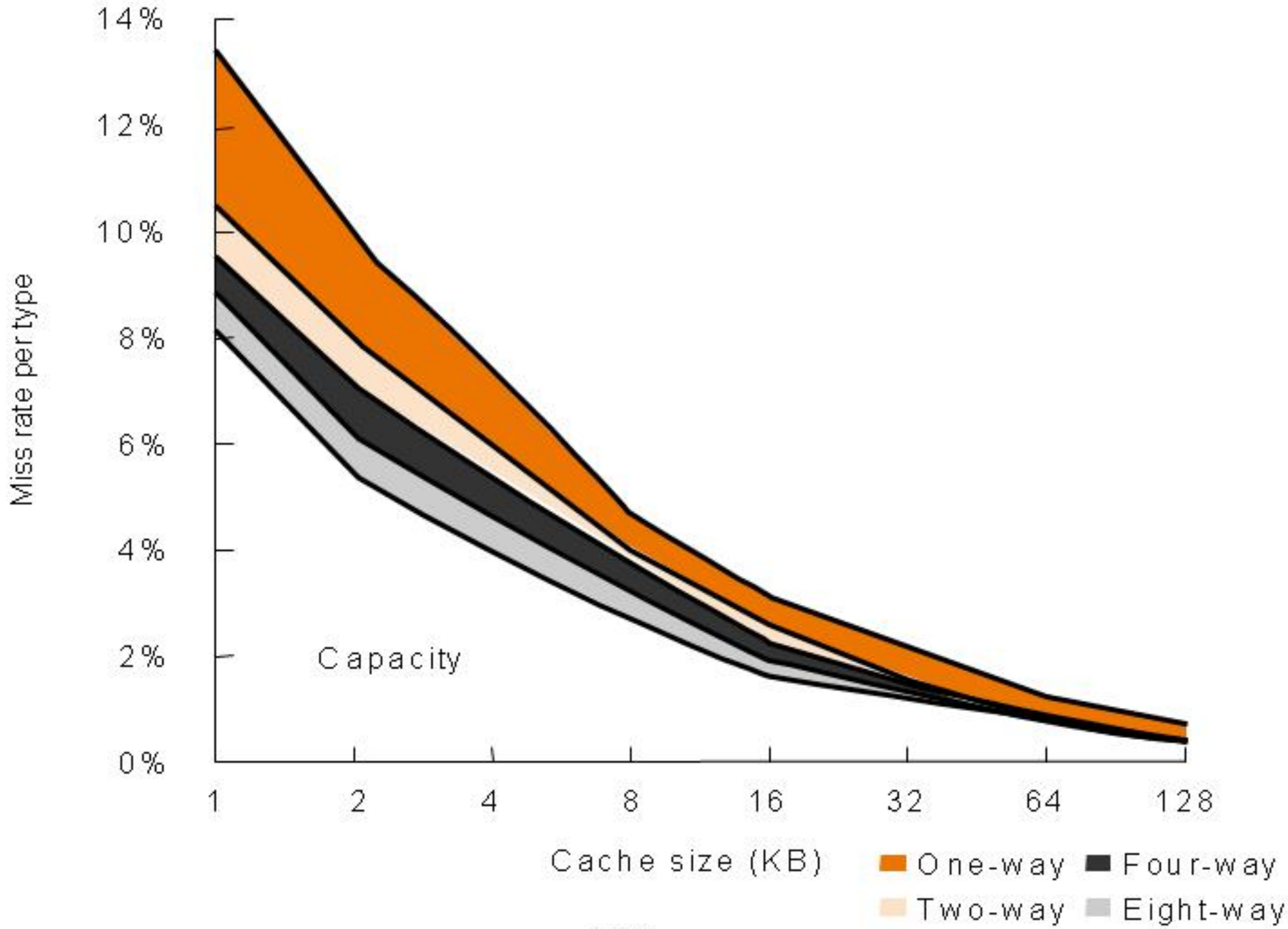
- capacity miss

  (z důvodu malé kapacity cache)

- conflict miss

  (příčinou je konflikt)

Četnost výpadků a jejich příčiny



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 (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 (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.

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 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

- 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 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

- 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

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

- 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

- 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…

$$\frac{\text{Výkon(rychlé\_hodiny)}}{\text{Výkon(pomalé\_hodiny)}} = \frac{I \times CPI_{fast} \times ClockCycle}{I \times CPI_{slow} \times \frac{ClockCycle}{2}} = \frac{3.36}{4.75 \times \frac{1}{2}} = 1.41$$

- Počítač s dvojnásobnou hodinovou frekvencí je 1.41 krát rychlejší. Kdyby neexistovaly výpadky cache, byla by jeho rychlost dvojnásobná!

Cache s flexibilnějším umísťováním bloků

- 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

- 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

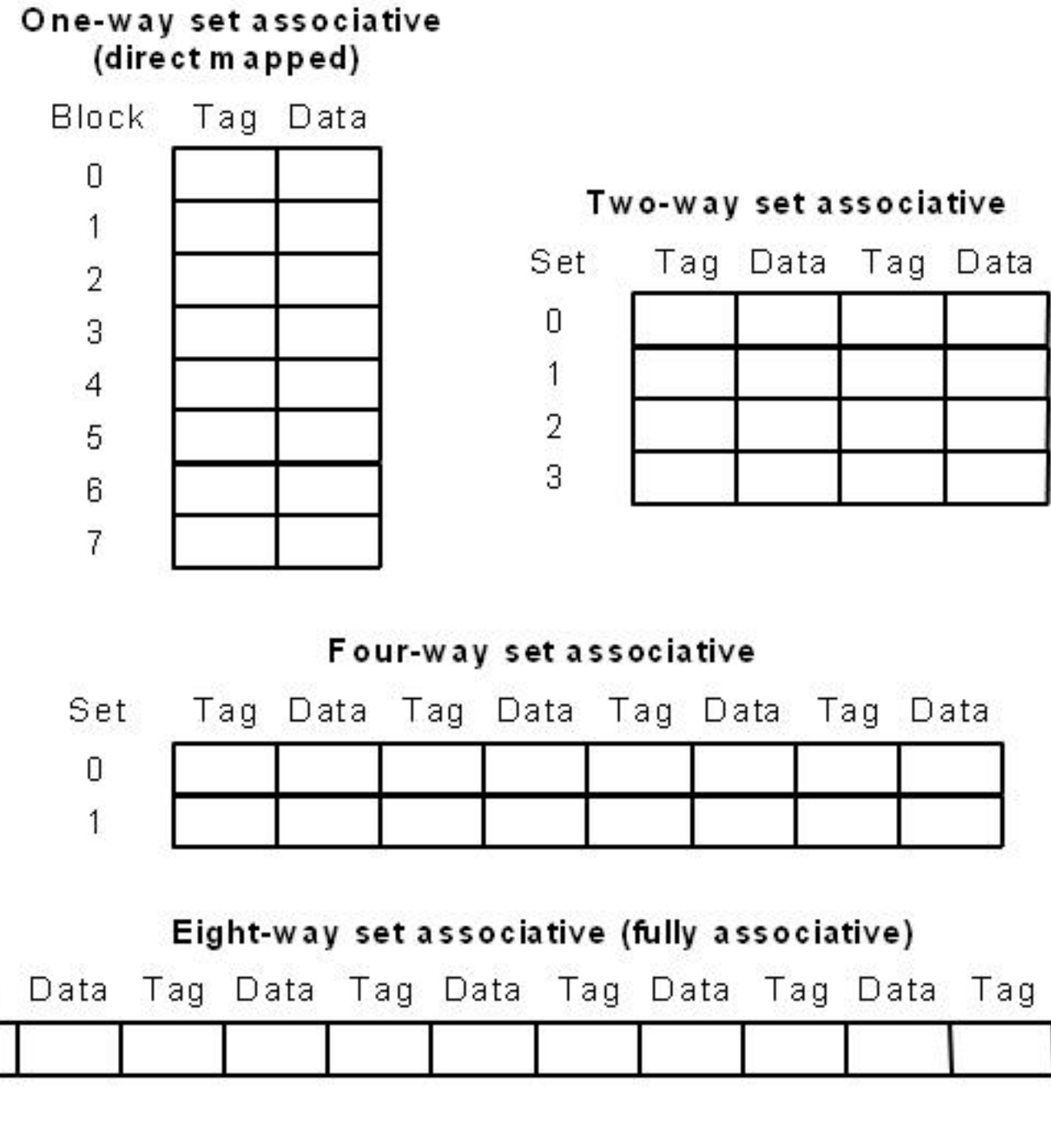
- Praktičtější řešení představují *částečně asociativní cache*, které představují kompromisní řešení mezi přímo mapovanou a plně asociativní organizací.
- U tohoto typu cache existuje pevný počet míst (2 nebo více) kde může být daný blok umístěn. Existuje-li pro každý blok n možných pozic, nazývá se 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.

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 cache

- Cache s 8-bloky můžeme organizovat…



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

- 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…

- 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

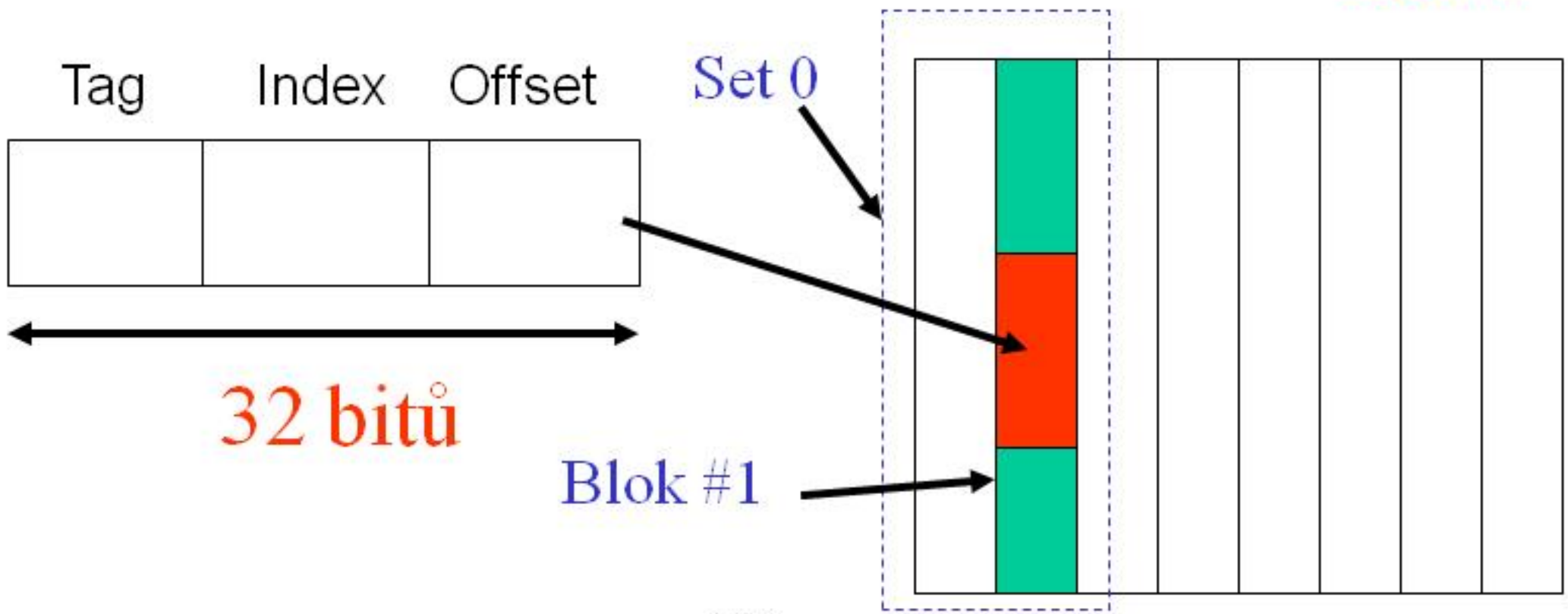
- **Zobecňuje všechna mapovací schémata cache**
  - Předpokládejme, že cache obsahuje *N* bloků
  - 1- cestná částečně asociativní cache: Přímé mapování
  - *M*-cestná částečně asociativní cache: Je-li M = N, pak se jedná o plně asociativní cache
- **Výhoda**
  - Zvyšuje úspěšnost – klesá „miss rate“ (více míst, kde lze nalézt B)
- **Nevýhoda**
  - Narůstá „hit time“ (více míst, kde je třeba hledat B)
  - Složitější hardware

Činnost částečně asociativní cache

Složky adresy cache:

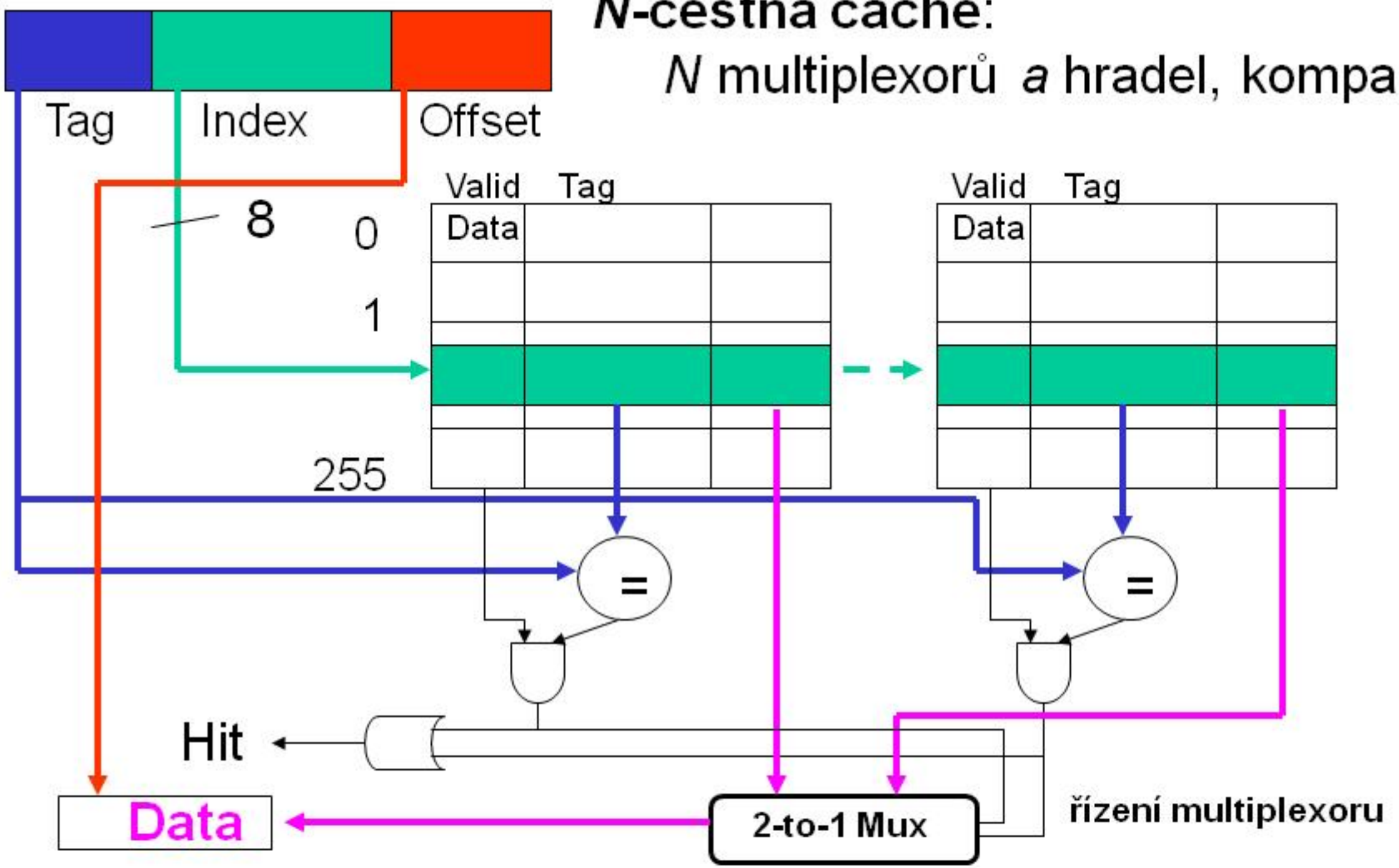
- *Index i*: Vybírá množinu S_i
- *Tag*: Použit pro výběr hledaného bloku, porovnáním *n* bloků ve vybrané množině S
- *Blok Offset*: Adresa cílové položky dat uvnitř bloku

