

# **Proudové zpracování instrukcí I. Celočíselná *pipeline* RISC**

**© Ing. Miloš Bečvář**

**s využitím slajdů prof. Davida Pattersona**

**© CS152, University California at Berkeley, 1996**

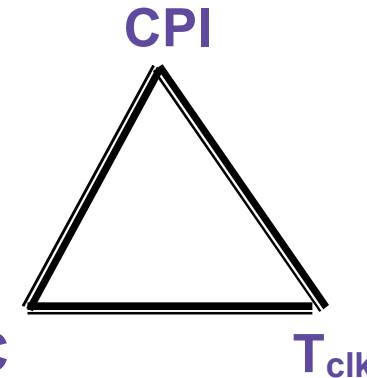
## Osnova přednášky

- Návrh jednoduché datové cesty a řadiče pro **DLX** (s použitím znalostí z X36JPO)
- Návrh proudově pracující datové části a řadiče **DLX**
- Porovnání výkonnosti různých organizací procesoru **DLX**
- Problémy vzniklé použitím proudového zpracování instrukcí (**hazardy proudového zpracování**)

# Opakování: Výkonnostní rovnice CPU

## ° Výkonnost procesoru závisí na:

- Počtu instrukcí (IC)
- Počtu taktů na instrukci (CPI)
- Periodě hodinového signálu ( $T_{\text{clk}}$ )



## ° Organizace CPU ovlivňuje:

- Periodu hodinového signálu ( $T_{\text{clk}}$ )
- Počet taktů na instrukci (CPI)

## ° Výkonnostní rovnice CPU (doba provádění programu):

$$T_{\text{CPU}} = IC * CPI * T_{\text{clk}}$$

# Návrh CPU – obecný postup (viz X36JPO)

1. **Analýza ISA => požadavky na datovou část**
  - Sémantika instrukcí vyjádřena jako „přesuny“ mezi registry
  - Datová část musí obsahovat minimálně všechny registry obsažené v ISA (také zvané *architekturní registry*)
  - Datová část musí podporovat všechny „přesuny“ mezi registry
2. **Výběr komponent datové cesty, definice časování**  
(registry řízené hranou/hladinou, jednofázové či vícefázové hodiny)
3. **Volba propojení elementů datové části**  
(sběrnice nebo multiplexery či kombinace)
4. **Návrh řadiče pro danou datovou cestu**

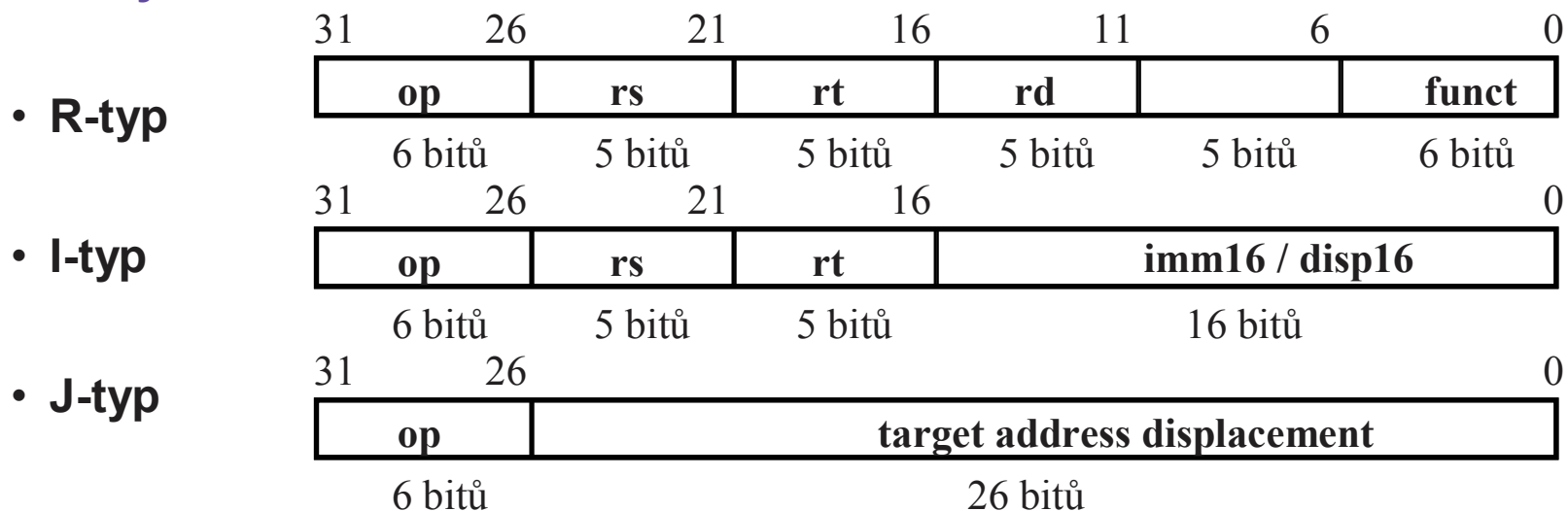
# “Typický 32-bitový RISC” - DLX

- 32-bit kódování instrukcí pevné délky (**3 formáty**)
- 32 32-bitových univerzálních registrů (**R0=0**)
- 32 FP registrů, registry dvojnásobné přesnosti v párech
- Registr – Registr (Load-Store) (3,0) GPR ISA
- ALU operace jsou typu registr-registr a registr-přímý operand (**16-bitů přímý operand**)
- **Jeden** adresní mód pro instrukce load/store: **bázovaný + 16-bitový offset (base/displacement)**
- Jednoduché PC-relativní podmíněné skoky, **16-bitový offset (displacement)**

Podobně : SPARC, MIPS, HP PA-RISC, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

# DLX - formáty instrukcí

## ◦ 3 formáty instrukcí



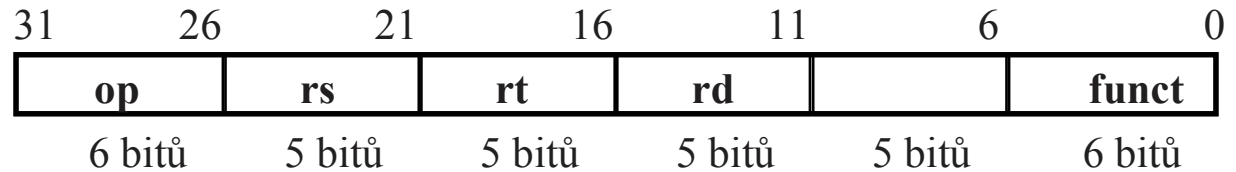
## ◦ Význam polí v kódu instrukce:

- **op**: operační znak (op-code)
- **rs, rt, rd**: čísla zdrojových a cílových registrů
- **funct**: specifikuje operaci u typu R – rozšíření pole “op”
- **imm16 / disp16**: přímý operand / displacement skoku, Load, Store
- **target address displacement**: displacement cílové adresy dlouhého skoku (resp. volání podprogramu)

# DLX – Podmnožina instrukcí pro dnešek

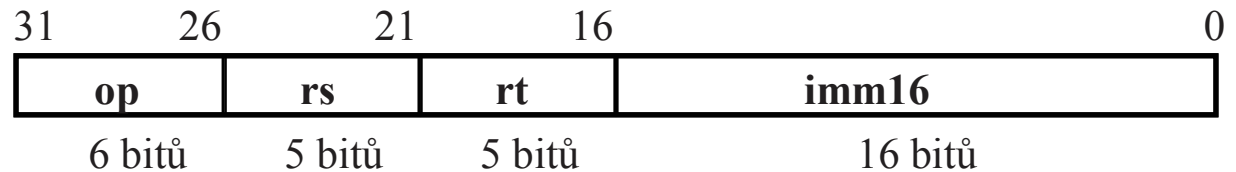
◦ **ADD a SUB**

- **ADD rd, rs, rt**
- **SUB rd, rs, rt**



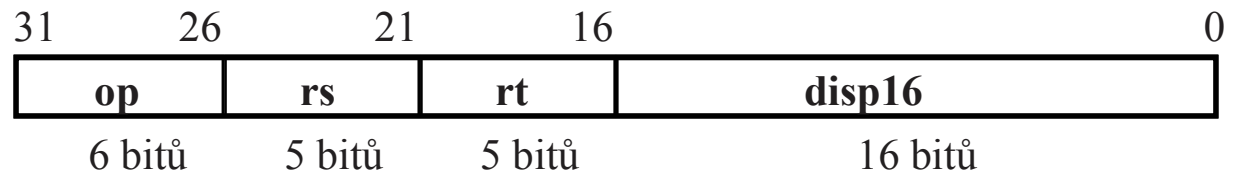
◦ **OR Immediate:**

- **ORI rt, rs, imm16**



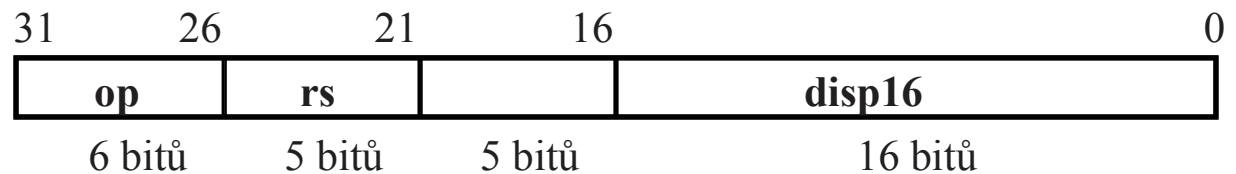
◦ **LOAD a STORE Word**

- **LW rt, disp16(rs)**
- **SW rt, disp16(rs)**



◦ **BRANCH:**

- **BEQZ rs, disp16**
- **BNEZ rs, disp16**



# Popis funkce instrukcí jako mezireg. „přesuny“

## Instr. „Meziregistrové přesuny“ (Register Transfer Level)

**ADD**       $R[rd] \leftarrow R[rs] + R[rt];$        $PC \leftarrow PC + 4$

**SUB**       $R[rd] \leftarrow R[rs] - R[rt];$        $PC \leftarrow PC + 4$

**ORI**       $R[rt] \leftarrow R[rs] \text{ or } ZX(\text{imm16});$        $PC \leftarrow PC + 4$

**LW**       $R[rt] \leftarrow \text{MEM}[ R[rs] + SX(\text{disp16})];$        $PC \leftarrow PC + 4$

**SW**       $\text{MEM}[ R[rs] + SX(\text{disp16}) ] \leftarrow R[rt];$        $PC \leftarrow PC + 4$

**BEQZ**      if (  $R[rs] == 0$  ) then  $PC \leftarrow PC + 4 + SX(\text{disp16})]$   
                    else  $PC \leftarrow PC + 4$

**BNEZ**      if (  $R[rs] != 0$  ) then  $PC \leftarrow PC + 4 + SX(\text{disp16})]$   
                    else  $PC \leftarrow PC + 4$

**SX** ... sign extended, **ZX** ... zero extended



# Komponenty datové části (opakování z X36JPO)

## ° **Kombinační logika :**

- hradla, multiplexery, třístavové budiče, ...
- paměti ROM

## ° **Sekvenční logika :**

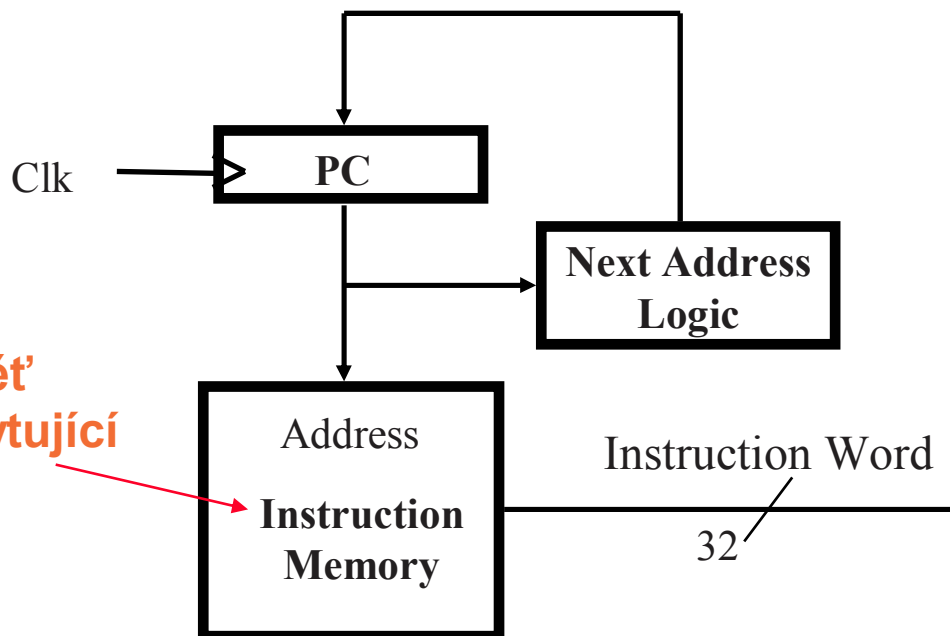
- registry (používáme pouze hranově řízené DFF)
- víceportová registrová pole
- zapisovatelné synchronní paměti

**Všimněte si, že registrové pole (paměť) se chová jako kombinační logika z hlediska čtení ale je to sekvenční logika pro zápis !**

# Datová cesta pro načtení instrukce (Instr.Fetch)

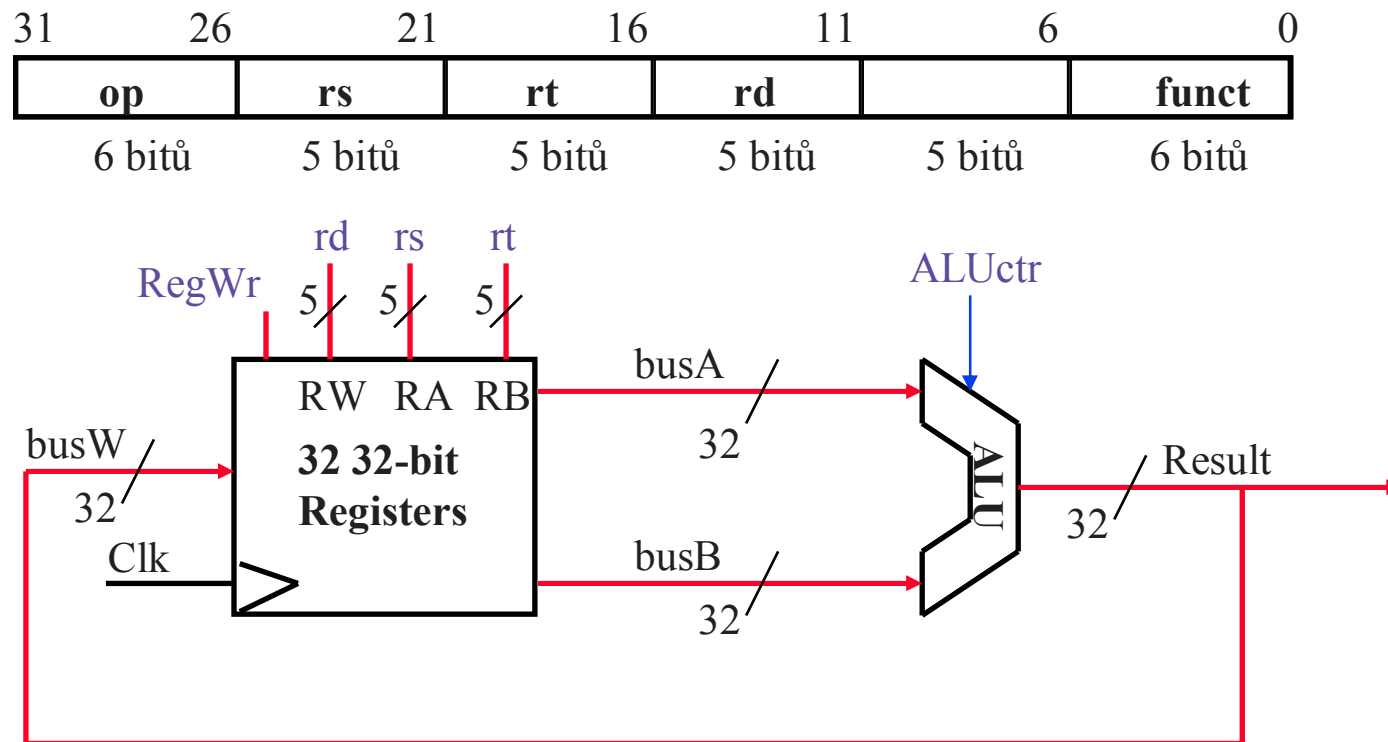
- Společná pro všechny instrukce
  - Čtení instrukce:  $\text{mem}[\text{PC}]$
  - Logika adresy následující instrukce :
    - **Implicitně:**  $\text{PC} \leftarrow \text{PC} + 4$
    - **Podmíněný skok, volání podprogramu:**  $\text{PC} \leftarrow \text{target address.}$

