

Úvod do organizace počítače

Pipelining

Provádění a řízení výpočtu ve
struktuře pipeline

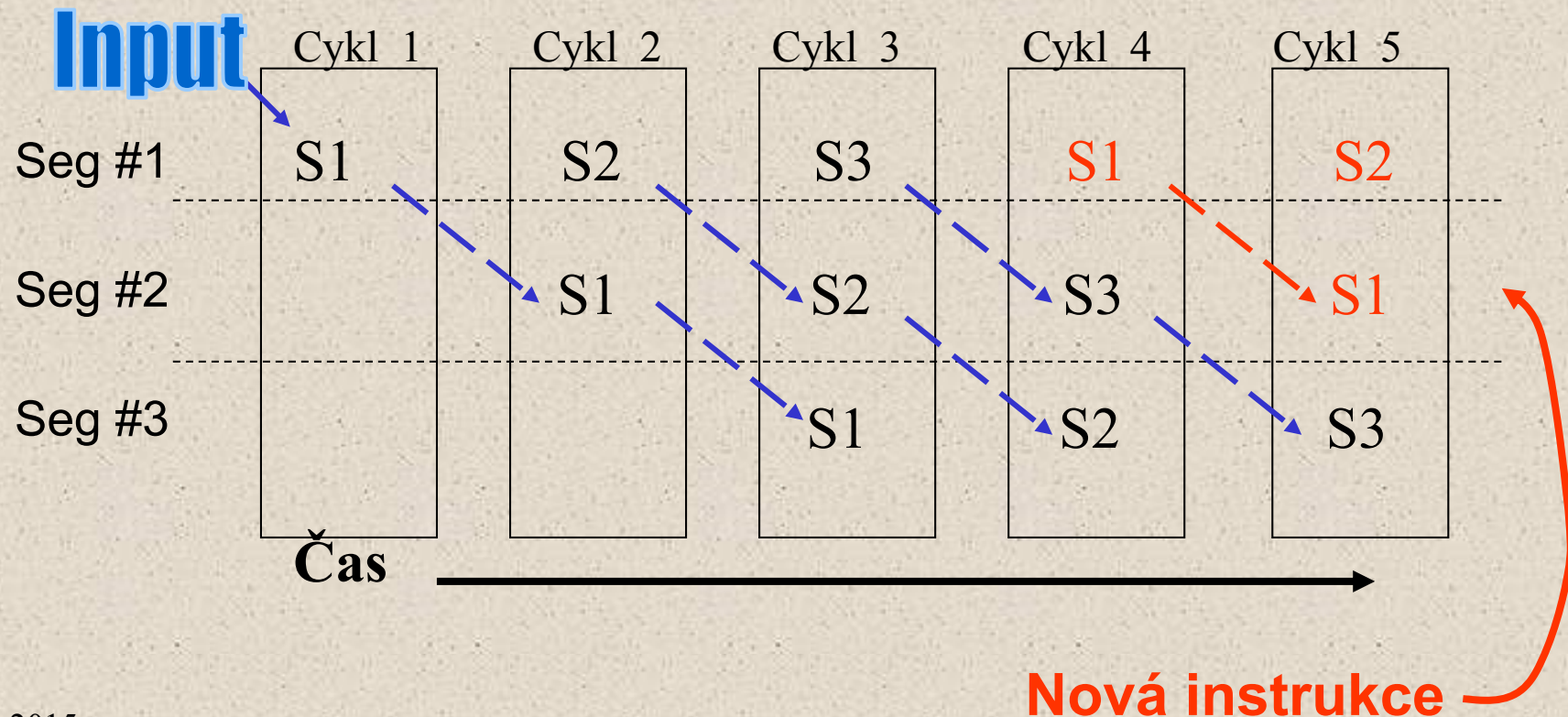
Řízení pipeline a **hazardy**

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 = $k \times$ počet kroků (stupňů)
 - Teorie: k je velká konstanta
 - Realita: Pipelining provází "další náklady"

Příklad pipeliningu

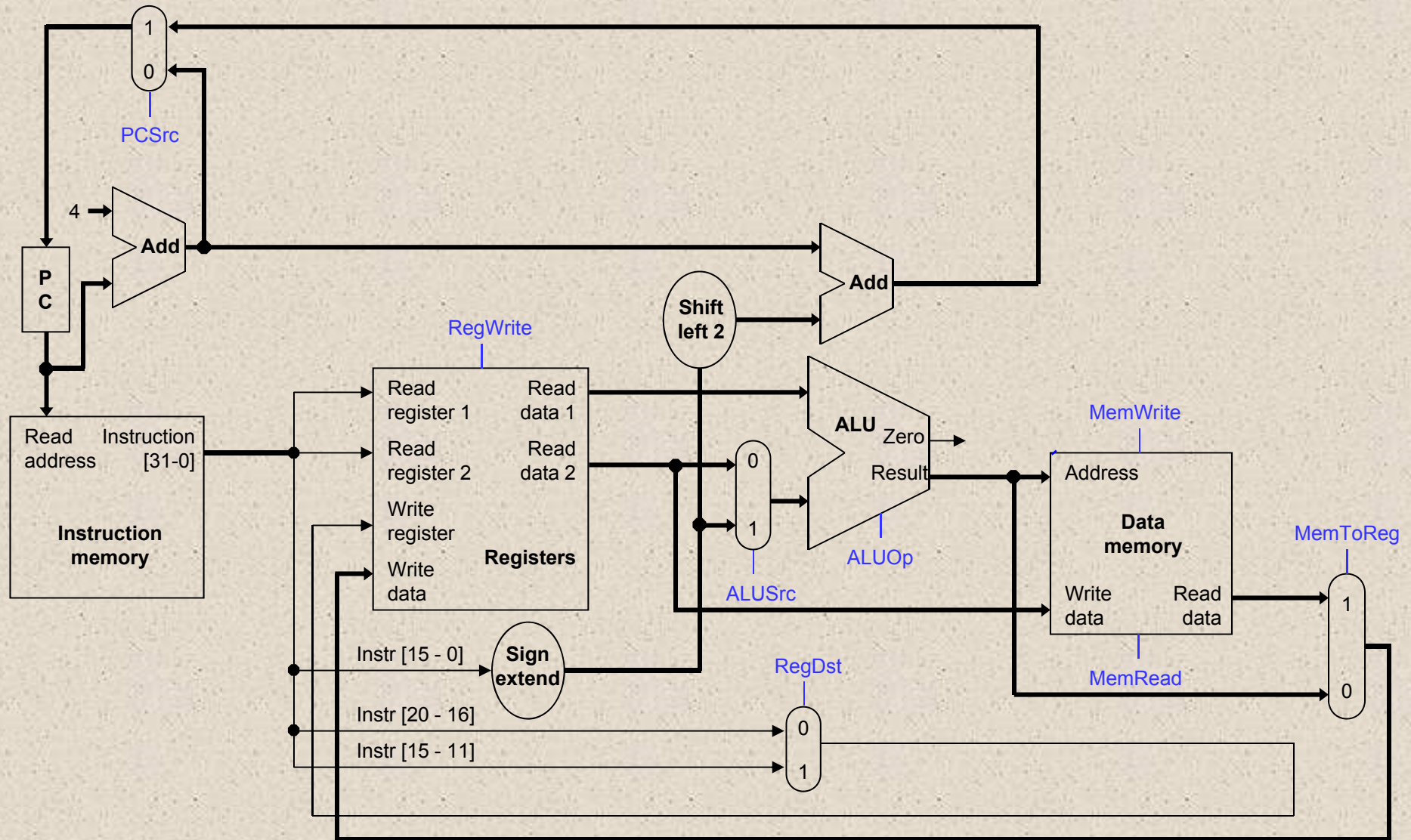
- Předpoklad: Jeden instrukční formát (snadné)
- Předpoklad: Každá instrukce má 3 kroky S1..S3
- Předpoklad: Pipeline má 3 segmenty (jeden/krok)



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

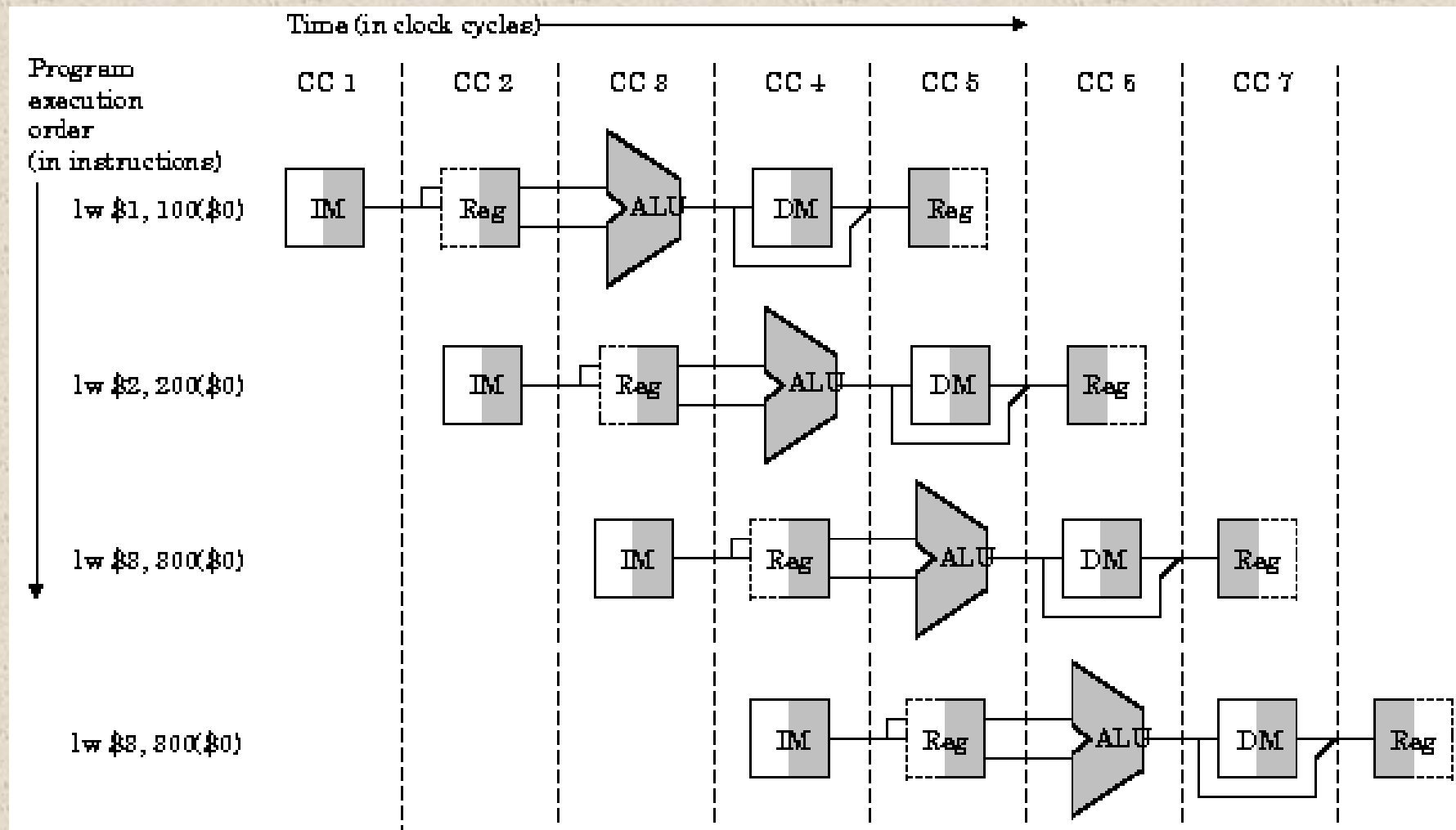
ID/EX

EX/MEM

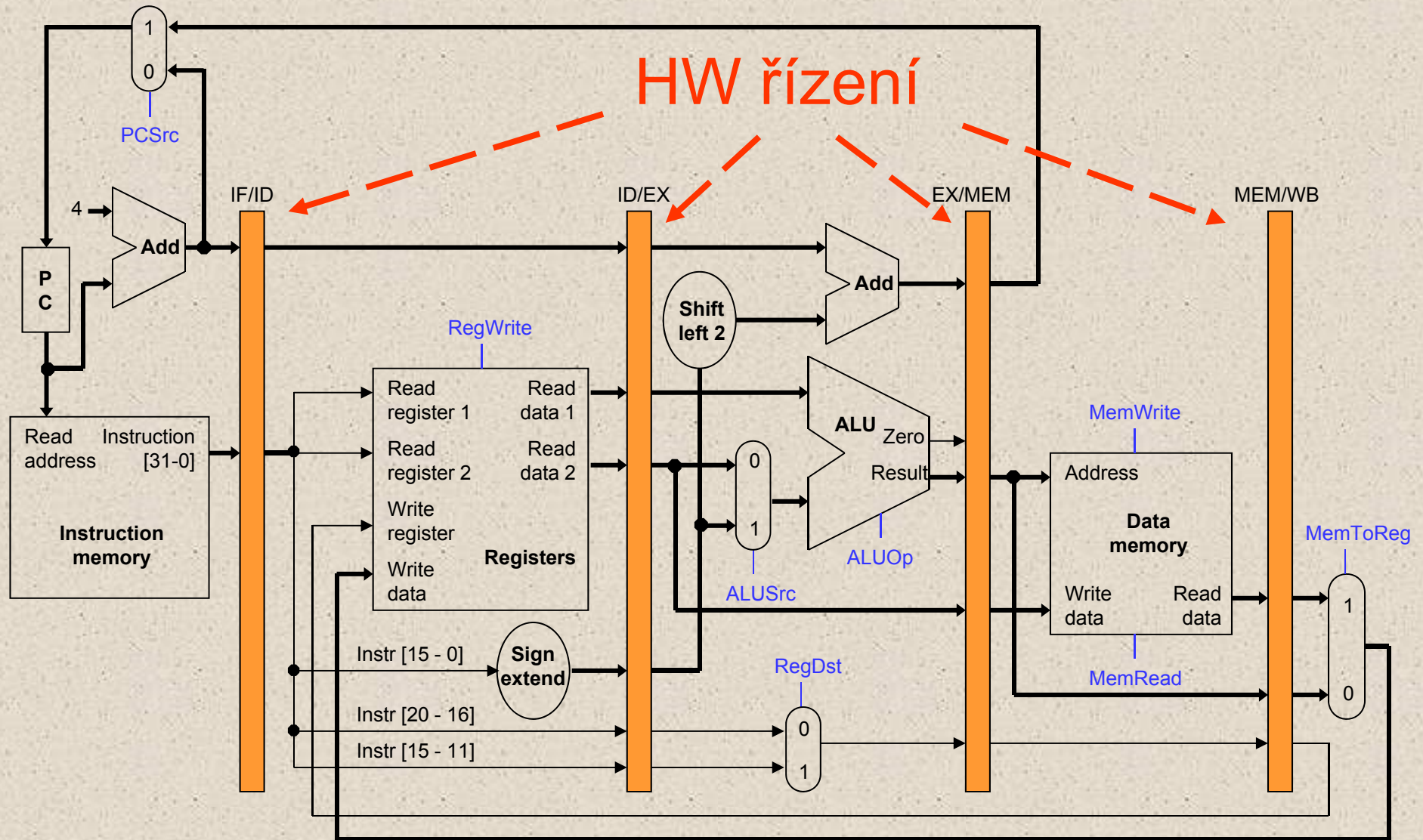
MEM/WB

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

Výpočetní struktura – uspořádání pipeline



Jednotka - režim pipeline

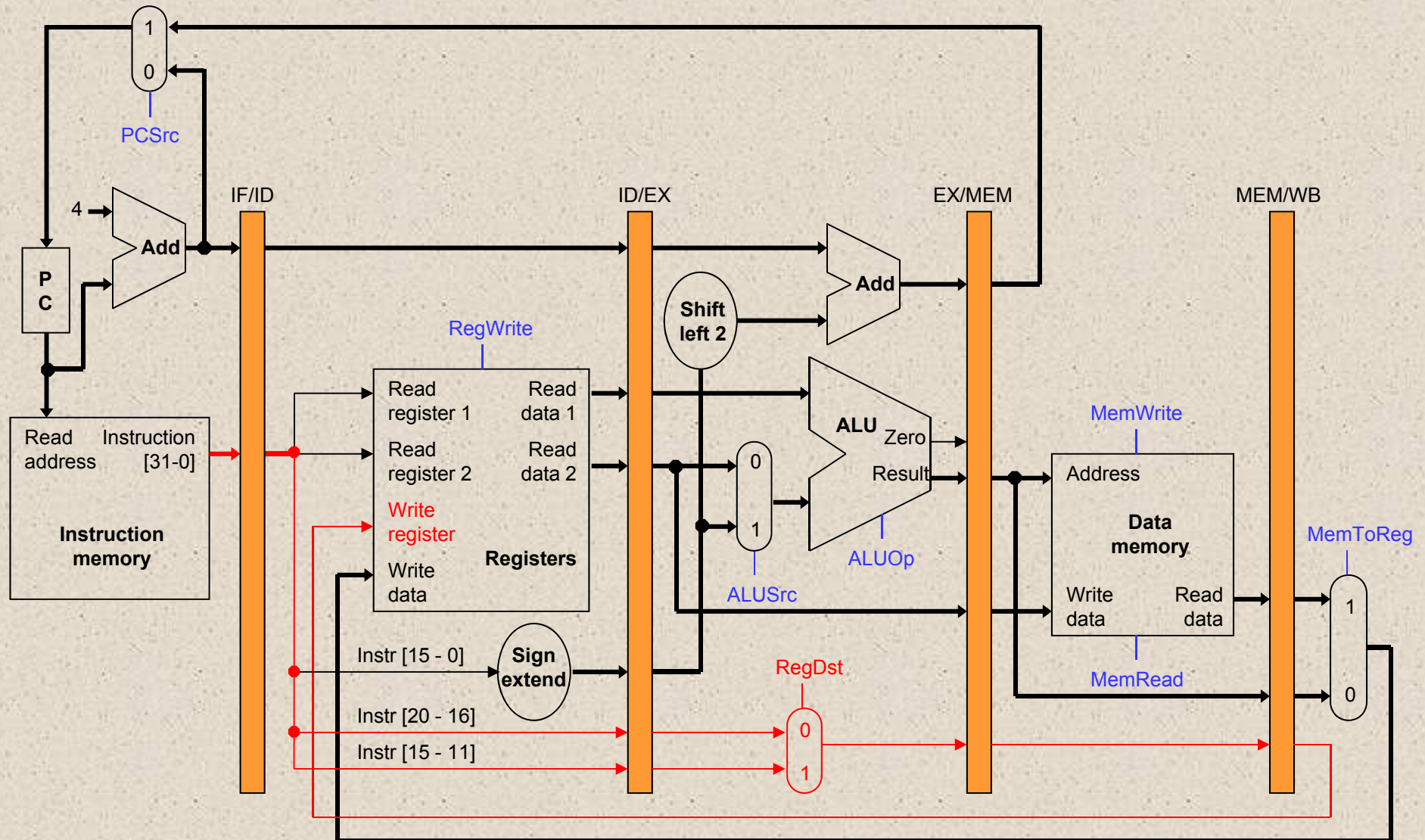


Přesun rozpracovaných dat

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.