Dvoucestná cache



Příklad: Hardware *2-cestné* cache

Adresa pro cache

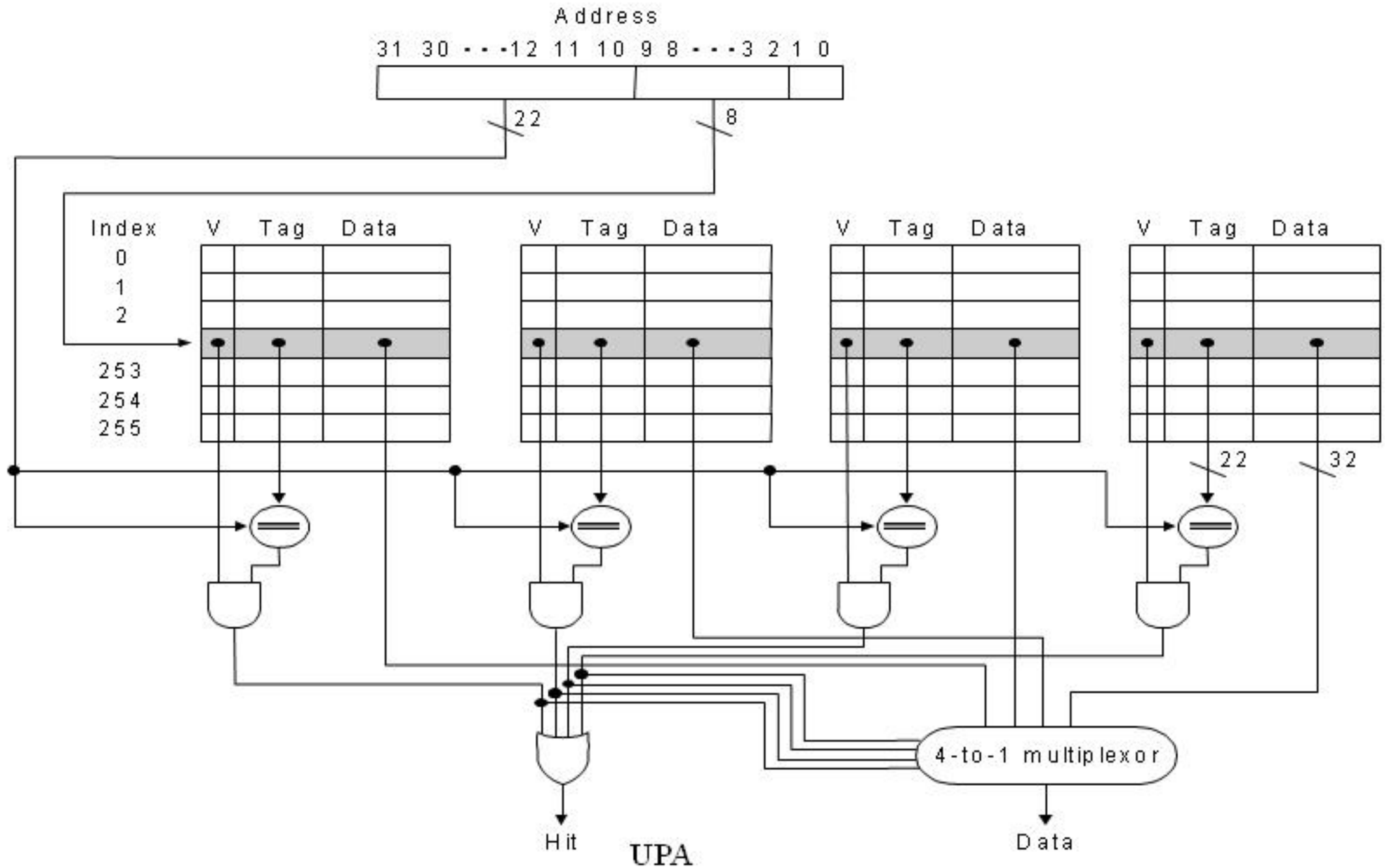


***N*-cestná cache**:

N multiplexorů *a* hradel, komparátory

Nalezení bloku u 4-cestné cache

- Příklad (4-cestná částečně asociativní cache)...



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

- 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

- 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

- 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

- 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

- 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.

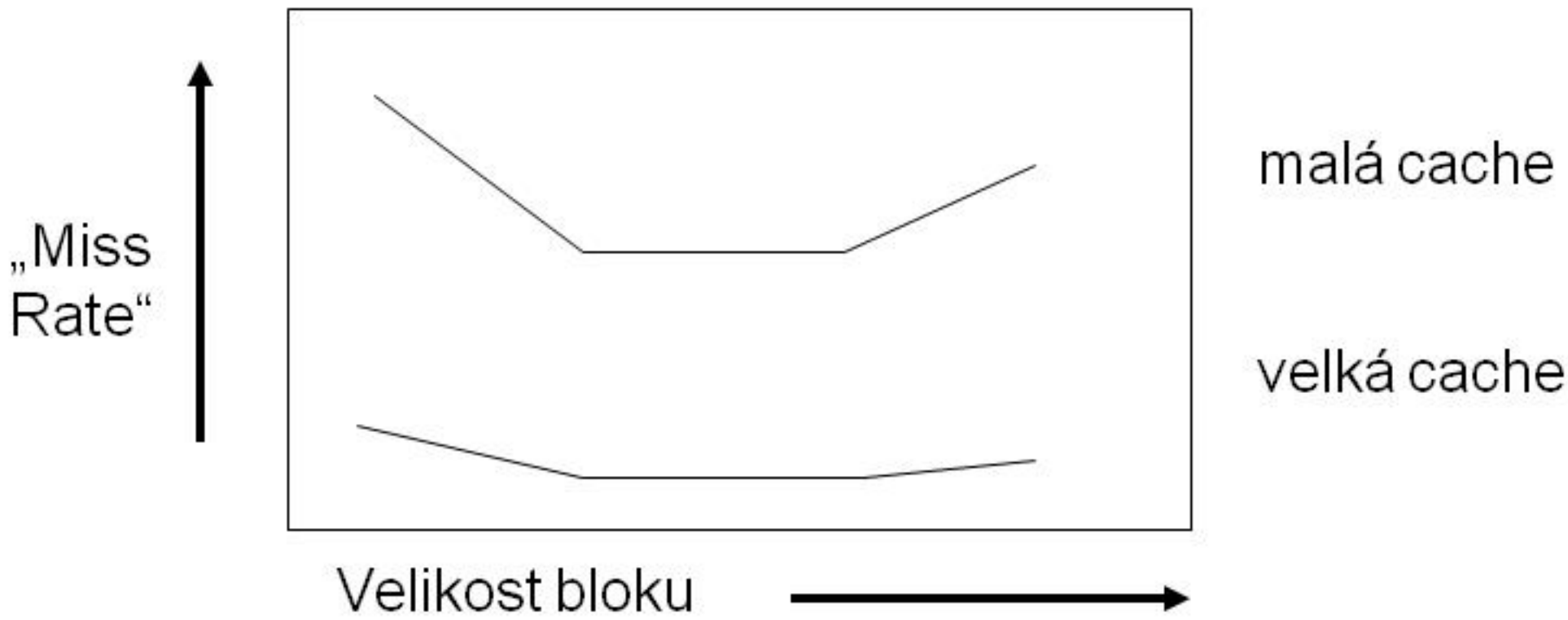
Uvažujeme-li „Hit Time“

- 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“

***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**



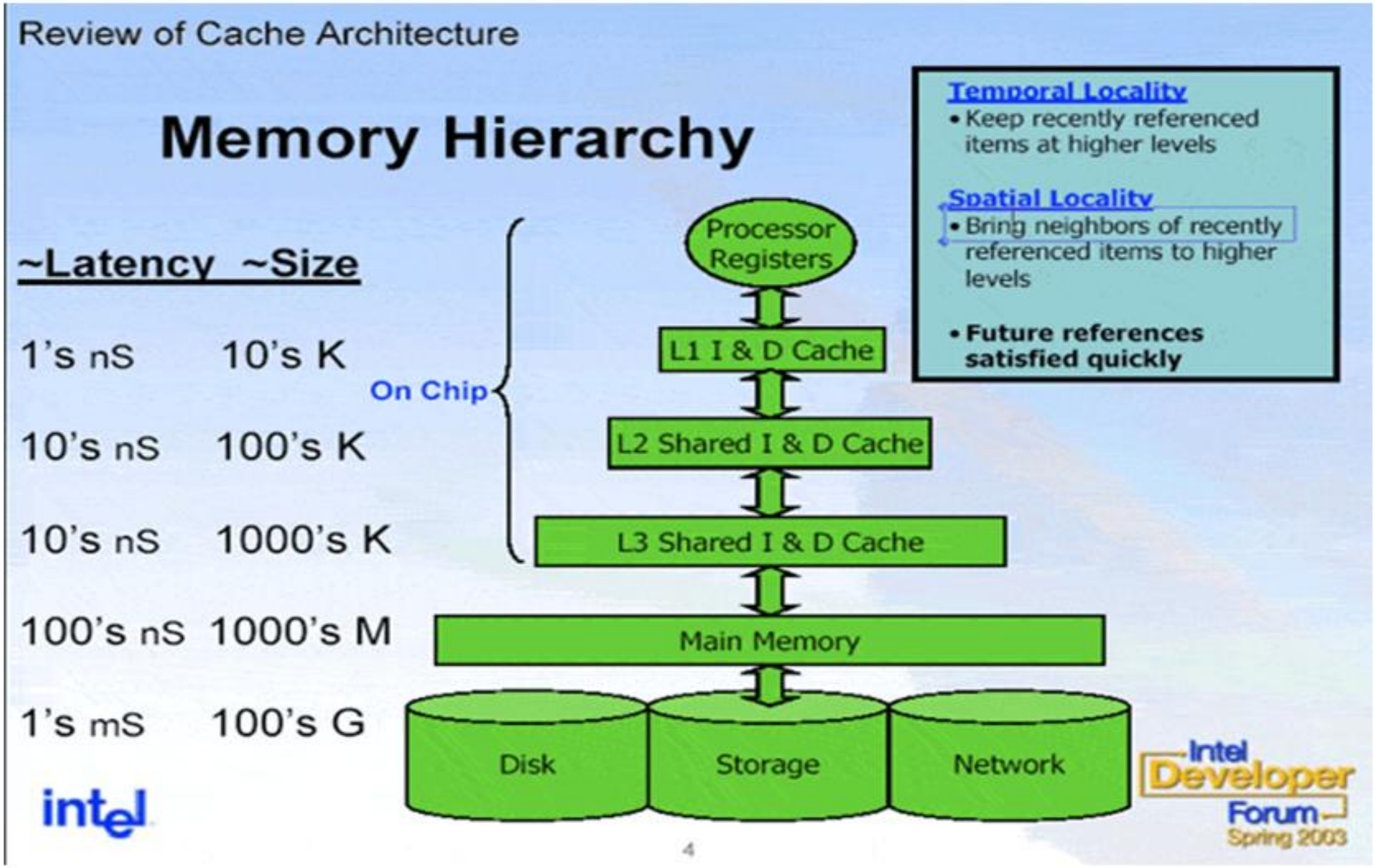
Návrh víceúrovňové cache

- 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

- 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

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

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 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

- 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

- 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.

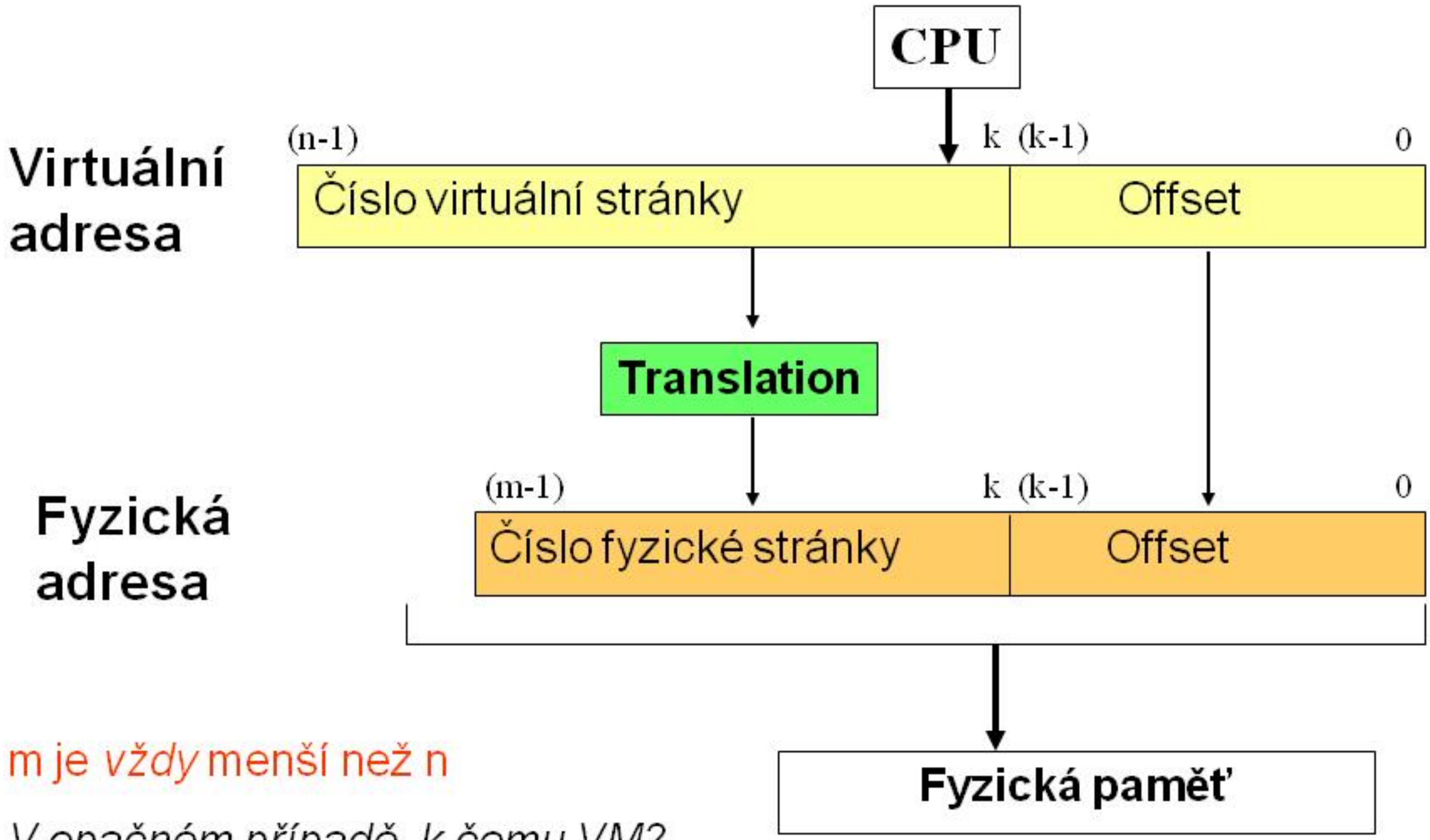
Další motivace

- 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 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

- 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“ přítomných v paměti musel být menší než velikost fyzické paměti.
- To byl podstatný nedostatek a překážka pro každého programátora.