Ideální instrukční paměť  
(Perfektní Cache poskytující  
instrukci každý takt)



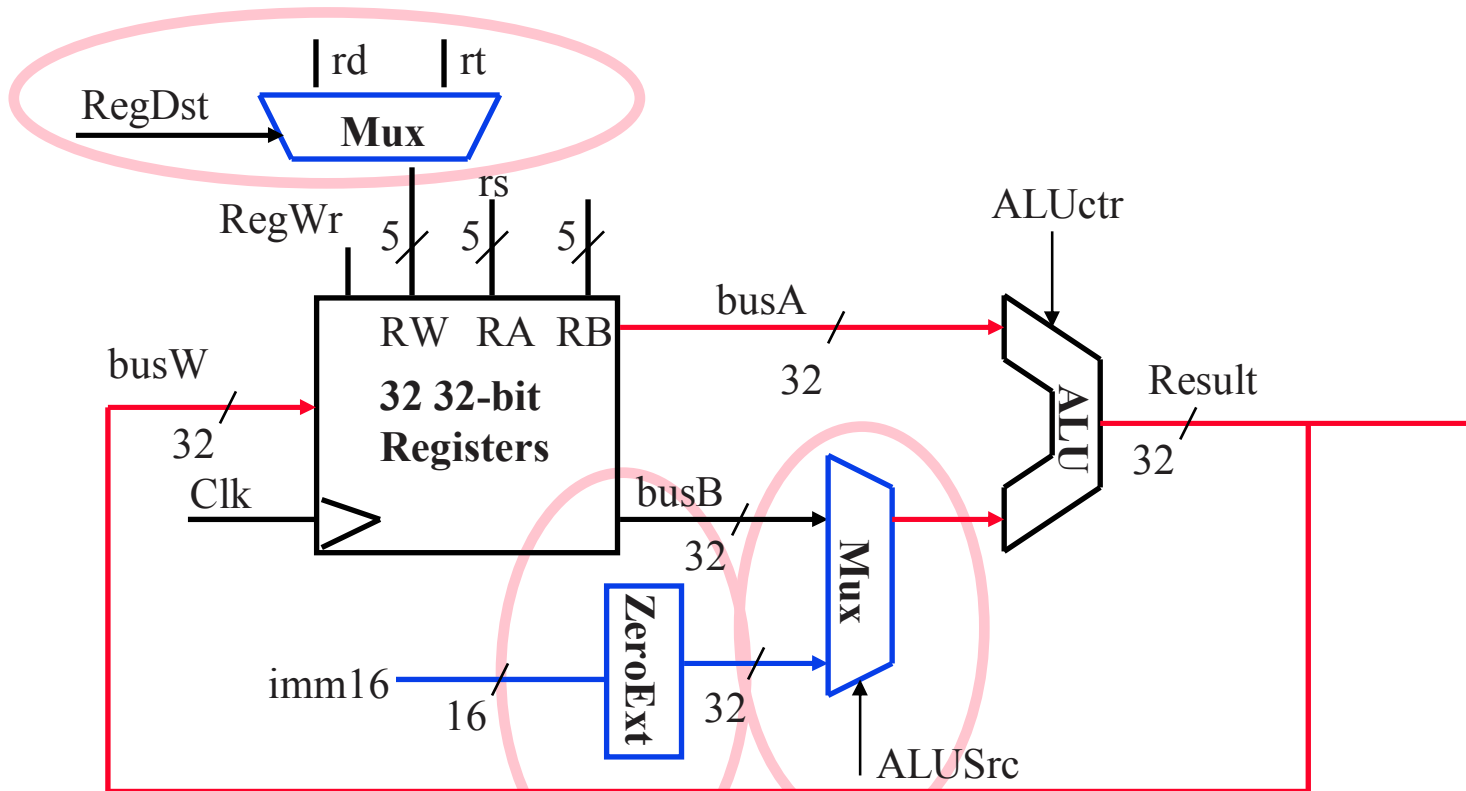
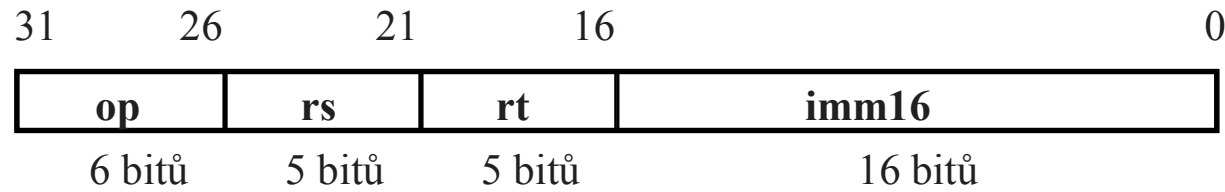
# Datová část pro registr-registr ALU instrukci

- $R[rd] \leftarrow R[rs] \text{ op } R[rt]$       Př: **ADD**     $rd, rs, rt$ 
  - **RA, RB, a RW** – připojeny k číslům registrů **rd, rs, rt**
  - **ALUctr a RegWr**: řídicí signály (dekódovány z polí **op** a **funct**)



# ALU instrukce s přímým operandem

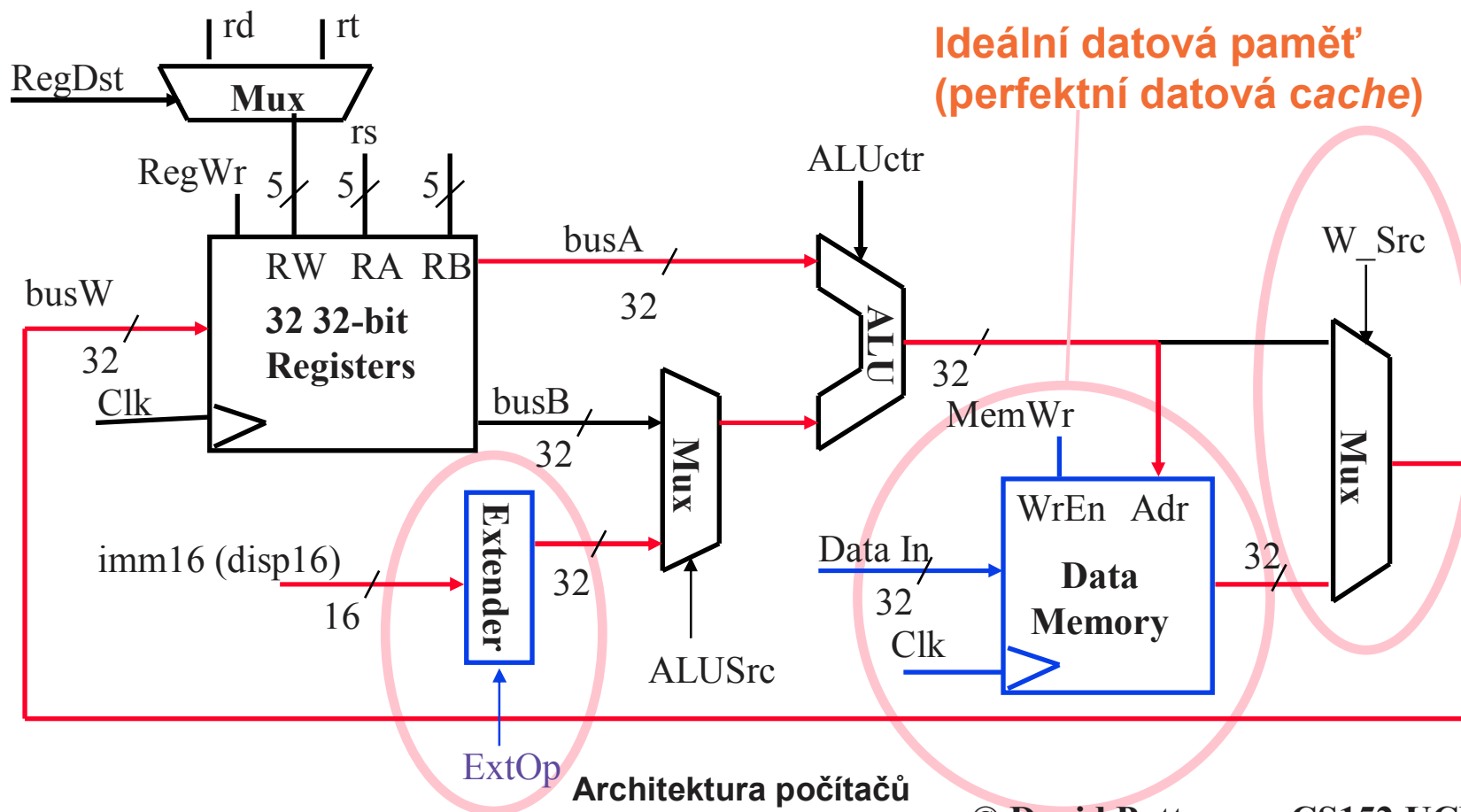
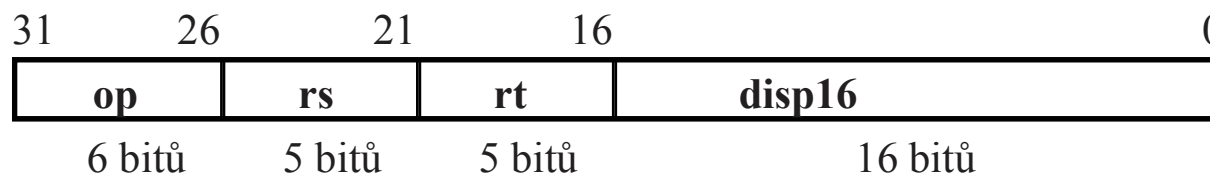
◦  $R[rt] \leftarrow R[rs] \text{ op } ZX[imm16]$



Architektura počítačů

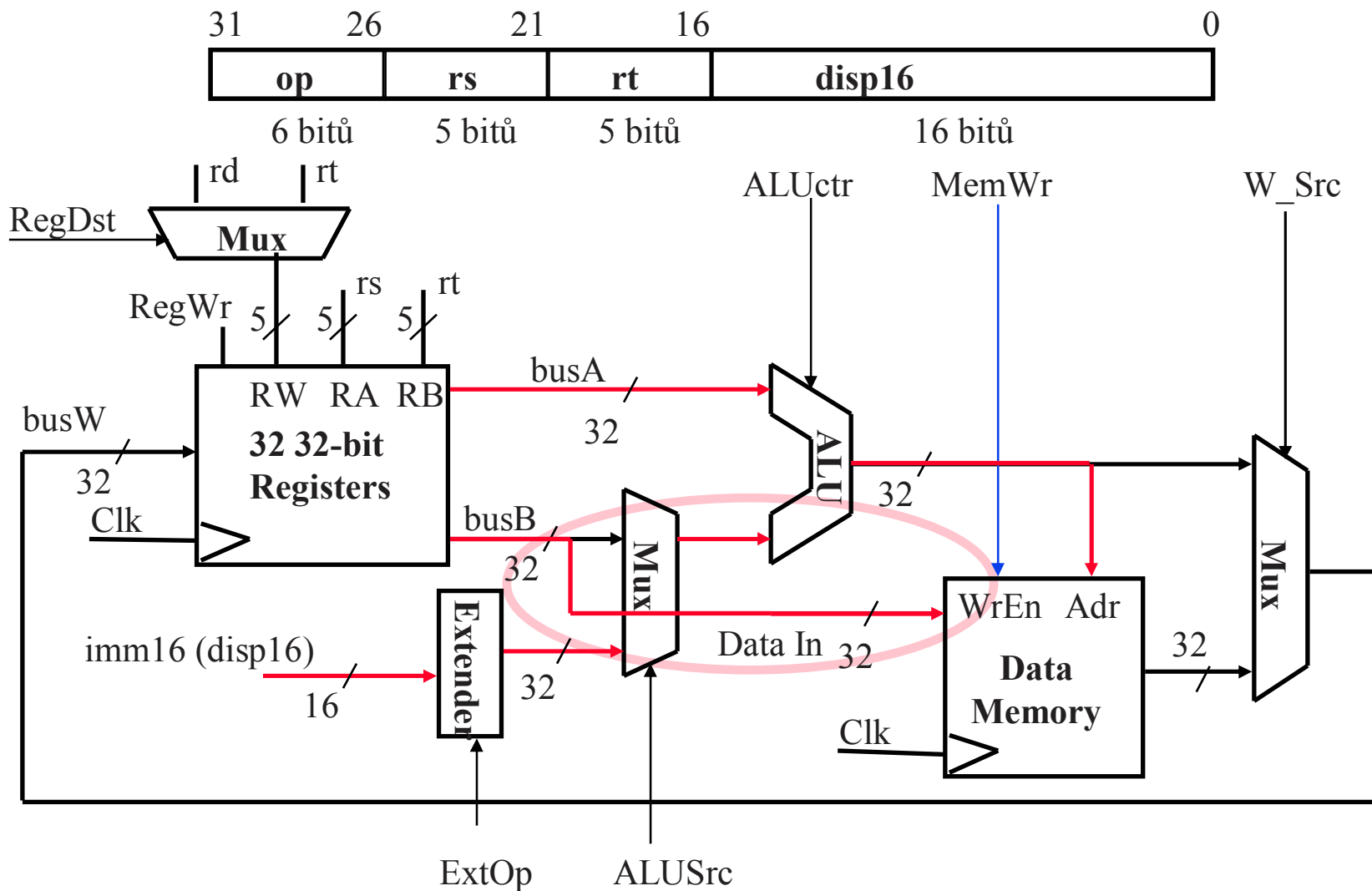
# Instrukce „Load“

◦  $R[\underline{rt}] \leftarrow \text{Mem}[ R[\underline{rs}] + \text{SX}[\text{disp16}] ]$       Př.: LW     $rt, \text{disp16} (rs)$