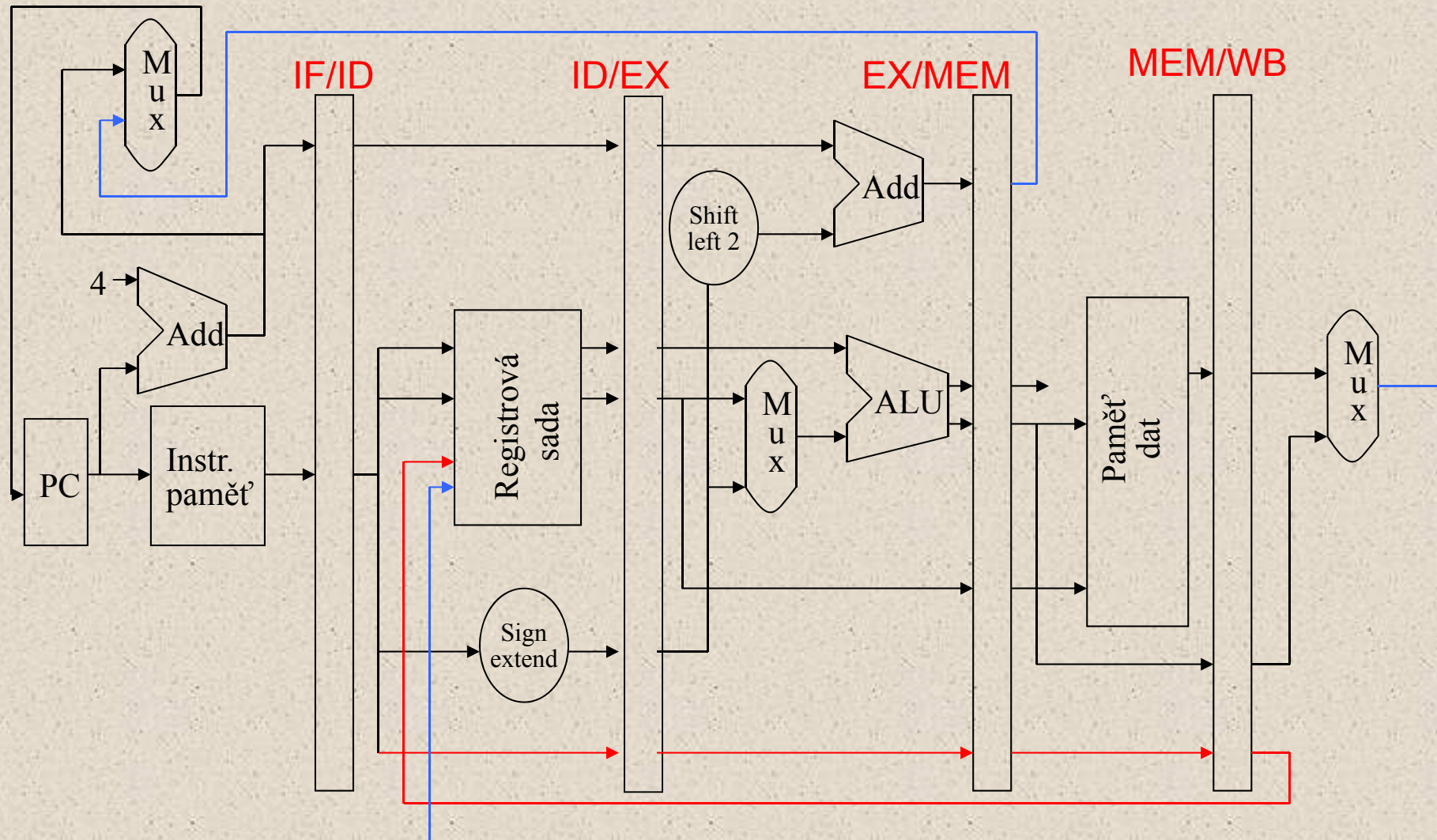
Registr adresy výsledku



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 = $k \times$ počet kroků (stupňů)
 - Teorie: k je velká konstanta
 - Realita: Pipelining provází “další náklady”

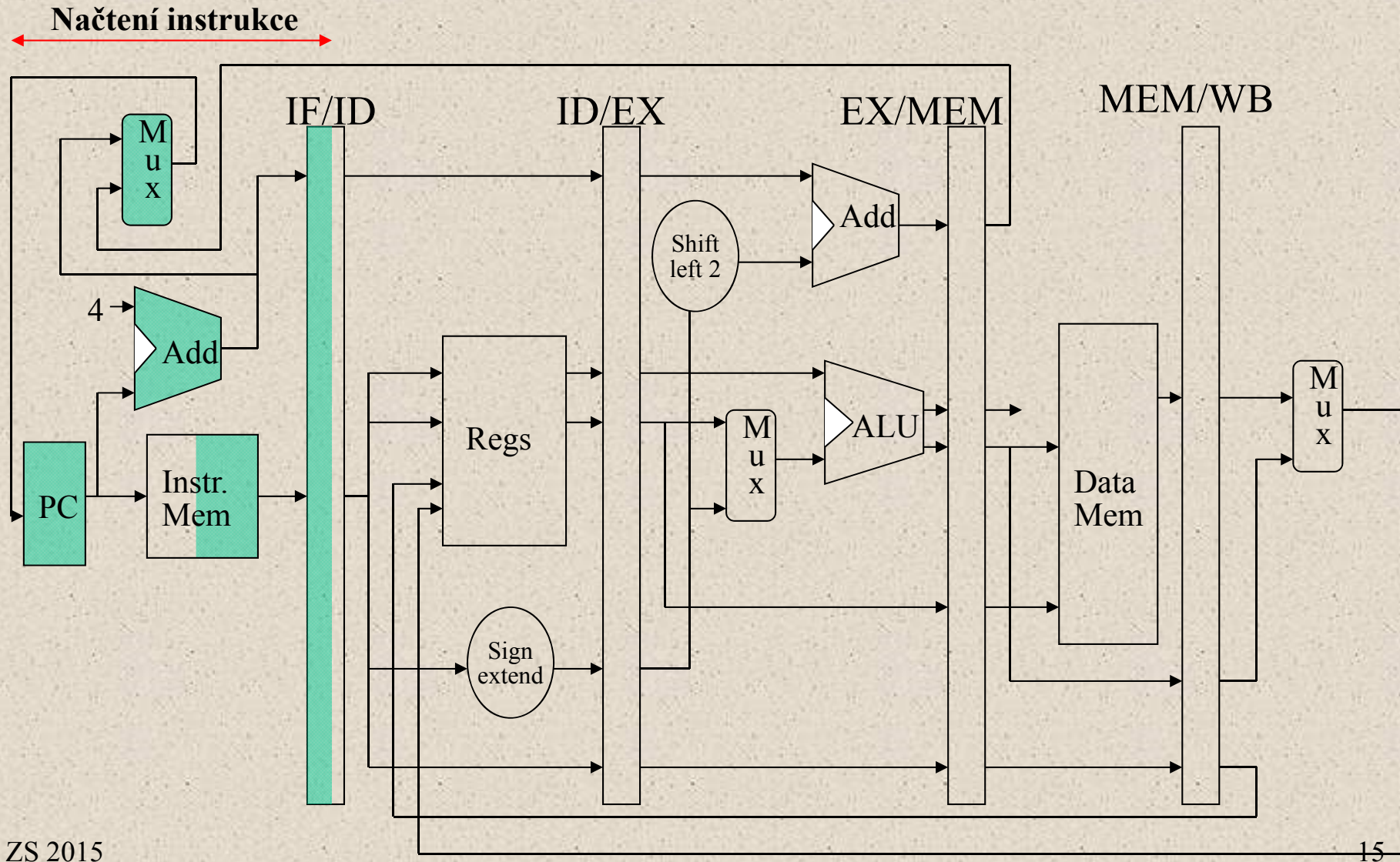
Jednotka v režimu pipeline



Pipeline u MIPS

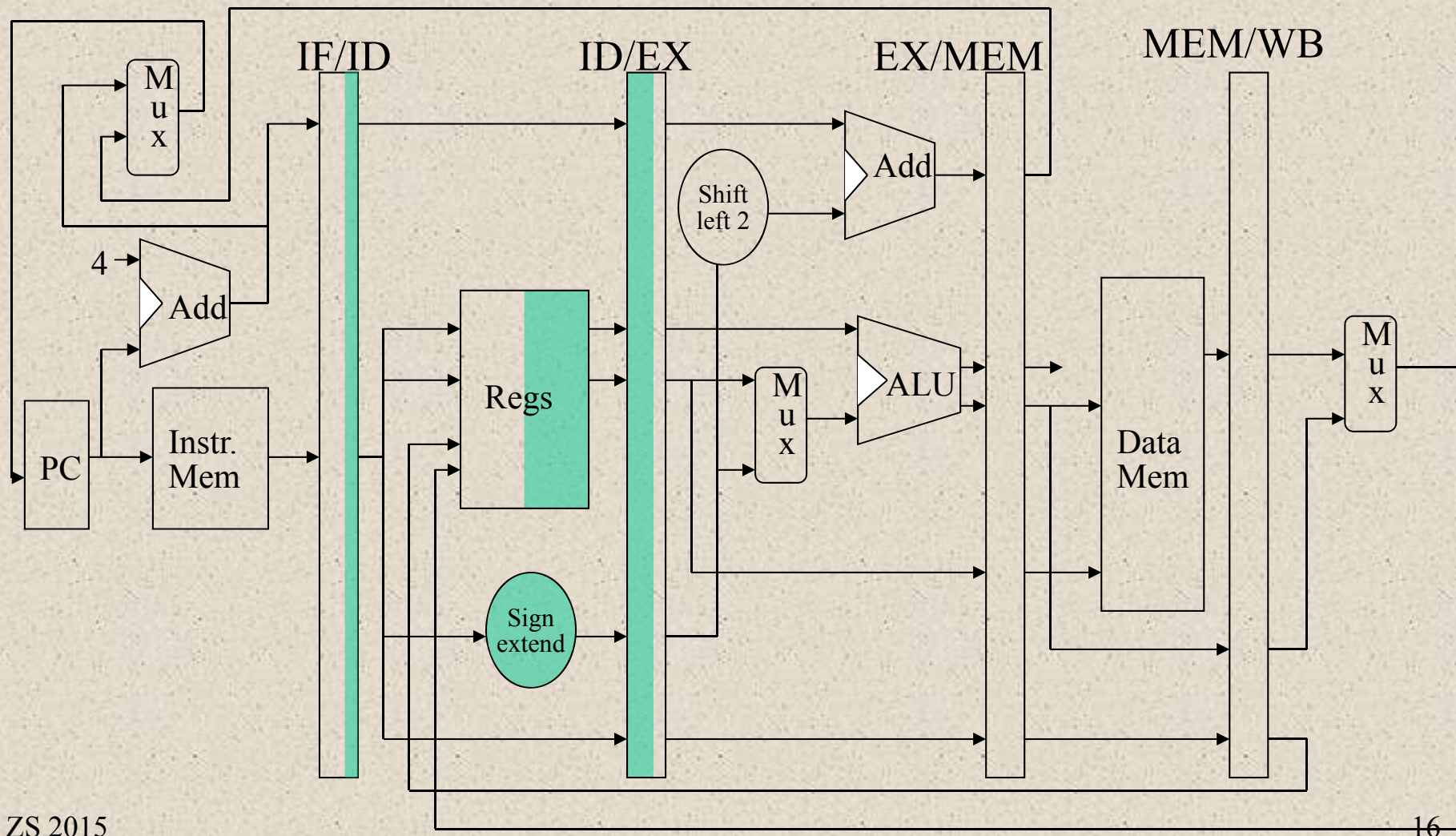
- Podmnožina MIPS
 - *Přístup do paměti*: lw a sw
 - *Aritmetické a logické instrukce*: and, sub, and, or, slt
 - *Instrukce větvení*: beq
- Kroky (segmenty pipeline struktury)
 - IF:** načtení instrukce z paměti
 - ID:** dekódování instrukce a čtení registrů
 - EX:** provedení operace nebo výpočet adresy
 - MEM:** přístup k operandu do datové paměti
 - WB:** zápis výsledku do registru

Provedení lw (1/5)

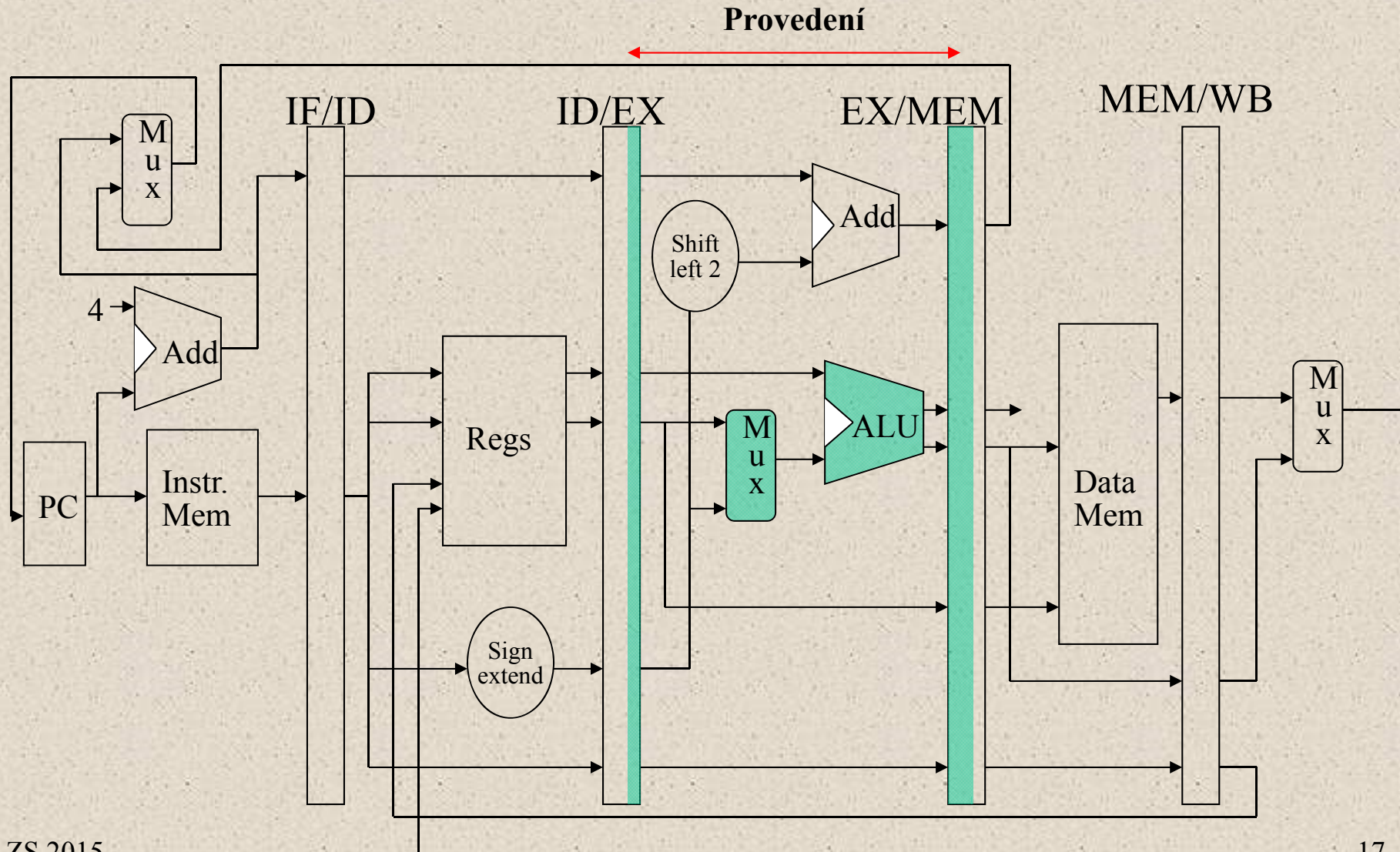


Provedení lw (2/5)