Činnost virtuální paměti



m je *vždy* menší než n

V opačném případě, k čemu VM?

„Cena“ virtuální paměti

Nedostupnost stránky se nazývá výpadek (miss).

- **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

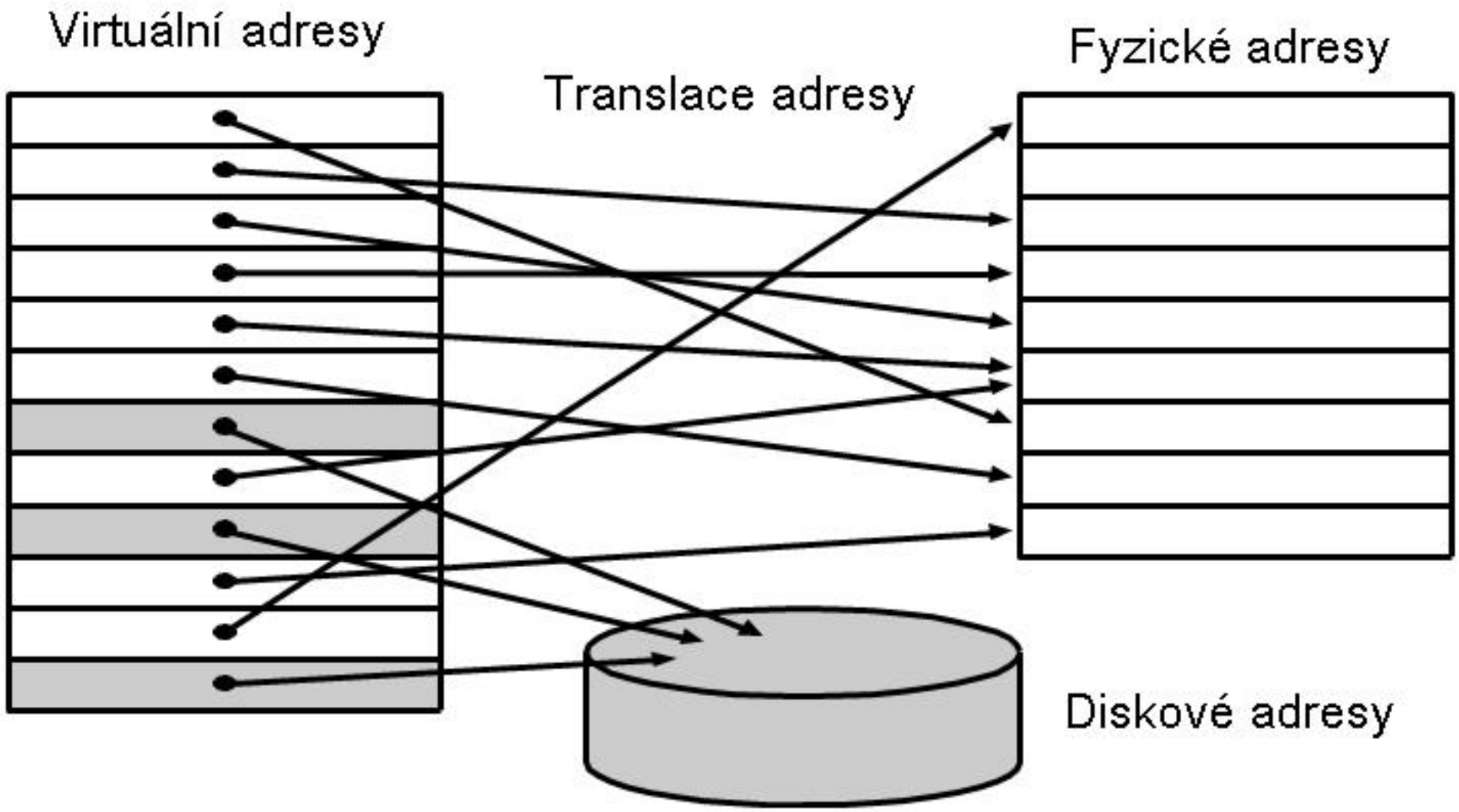
Terminologie

- 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

- 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

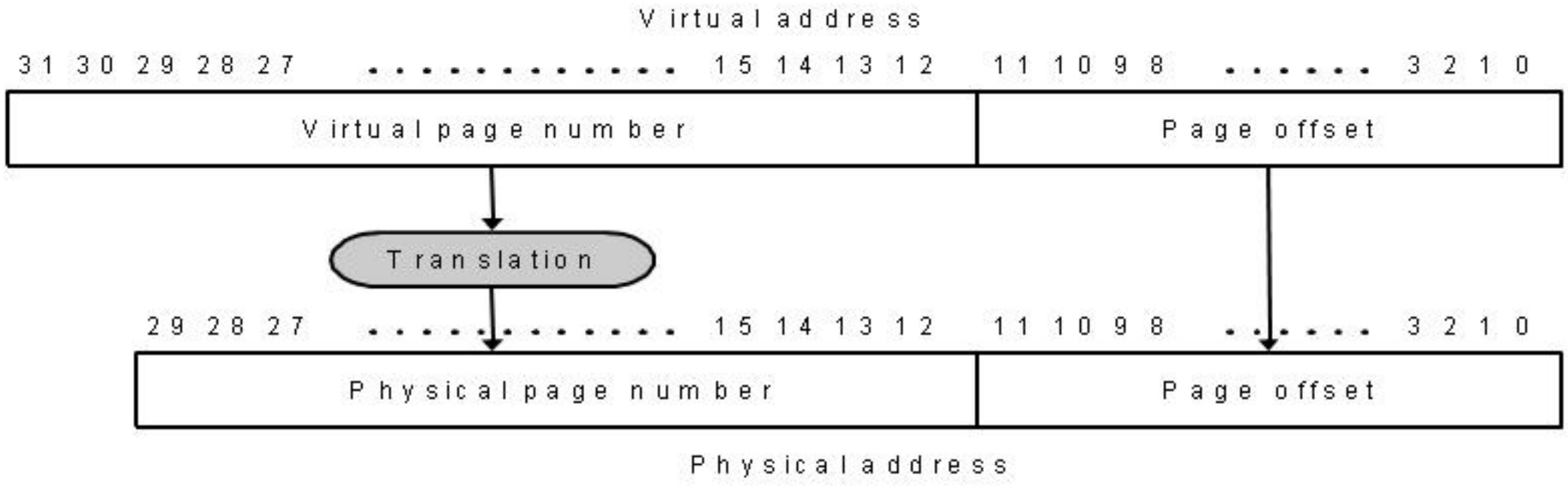


- Tento program obsazuje spojitý prostor ve virtuálním adresním prostoru, nikoliv však ve fyzickém.

Translace adresy

- 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

- 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-through pro VM nelze použít, protože zápis trvá příliš dlouho. Systémy VM používají techniku Write-back.

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...

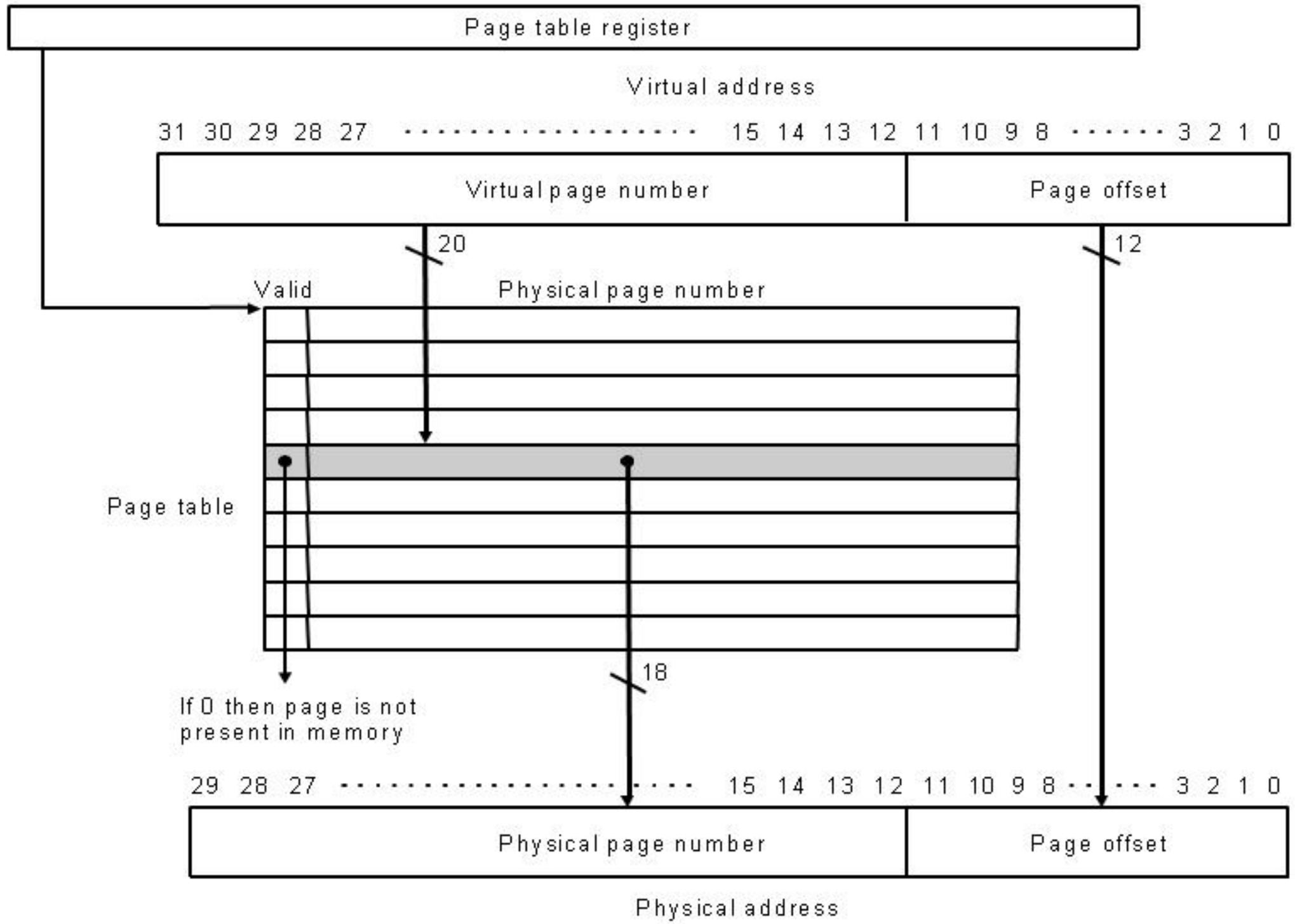
Umístění stránky

- Jak se umístí stránka do fyzické paměti?
- Umístění stránky je možné do libovolného místa, 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

- 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



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

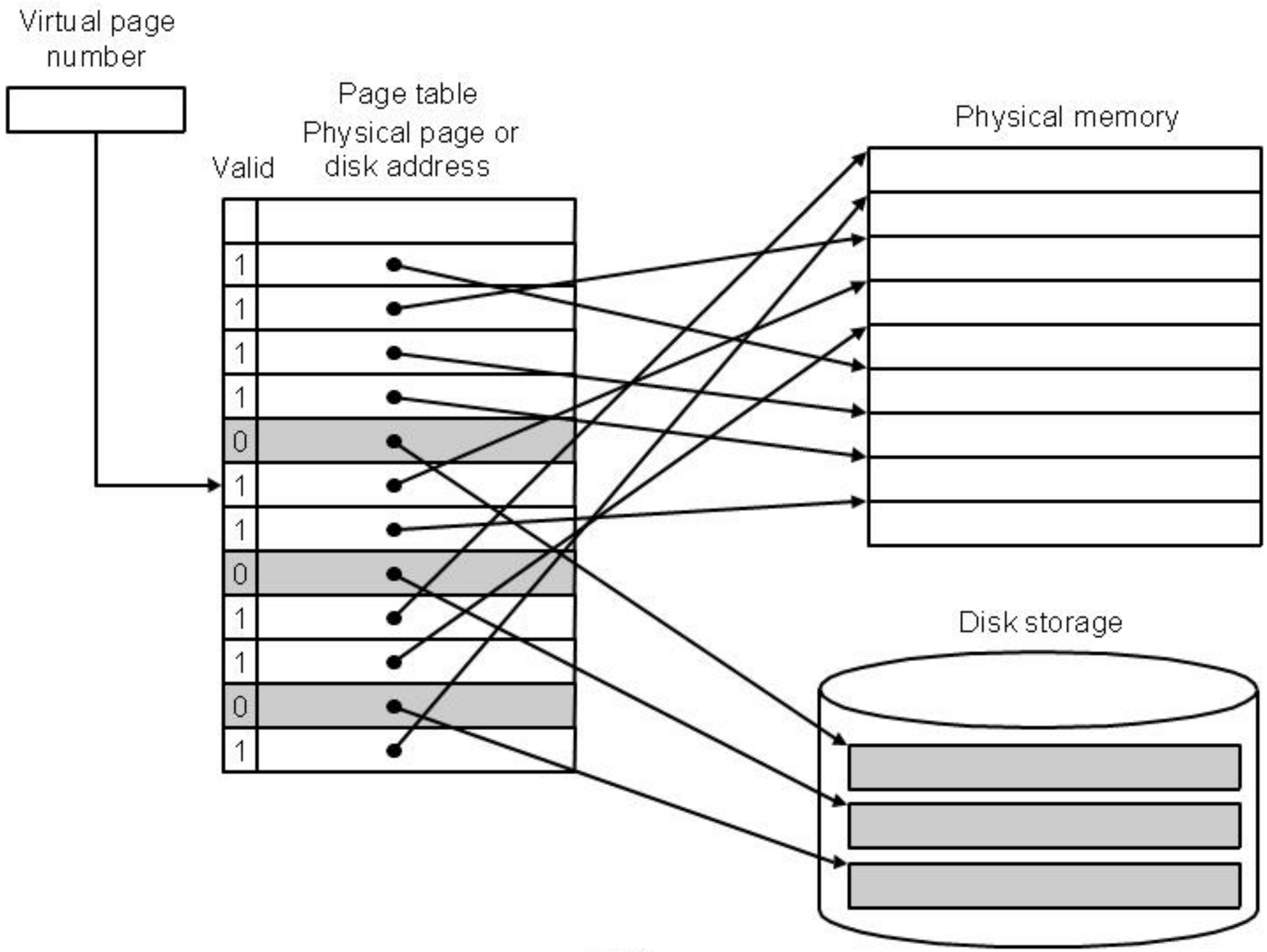
- Co se stane při výpadku stránky?
- Operačnímu systému je předáno řízení, ten musí najít požadovanou stránku na disku a umístit ji do hlavní paměti.
  (rozdíl oproti cache !)
- Předání řízení a výměna se zajišťuje s využitím výjimek.
- Přirozeně je nutné udržovat informaci o poloze každé stránky na disku.
- Nevíme dopředu, kdy se bude stránka v paměti vyměňovat.
- OS vytváří datovou strukturu pro uložení každé virtuální stránky na disk.
  - Stránková tabulka obsahuje obvykle 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

- 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. …

Výpadek stránky a tabulka stránek



Implementace LRU

- O úspěšnosti VM rozhoduje také algoritmus výměny stránek. Jedním z nich je 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 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

- 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.