# Instrukce „Store“

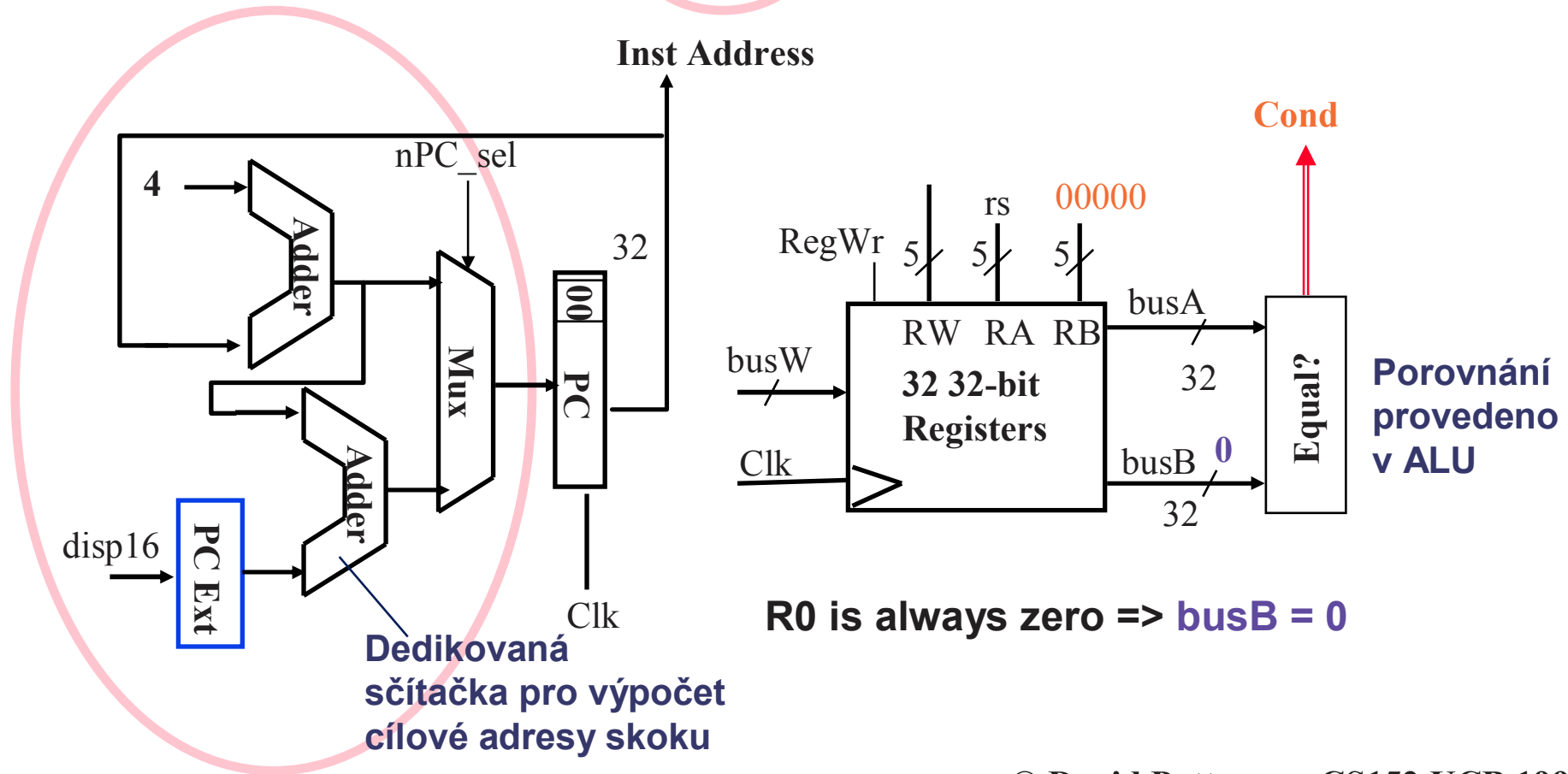
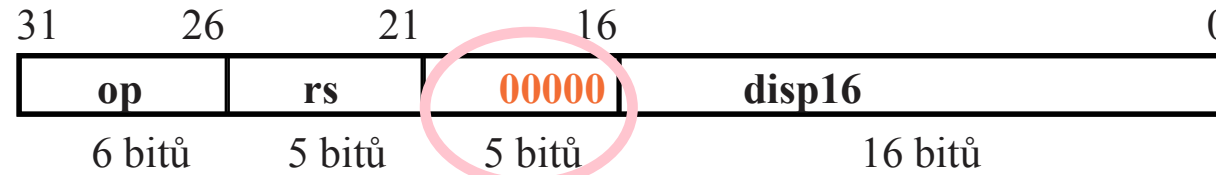
◦  $\text{Mem}[R[\text{rs}] + \text{SX}[\text{disp16}]] \leftarrow R[\text{rt}]$       Př.: SW    rt, disp16 (rs)





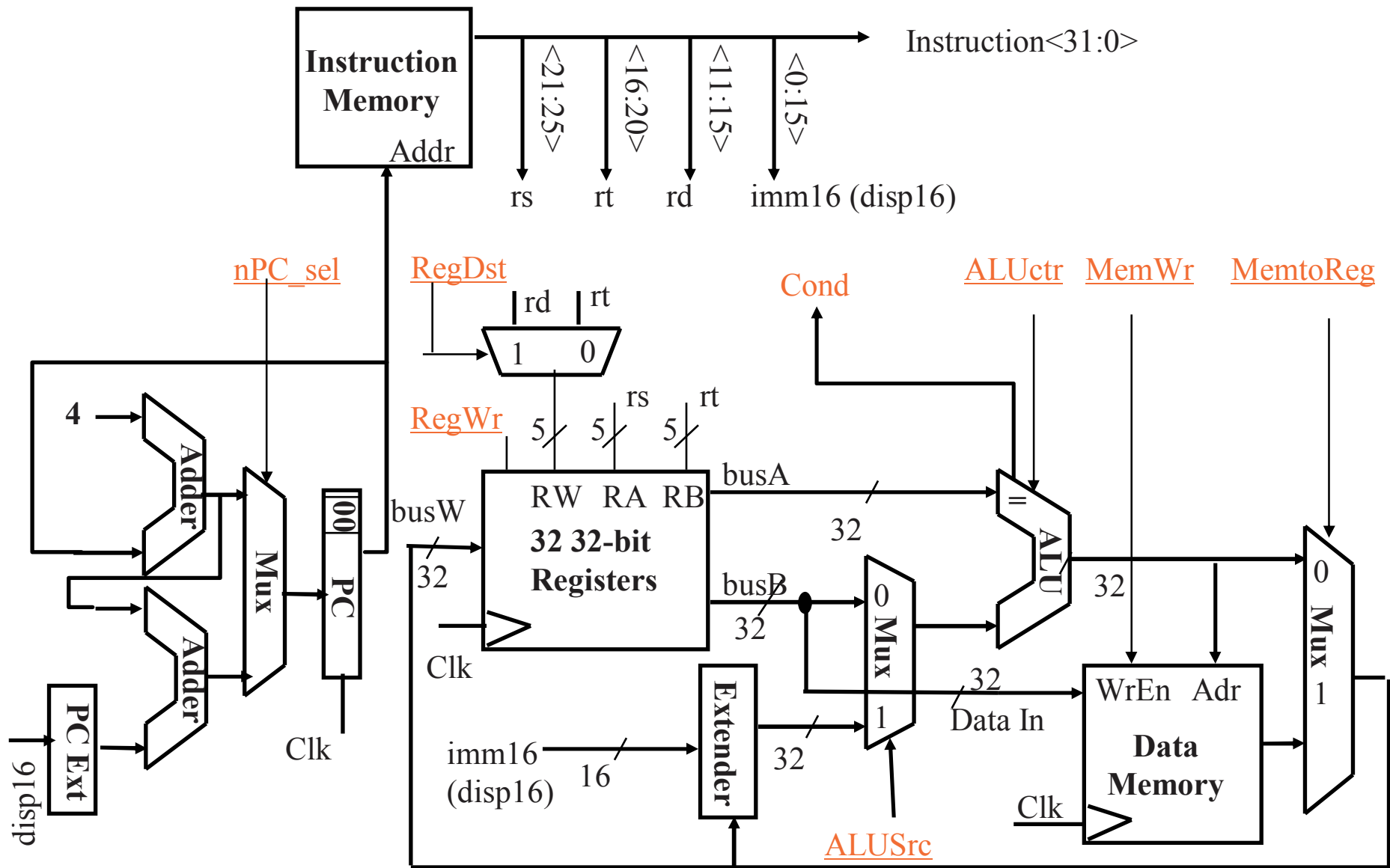
# Datová část pro instrukci podmíněného skoku

◦ **BEQZ** rs, disp16

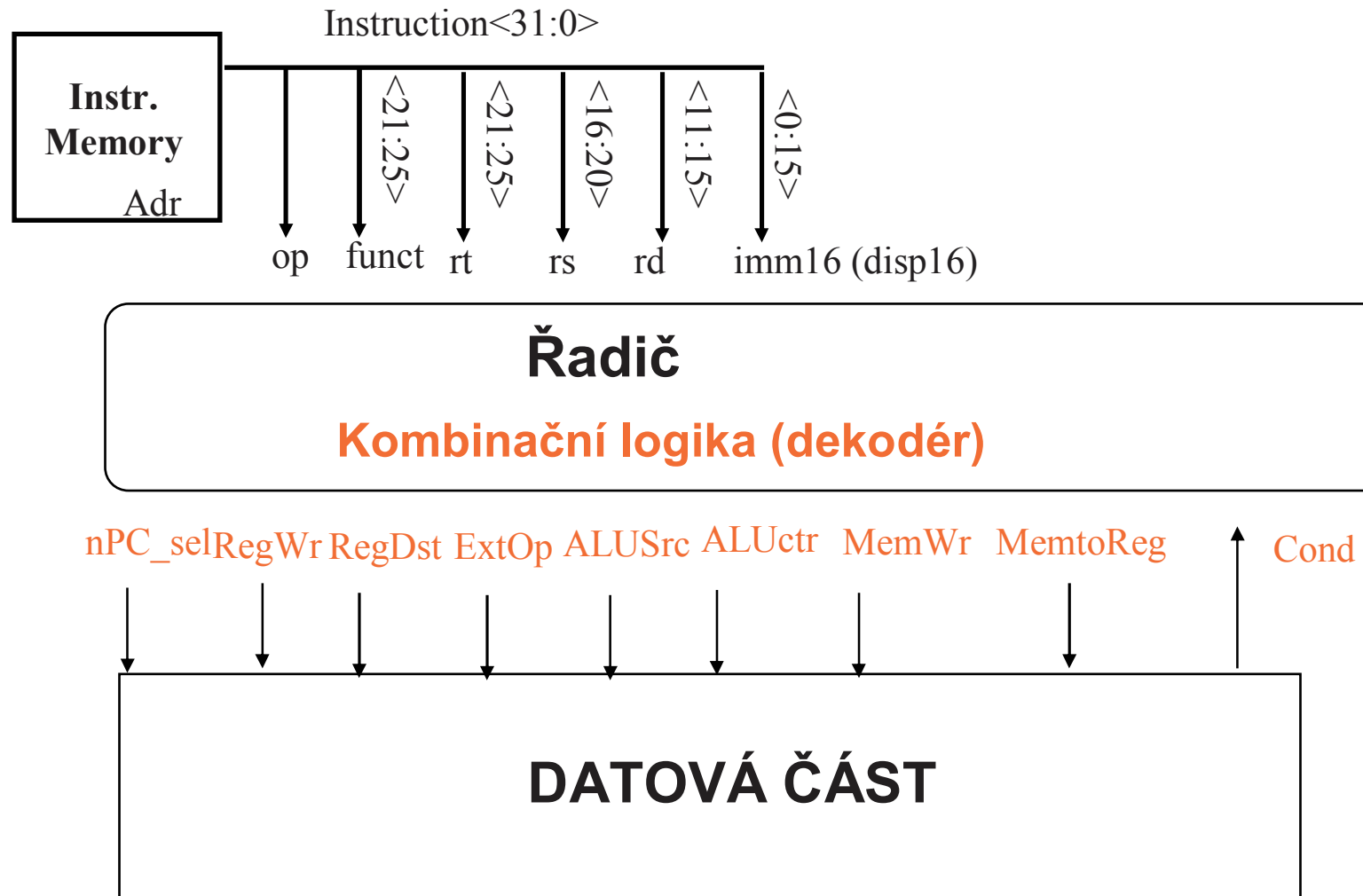




# Vše pohromadě – jednoduchá datová část DLX



# Řadič pro jednotaktový procesor

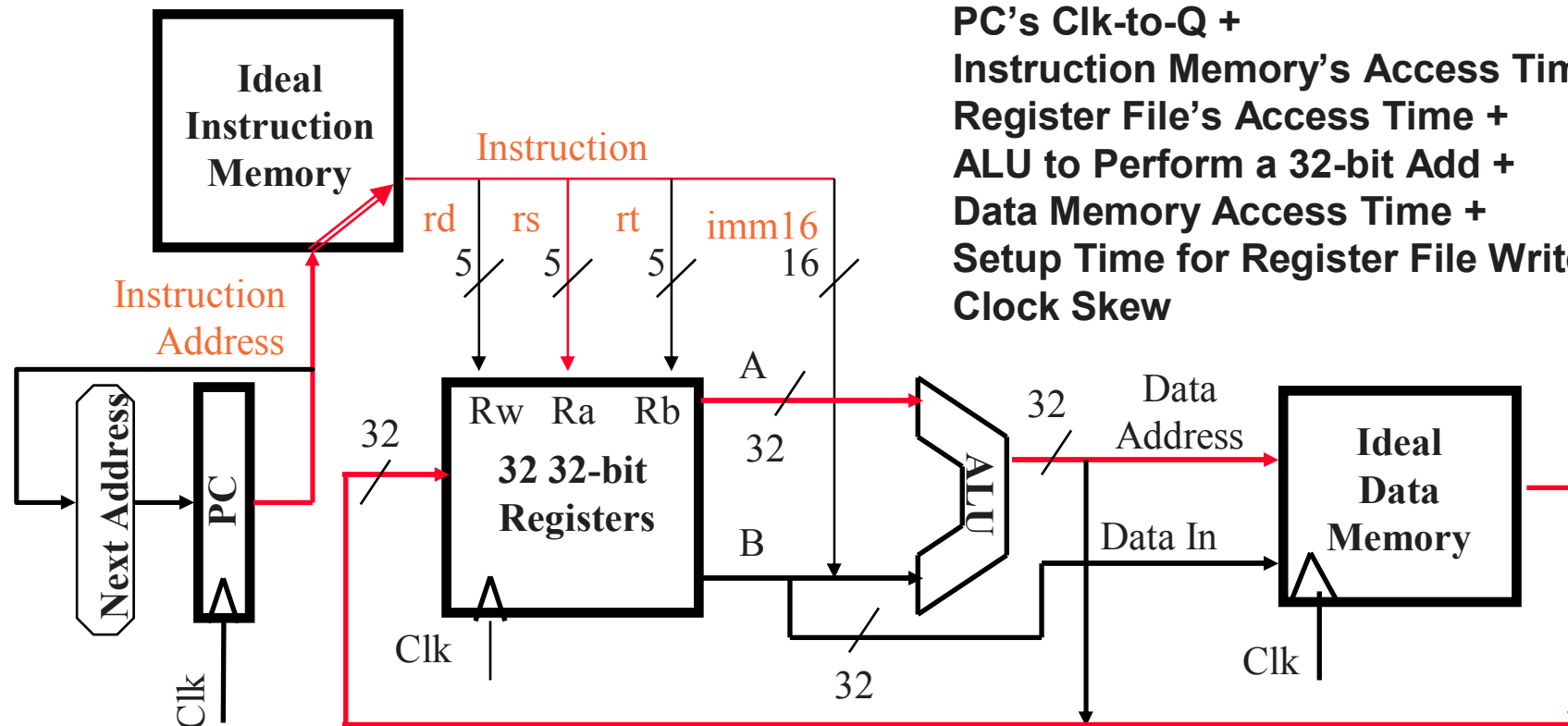


# Charakteristika jednotaktového procesoru

- Jednoduchá implementace
- **CPI=1**, ale  $T_{clk}$  je velmi dlouhá !!
- $T_{clk}$  záleží na **nejdelší cestě v kombinační logice** – instrukce LOAD

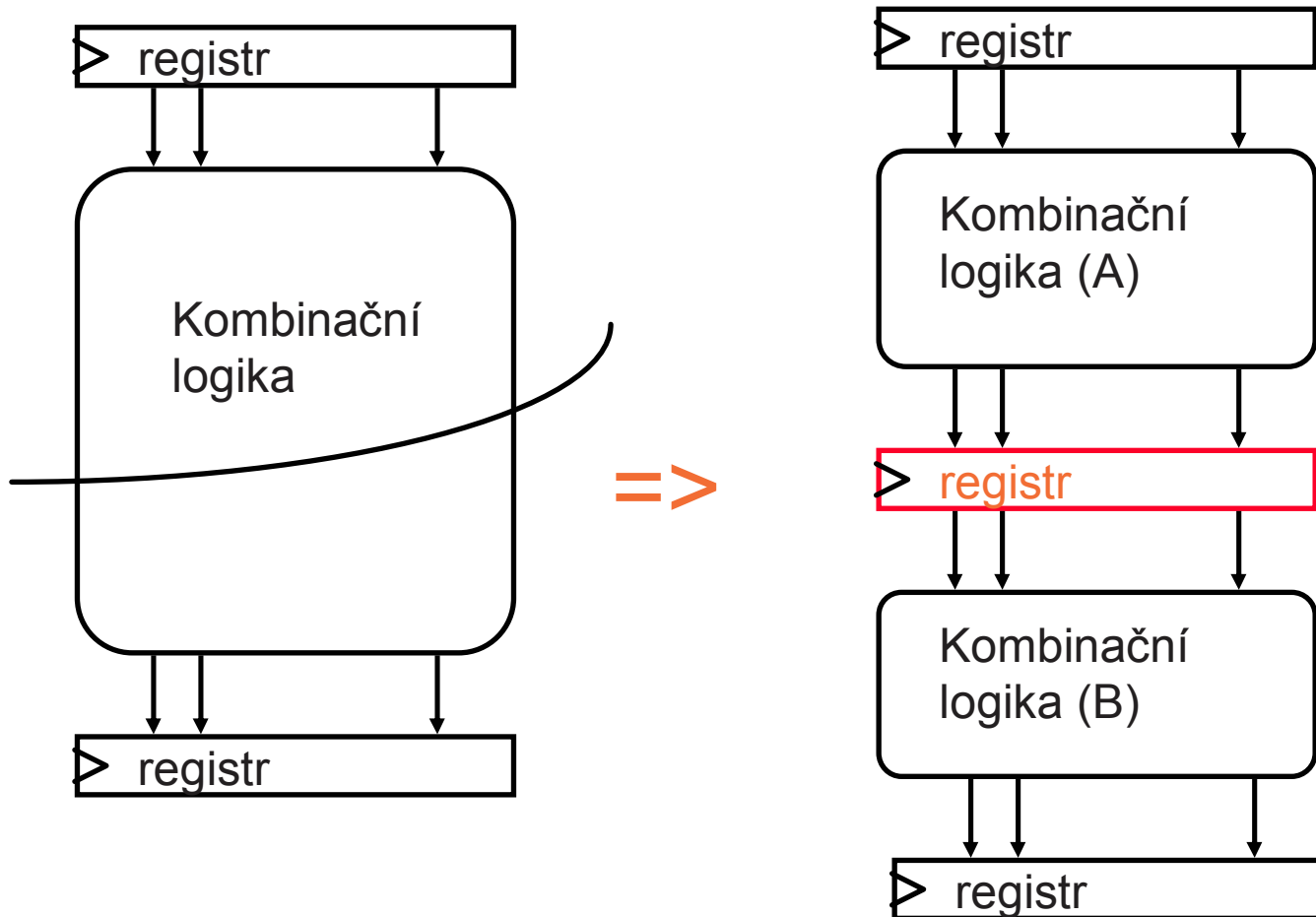
**Kritická cesta (instrukce Load) =**

PC's Clk-to-Q +  
Instruction Memory's Access Time +  
Register File's Access Time +  
ALU to Perform a 32-bit Add +  
Data Memory Access Time +  
Setup Time for Register File Write +  
Clock Skew