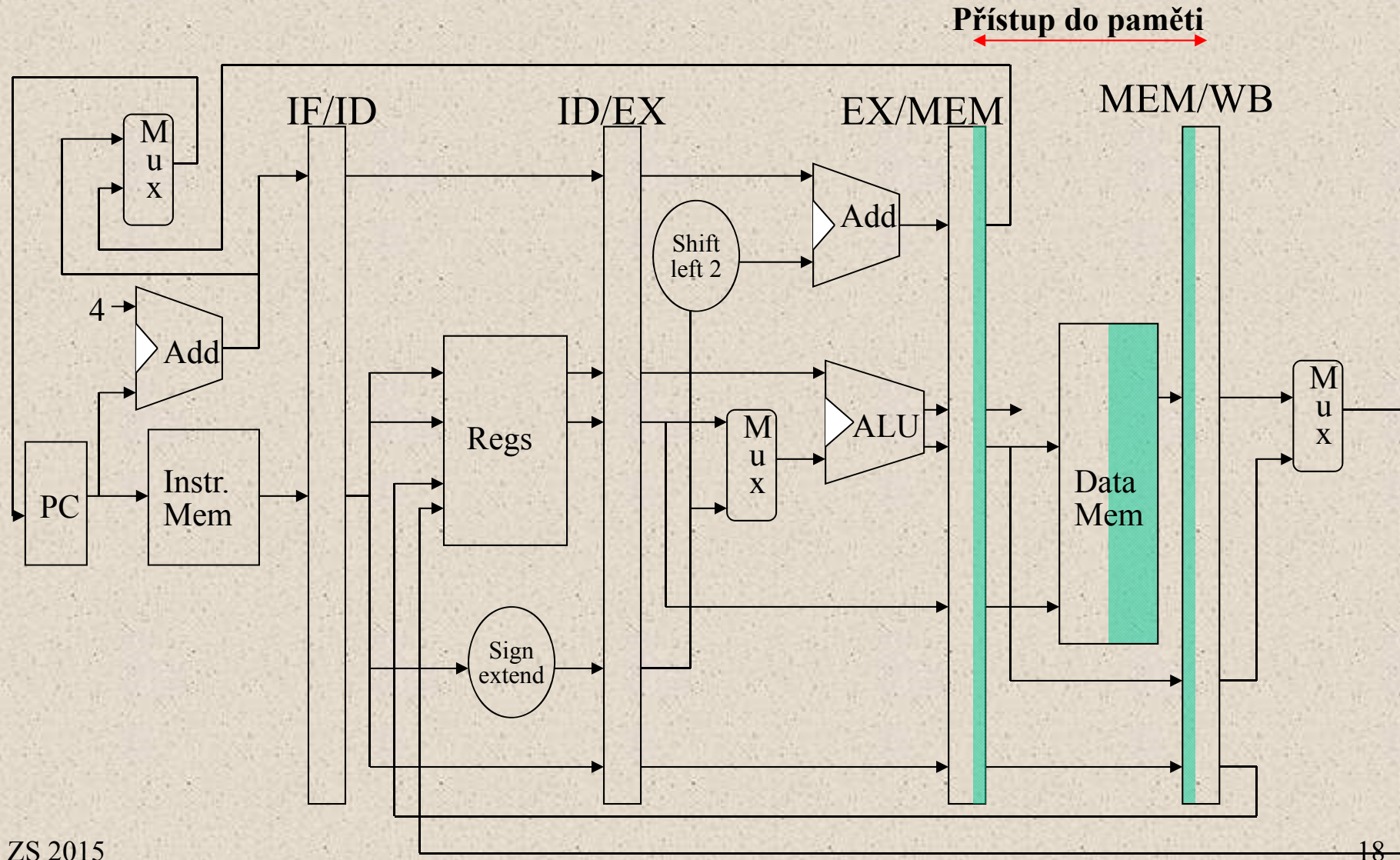
Dekódování instrukce



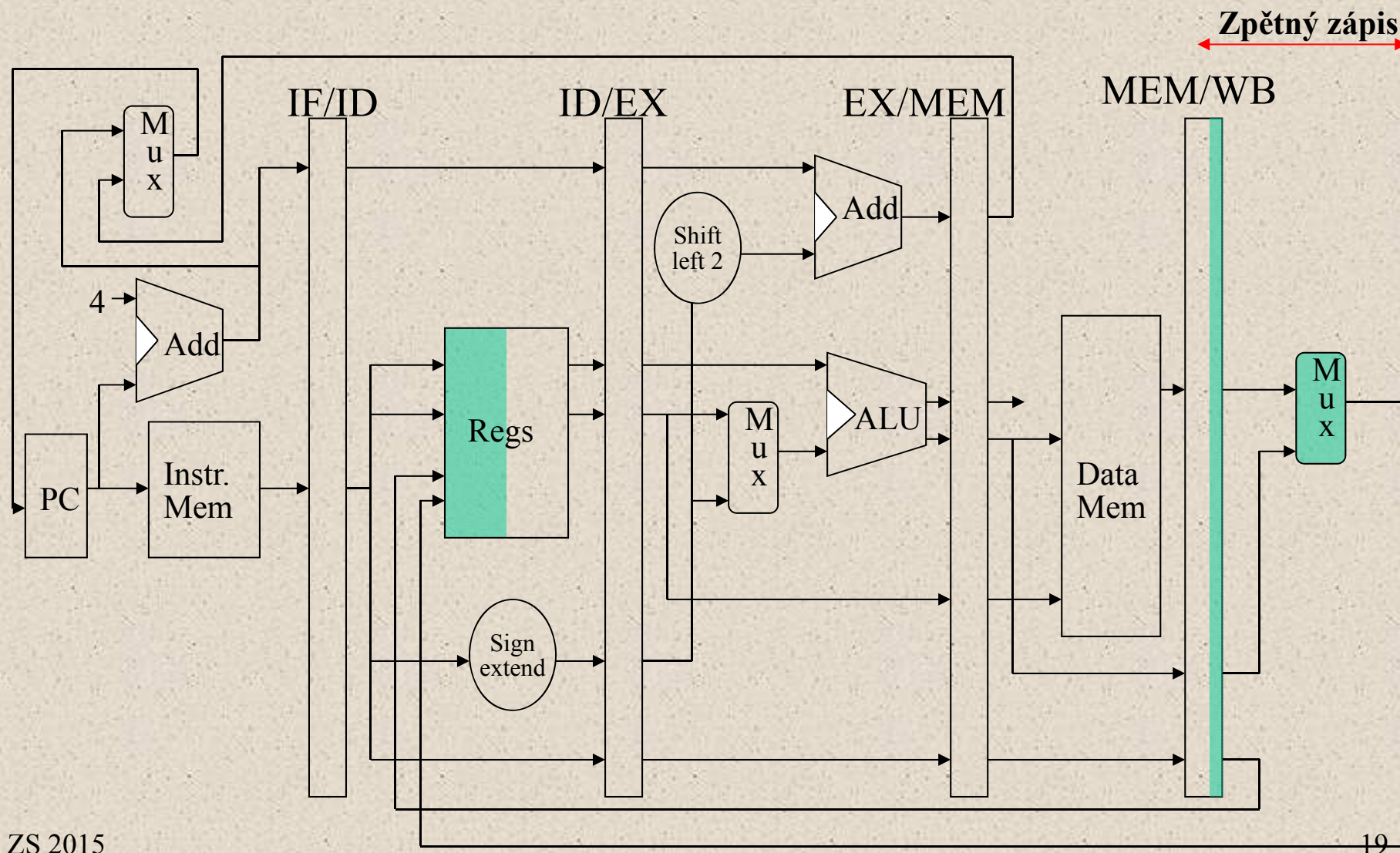
Provedení lw (3/5)



Provedení lw (4/5)



Provedení lw (5/5)



Zdroje použité pro vykonání lw

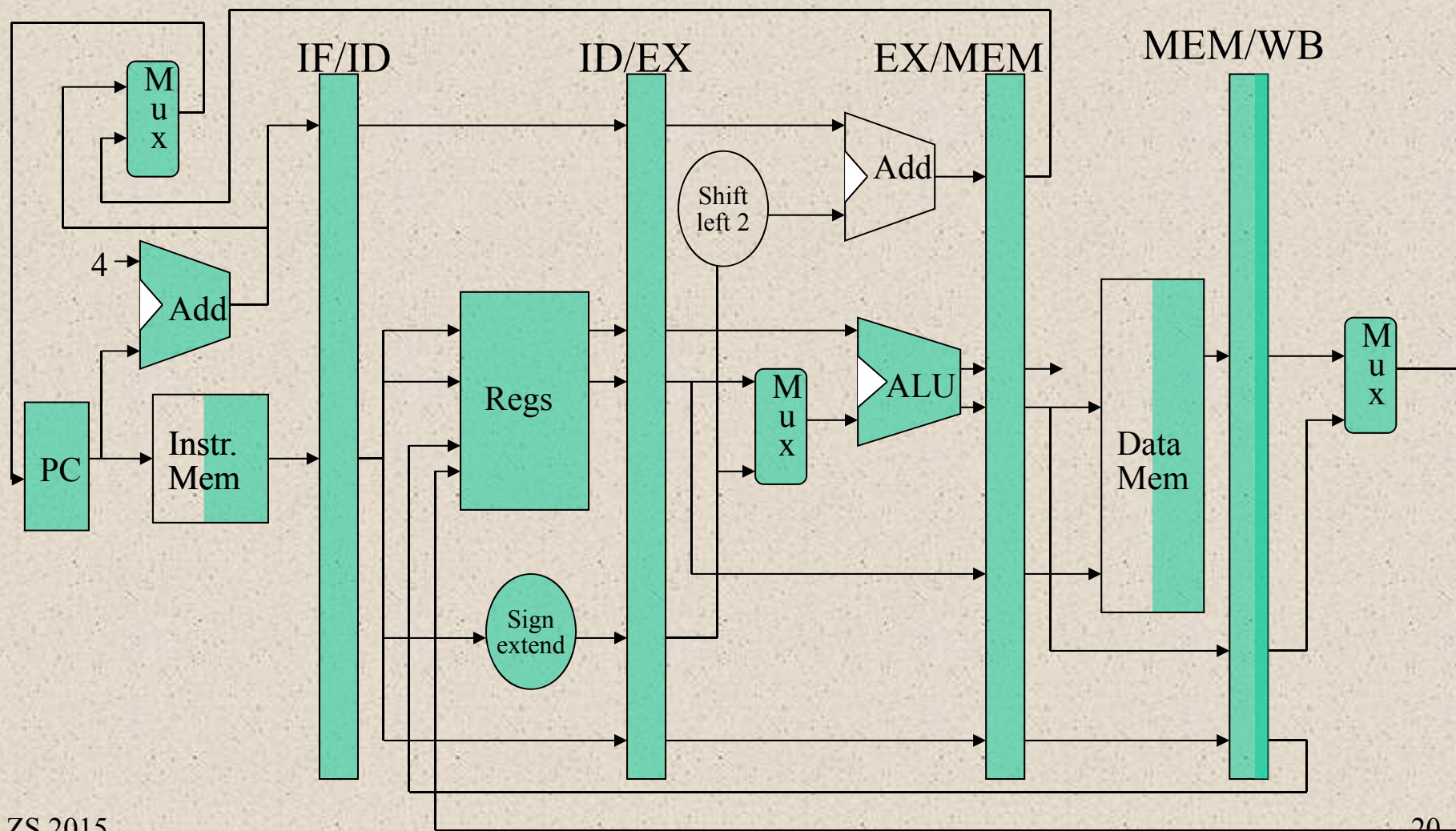
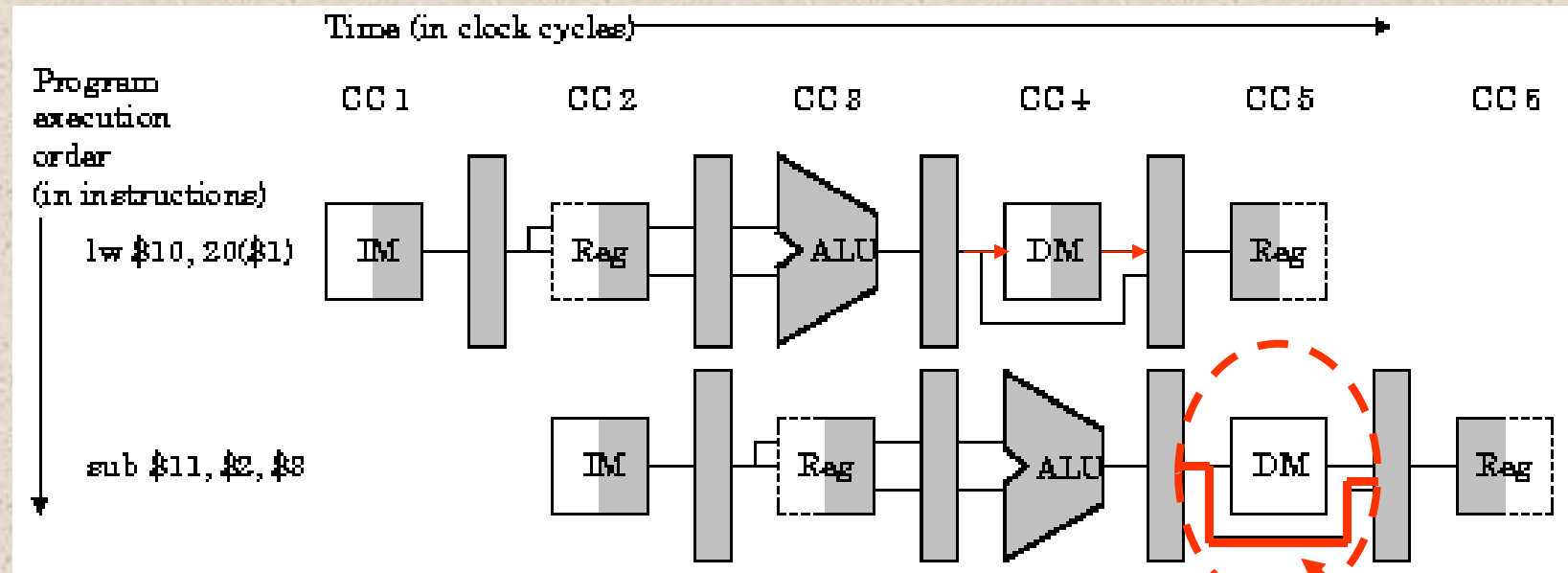


Diagram – časový rozvoj



Bypass

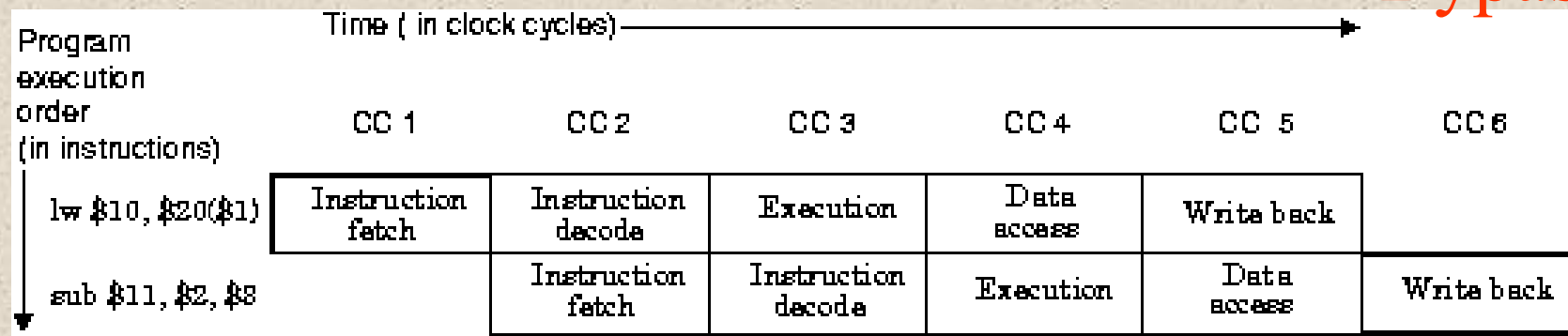


Diagram jednoho cyklu (cykl 1)