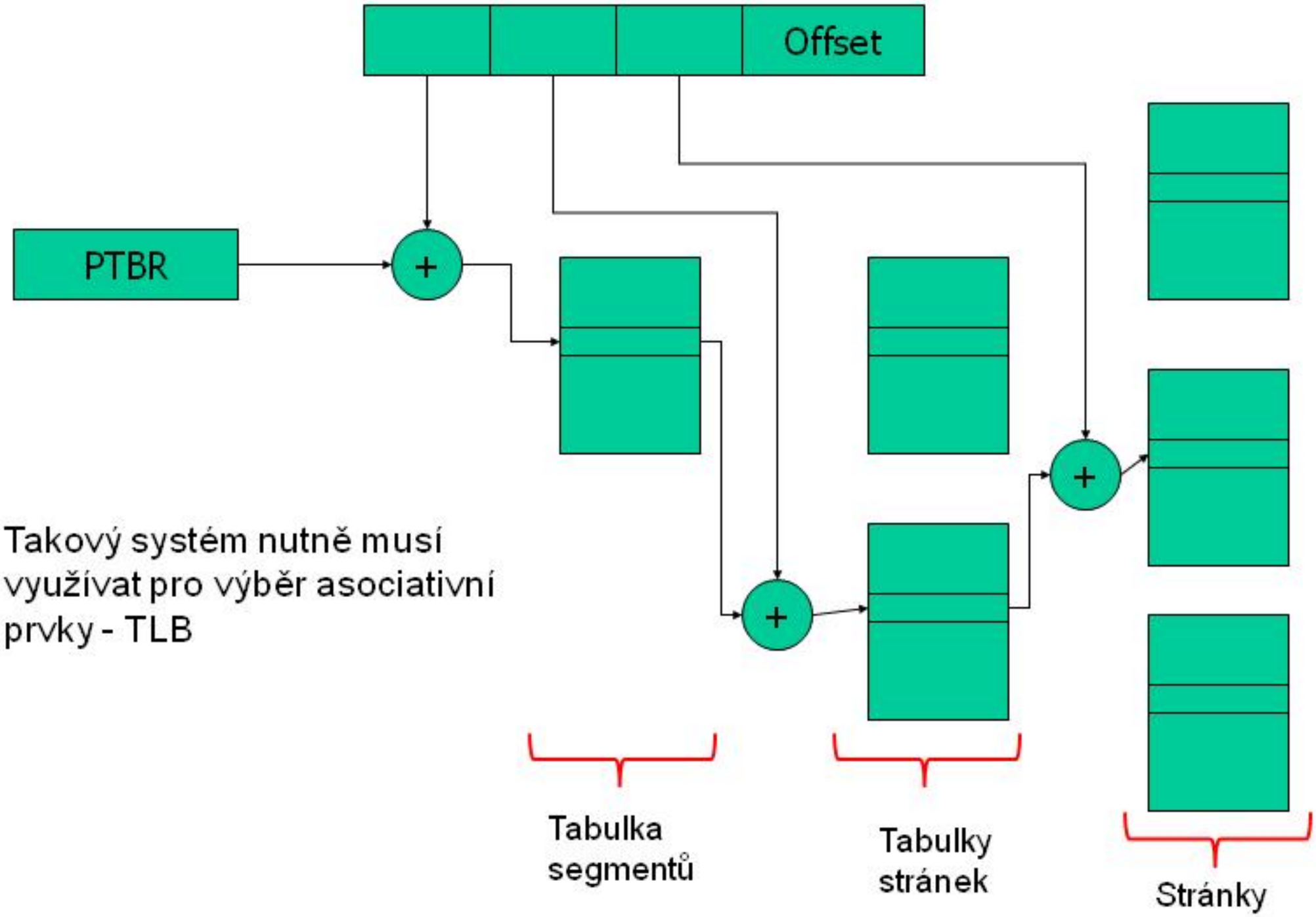
Složitější techniky

- 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

- 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.

Víceúrovňový systém



Takový systém nutně musí využívat pro výběr asociativní prvky - TLB

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

- 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, známá z úrovně cache, je nepoužitelná.

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

- 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.

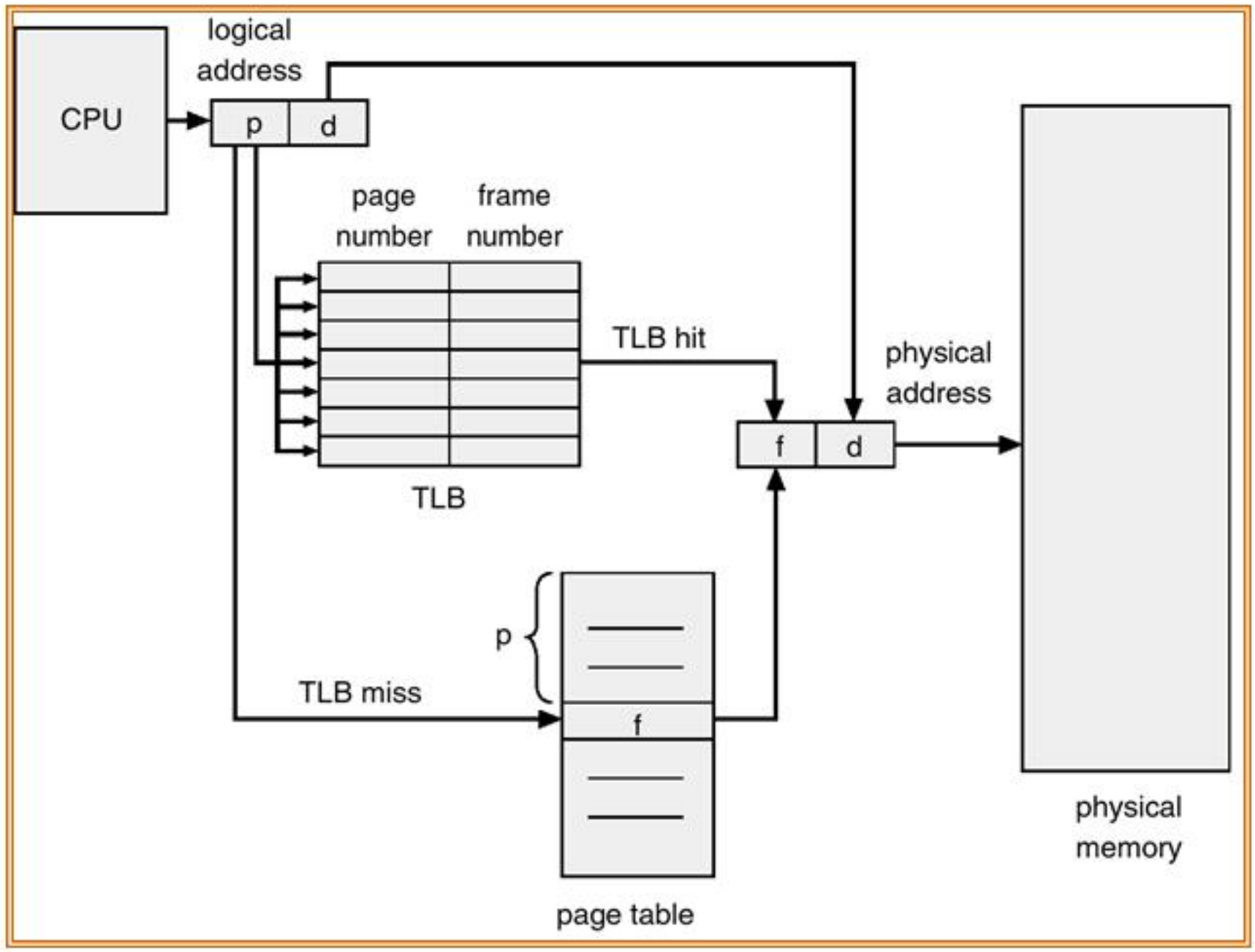
Translation-Lookaside Buffer (TLB)

- 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.

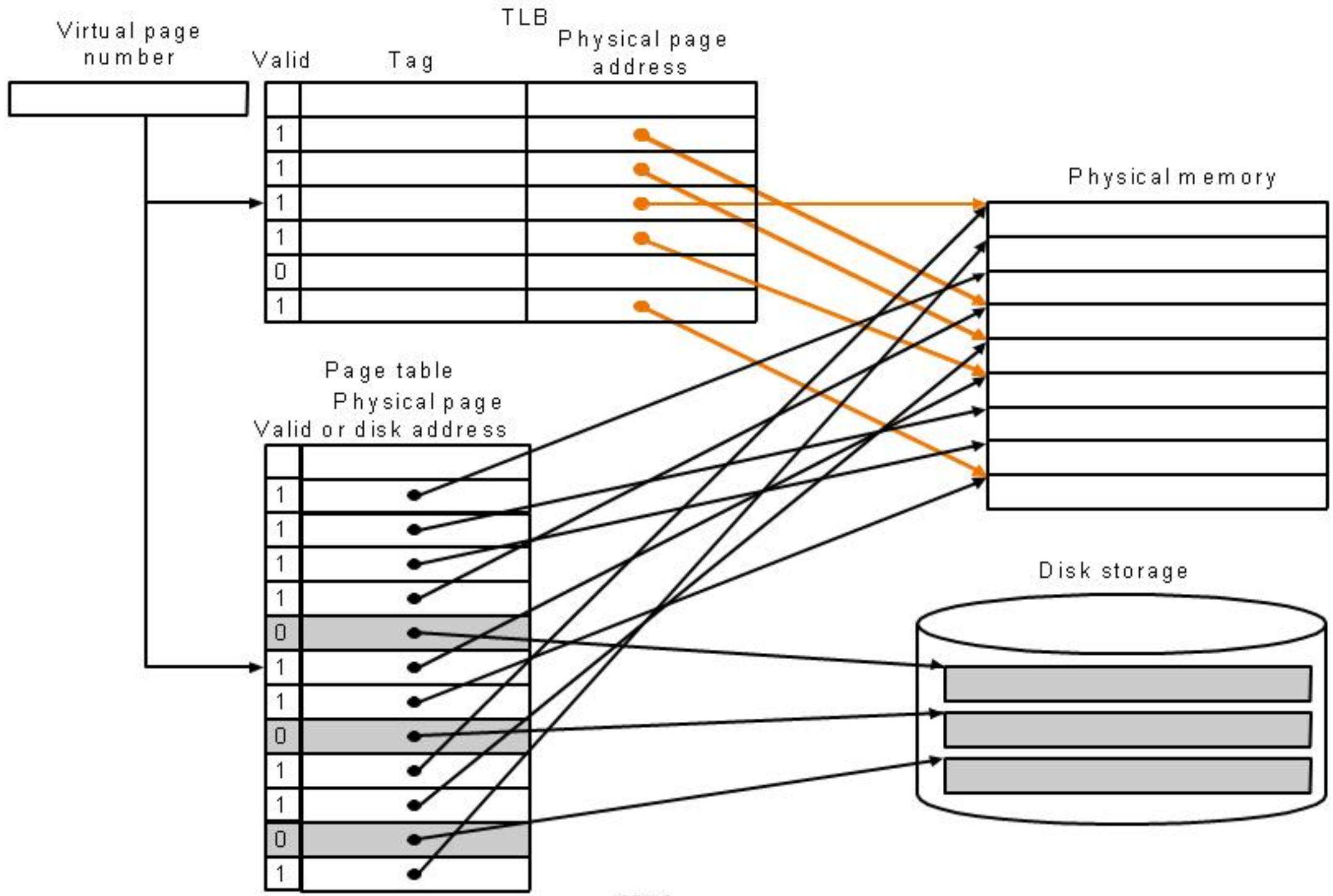
Virtuální paměť a TLB



Při neúspěchu v TLB pokračuje transformace výběrem ze stránkové tabulky

p … page
d … displacement (jinak offset)
f … frame

Virtuální paměť a 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 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 TLB a výpadky stránek

- Je jednoduché určit, který z uvedených dvou případů nastal.
- Zpracováváme-li TLB miss, nahlédneme do tabulky stránek, abychom vyšetřili vstupní bod stránky do TLB.
  - Má-li nalezený vstupní bod nastaven bit platnosti, můžeme získat fyzické číslo stránky z tabulky stránek a zapsat je do TLB.
  - Nemá-li nalezený vstupní bod nastaven bit platnosti, stránka se nenachází v paměti a jedná se o výpadek stránky.
- Nastane-li TLB miss, ale nikoliv výpadek stránky, lze toto situaci ošetřit v software nebo v hardware. Jde-li o výpadek stránky, je volán OS.

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 miss v TLB může indikovat dvě různé možnosti:
  1. Stránka je přítomna v paměti a pouze je třeba vytvořit chybějící přístupový bod v TLB pomocí tabulky stránek.
  2. Stránka není přítomna v paměti a pak je třeba předat řízení OS, který se musí vypořádat s výpadkem stránky.

Výpadky TLB a výměna

- Některé 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).

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

- 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.