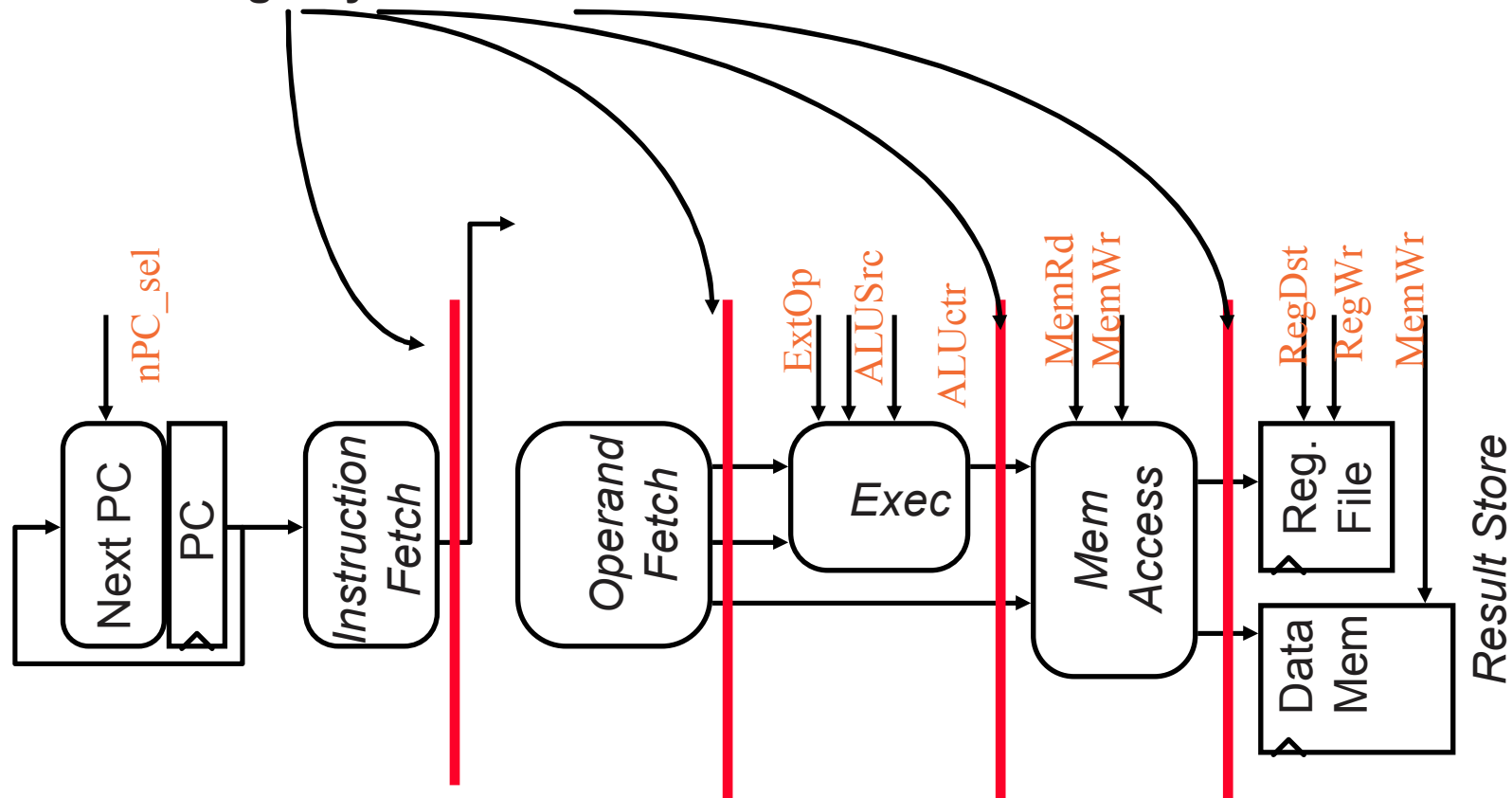
# Zkrácení $T_{clk}$ vložení dodatečných registrů

- Obecné pravidlo návrhu:



# Vložení registrů do datové části

- Přidané registry



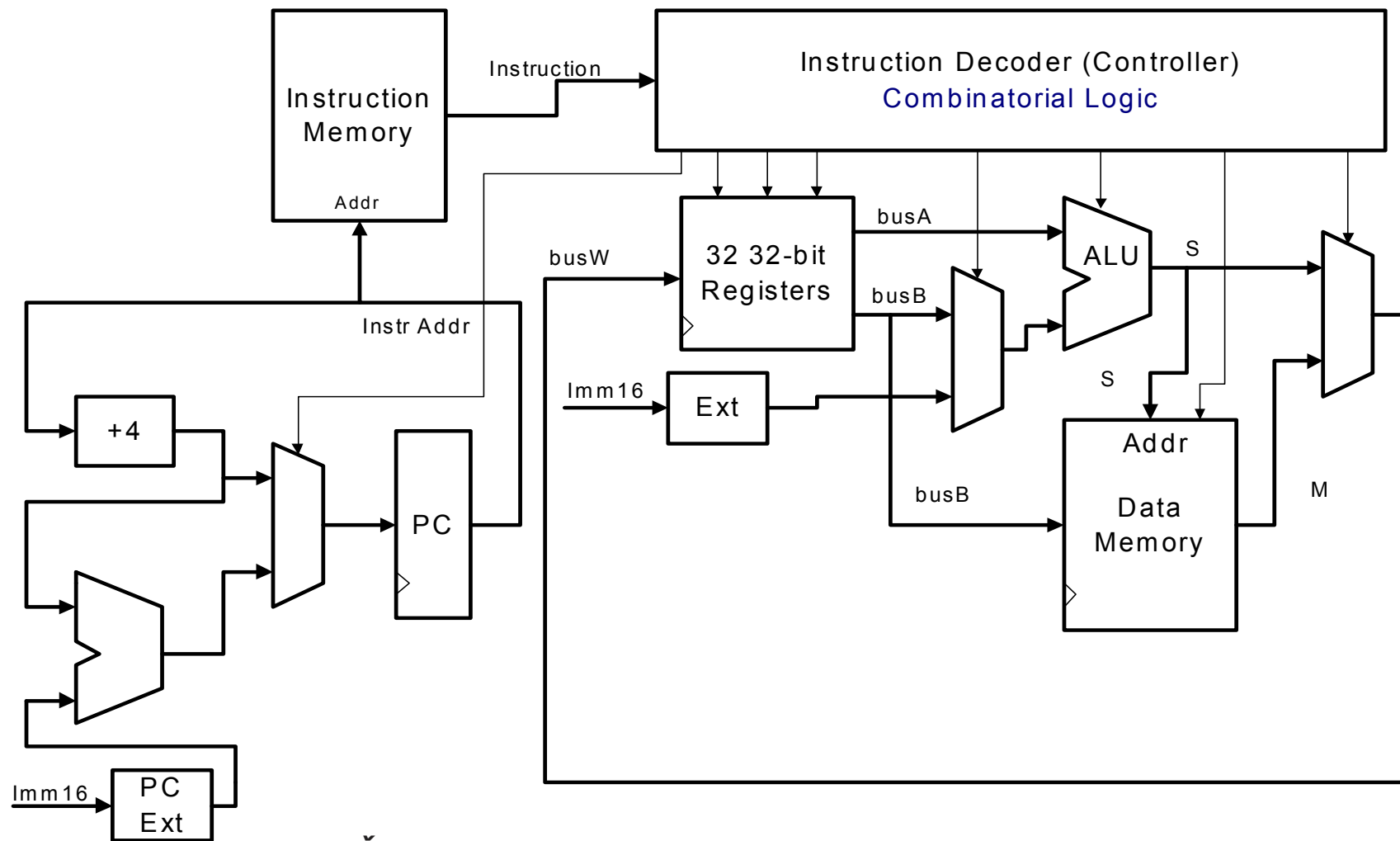
Instrukční  
Registr

Registry  
Operandů

Výsledek ALU  
Registr adresy

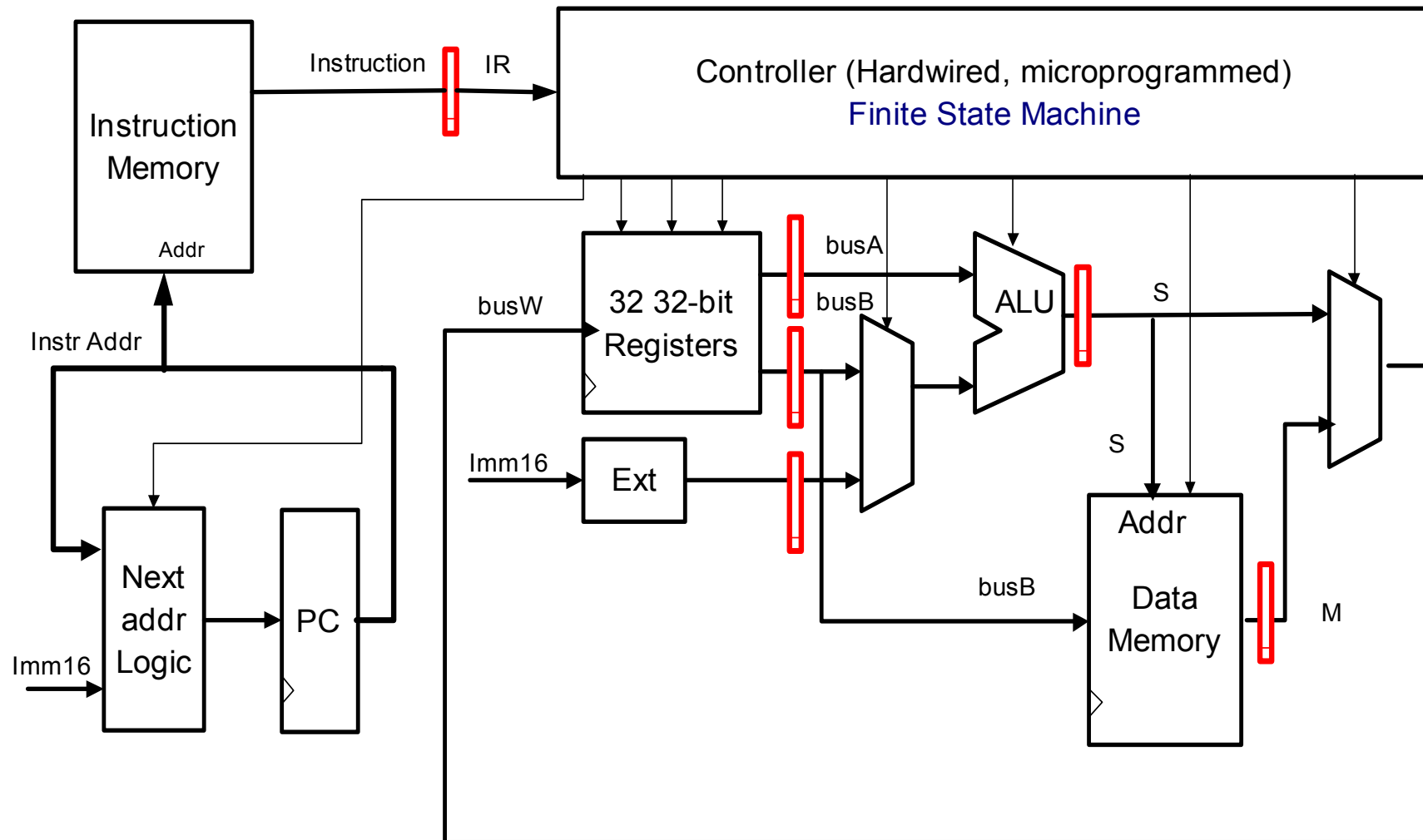
Registr  
dat z paměti


# Překreslení jednotaktového CPU



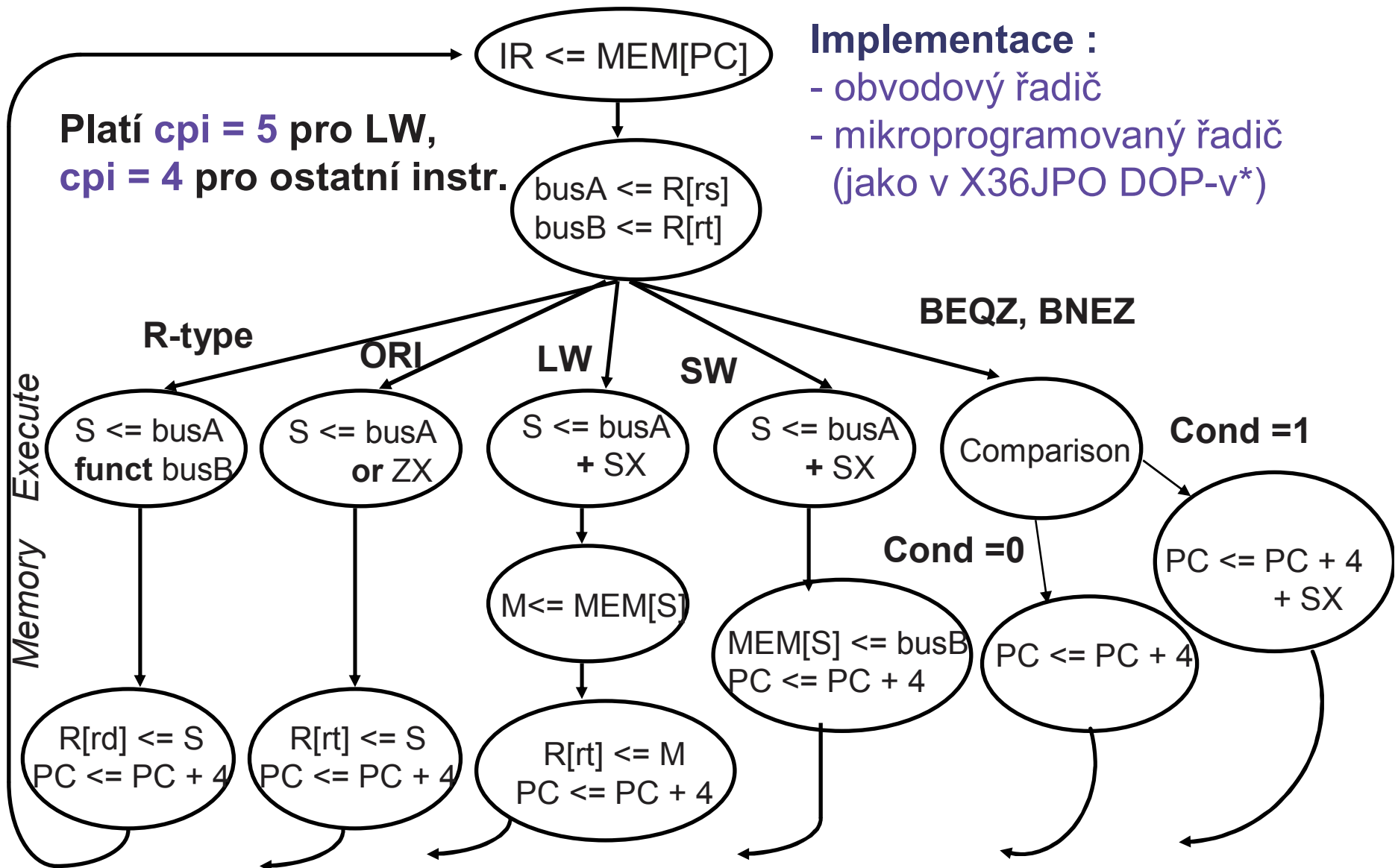
*(Řídící signály vynechány pro přehlednost; imm16 = disp16)*

# Vícetaktová datová část a řadič



 Added Registers

# Řadič vícetaktového CPU jako konečný automat





# Charakteristika vícetaktového CPU

- $T_{clk} = 1/5$  (hodně optimistické)
- cpi: Load instrukce = 5 taktů  
ALU instr., skoky, store = 4 takty,
- Load instrukce tvoří typicky 26 % instrukcí  
(SPECInt1996 průměr na DLX)
- $CPI = 4 * 0.74 + 5 * 0.26 = 4.26$
- Zrychlení oproti jednotaktovému CPU =  $5/4.26 = 1.17$
- Vícetaktový procesor je univerzálnější, umožňuje implementovat i komplexní instrukce např. pomocí mikroprogramování.