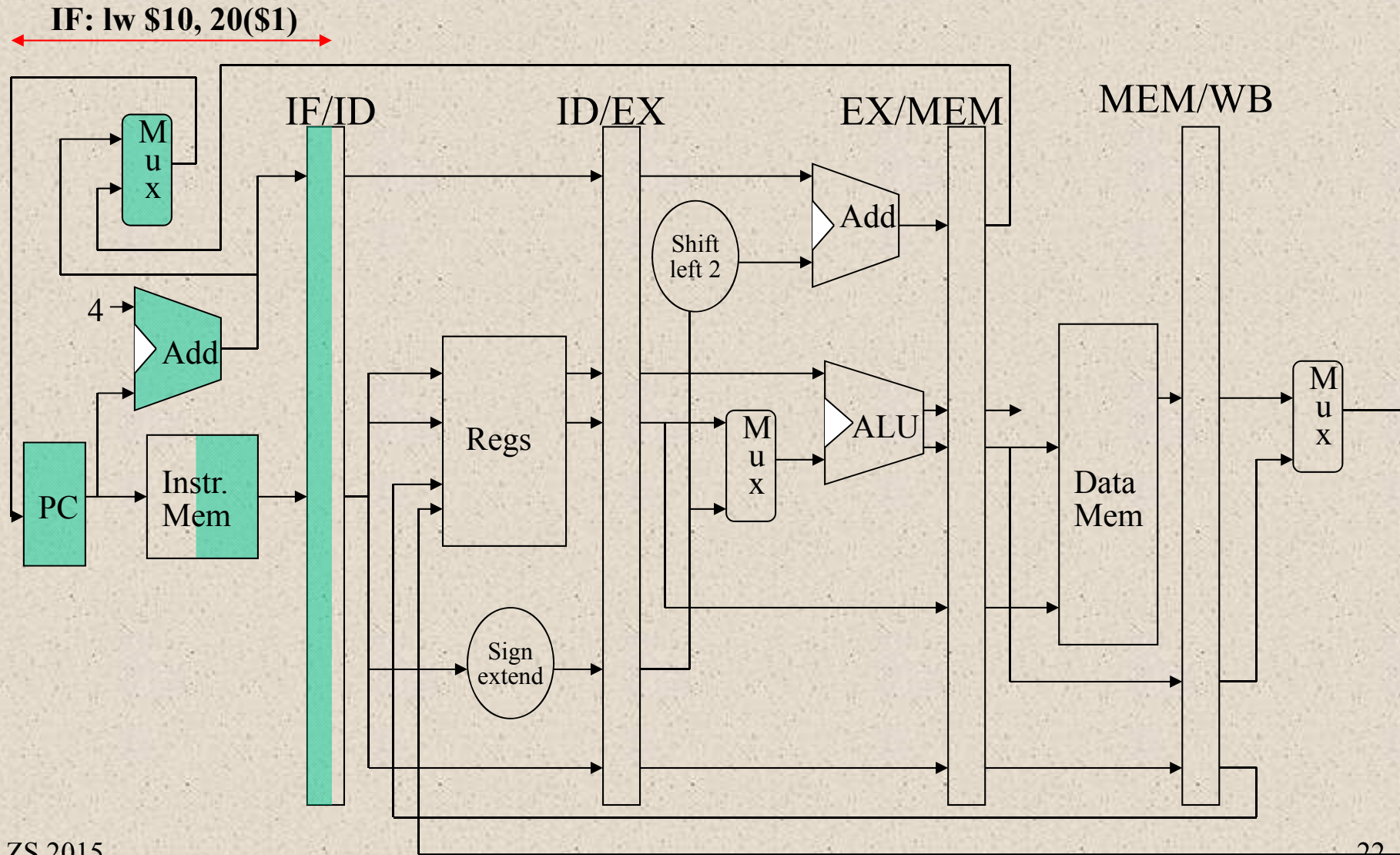


Diagram jednoho cyklu (cykl 2)

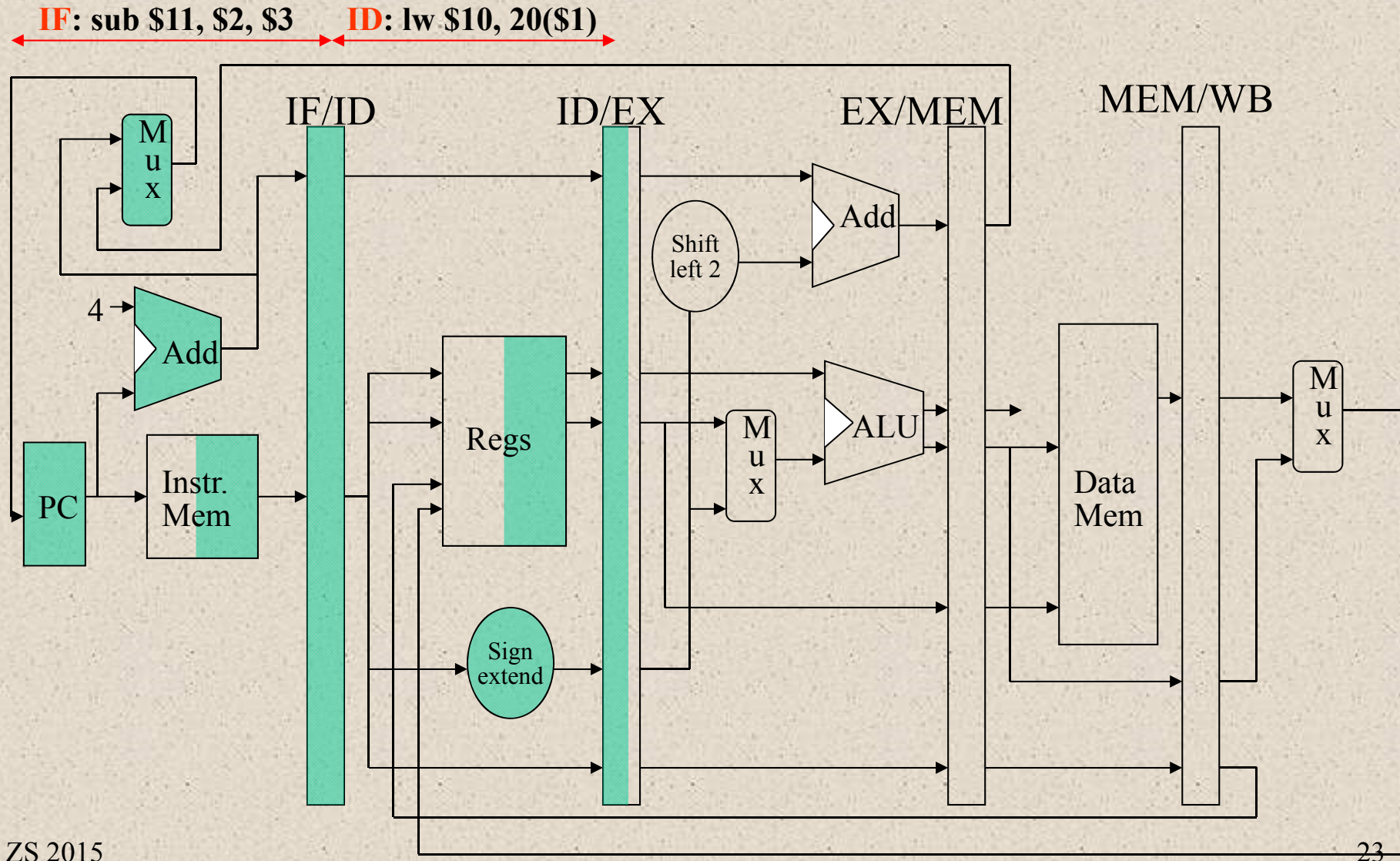


Diagram jednoho cyklu (cykl 3)

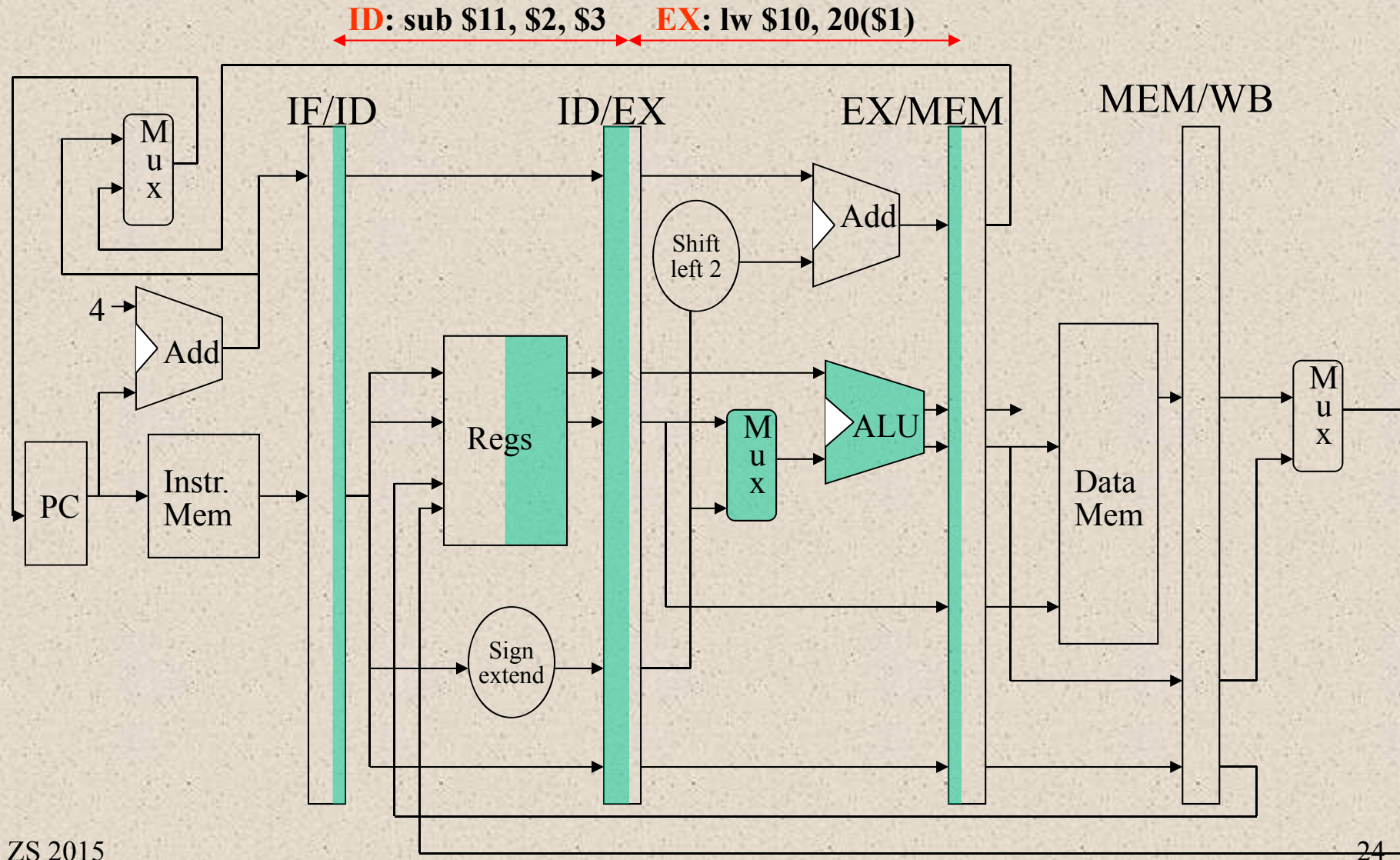


Diagram jednoho cyklu (cykl 4)

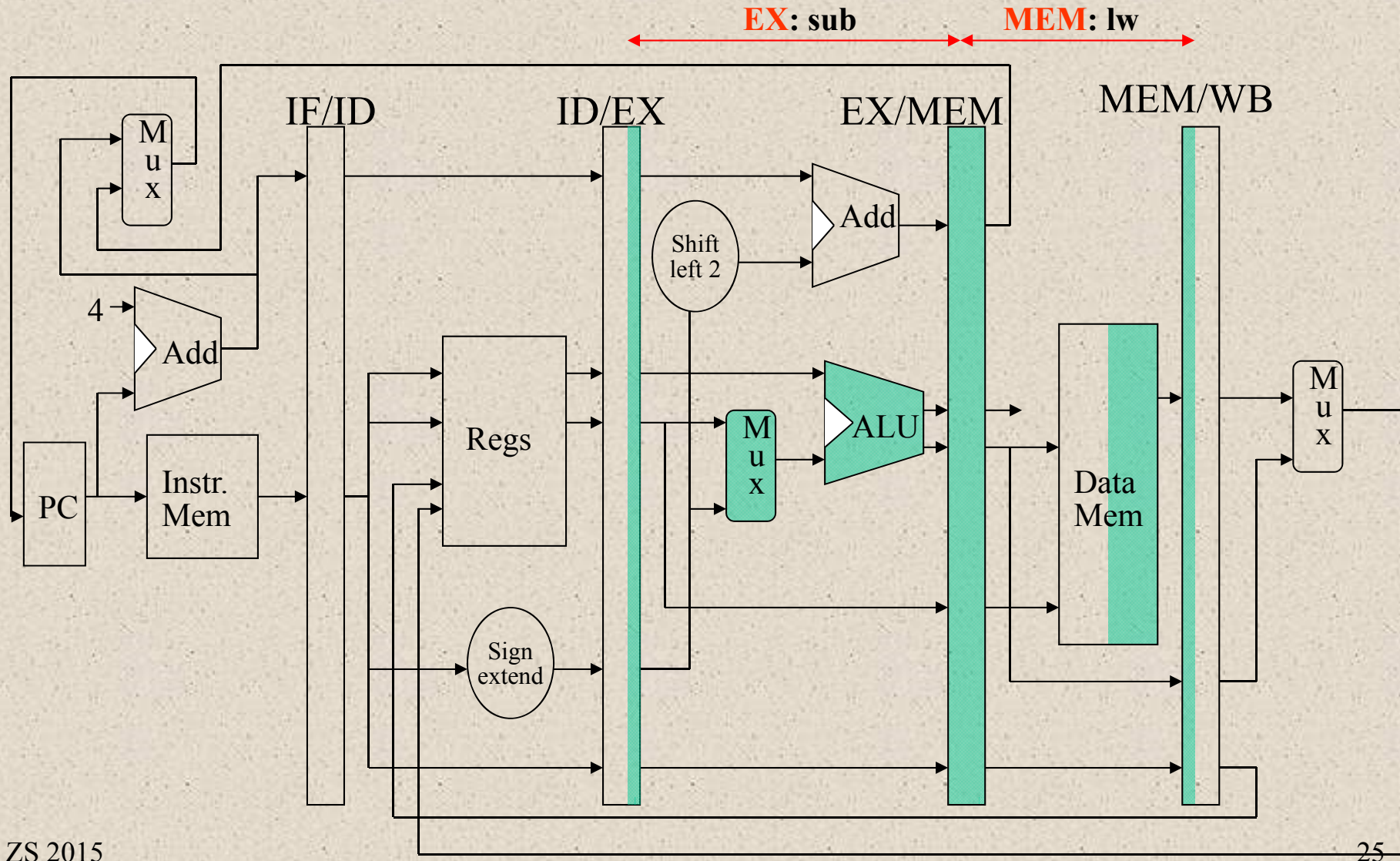


Diagram jedného cyklu (cykl 5)