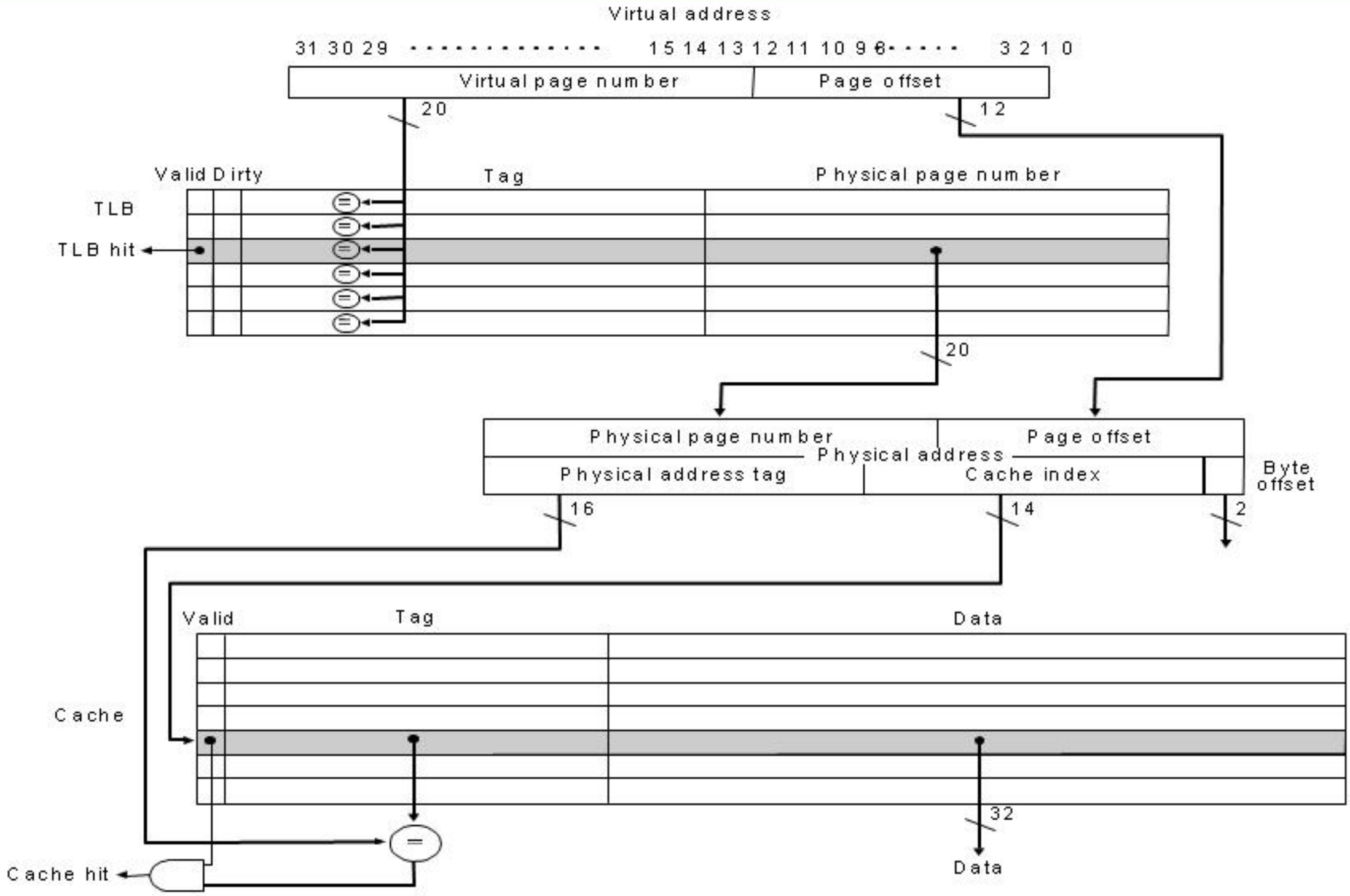
Výjimka výpadku stránky

- Nastane-li výpadek stránky, předá mechanizmus ošetření výjimky řízení OS (handler ošetření výpadku stránky).
- Výpadek stránky je rozpoznán přibližně v době cyklu, kterým se provádí přístup do paměti.
  - Jakmile vstoupíme do handleru ošetření výjimky výpadku stránky (určeno registrem Cause), OS uloží celý stav probíhajícího procesu.
  - OS zveřejní virtuální adresu, kde byl způsoben výpadek (využitím EPC nebo instrukcí a instrukci na EPC).

Výjimka - výpadek TLB

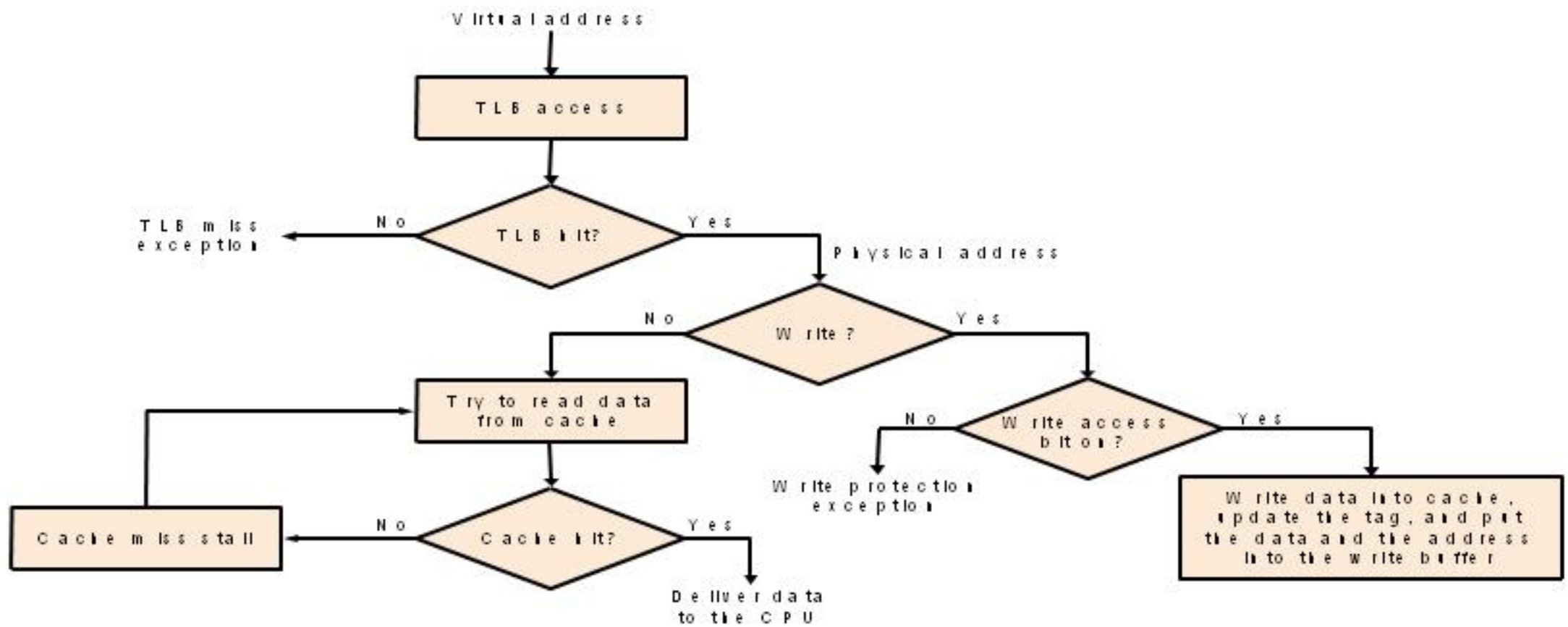
- Nastane-li výjimka výpadku TLB, hardware uloží číslo stránky reference do speciálního registru a generuje výjimku.
- OS zahájí obsluhu výpadku. Handler výpadku TLB zjistí ve stránkové tabulce uložené virtuální číslo stránky a registr stránkové tabulky.
  - Pravý výpadek stránky nastane, neobsahuje-li stránková tabulka fyzickou adresu.
- Užitím speciálního souboru instrukcí MIPS, OS aktualizuje TLB zapsáním fyzické adresy ze stránkové tabulky do nové položky TLB.

DECStation 3100 TLB a virtuální paměť



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

- 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

- 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

- 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

Restart versus přerušení instrukcí

- MIPS instrukce jsou *restartovatelné* – jestliže některá způsobí výpadek stránky, je zastavena uprostřed provádění a OS ošetří výpadek stránky.
- Jakmile je výpadek stránky ošetřen, instrukce je startována od počátku.
- Toto lze provést u procesorů s jednoduchým instrukčním souborem jako je MIPS. U složitějších architektur musí být instrukce přerušena, uložen kontext a později se dokončuje rozpracovaná instrukce.
  - Toto je nutné, protože některé instrukce mohou pracovat s tisíci datovými slovy (např. blokové přenosy).
  - Vyžaduje to pak mnoho HW prostředků pro uložení stavu a složité interakce mezi HW a OS.

Ochrana paměti

- Jednou z velmi důležitých vlastností virtuální paměti je možnost sdílení jednoho paměťového prostoru více procesy. Vyžaduje to ale ochranu paměti mezi jednotlivými procesy a OS.
  - Žádný proces nesmí zapisovat do adresního prostoru jiného procesu.
  - Žádný proces nesmí číst z adresního prostoru jiného procesu.
  - Procesy musí mít chráněný svůj virtuální adresní prostor. Jinak, VM má své výhody i nevýhody.
- VM je klíčem k *izolaci procesů*.
- Pole bitů pro ochranu jsou součástí *page descriptorů*, popř. *segment descriptorů*

Implementace izolace procesů

- Každý proces má svůj virtuální adresní prostor.
- Jestliže OS organizuje stránkové tabulky tak, že nezávislé virtuální stránky mapuje do disjunktních fyzických stránek (nesdílí se), pak žádný proces nemůže přistupovat k datům jiného procesu.
- Žádný proces nesmí manipulovat se svými stránkovými tabulkami, ale OS může.
- Z tohoto důvodu umísťují moderní operační systémy stránkové tabulky do vlastního adresního prostoru (kernel memory).

Závěr

- Použití cache zlepšuje efektivitu paměti:
  - Omezuje často se vyskytující operace read/write na přístup do rychlé paměti
  - „Shlukuje“ přístupy registr-hlavní paměť
  - Mění charakter chování procesoru vzhledem ke sběrnici
  - Vytěžuje I/O pipeline a tak *maximalizuje propustnost hierarchického paměťového systému*
- Použití virtuální paměti je výhodné, protože:
  - Mapuje velký symbolický adresní prostor na malý fyzický adresní prostor – mechanizmus podobný cache

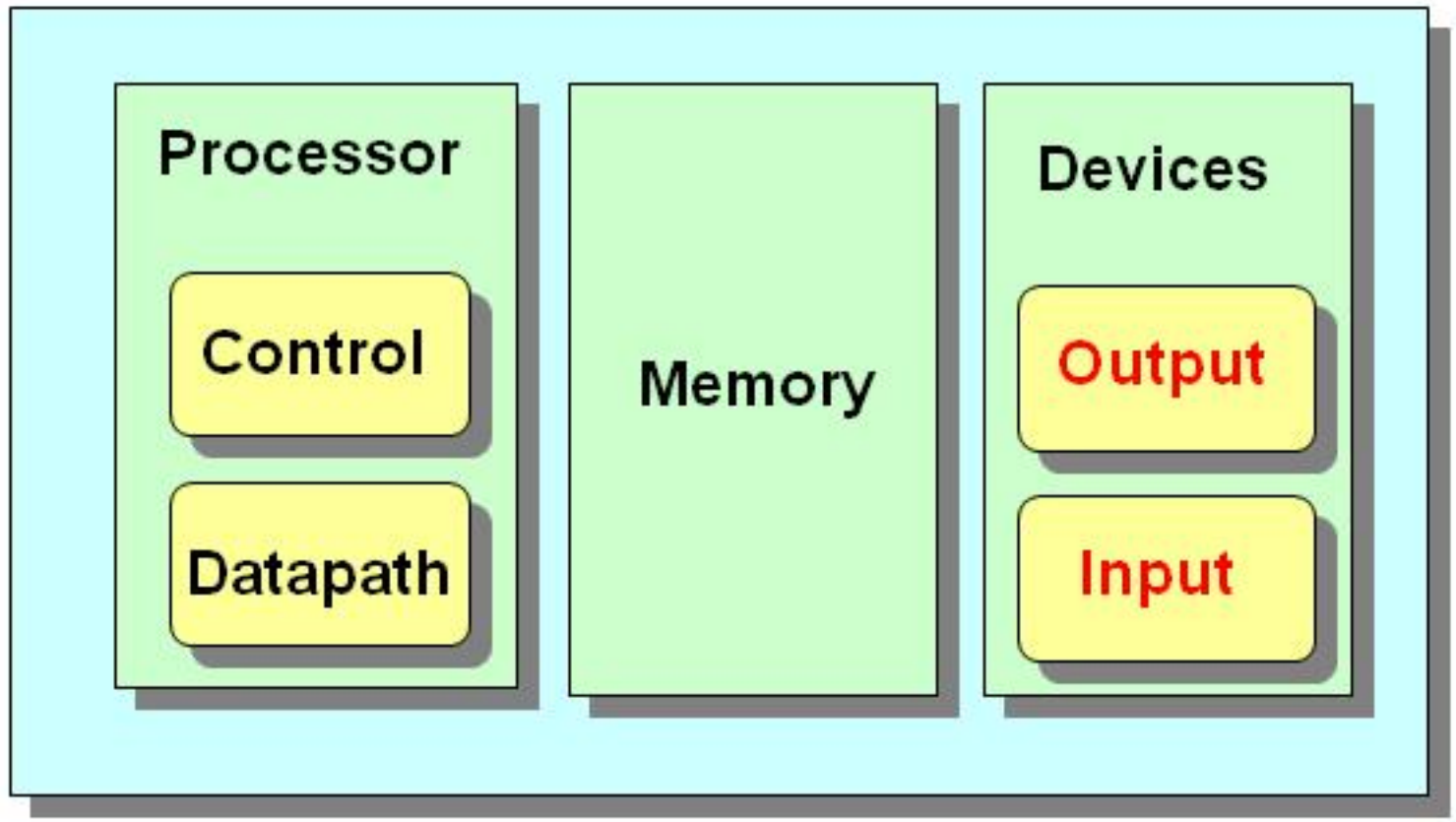
Příště: I/O a sběrnice

Úvod do organizace počítače

I/O zařízení a sběrnice

[K přípravě využita kniha: *Computer Organization and Design*, Patterson & Hennessy, © 2010]

Opakování: Základní části počítače



- Důležité metriky I/O systému
  - Výkon
  - Rozšiřitelnost
  - Spolehlivost
  - Cena, velikost, váha