## Využití komponent datové části

**Instruction Memory** – použit v taktu Instruction Fetch

**Register File** – použito v taktu Operand Fetch a Result Store

**ALU** – použito v taktu Execute (také pro výpočet adresy L/S )

**Data Memory** – použito v taktu Memory Access

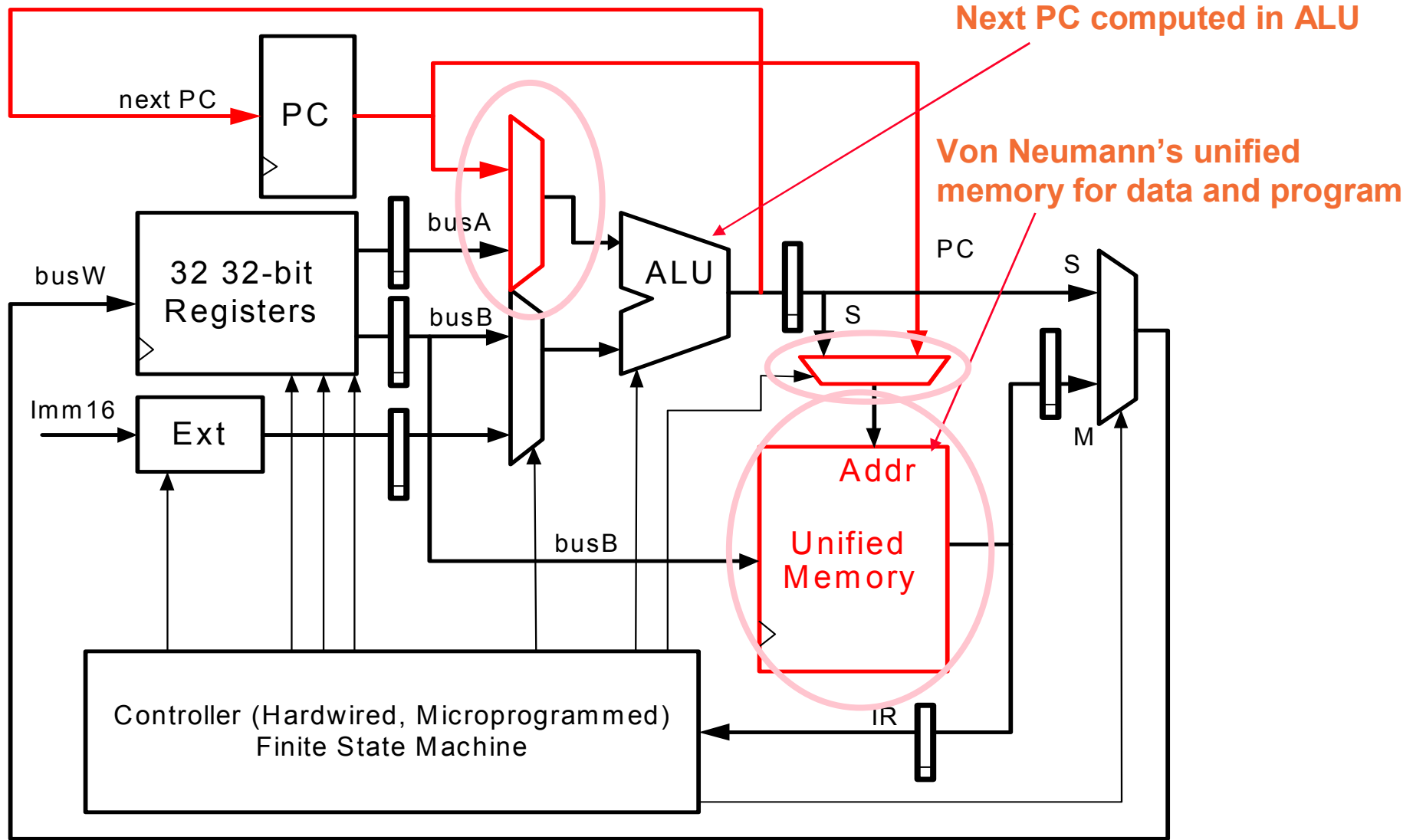
**Next PC logic** – použito v taktu Result Store nebo  
v taktu Memory Access (instrukce store)

Průměrné využití komponenty je v jednom taktu  
na instrukci  $(1/4.26) = 23 \%$

### Optimalizace komponent - sdílení:

- Sloučení paměti instrukcí a dat,
- „Next PC“ může být spočítán v ALU.

# Alternativní datová část – sdílení komponent - stejně výkonná + menší



Architektura počítačů

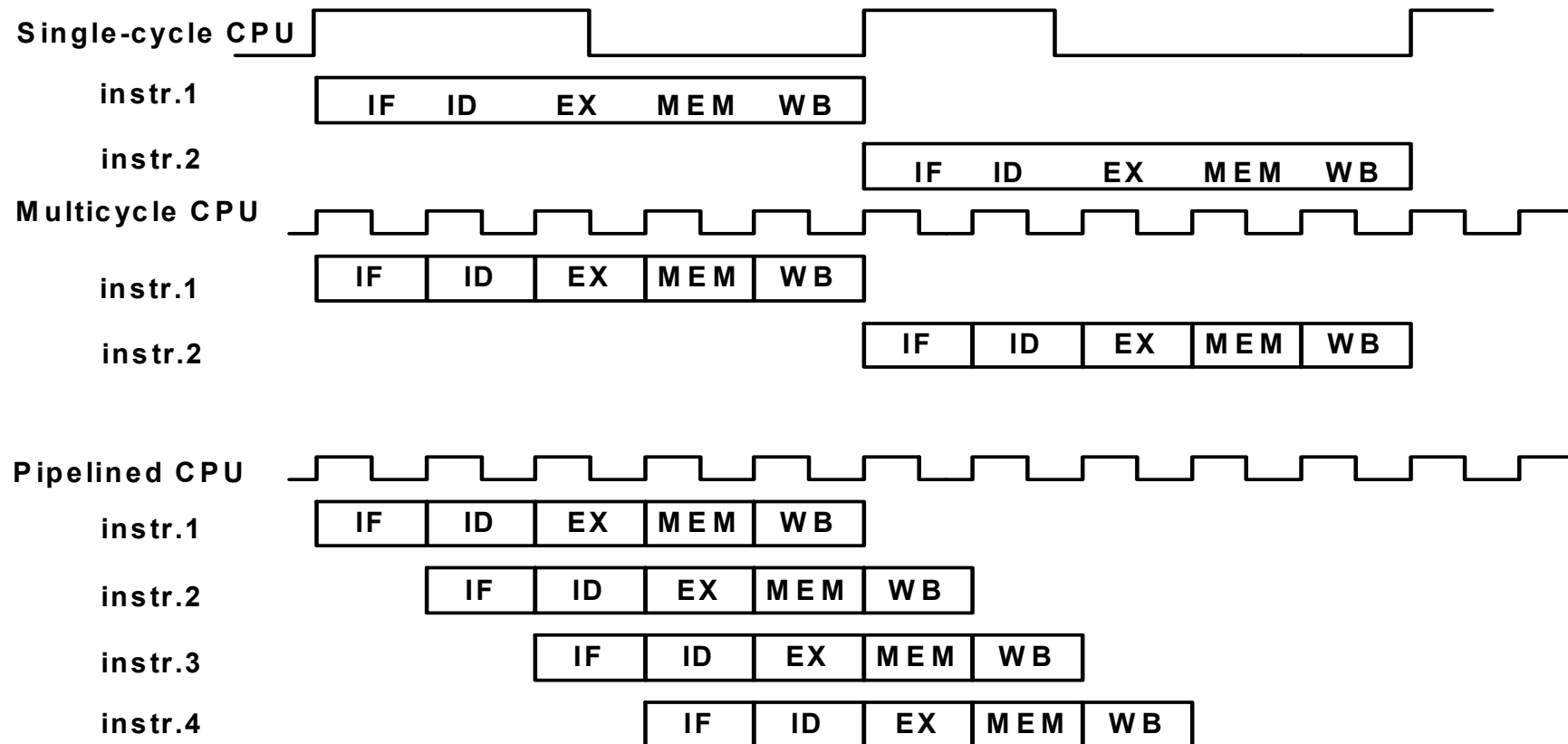
# Shrnutí jednotaktového a vícetaktového CPU

- Dosud nebylo v zásadě *nic nového* – pouze jsme použili postupy známé z X36JPO
- Alternativní datová část vícetaktového DLX je podobná procesorům probíraným v X36JPO
- Všechny varianty CPU jsou *plně sekvenční* – instrukce je dokončena než je následující započata (nazýváme je “*konvenční CPU architektury*”)

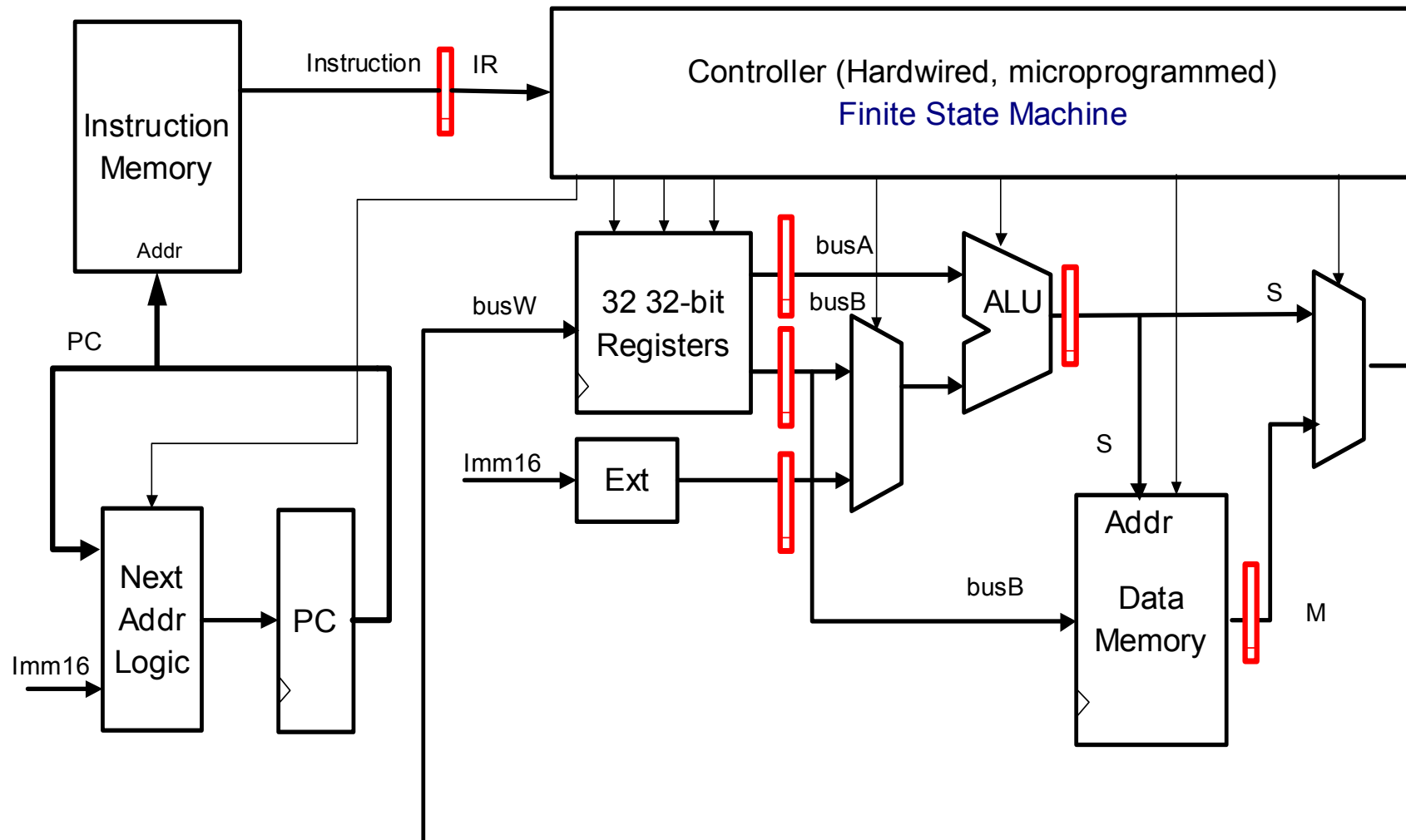
Abychom dosáhli lepší výkonnosti DLX musíme použít techniky *proudového zpracování (pipelining)*


# Proudové zpracování instrukcí - motivace

- Komponenty datové části nebyly plně využity ve vícetaktovém CPU.
- Pokusme se je použít *v každém taktu* místo jejich optimalizace sdílením