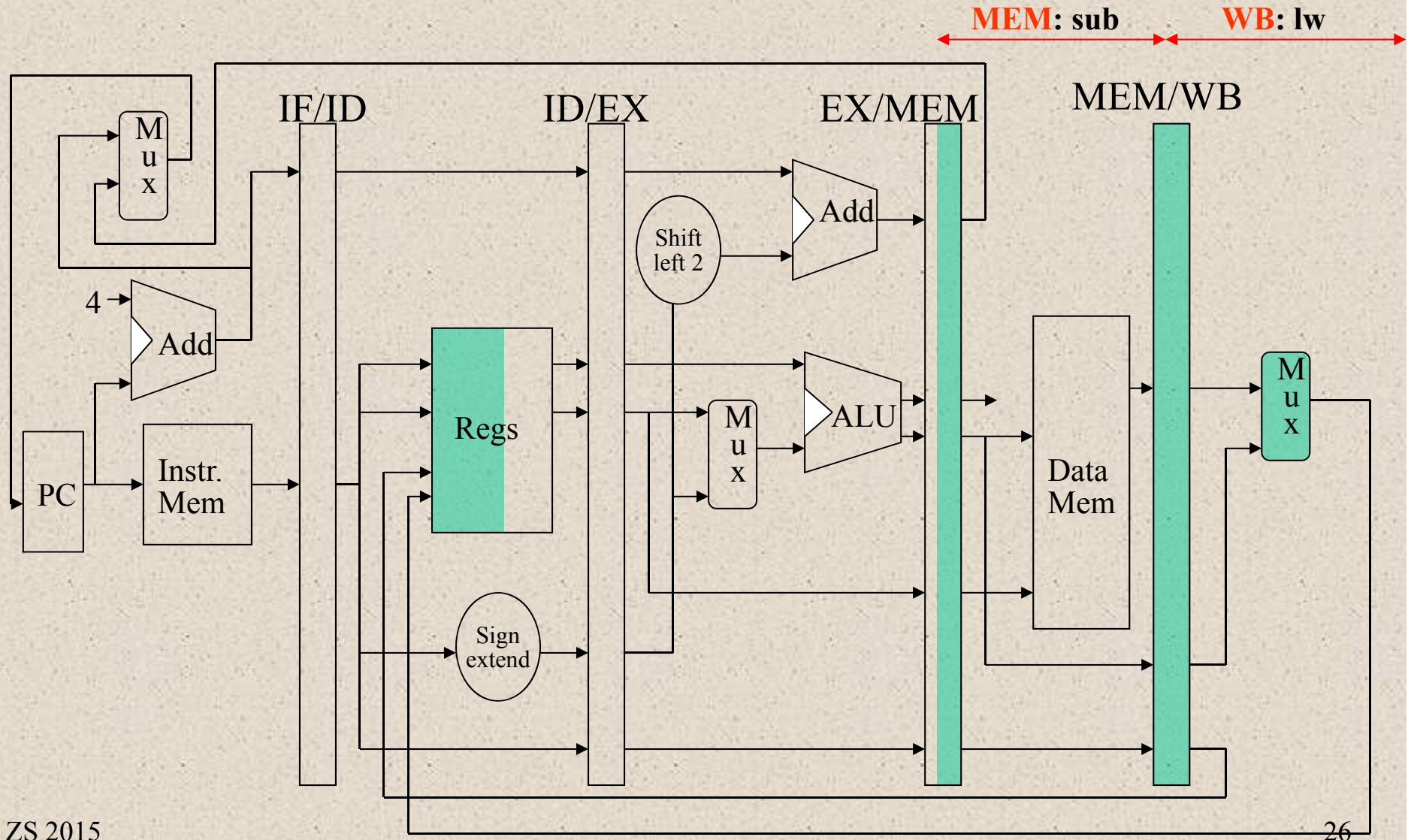
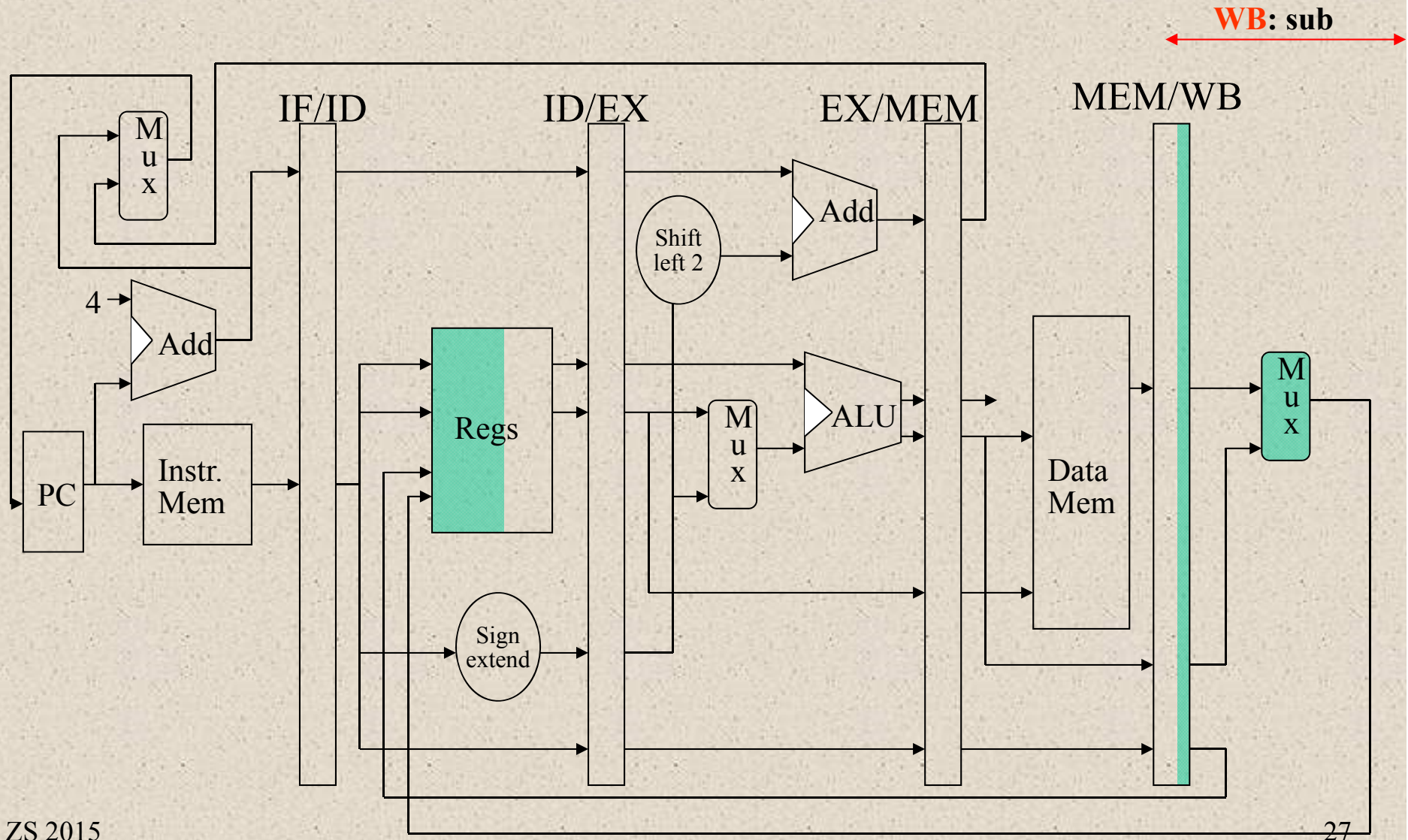
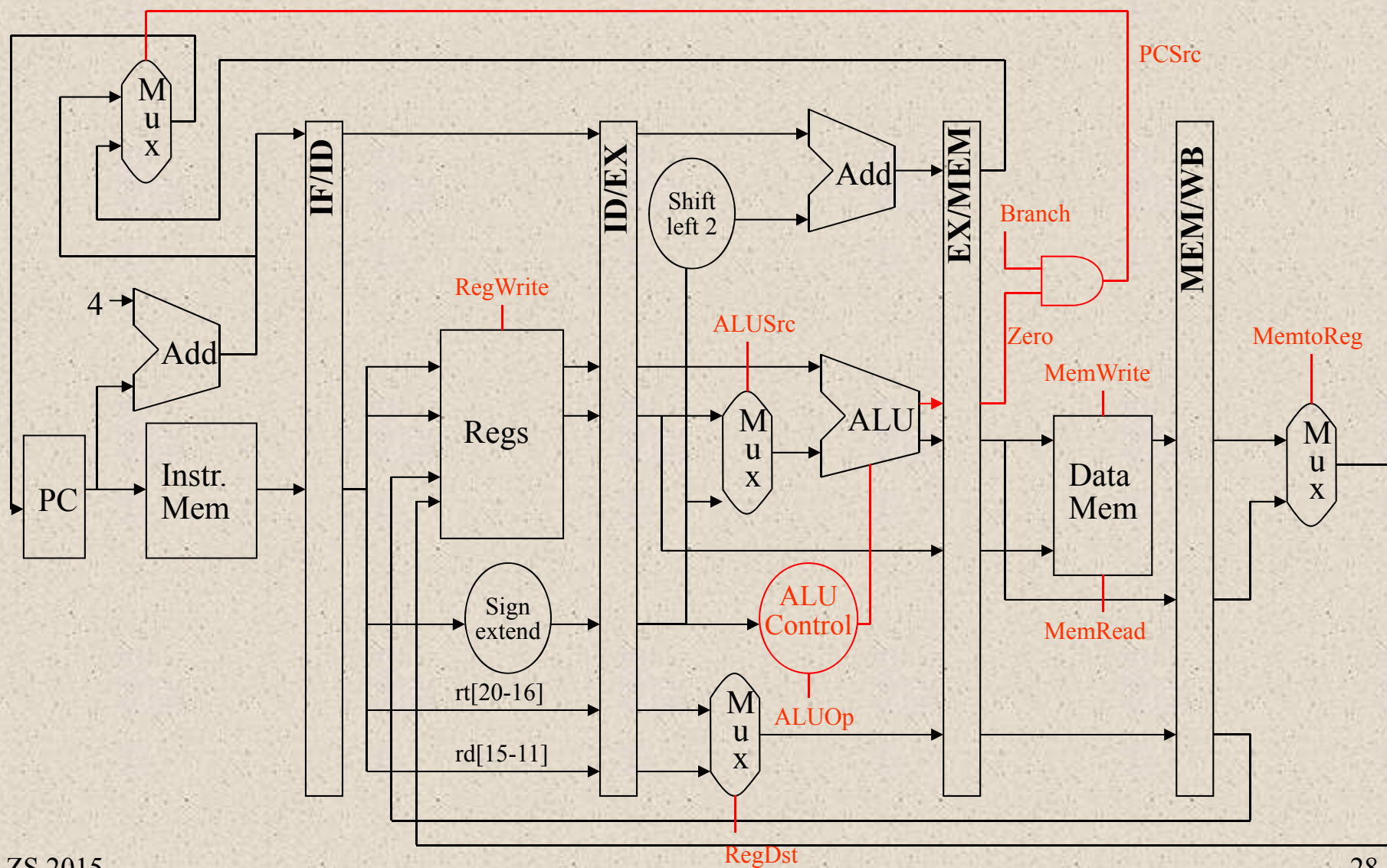


Diagram jednoho cyklu (cykl 6)



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

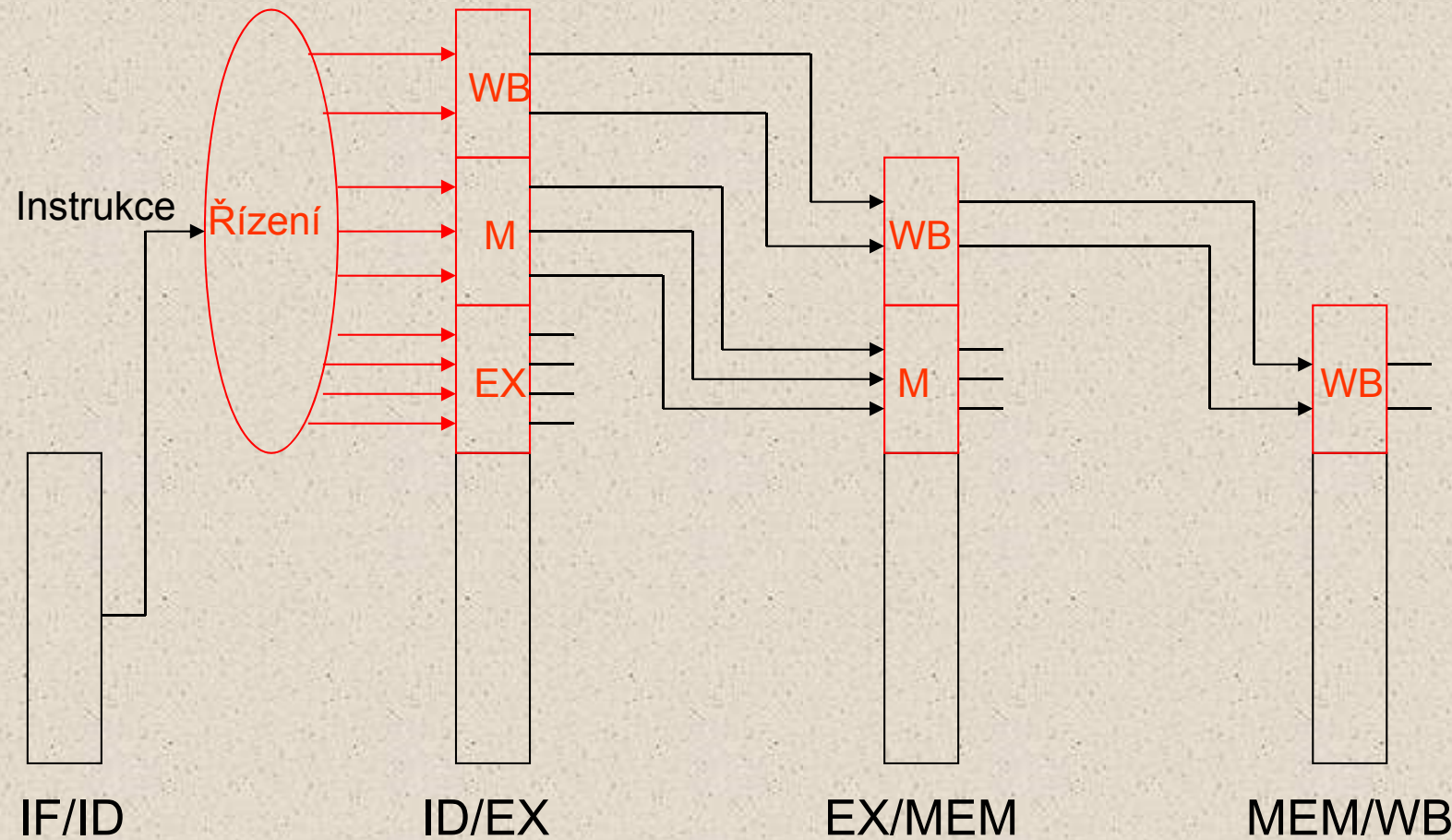
Řídící linky

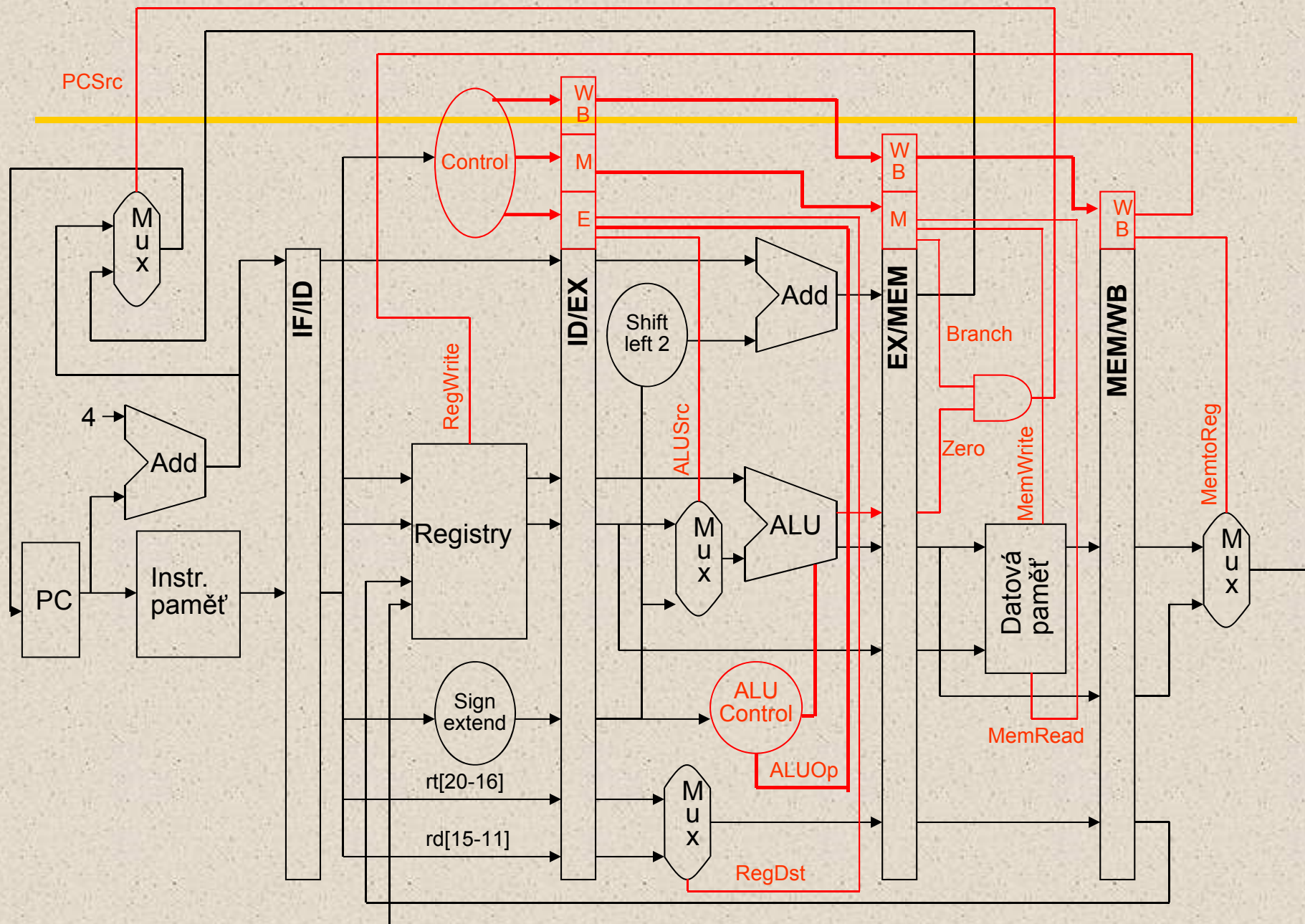
Instrukce	Řídící linky prováděcích stupňů				Řídící linky přístupu do paměti			Řídící linky zpětného zápisu	
	Reg Dst	ALU Op1	ALU Op2	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem 2Reg
R-formát	1	1	0	0	0	0	0	1	0
Lw	0	0	0	1	0	1	0	1	1
Sw	x	0	0	1	0	0	1	0	x
Beq	x	0	1	0	1	0	0	0	x

Implementace řízení

- 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






Příklad posloupnosti instrukcí

Příklad zpracovávané posloupnosti instrukcí:

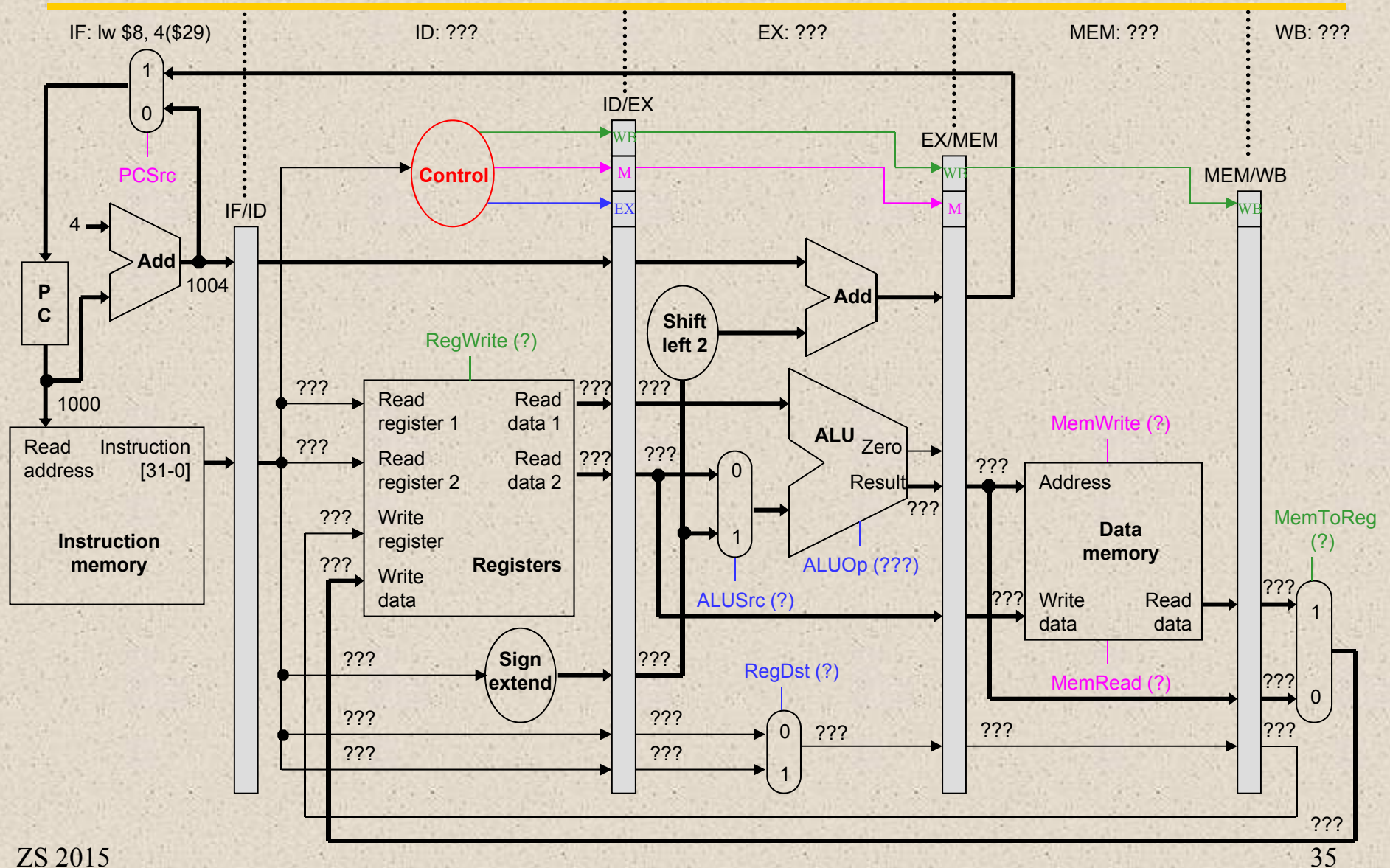
Adresy
dekadicky



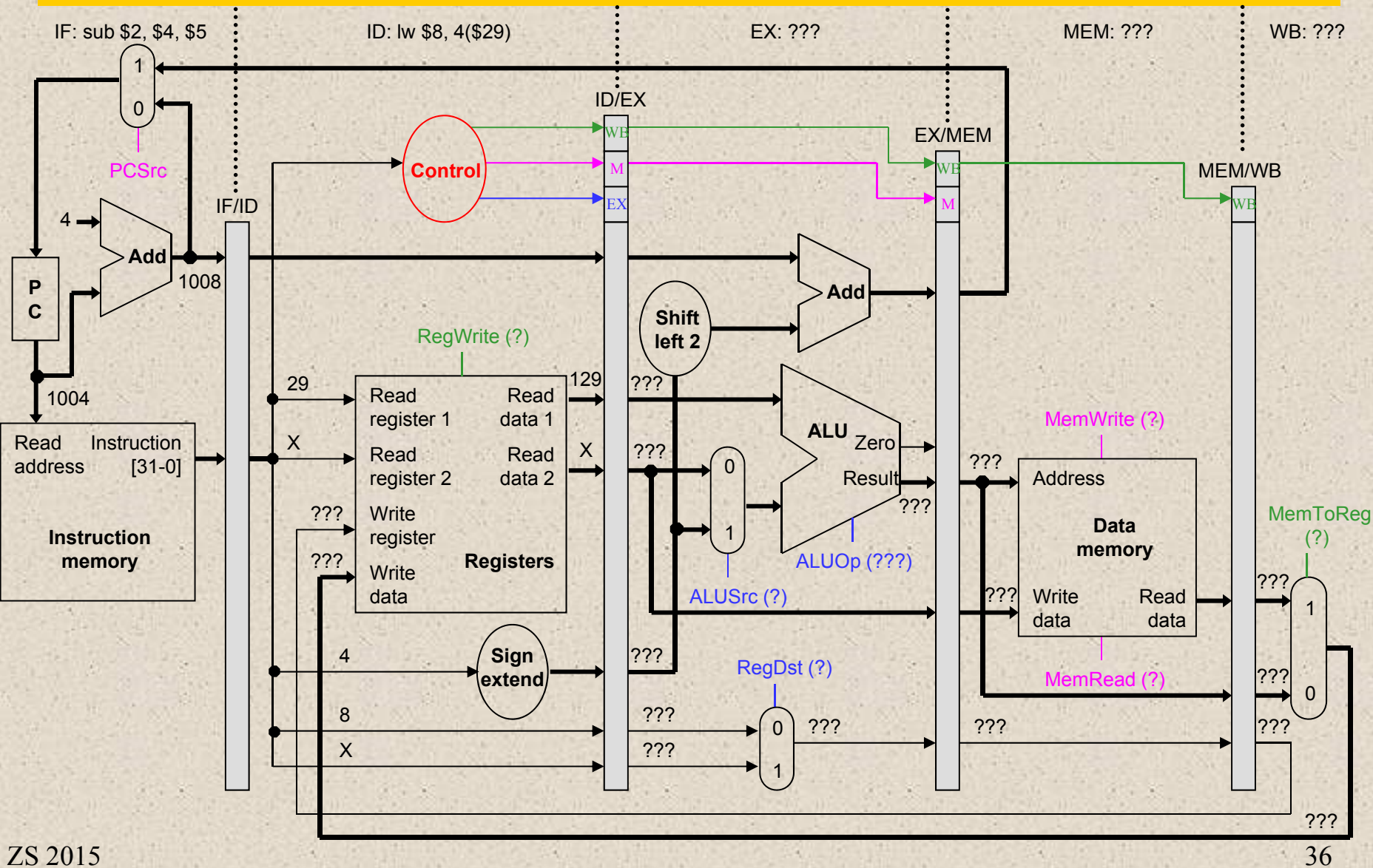
1000:	lw	\$8, 4(\$29)
1004:	sub	\$2, \$4, \$5
1008:	and	\$9, \$10, \$11
1012:	or	\$16, \$17, \$18
1016:	add	\$13, \$14, \$0

- 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.

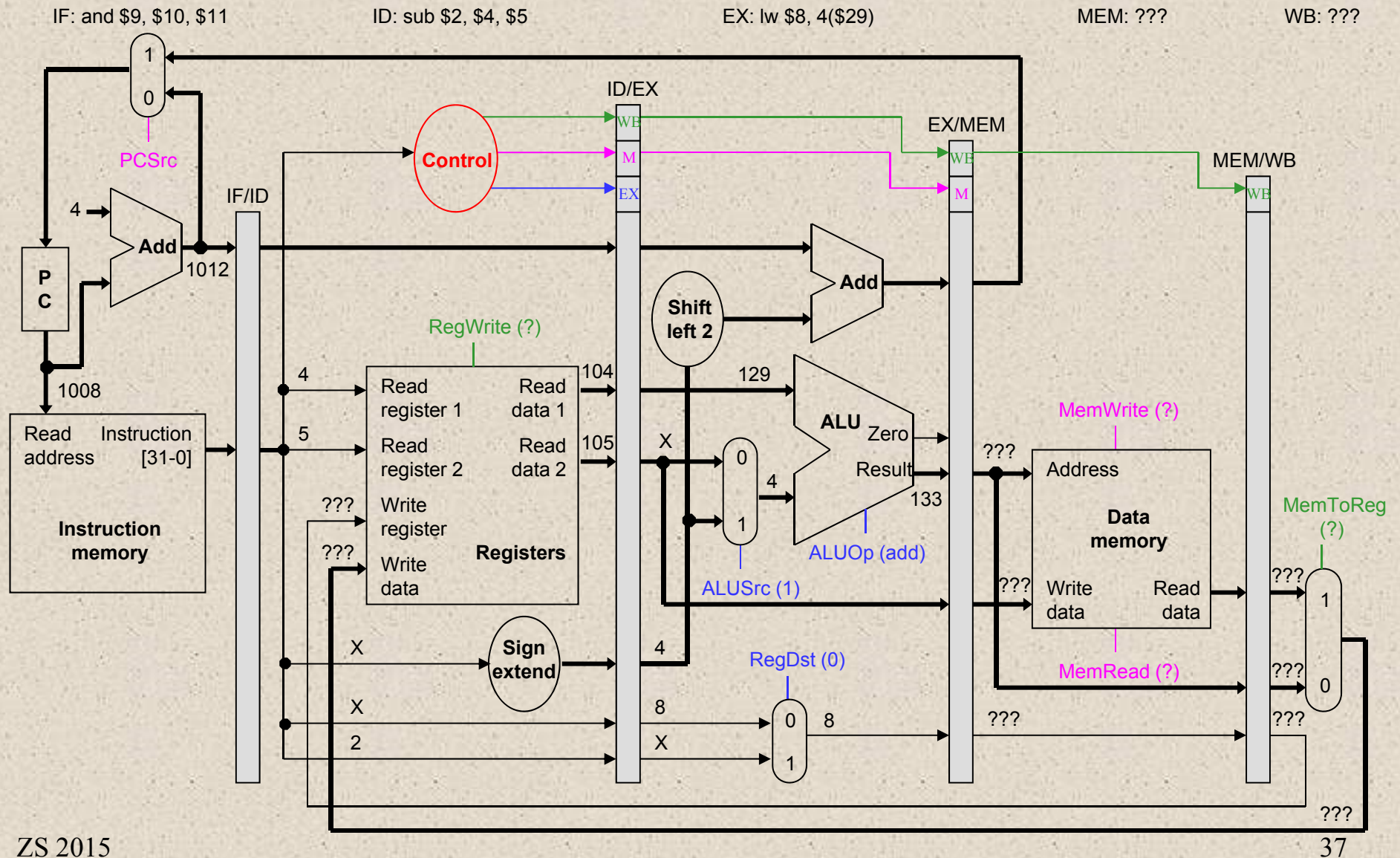
Cykl 1 (plnění)



Cykl 2



Cykl 3



Cykl 4

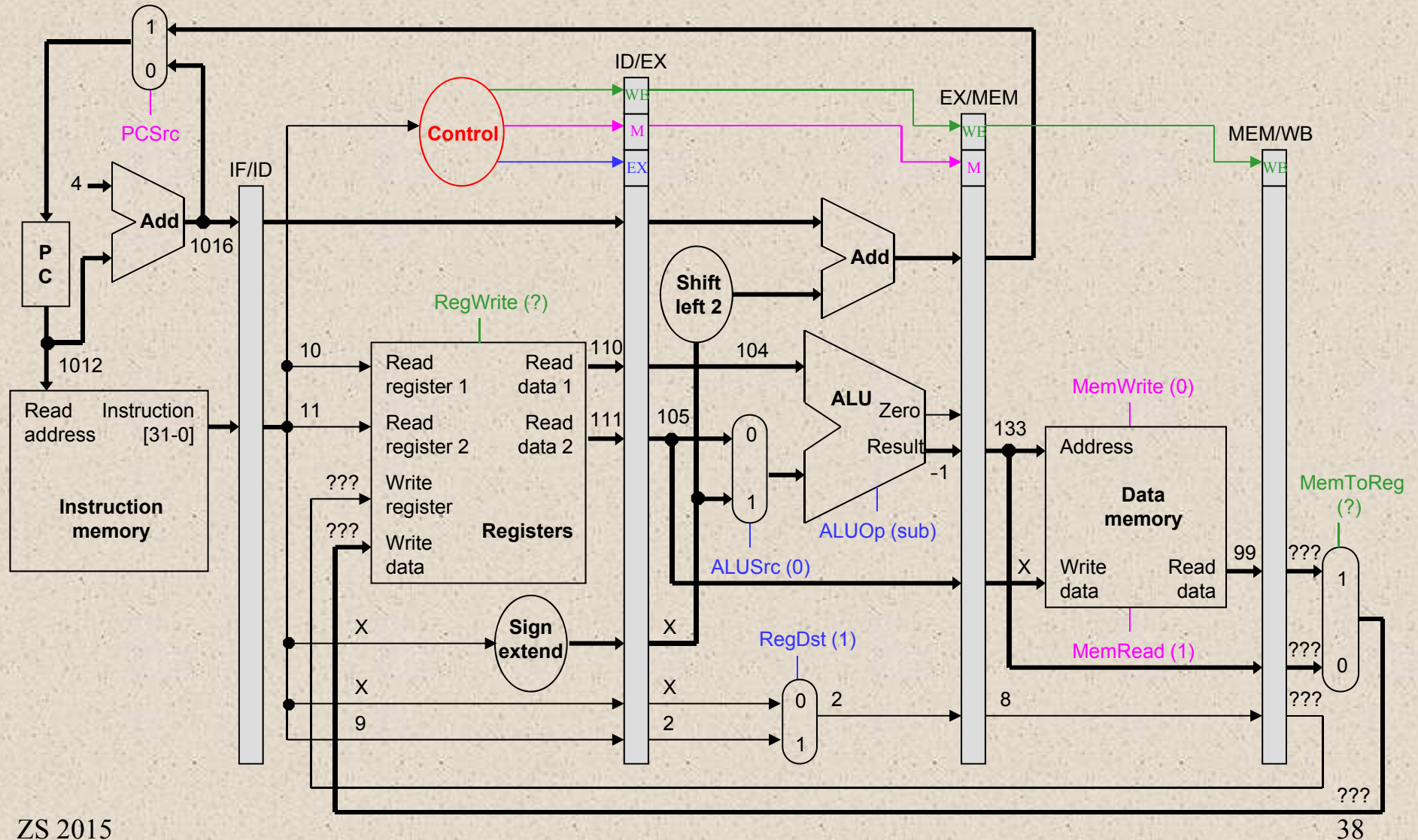
IF: or \$16, \$17, \$18

ID: and \$9, \$10, \$11

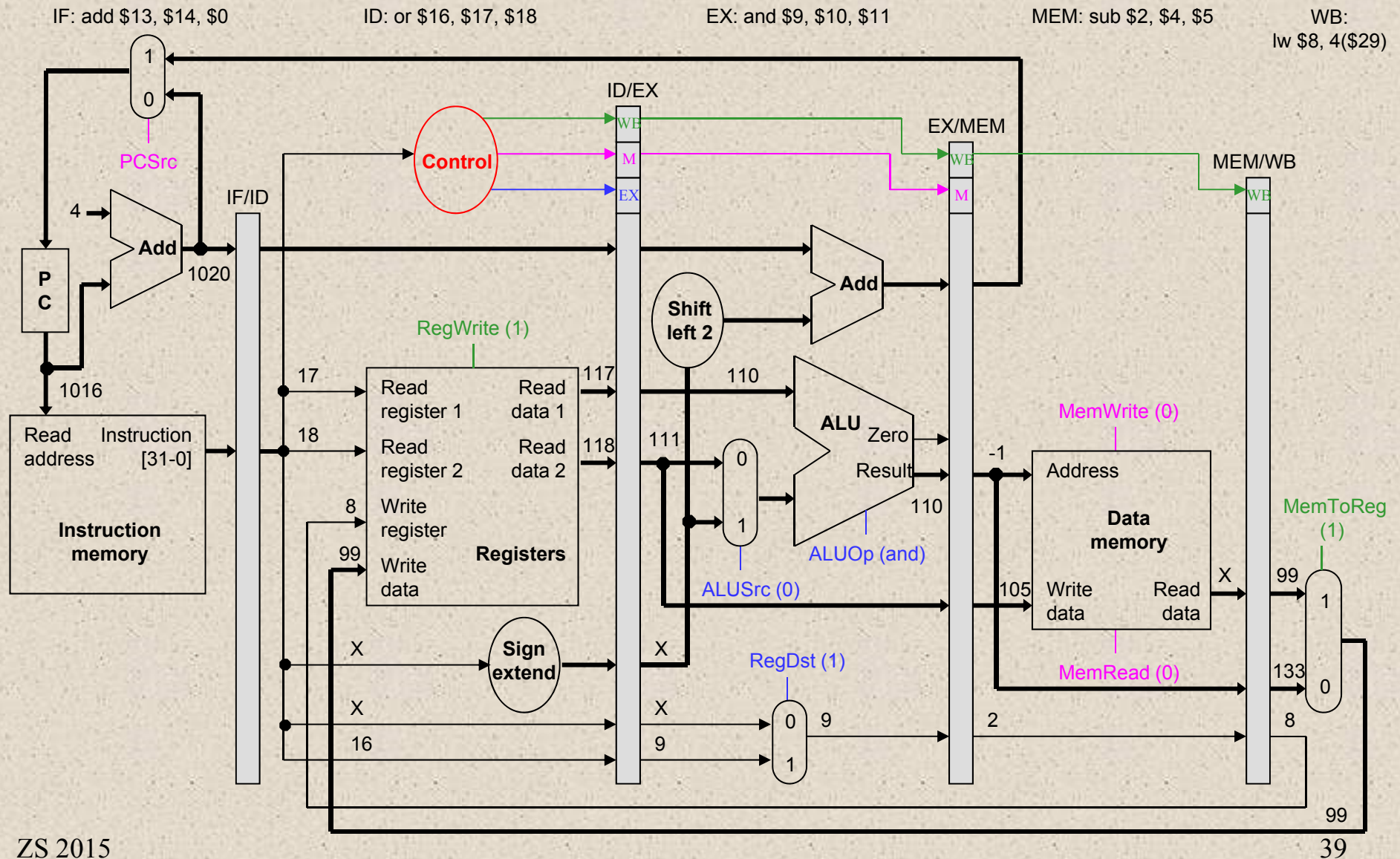
EX: sub \$2, \$4, \$5

MEM: lw \$8, 4(\$29)