Vstupní a výstupní zařízení

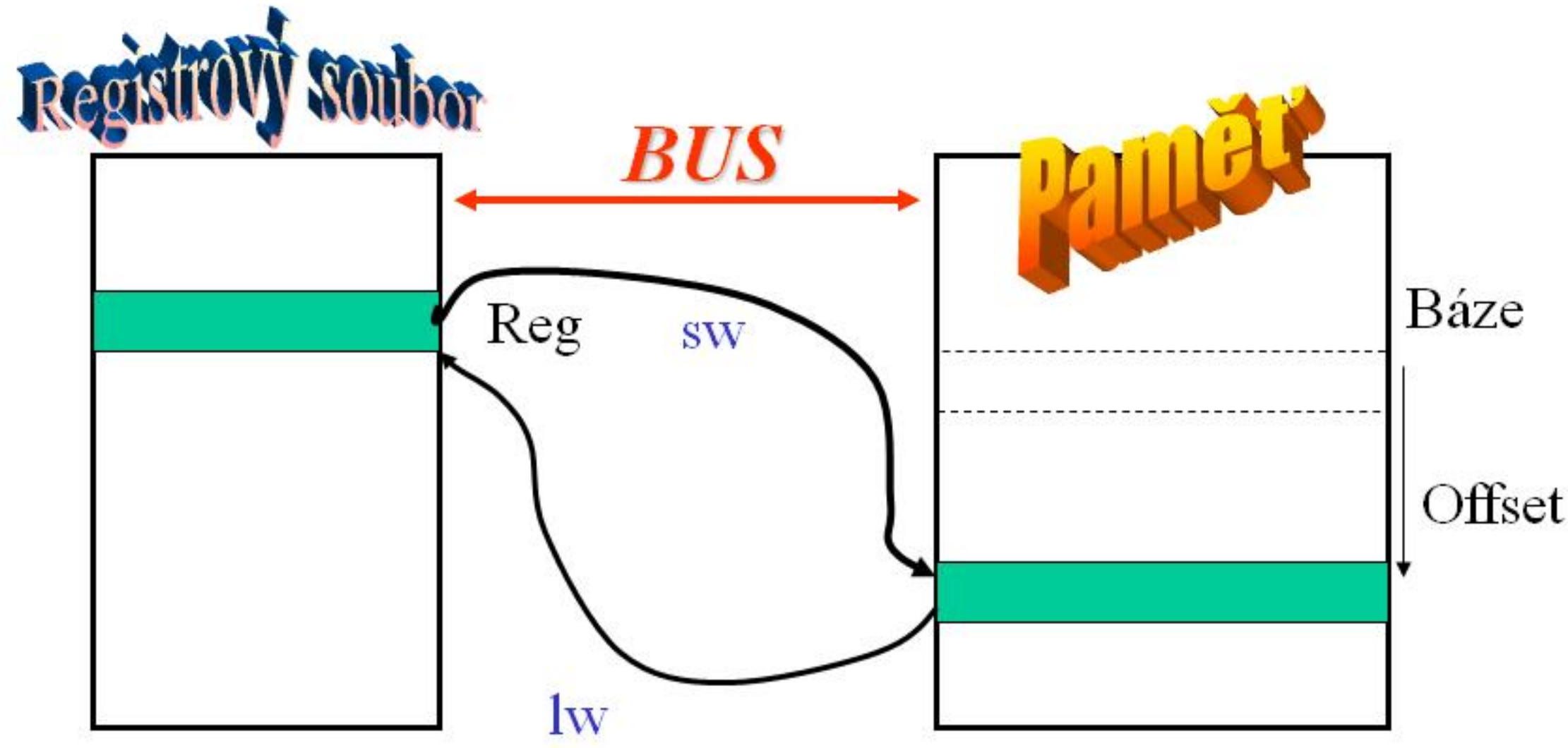
- I/O zařízení jsou velmi různorodá z hlediska
  - Chování – vstup, výstup nebo paměť
  - Partner – člověk nebo stroj
  - Rychlost přenosu dat – špičková rychlost, se kterou jsou data přenášena mezi I/O zařízením a hlavní pamětí nebo procesorem

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

- Vysílač/Přijímač: Zdroj/Příjemce dat
- Kanál: Cesta pro data nebo instrukce
- Řadič: Master řídí akce sběrnice



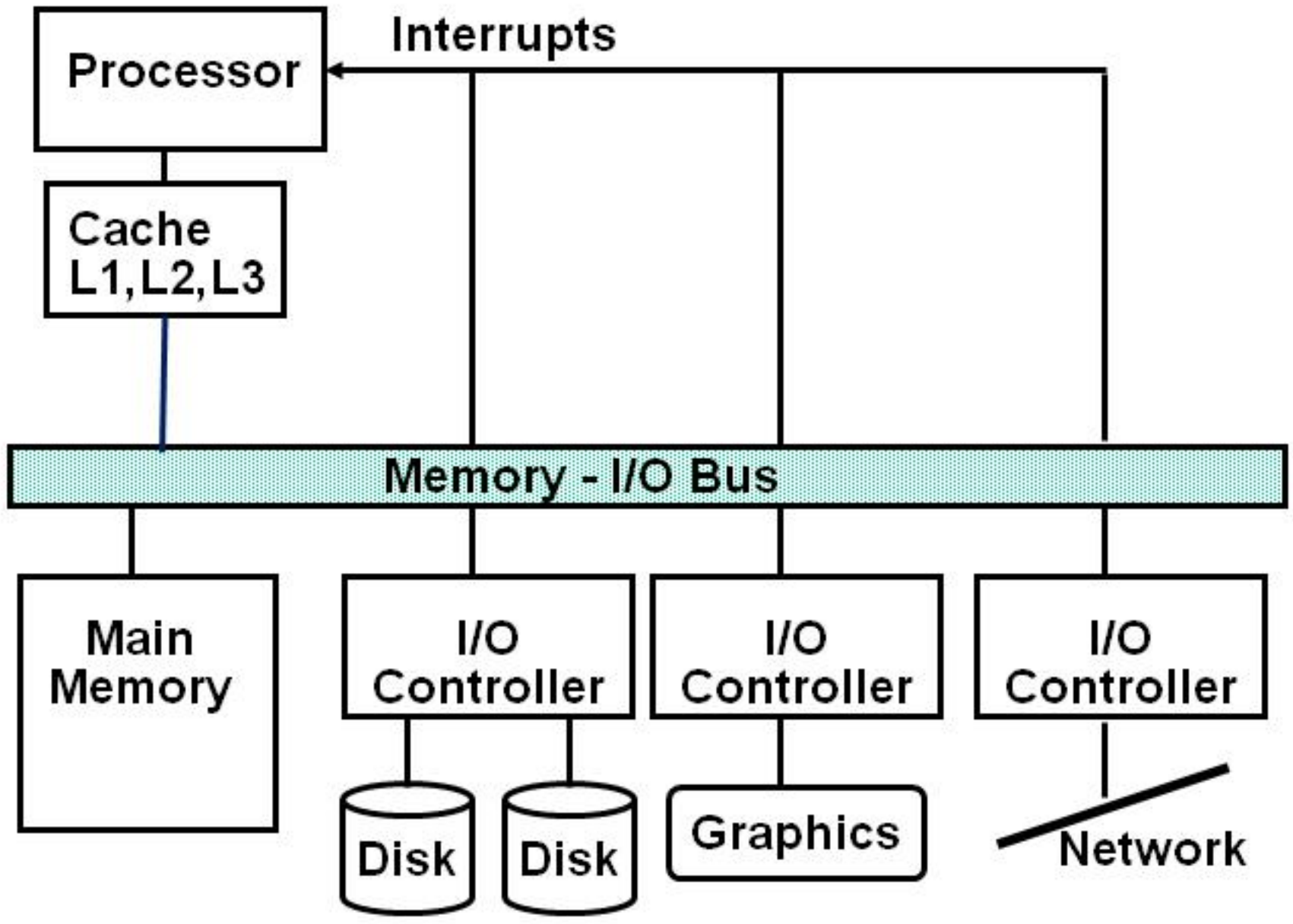
Fyzické sběrnice

- **Paralelní sběrnice**
  - *N* paralelních vodičů:

  - 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 (???)

Dnes už příliš neplatí

Typický I/O systém



Přehled organizace 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

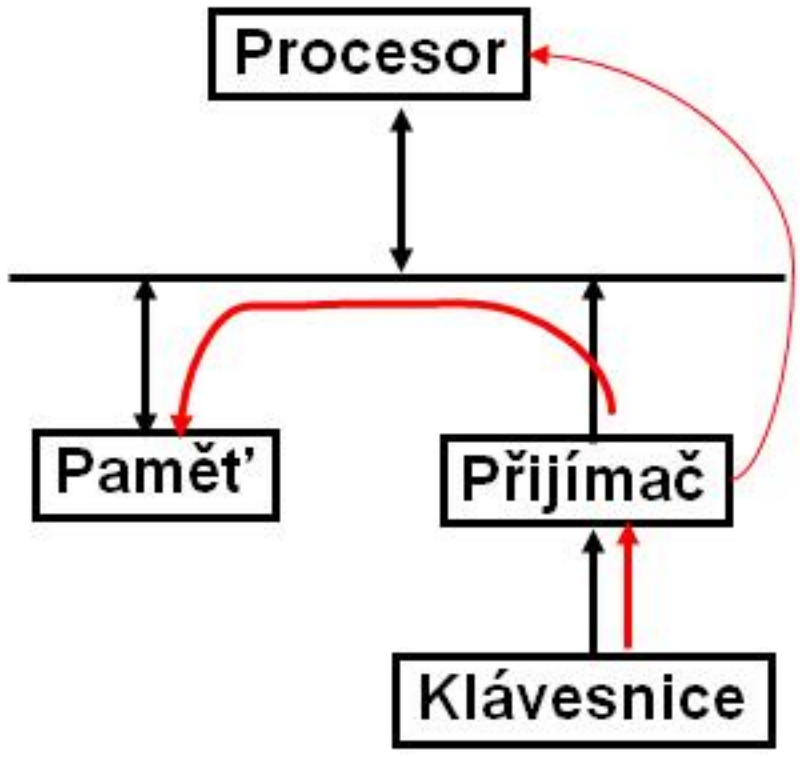
1. **Polled**
   - Programová kontrola stavu I/O zařízení a následné plánování činnosti volných I/O zařízení
   - *Dotaz na stav (status)* (busy, wait, idle, dead…)
2. **Interrupt-Driven**
   - Obsluha I/O na požadavek, vyjádřený interruptem
   - *Vyžaduje **frontu** na uložení čekajících I/O žádostí*
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)

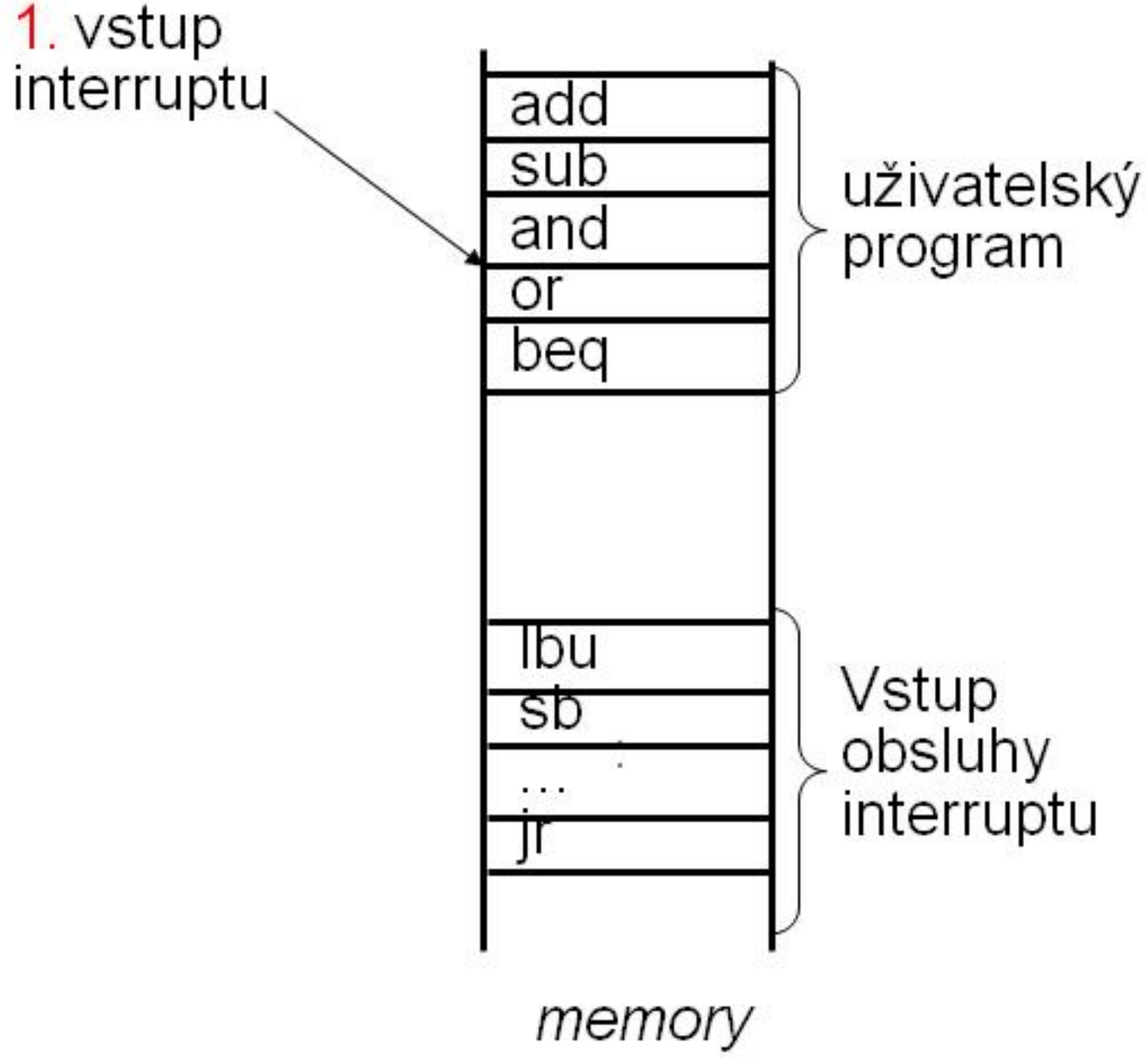
Registry I/O zařízení zaujímají vlastní adresní prostor nebo bývají mapovány do oblasti paměťového adresního prostoru.