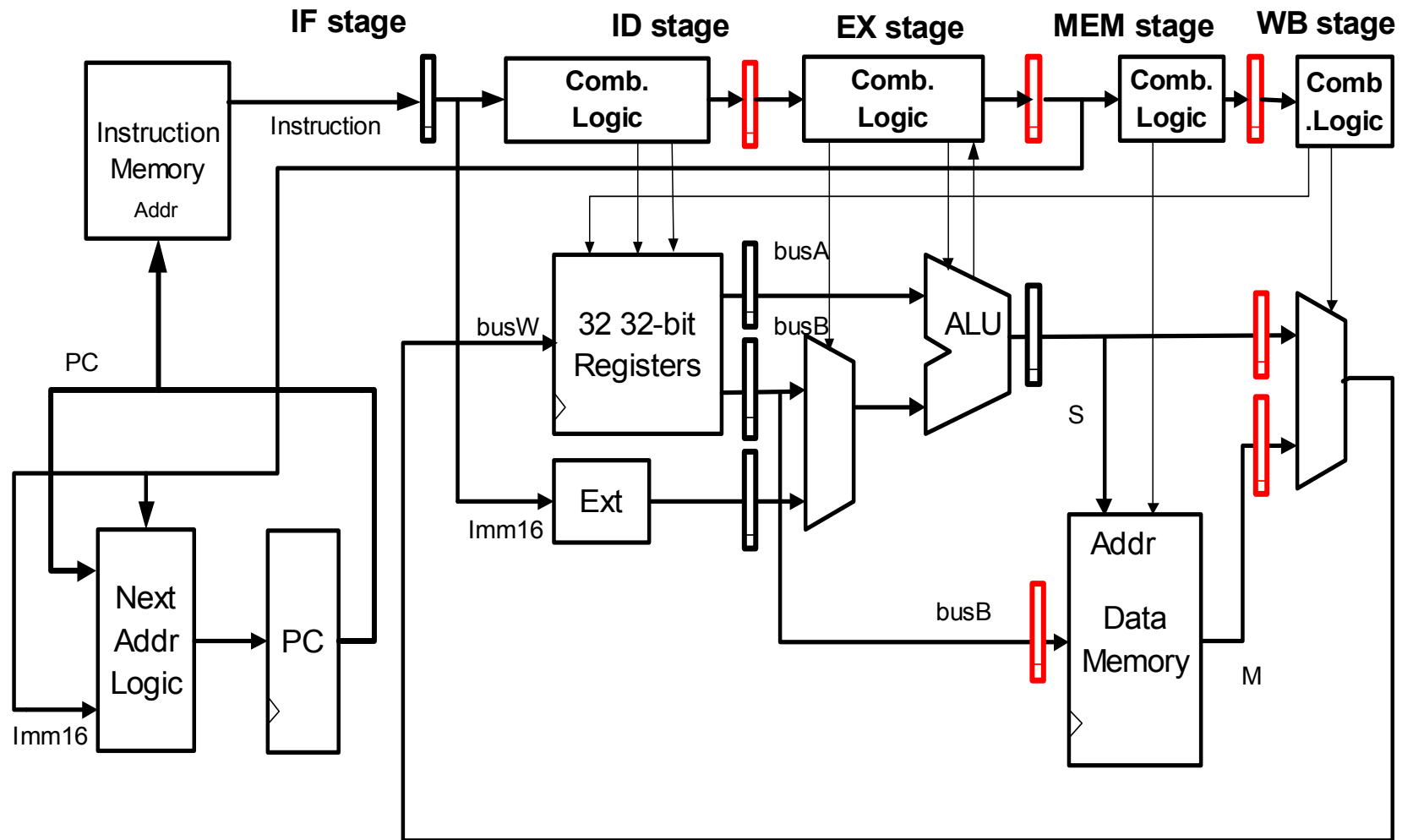


# Vícetaktová datová část s řadičem



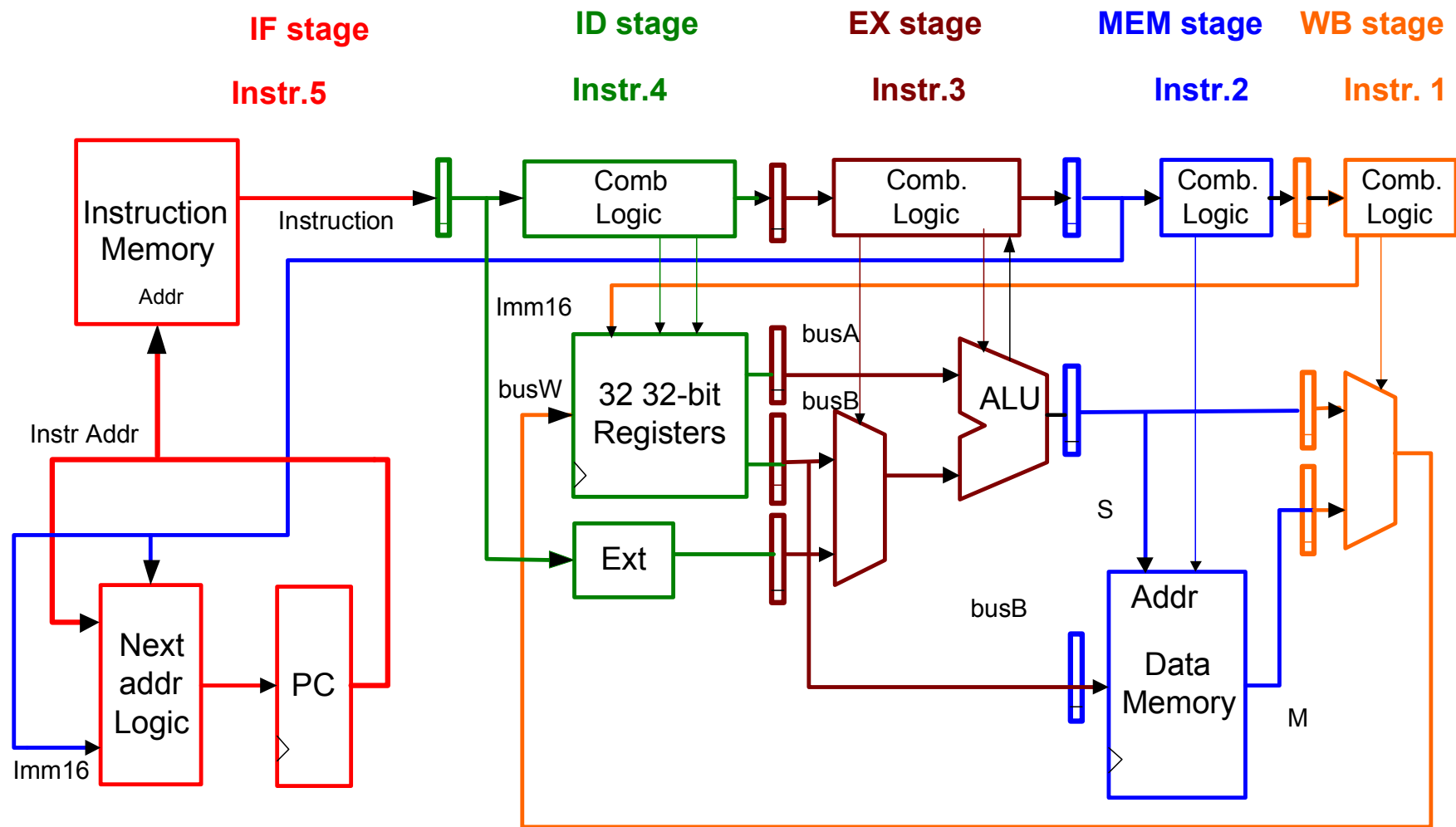
 Added Registers

# Proudově pracující datová část a řadič



Potenciál zrychlení = počet stupňů pipeline.

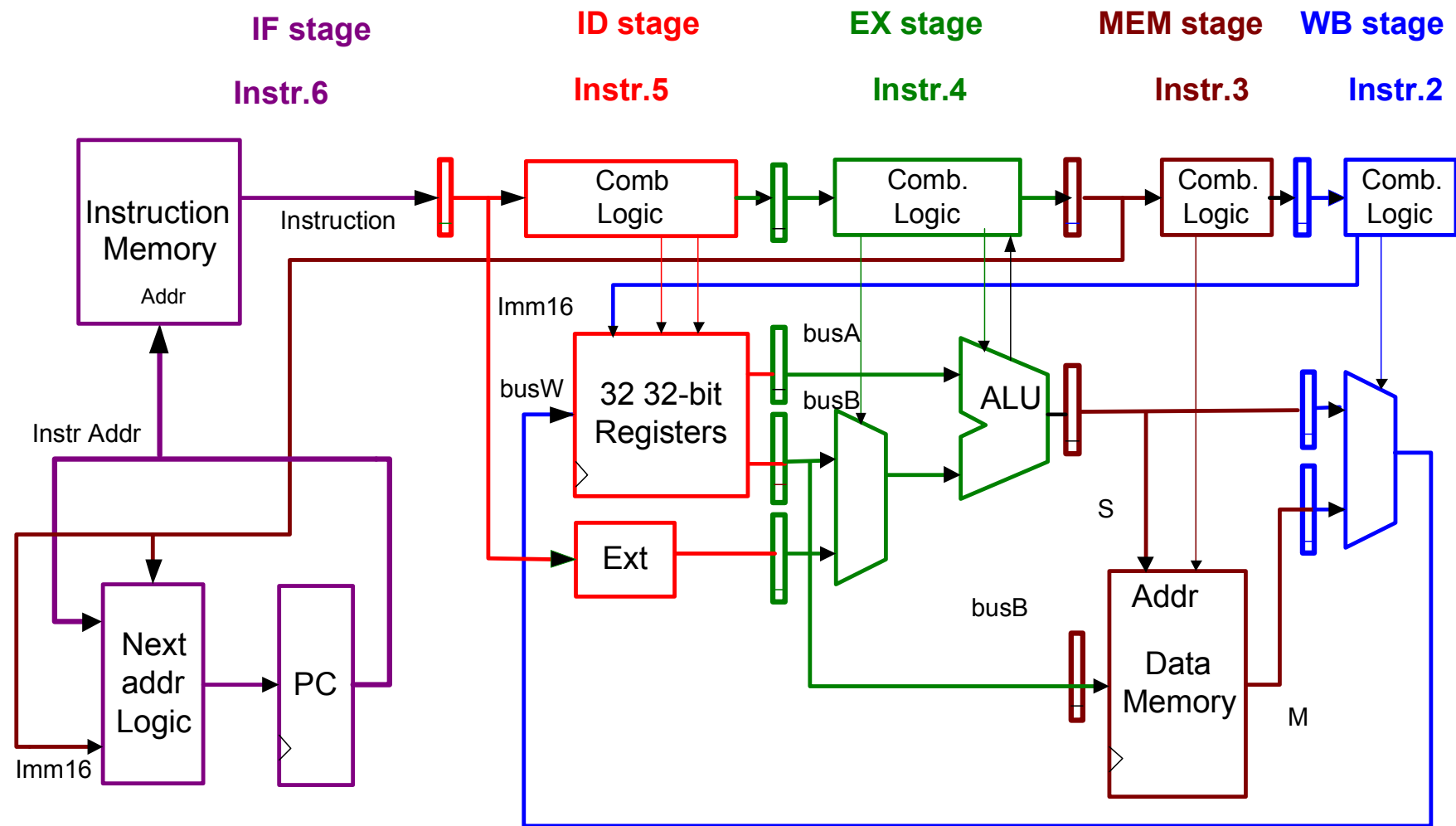
# Proudově pracující datová část a řadič 5.takt



Potenciál zrychlení = počet stupňů pipeline.

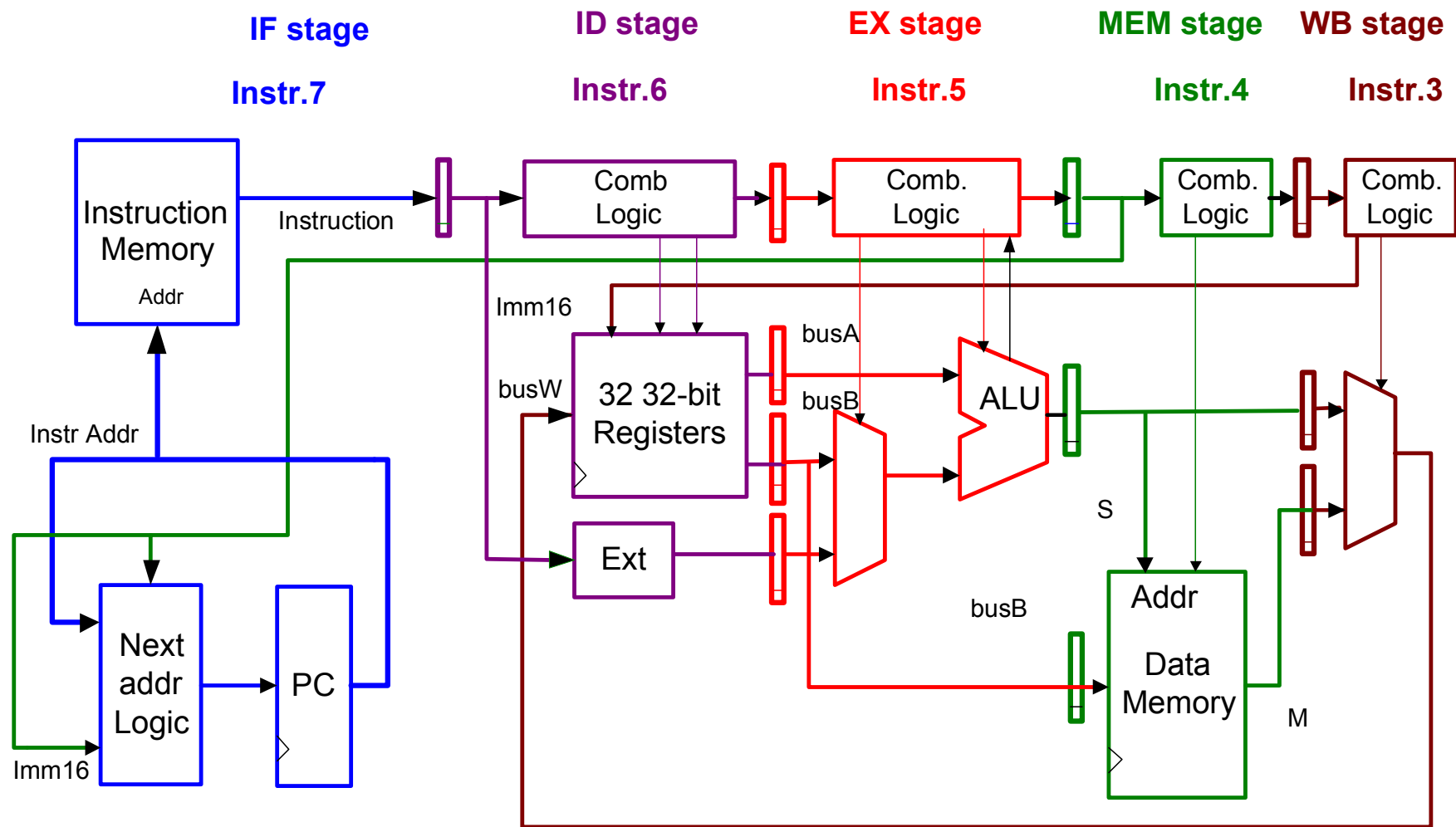


# Proudově pracující datová část a řadič **6.takt**



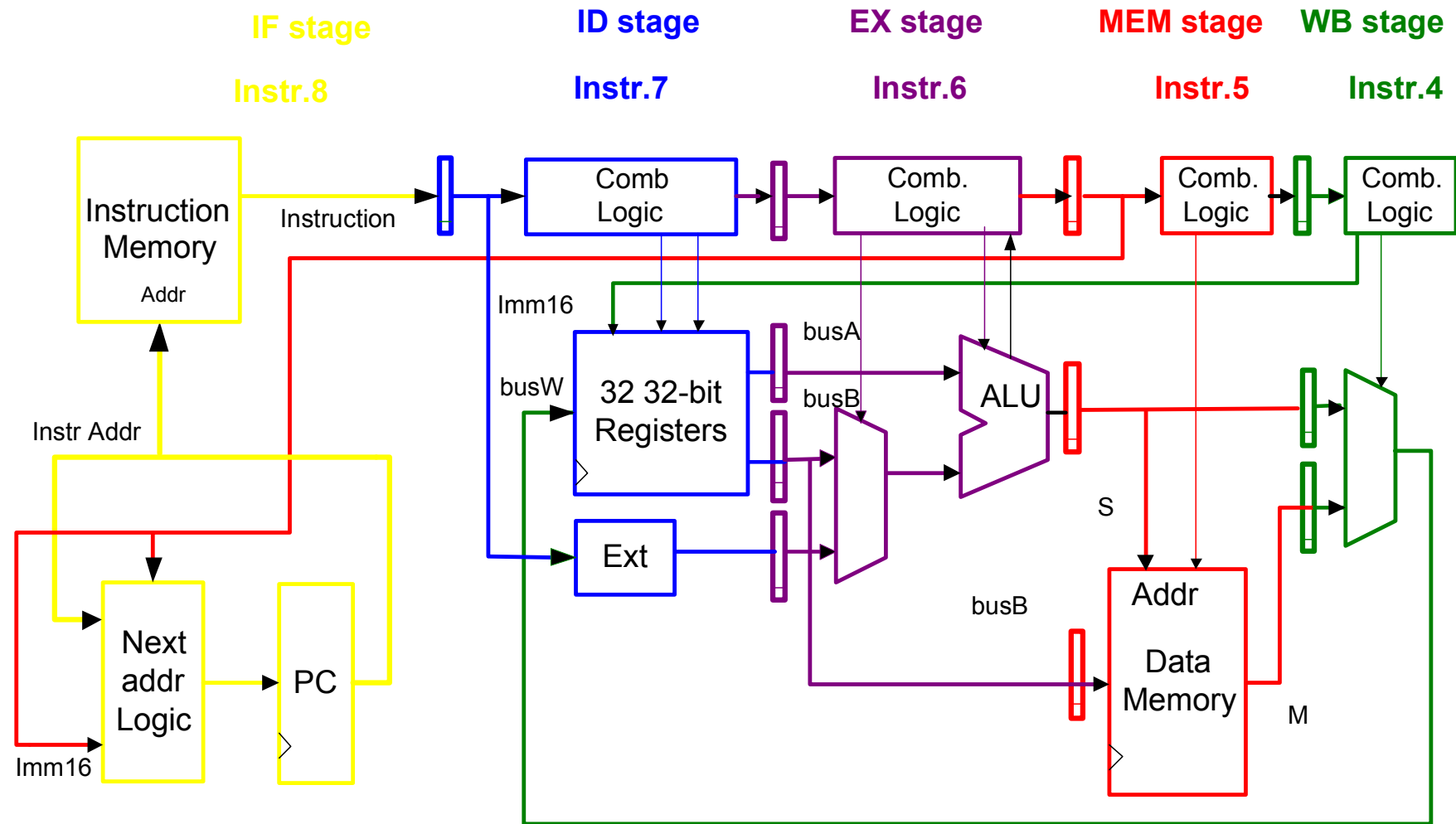
Potenciál zrychlení = počet stupňů pipeline.