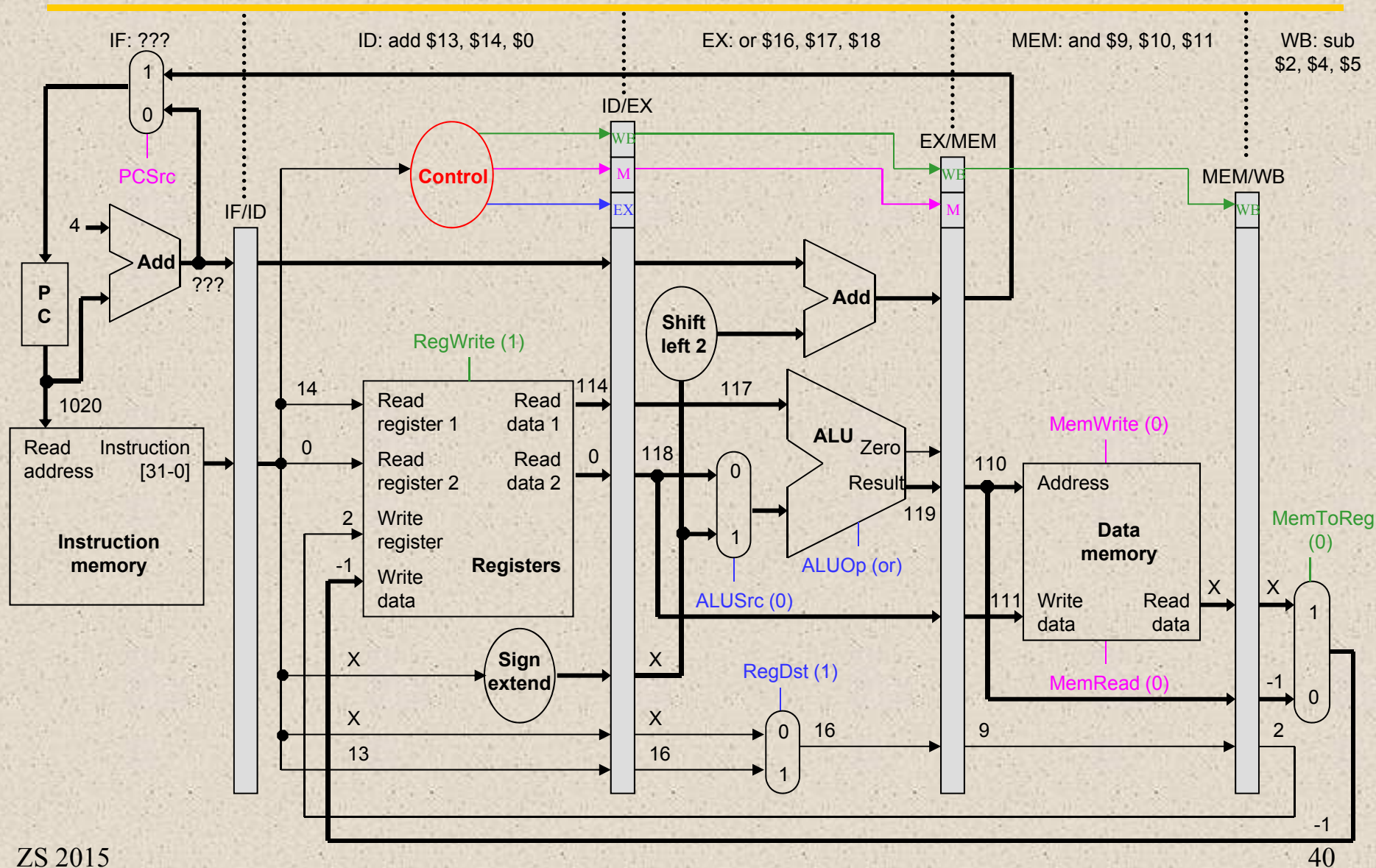
WB: ???



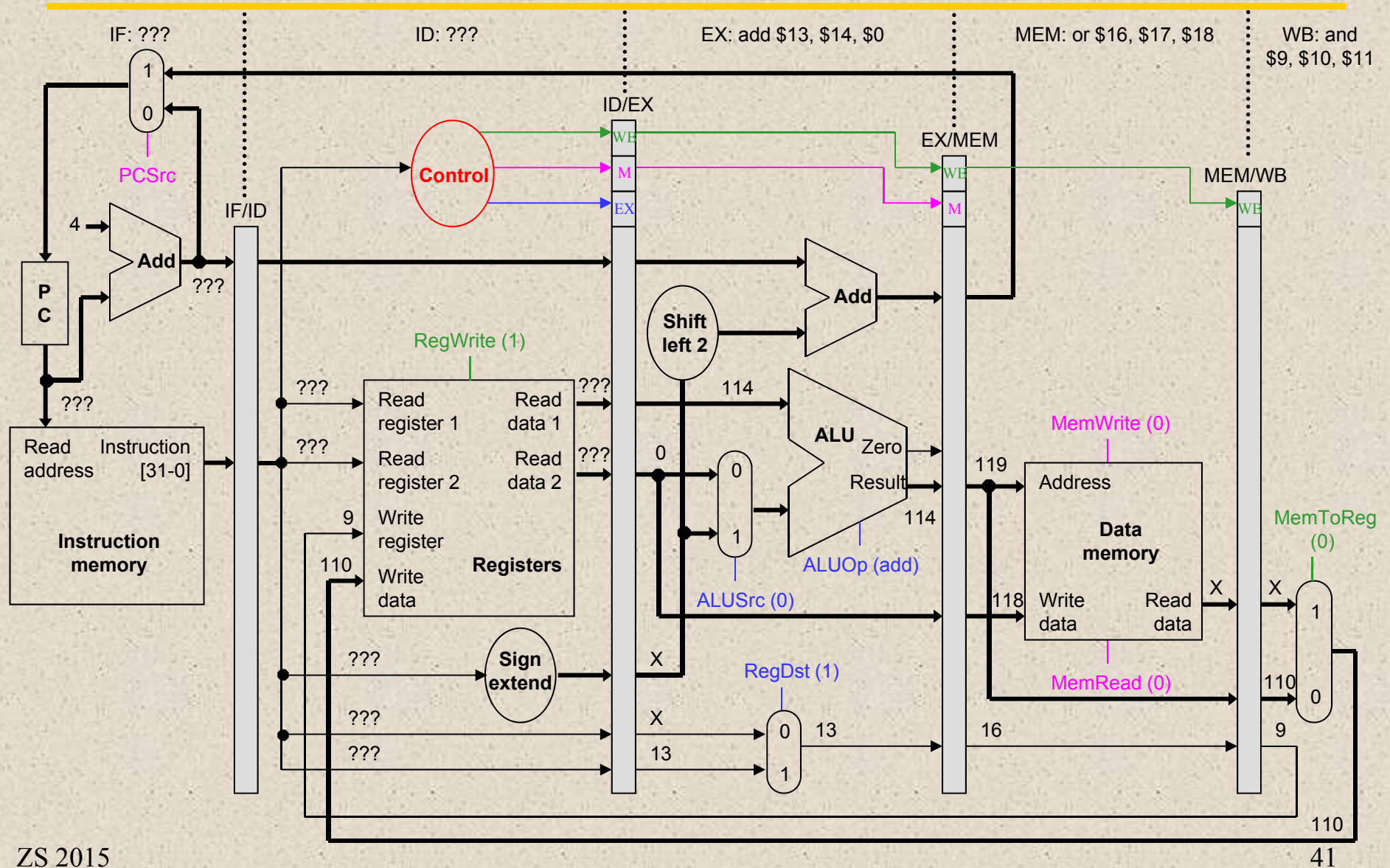
Cykl 5 (zaplněno)



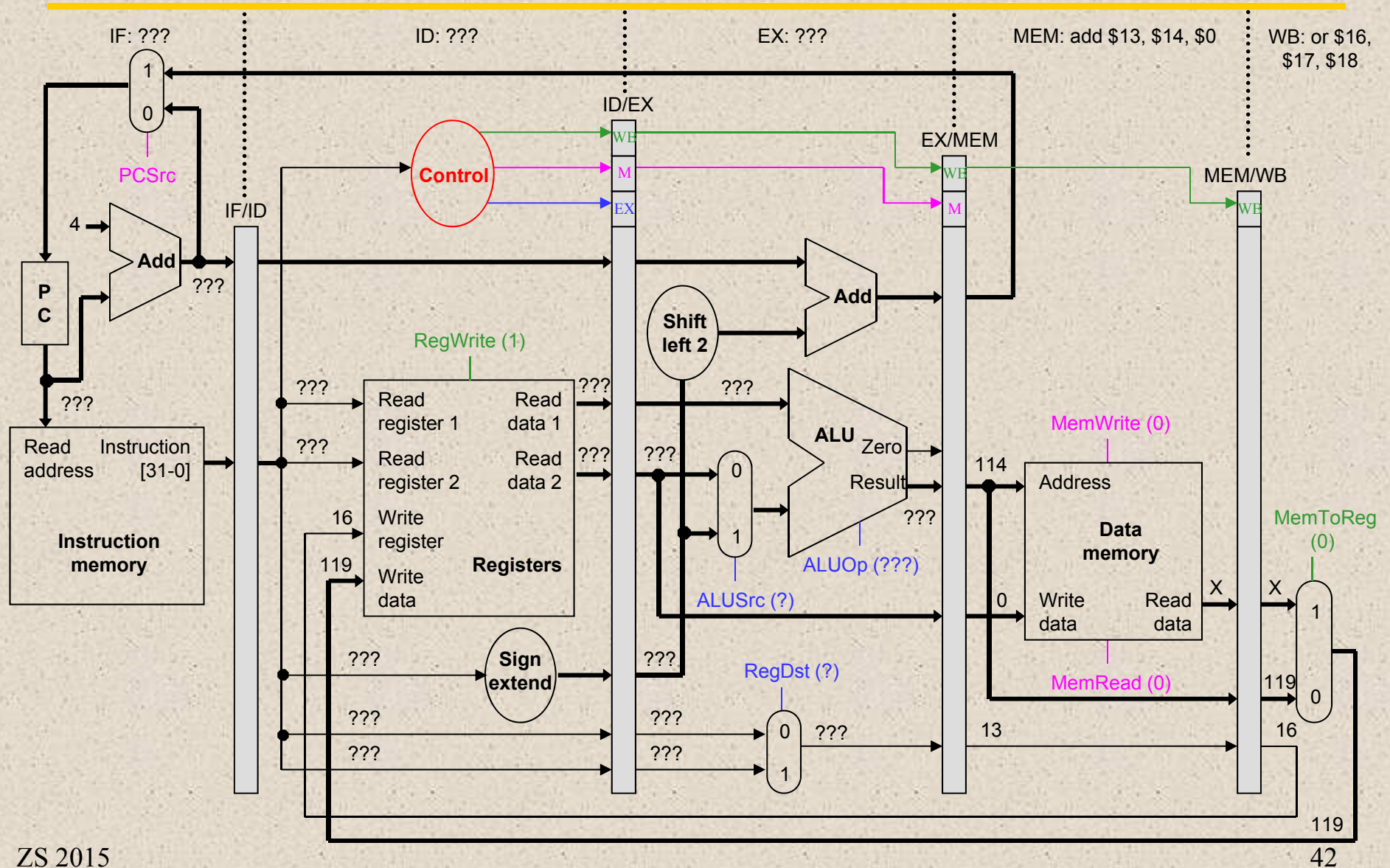
Cykl 6 (vyprazdňování)



Cykl 7



Cykl 8



Cykl 9

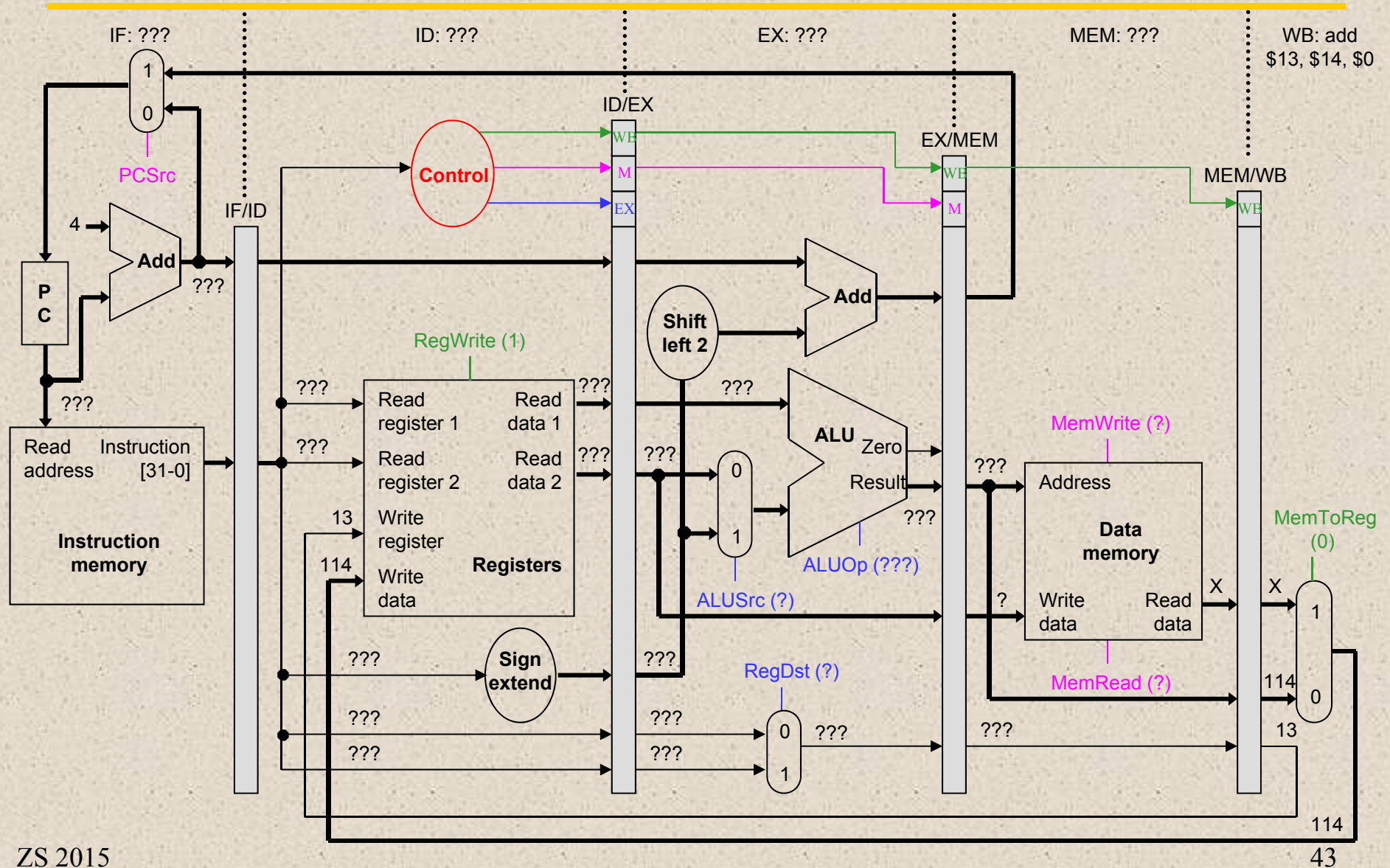
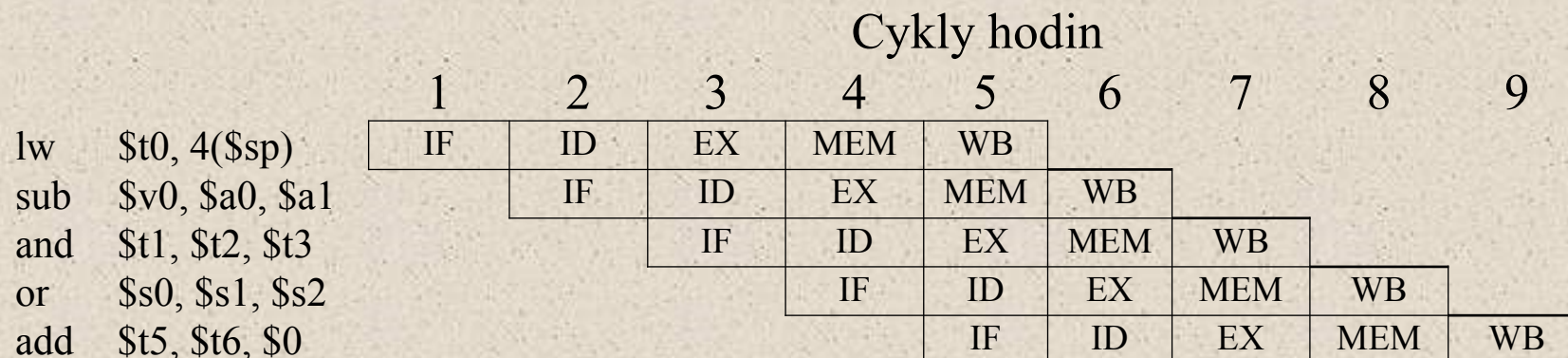