Komunikace I/O zařízení a procesoru

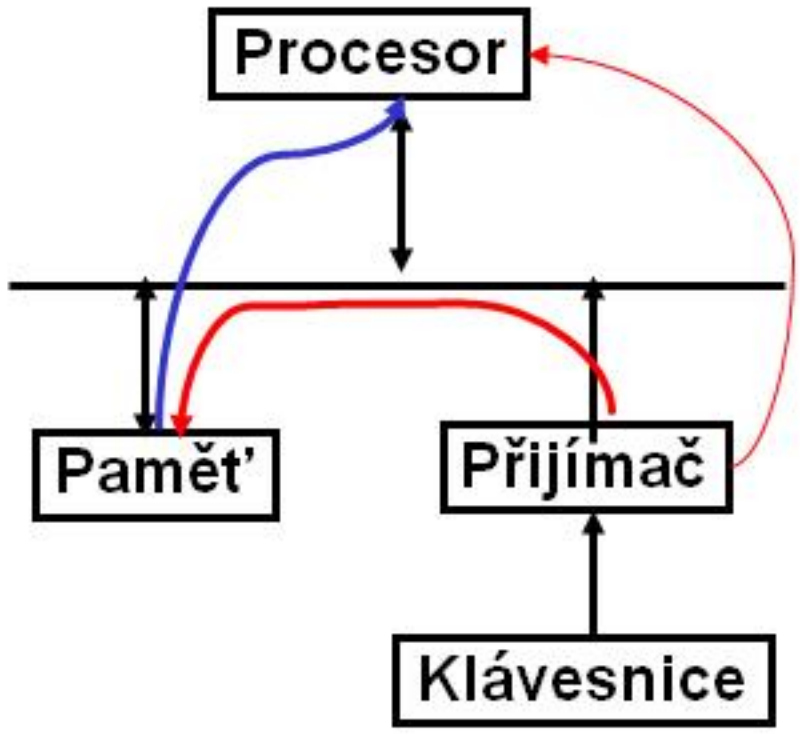
- **Jak procesor ovládá I/O zařízení**
  - Speciální I/O instrukce
    - Specifikují jak zařízení, tak vlastní příkaz
  - I/O mapované do paměťového prostoru
    - Část adresního prostoru (obvykle horní adresy) je přiřazena I/O zařízením
    - Čtení a zápis na tato paměťová místa se interpretují jako 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

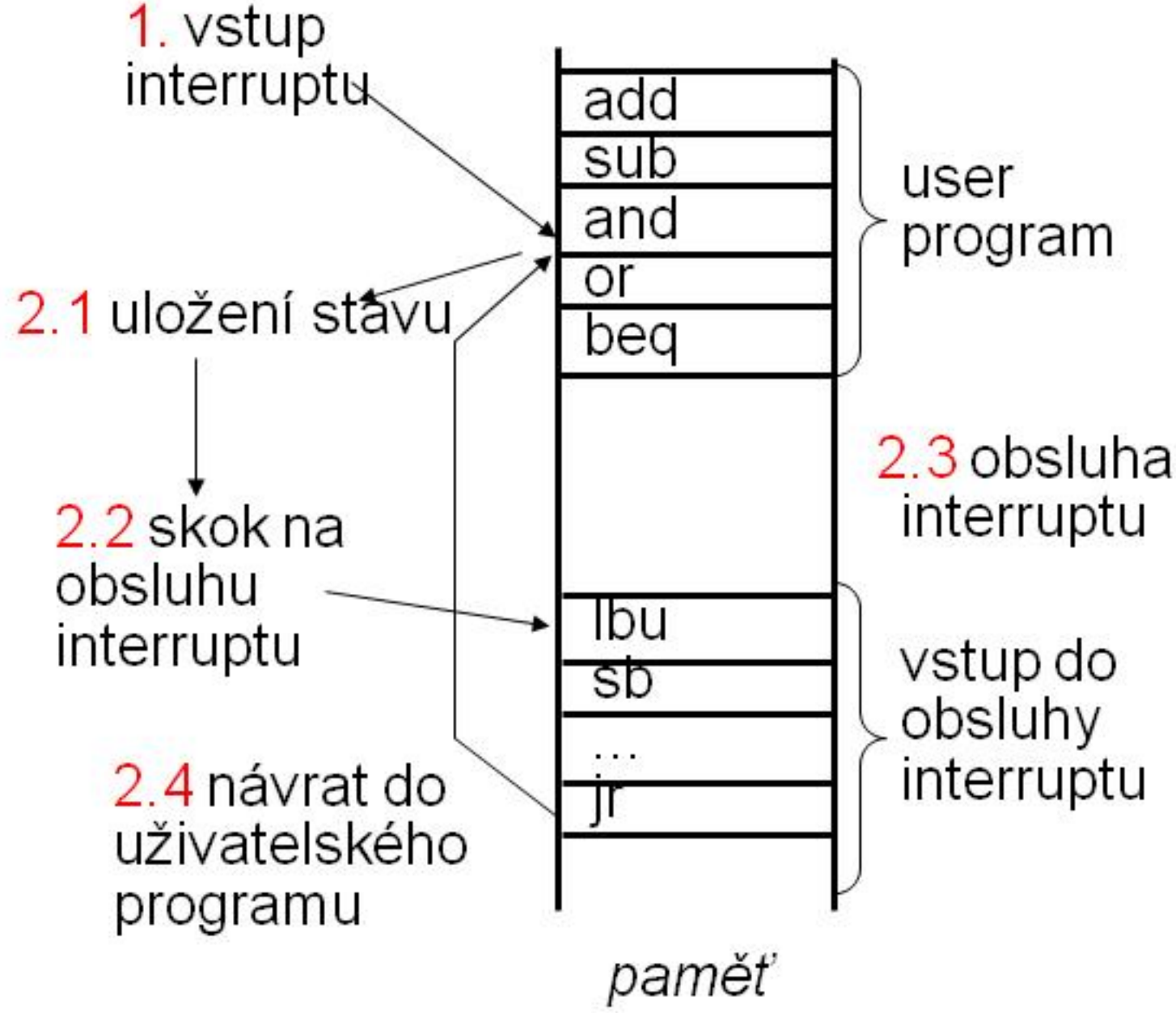
Vstup s využitím interruptů



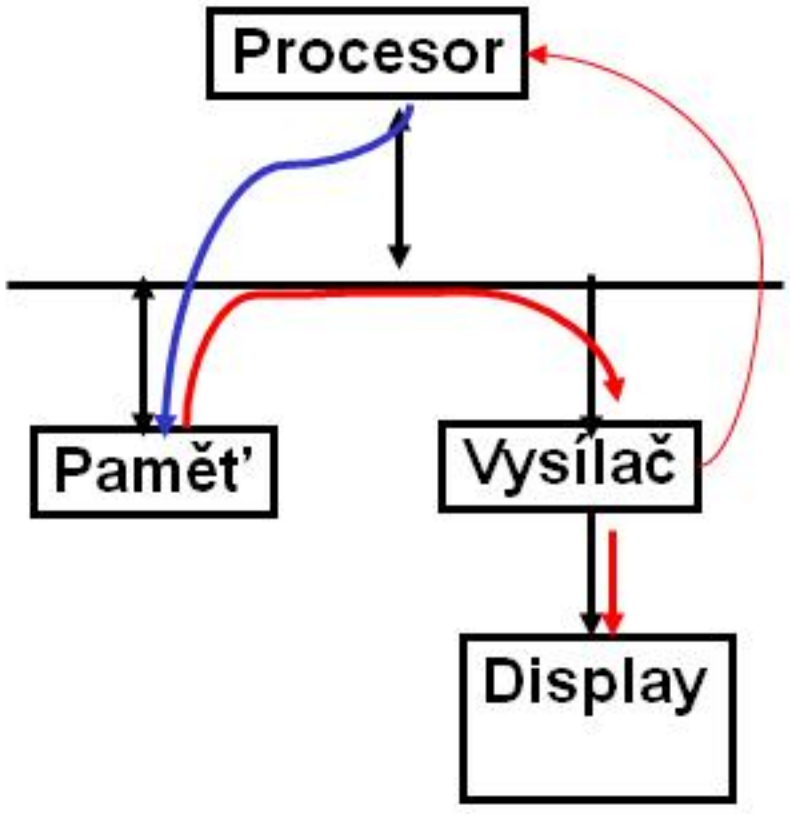


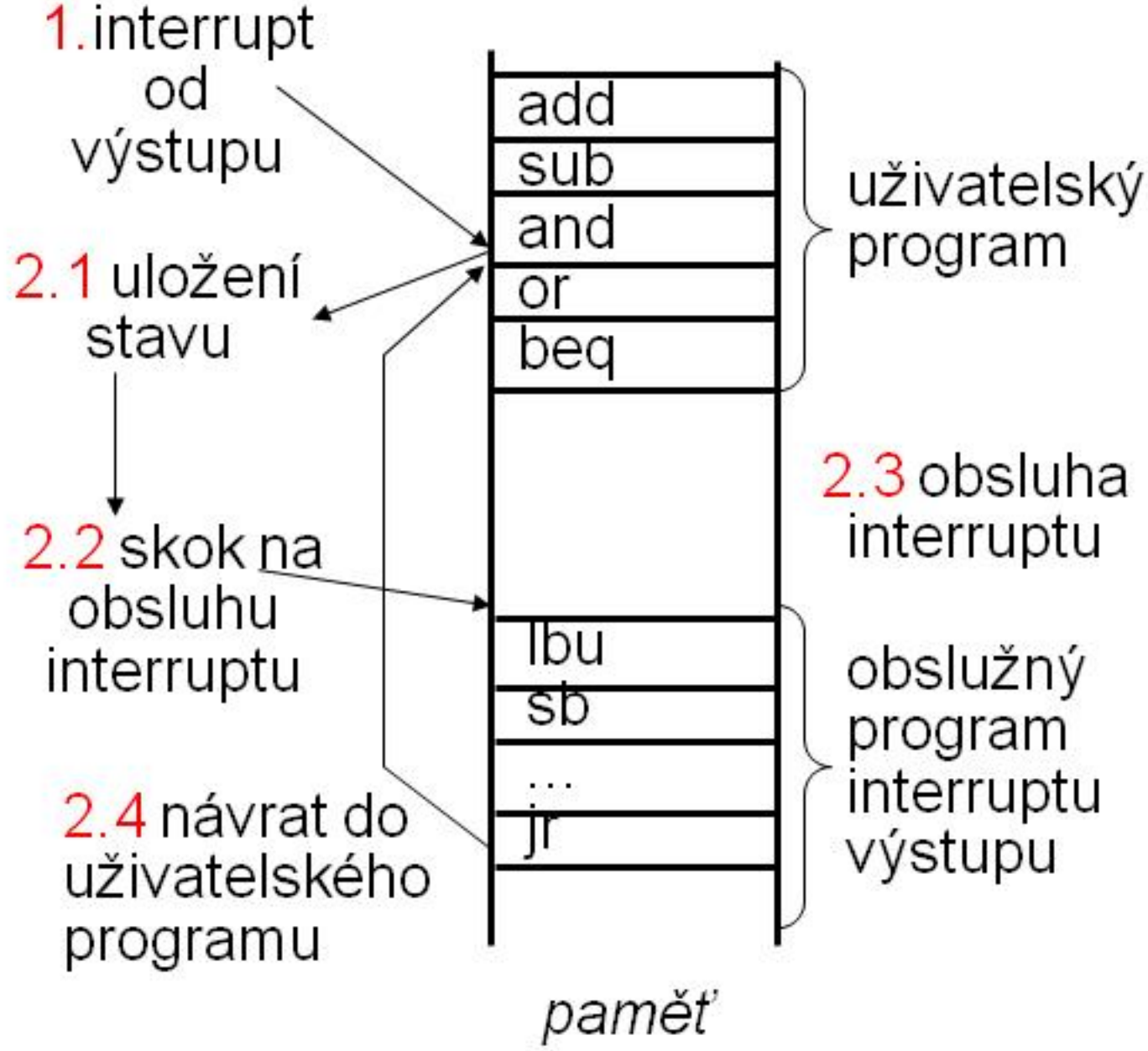
Vstup s využitím interruptů





Výstup s využitím interruptů





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)

- 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 initicializuje 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

- 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

- 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 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
  - 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)
  - 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

- **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

- **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

- 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 $3.1O^9$ 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:

  - 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:

  - Diskové SCSI I/O kontroléry DMA s přenosovou rychlostí 320 MB/s, které obsluhují až 7 disků/kontrolér
  - Diskové jednotky s přenosovou rychlostí (při operaci 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 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 $3.1O^9$ 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

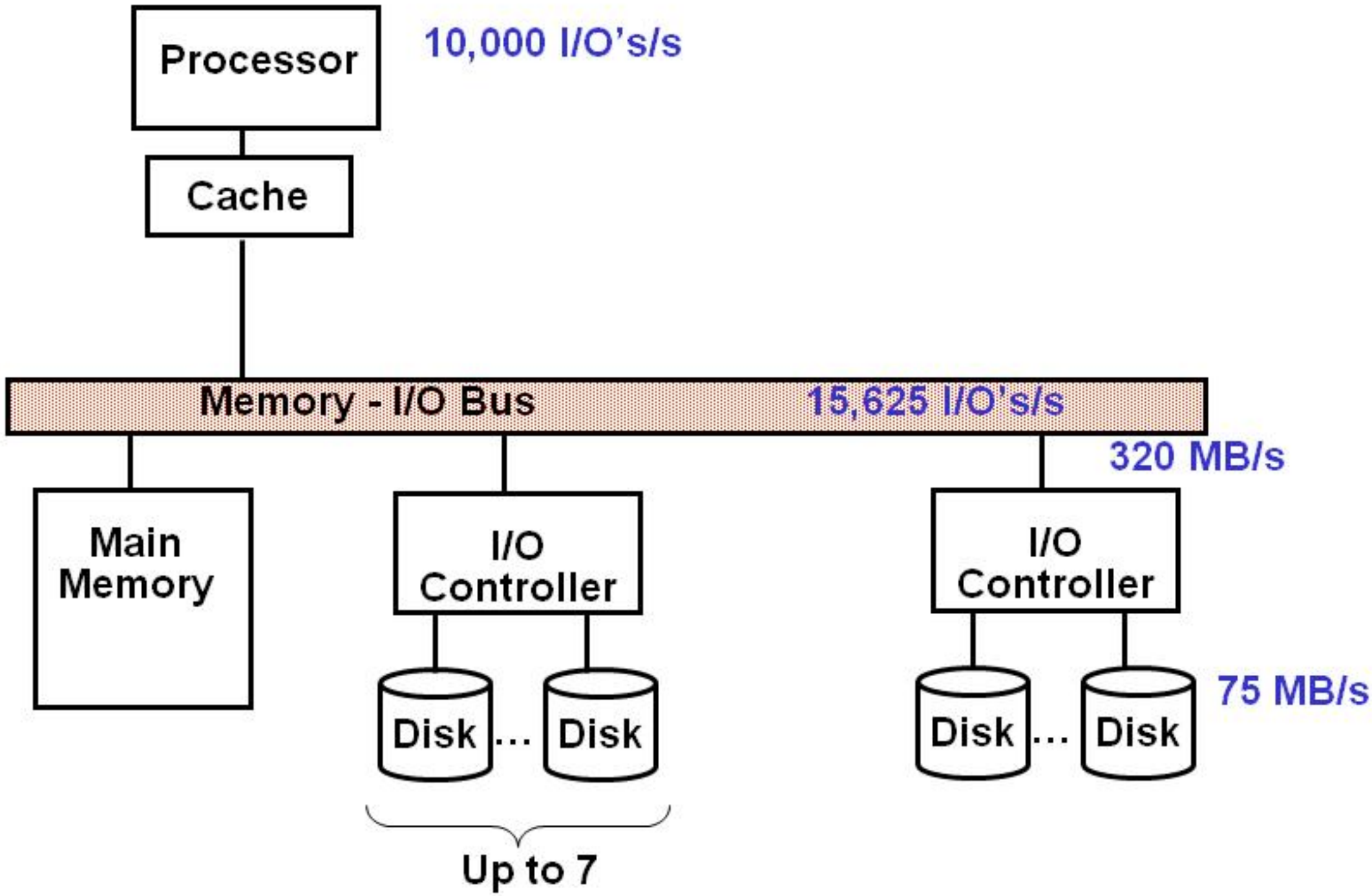
Každá disková I/O čte nebo zapisuje 64 KB, takže maximální rychlost I/O sběrnice je:

$$\frac{\text{Bus bandwidth}}{\text{Bytes per I/O}} = \frac{1000 \times 10^6}{64 \times 10^3} = 15{,}625 \text{ I/O's/s}$$