# Proudově pracující datová část a řadič 7.takt



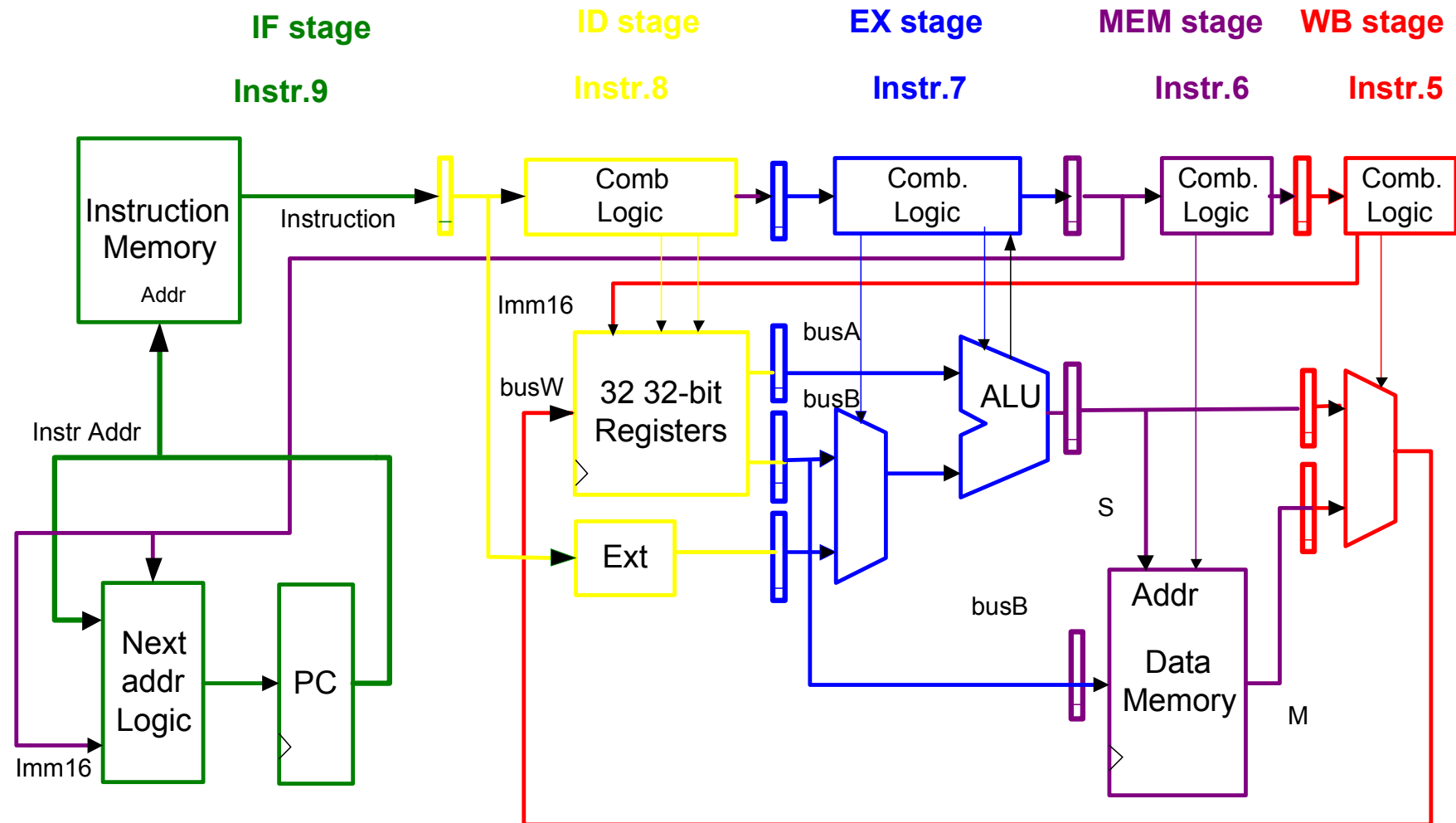
Potenciál zrychlení = počet stupňů pipeline.

# Proudově pracující datová část a řadič 8.takt



Potenciál zrychlení = počet stupňů pipeline.

# Proudově pracující datová část a řadič 9.takt



Potenciál zrychlení = počet stupňů pipeline.

# Řadič proudově pracujícího DLX

- Distribuován mezi stupni *pipeline* (tzv. *data stationary control*)  
– Instrukce prochází stupni zpracování společně se svými daty
- Nějaký klasický sekvenční řadič je potřeba pro korektní obsluhu výjimek (přerušeni), datových hazardů a dalších funkcí.  
(pro jednoduchost není znázorněn).

Řadič proudově pracujícího DLX je relativně jednoduchý neboť DLX je RISC procesor

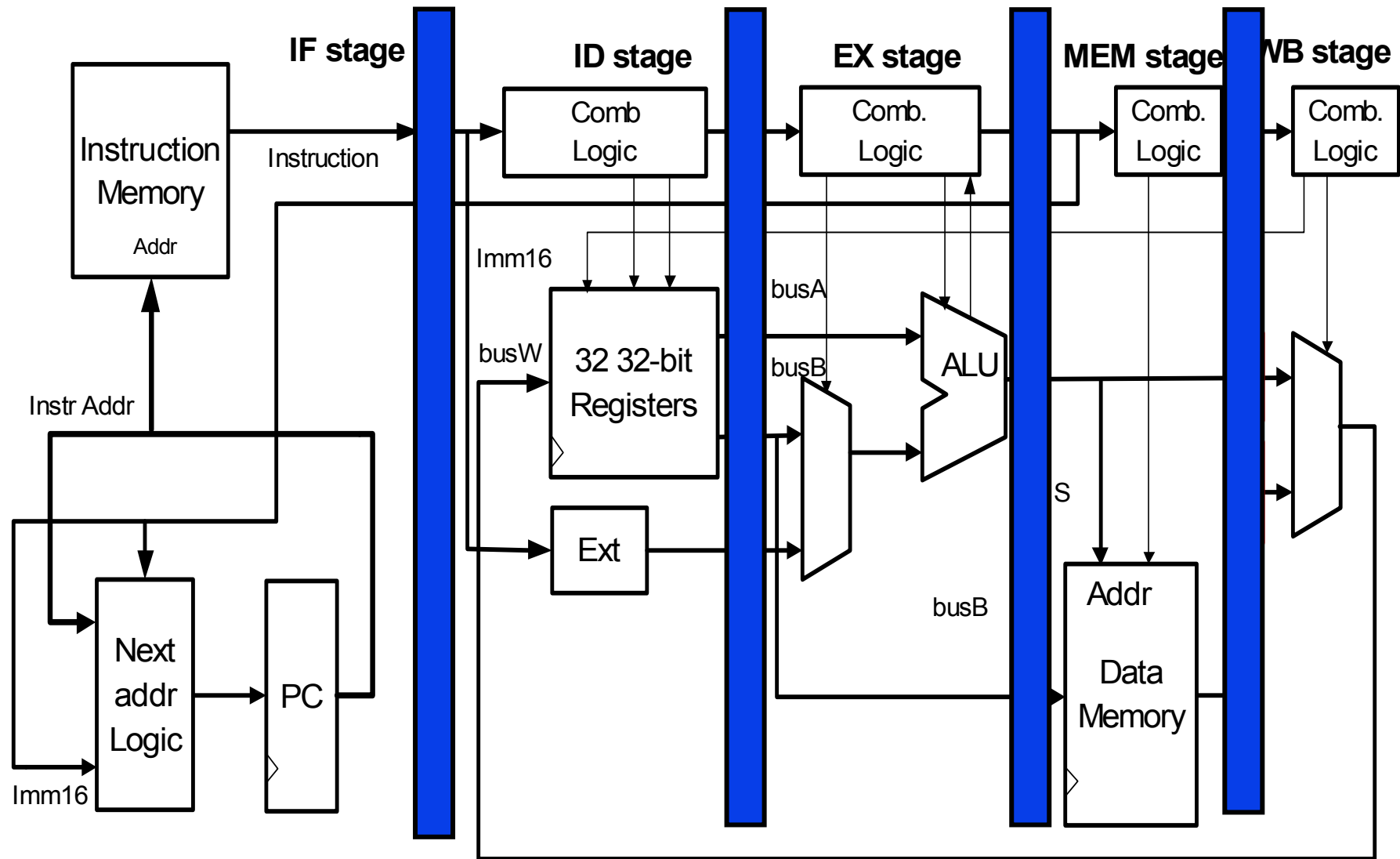
- Všechny instrukce trvají stejný nebo podobný počet taktů
- Jednoduché kódování instrukcí pevné délky
- Pouze instrukce Load/Store přistupují do paměti
- Procesor podporuje jen zarovnaná data a instrukce

## Chování DLX *pipeline* podrobněji

Stupeň pipeline	ALU instrukce	Load / Store instrukce	Branch instrukce
Instruction Fetch (IF)	Čtení instrukce	Čtení instrukce	Čtení instrukce
Instruction Decode (ID)	Čtení operandů a dekódování instrukce	Čtení operandů a dekódování instrukce	Čtení operandů a dekódování instrukce
Execute (EX)	Výpočet výsledku	Výpočet adresy	Vyhodnocení podmínky
Memory Access (MEM)	Nic, jen kopírování výsledku	Přístup do paměti	Výpočet cílové adresy a změna PC
Write Back (WB)	Zápis výsledku do registrového pole	Zápis výsledku do reg. pole (pouze Load)	Nic

Toto je “tradiční 5-stupňová celočíselná RISC *pipeline*” (též *pipeline* ve stylu procesoru MIPS)

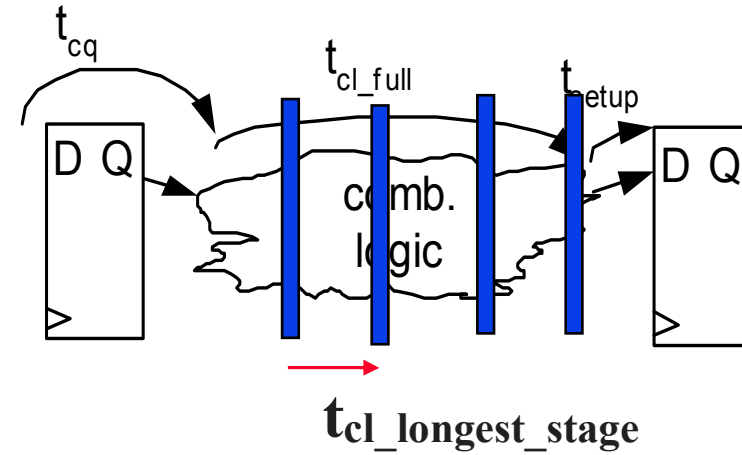
# Jiný pohled na *pipeline* - registry mezi stupni



# Proudové zpracování – analýza zrychlení (1)

$$T_{\text{CPU}} = IC * CPI * T_{\text{clk}}$$

- **IC** je stejné pro všechny varianty CPU
- **S** je počet stupňů proud. zpracování



$$T_{\text{clk\_nopipe}} = t_{cq} + t_{cl\_full} + t_{setup} + t_{skew}$$

$$T_{\text{clk\_pipe}} = t_{cq} + t_{cl\_longest\_stage} + t_{setup} + t_{skew}$$

$$t_{cl\_longest\_stage} \approx \frac{t_{cl\_full}}{S} \quad t_{setup} + t_{cq} + t_{skew} \ll \frac{t_{cl\_full}}{S}$$

$$T_{\text{clk\_pipe}} \approx \frac{T_{\text{clk\_nopipe}}}{S}$$

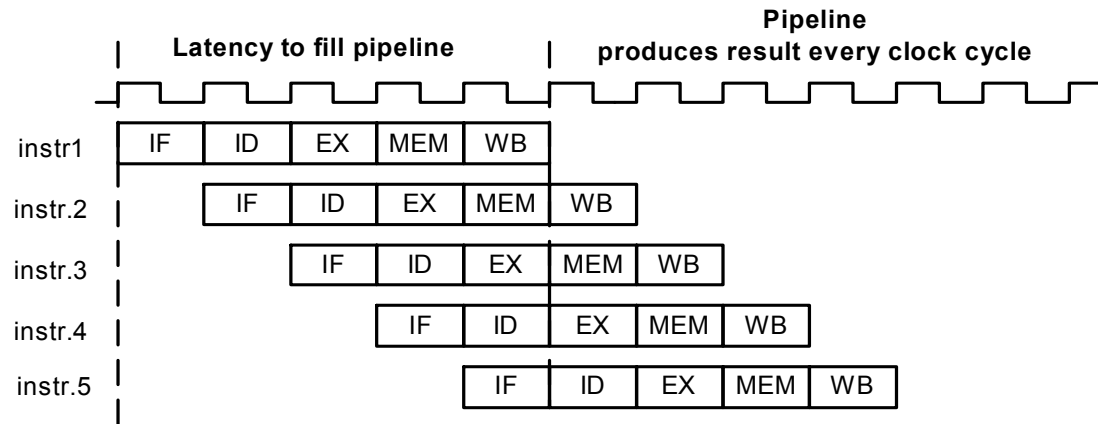
**Pozn:** **cl** = combinatorial logic, **cq** = clock to Q delay of register

$t_{skew}$  = rozptyl příchodu aktivních hran clk k registrům



# Proudové zpracování – analýza zrychlení (2)

- Kolik je CPI proudově pracujícího procesoru ?
- **S** je počet stupňů pipeline,  
**IC** je počet provedených instrukcí



$$\text{Cycles}_{IC\_instructions} = S + IC - 1$$

$$\text{CPI} = \frac{\text{Cycles}_{IC\_instructions}}{IC} = \frac{S - 1}{IC} + 1$$

$$\text{Pro velké } IC : \frac{S - 1}{IC} \ll 1 \Rightarrow \text{CPI}_{\text{pipeline\_ideal}} = 1$$

Ideální CPI pro proudově pracující procesor je rovno 1.

# Problémy, které přináší proudové zpracování

Proudové zpracování může narušit **sémantiku provádění** programu:

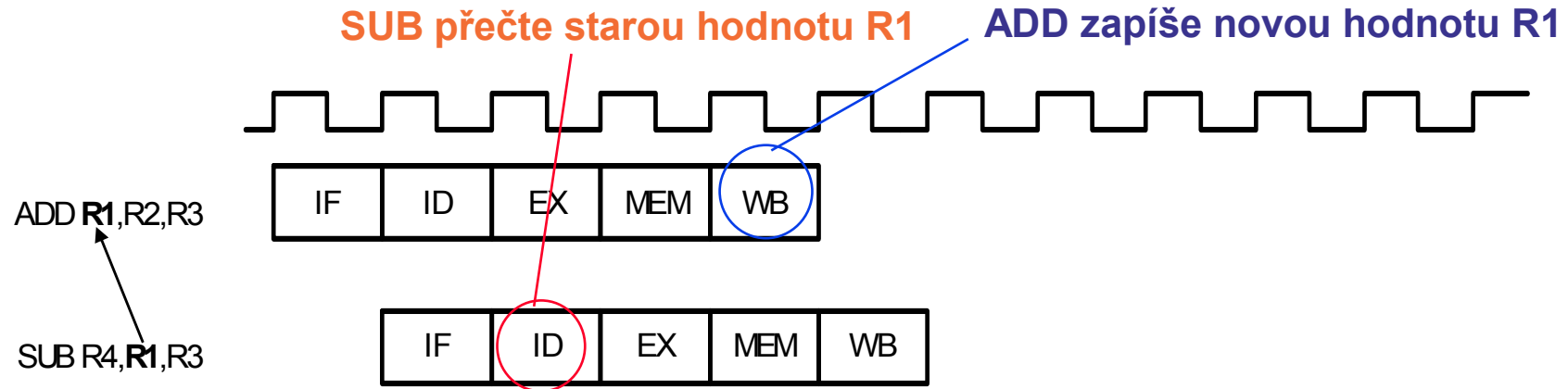
Sémantika provádění programu na von Neumannově počítači:  
Necht'  $i, j$  jsou dvě přirozená čísla,  $i < j$  potom  **$i$ -tá instrukce** v programovém pořadí **je provedena před  $j$ -tou instrukcí**.