  - 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



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

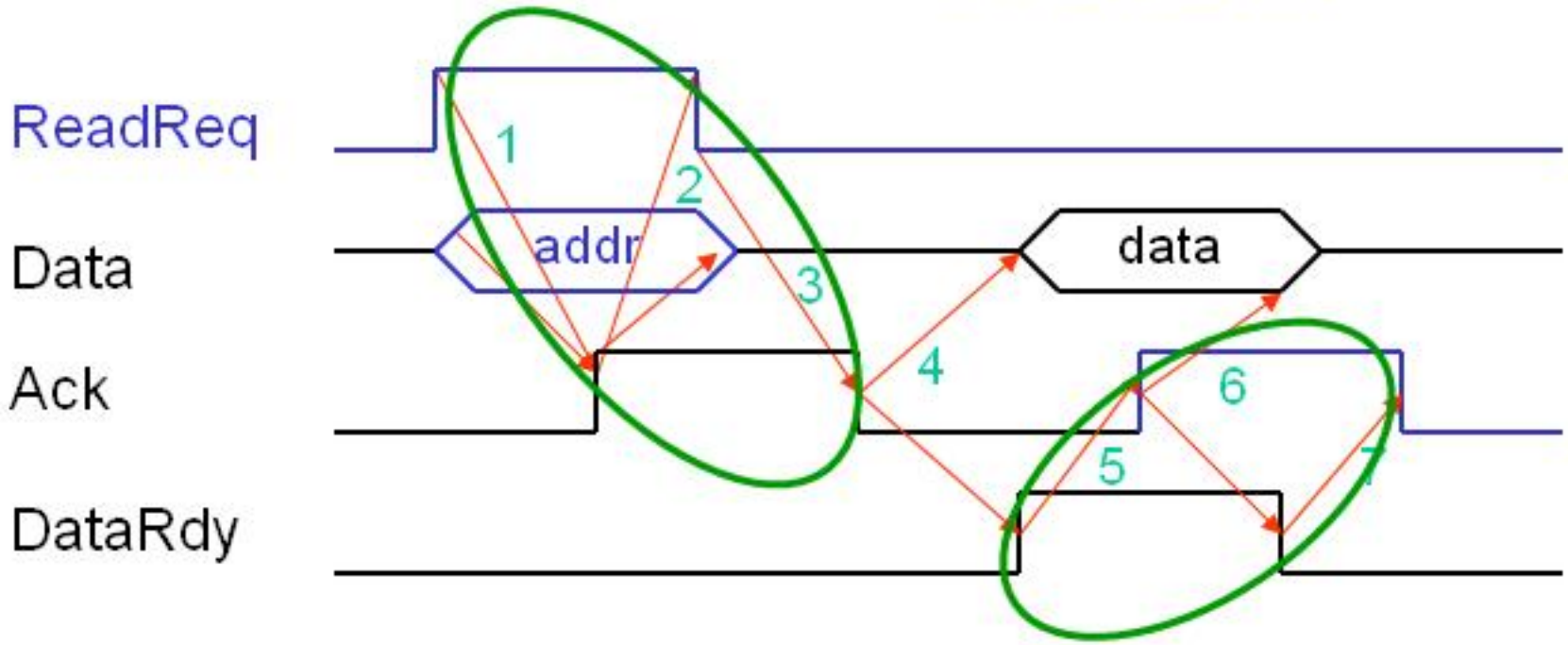
Pro výpočet počtu SCSI diskových kontrolérů potřebujeme znát střední přenosovou rychlost jednoho disku, abychom určili, zda můžeme připojit maximální počet 7 disků na SCSI kontrolér a že diskový kontrolér nebude saturovat sběrnici IO-paměť během DMA přenosu.

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 (read) dat z paměti na I/O zařízení



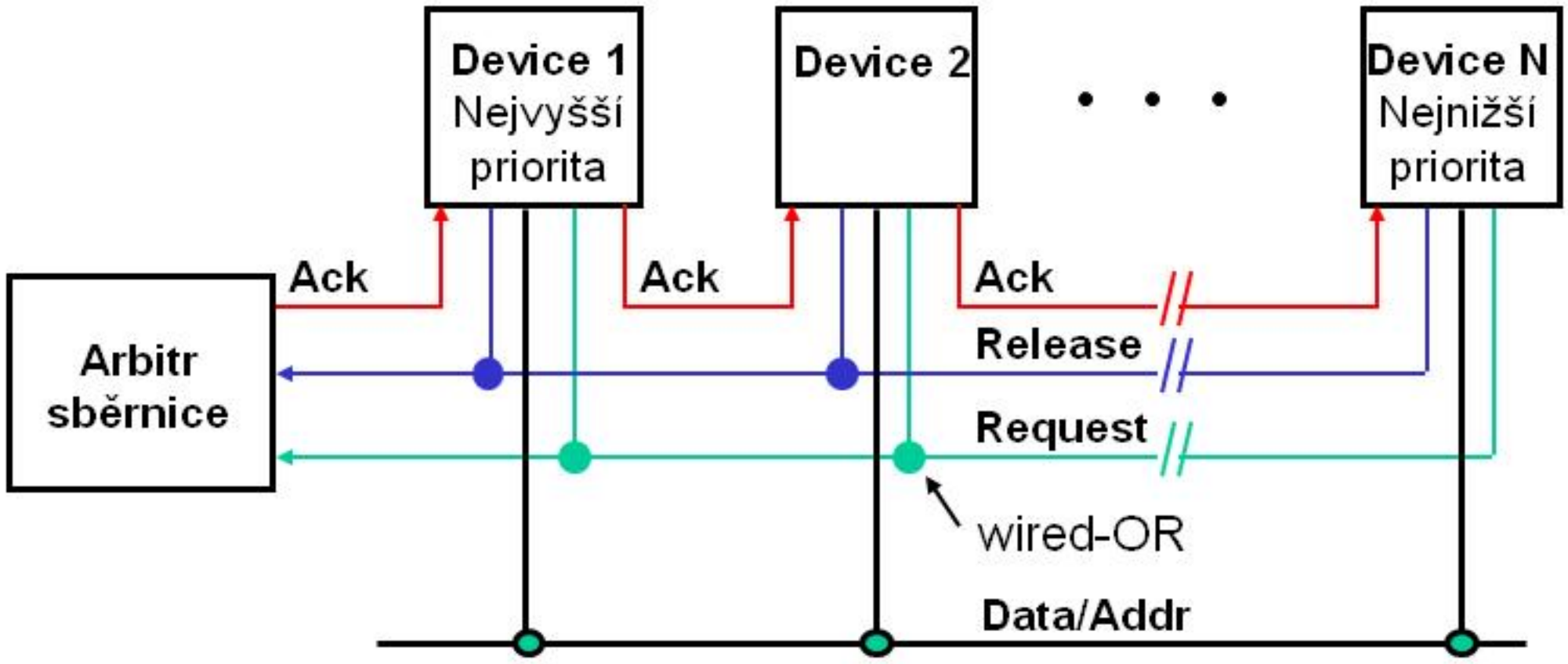
Zařízení I/O oznamuje požadavek nastavením ReadReq a vydáním addr na datových linkách.

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

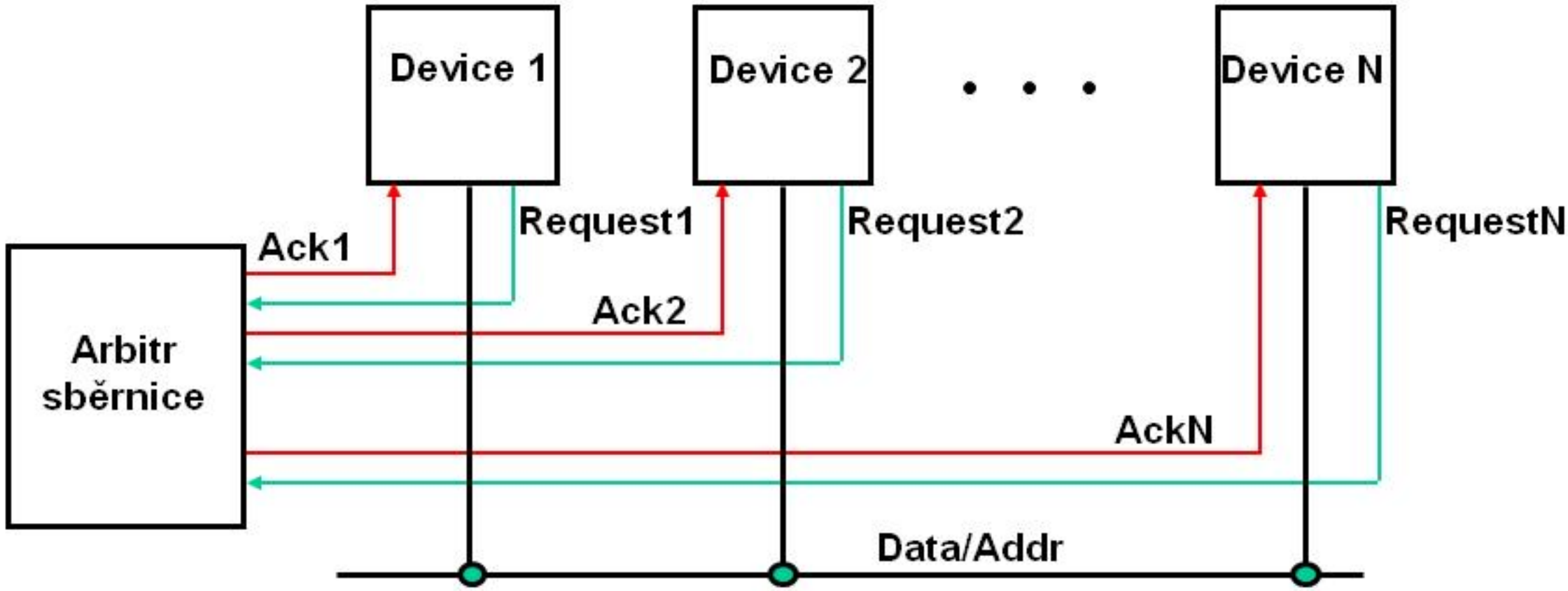
- Sběrnici může být schopno současně řídit větší množství zařízení => nutnost arbitrovat požadavky
- Arbitrační schémata se snaží zohledňovat:
  - Bus priority – zařízení nejvyšší priority by mělo být obslouženo nejdříve
  - „Fairness“ – i zařízení s nejnižší prioritou nesmí být odříznuto od sběrnice
- 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



- Výhoda: jednoduchost
- Nevýhody:
  - Nelze zajistit „spravedlnost“ – zařízení nízké priority mohou být trvale blokována
  - Pomalé – „daisy chain“ signál omezuje rychlost procesu přidělování

Centralizovaná paralelní arbitrace



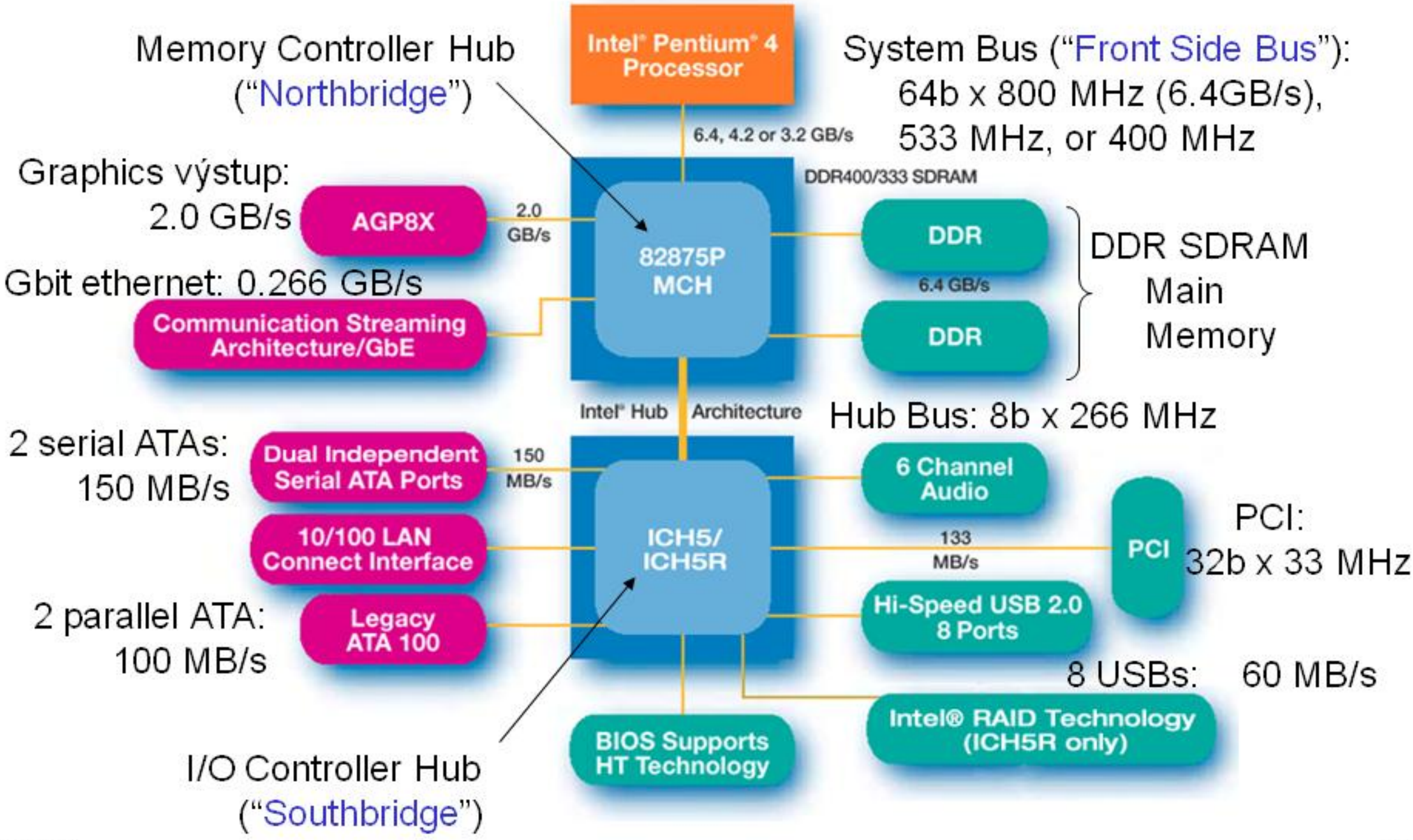
- Výhody: flexibilní, může zajistit „spravedlnost“
- Nevýhody: složitější hardware arbitru
- Používáno hlavně u všech sběrnic procesor-paměť a u rychlých I/O sběrnic

Šířka pásma sběrnice

- 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



Sběrnice z hlediska vývoje

- 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)

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 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 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
  - Šířka pásma (B), četnost bitových chyb (BER)
  - Četnost poruch (FR), cena poruch (FC)
- Výpočet aktuální šířky pásma (B')

$$B' = B * \underline{(1-BER)} - \underline{FC/(1-FR)}$$

Vliv chyb — Error

Vliv poruch — Failure

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 (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
  - 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:
  - 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)

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