Toto definuje **úplné sekvenční uspořádání** prováděných instrukcí a pipelining toto pořadí narušuje (všimněte si, že jednotaktový i vícetaktový procesor toto pořadí respektují).

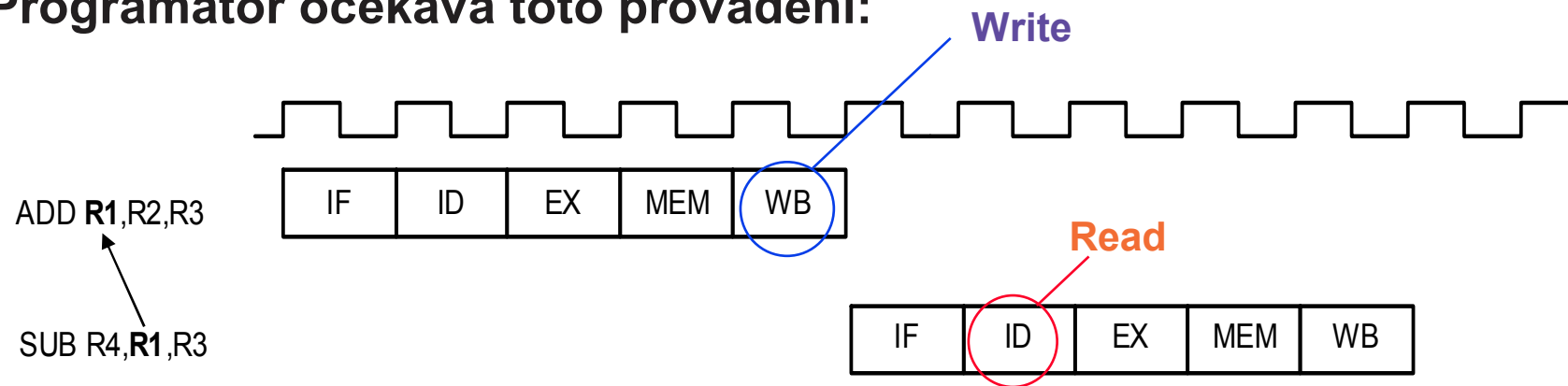
Dvě možné řešení:

- 1. Změnit sémantiku** – toto je obtížné až nemožné (Ale i tato cesta se využívá – viz procesory VLIW a EPIC)
- 2. Modifikace proudového zpracování** – výsledek je stejný jako v případě sekvenčního provádění (Toto je výhodné, neboť programátor v JSA nemusí vědět o proudovém zpracování.)

# Příklad narušení sémantiky

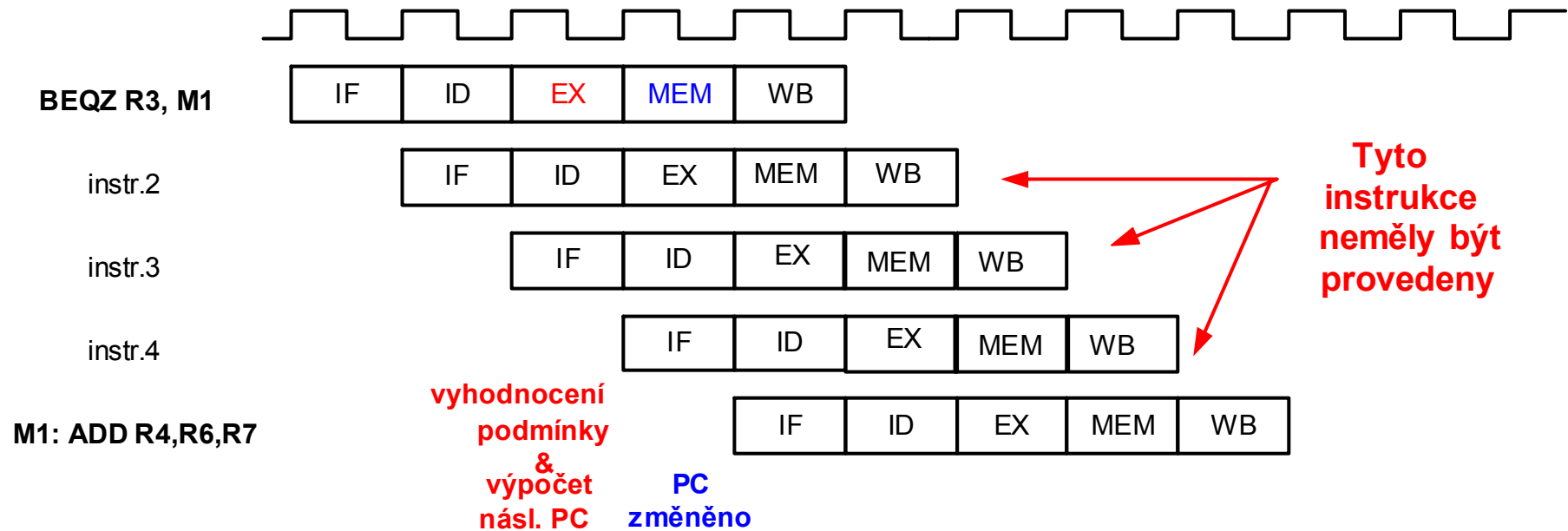


Programátor očekává toto provádění:



**Čtení R1 instr. SUB má být provedeno po zápisu R1 instrukcí ADD.**  
V pipeline toto pořadí není dodrženo. Tuto situaci nazýváme **Read after Write (RAW) hazard**.

# Jiný příklad narušení sémantiky



Toto je příklad narušení sémantiky řídicí instrukce (BEQZ)  
Tzv. **řídicí hazard (control hazard)**.

# Požadavky na korektnost provádění

Úplné sekvenční provádění není třeba, ale musí být zachovány následující podmínky:

Definice: Pro  $i < j$ :

## 1. Datové závislosti (Data Dependences)

Instrukce  $j$  je datově závislá na instrukci  $i$  iff instrukce  $j$  vytváří výsledek používaný instrukcí  $j$ .

## 2. Jmenné závislosti (Name Dependences též False data dependences)

Instrukce  $j$  je závislá po jménu na instrukci  $i$  iff instrukce  $j$  používá stejné místo k uložení výsledku ze kterého instrukce  $i$  čte nebo do něj zapisuje výsledek.

## 3. Řídící závislosti (Control Dependences)

Instrukce  $j$  je závislá na instrukci  $i$  iff provedení instrukce  $j$  závisí na výsledku provedení řídicí instrukce  $i$ .

Lze dokázat, že výsledek programu je stejný jako v případě sekvenčního provádění pokud tyto vlastnosti jsou respektovány.

# Hazardy v proudovém zpracování

(nemají nic společného s *hazardy v logických obvodech !!*)

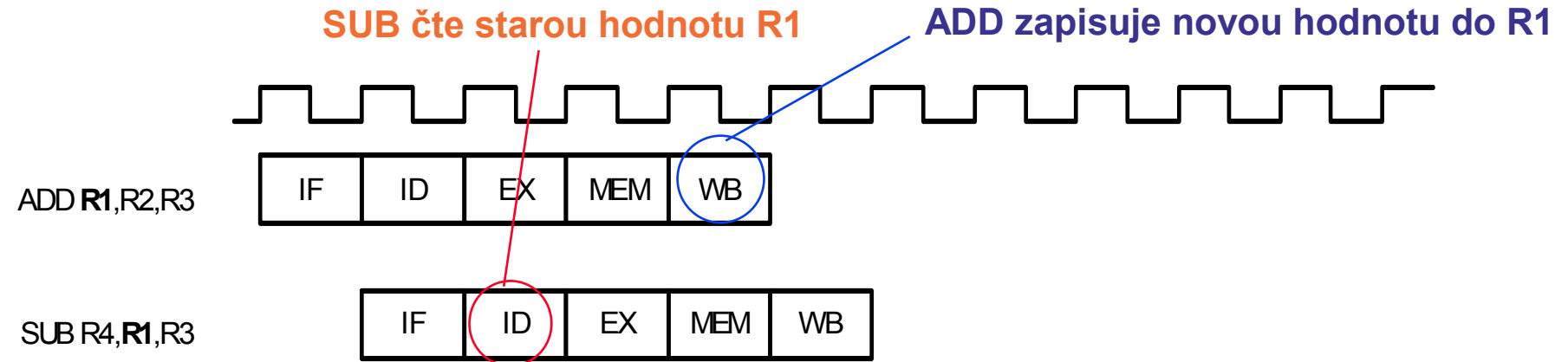
**Hazard v proudovém zpracování je potenciální narušení sémantiky provádění programu.**

- **Datové hazardy:** narušení **datových** nebo **jmenných závislostí**
- **Řídicí hazardy:** narušení **řídících závislostí**
- **Strukturní hazardy:** důsledek neúplné replikace *prostředků* v pipeline, kolize na sdílených prostředcích (paměti, porty reg. polí, funkční jednotky )

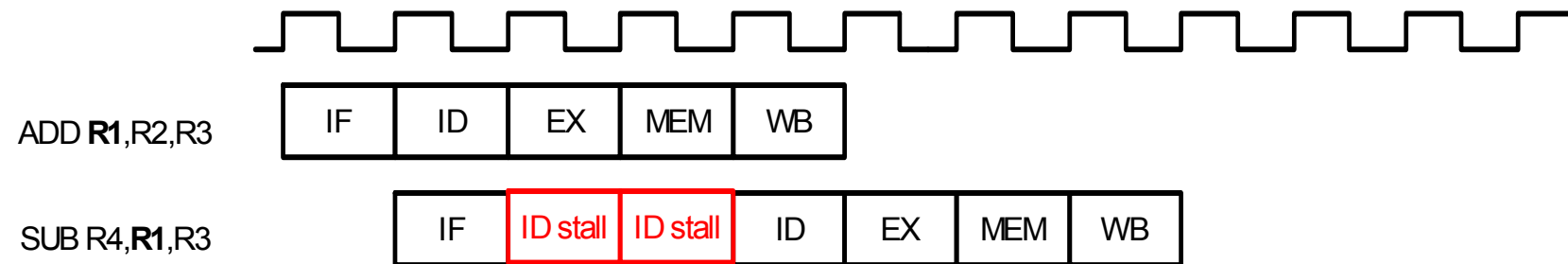
**Všimněte si, že datové a řídicí hazardy jsou důsledkem konfliktu mezi sémantikou programu a pipeliningem.**

**Strukturní hazardy jsou důsledkem nedokonalého pipelingu (více o nich příští týden).**

# Řešení datového hazardu pomocí pozastavení



**Řešení: Pozastavení pipeline v ID dokud není hazard odstraněn.**

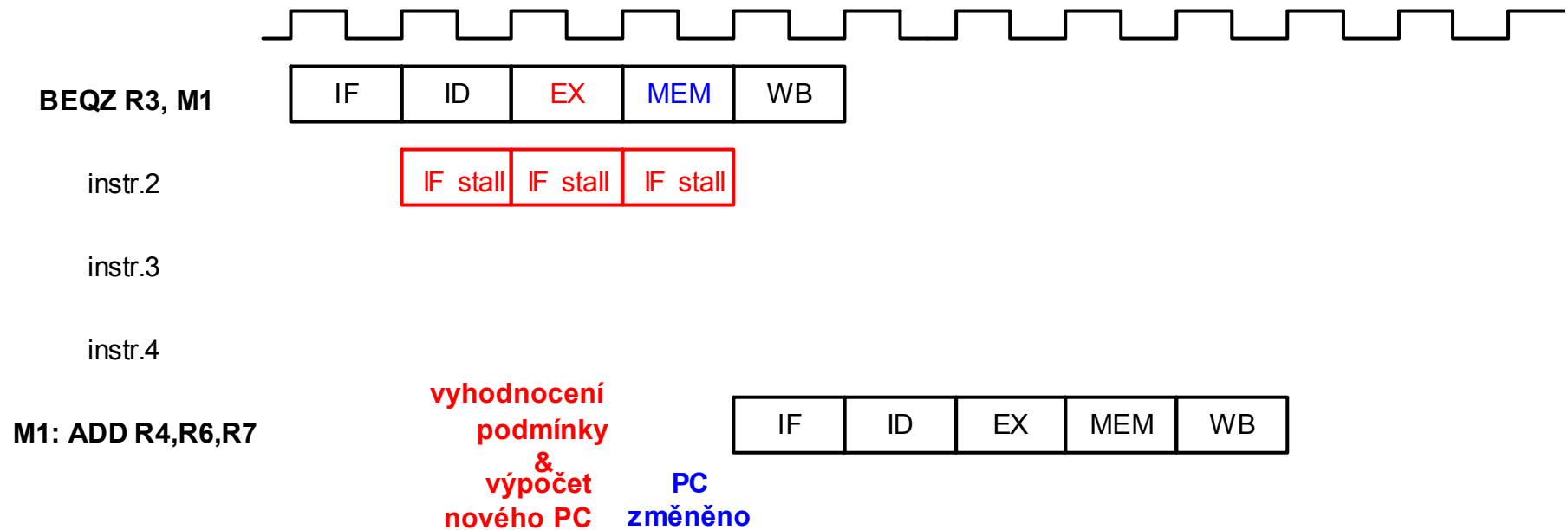


**SUB čeká v ID dokud ADD nezapiše novou hodnotu do R1**

*Pozastavení = pipeline stall, pipeline bubble insertion, pipeline interlock*

Architektura počítačů

# Řešení řídicího hazardu pomocí pozastavení



**Toto řešení znamená 3 takty pozastavení (3 stalls) na každou prováděnou instrukci podmíněného skoku.**



## Vliv pozastavení na CPI

- **CPI v ideálním proudovém zpracování je 1** (když je pipeline zaplněna, dokončí v každém taktu jednu instrukci, toto nezávisí na počtu stupňů pipeline)
- Každý hazard lze v principu odstranit pomocí pozastavení (někdy to lze i bez pozastavení ale o tom příště).
- CPI reálného proudově pracujícího procesoru je dáno jako

$$CPI_{\text{pipeline\_real}} = CPI_{\text{pipeline\_ideal}} + SPI \text{ (stalls per instruction)}$$

### Vliv faktorů SPI na CPI:

$$SPI = SPI_{\text{data}} + SPI_{\text{control}}$$

$SPI_{\text{control}}$  = dynamická četnost instrukce podmíněném skoku \* 3

Pokud **13 %** instrukcí jsou instrukce podmíněného skoku:

$$SPI_{\text{control}} = 0,13 * 3 = \mathbf{0,39}$$

*Výpočet  $SPI_{\text{data}}$  příští přednášku.*

# Zrychlení pomocí proudového zpracování

$$\text{Speedup} = \frac{T_{\text{cpu\_no\_pipeline}}}{T_{\text{cpu\_pipeline}}} = \frac{\text{IC} * \text{CPI}_{\text{no\_pipeline}} * T_{\text{clk\_no\_pipeline}}}{\text{IC} * \text{CPI}_{\text{pipeline\_real}} * T_{\text{clk\_pipeline}}}$$

$$\text{CPI}_{\text{single\_cycle}} = 1; \text{CPI}_{\text{pipeline\_real}} = \text{CPI}_{\text{pipeline\_ideal}} + \text{SPI}$$

Stalls per Instruction  
(pozastavování pipeline)

$$\text{Speedup} = \frac{1}{\text{CPI}_{\text{pipeline\_ideal}} + \text{SPI}} * \frac{T_{\text{clk\_no\_pipeline}}}{T_{\text{clk\_pipeline}}}$$

$$\text{Speedup} = \frac{S}{1 + \text{SPI}}$$

# Závěry

- Ukázali jsme si několik organizací procesoru DLX a srovnali jsme jejich výkonnost
- Ukázali jsme, že je možno snížit *CPI* na úkor  $T_{clk}$  (a naopak)
- Ukázali jsme si tradiční 5-stupňovou celočíselnou RISC pipeline
- Výkonnost proudového zpracování závisí na počtu stupňů pipeline *S* a četnosti pozastavování (*SPI*)
- K pozastavování dochází kvůli *strukturním, datovým a řídicím hazardům*

## Příští přednáška

- Řešení hazardů v celočíselné pipeline bez pozastavování
- Vícetaktové operace v pipeline –pipeline pro pohyblivou ř.č.