Diagram překrývání



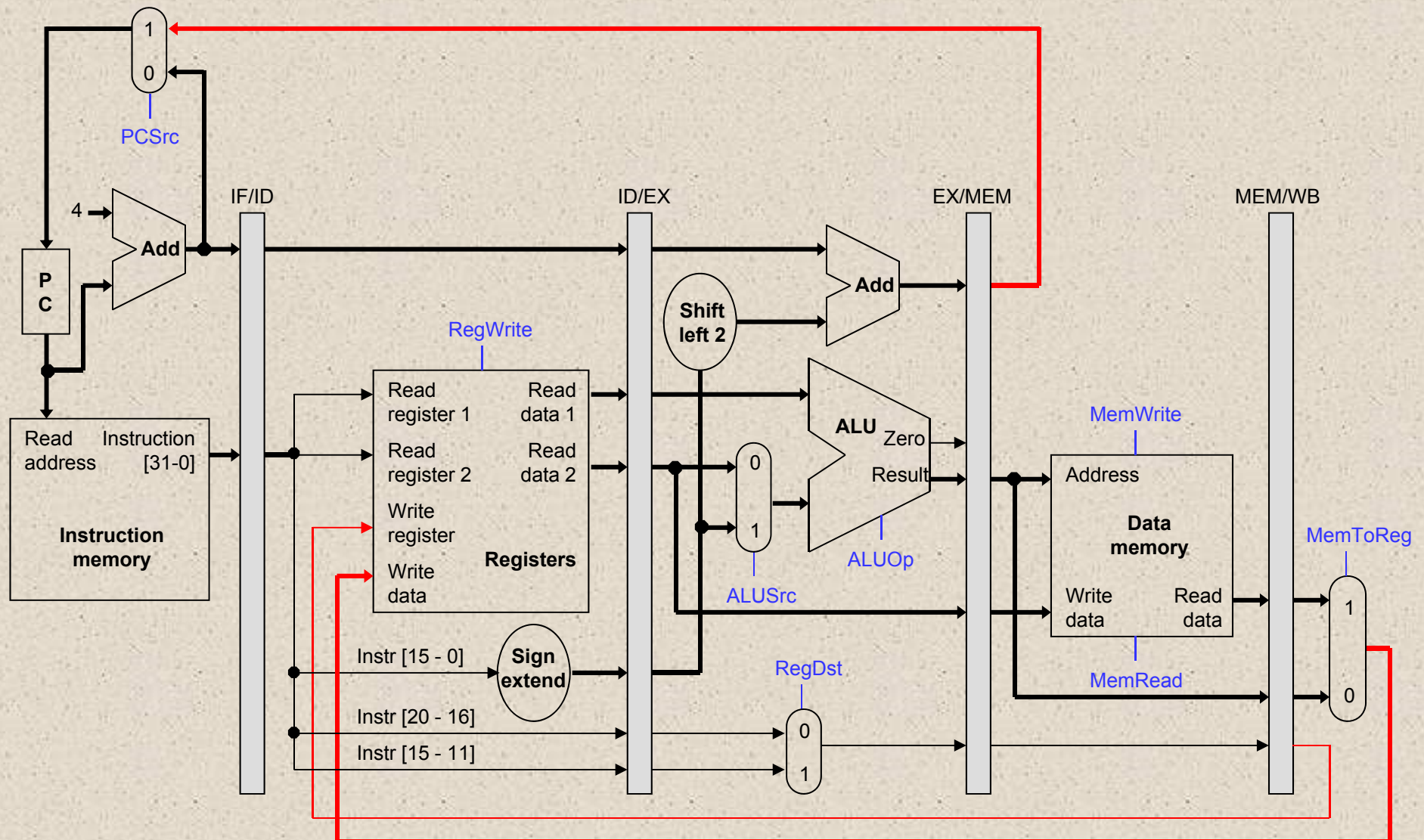
Porovnejte devět posledních snímků s diagramem nahoře.

- 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

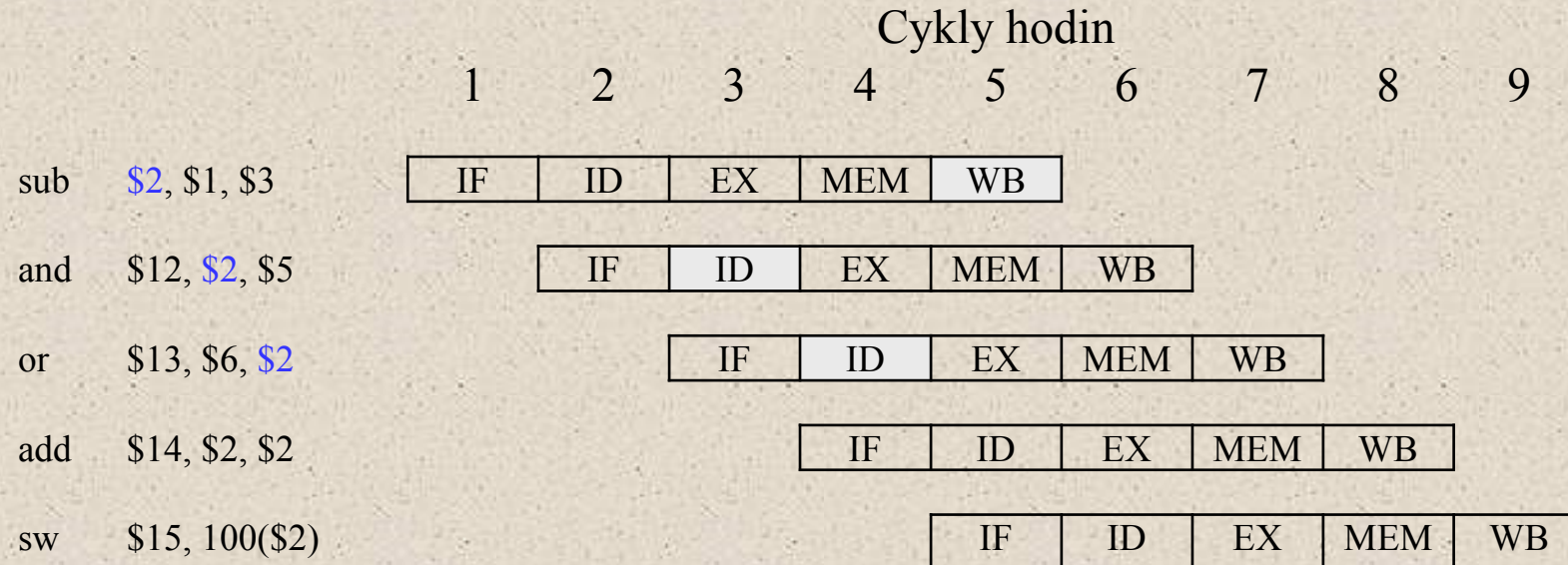
- Instrukce v příkladu jsou **nezávislé**.
 - Každá instrukce čte a modifikuje úplně jiné registry.
 - Taková posloupnost se snadno ošetřuje.
- Většina posloupností instrukcí v reálných programech ale **nejsou** nezávislé!

Příklad se závislostmi

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

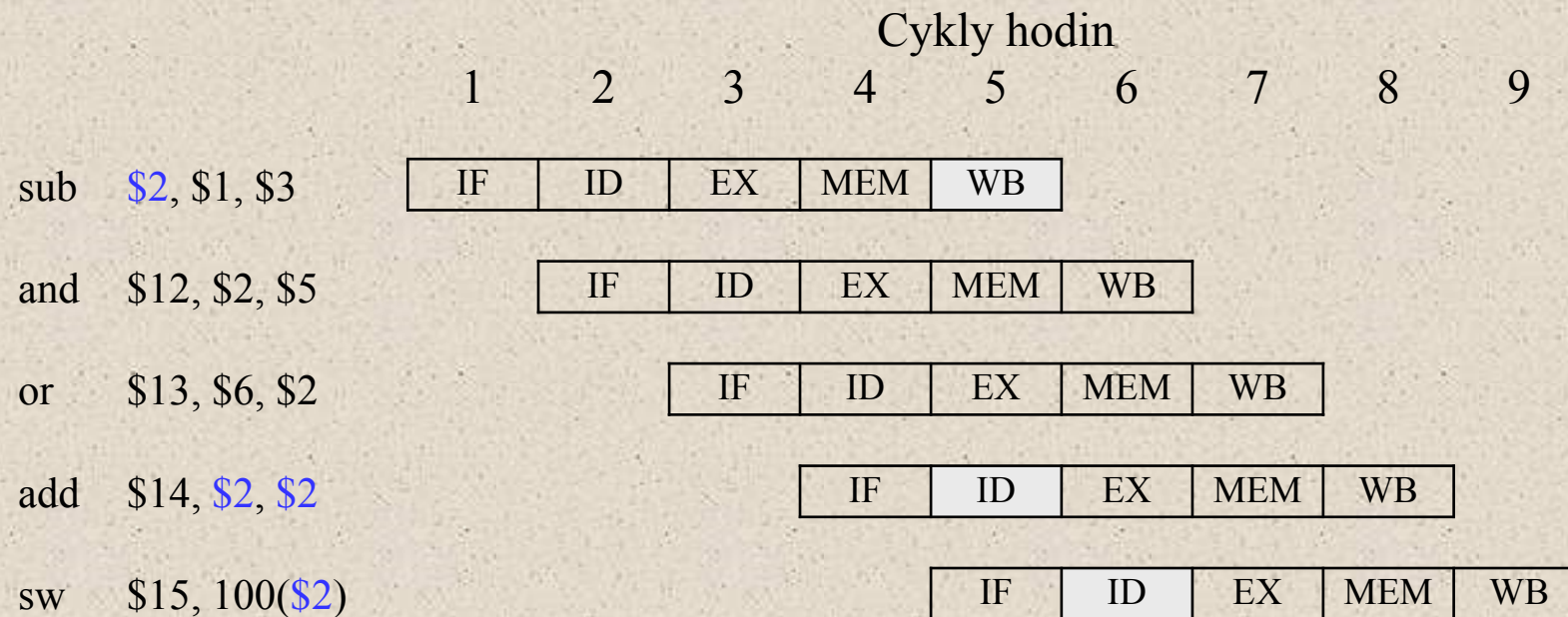
- Mezi instrukcemi je několik **závislostí**.
 - Prvá instrukce SUB ukládá hodnotu do \$2.
 - Tento registr je pak použit jako zdroj v dalších instrukcích.
- Pro jednocyklovou jednotku by to nepředstavovalo problém.
 - Každá instrukce je zakončena před započítáním další instrukce.
 - To zajistí, že instrukce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



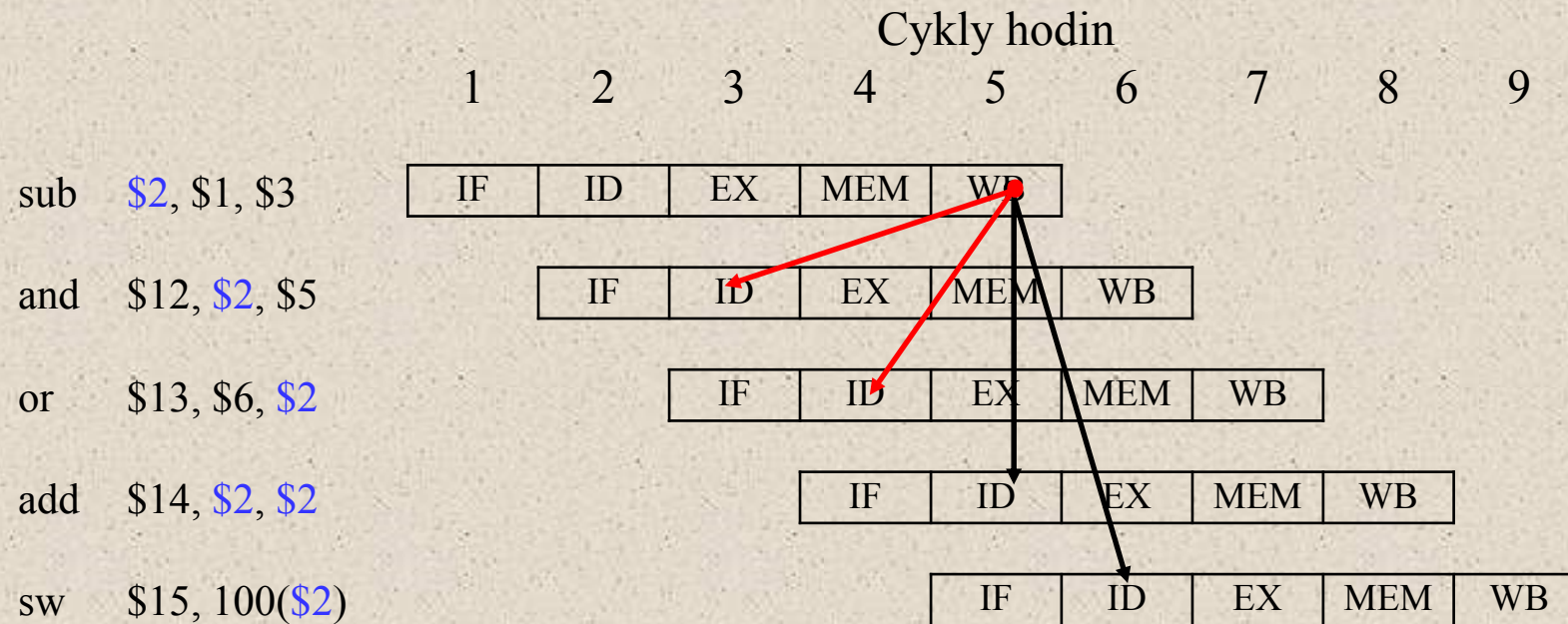
- Instrukce SUB zapisuje do registru \$2 až v cyklu 5. To způsobí dva **datové hazardy** v naší současné jednotce, pracující v režimu pipeline.
 - AND čte registr \$2 v cyklu 3. Protože SUB dosud nezapsala do \$2, bude přečtena *stará* hodnota \$2, nikoliv nová.
 - Podobně instrukce OR používá registr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

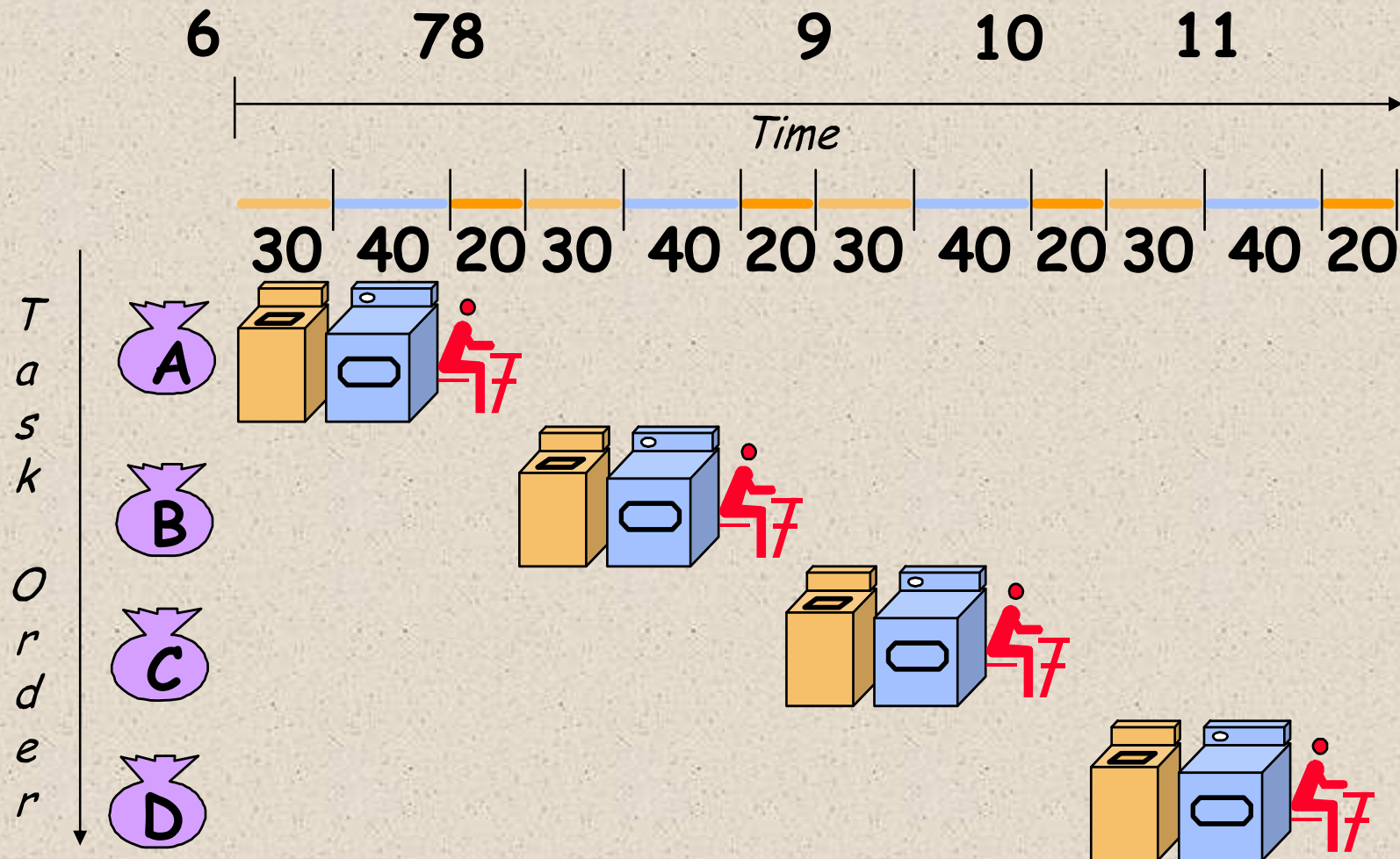
- **Strukturní**

- Různé instrukce se snaží používat současně stejné funkční jednotky (např. paměť, registrovou sadu)
- **Řešení**: duplikovat hardware

- **Řídící** (větvení)

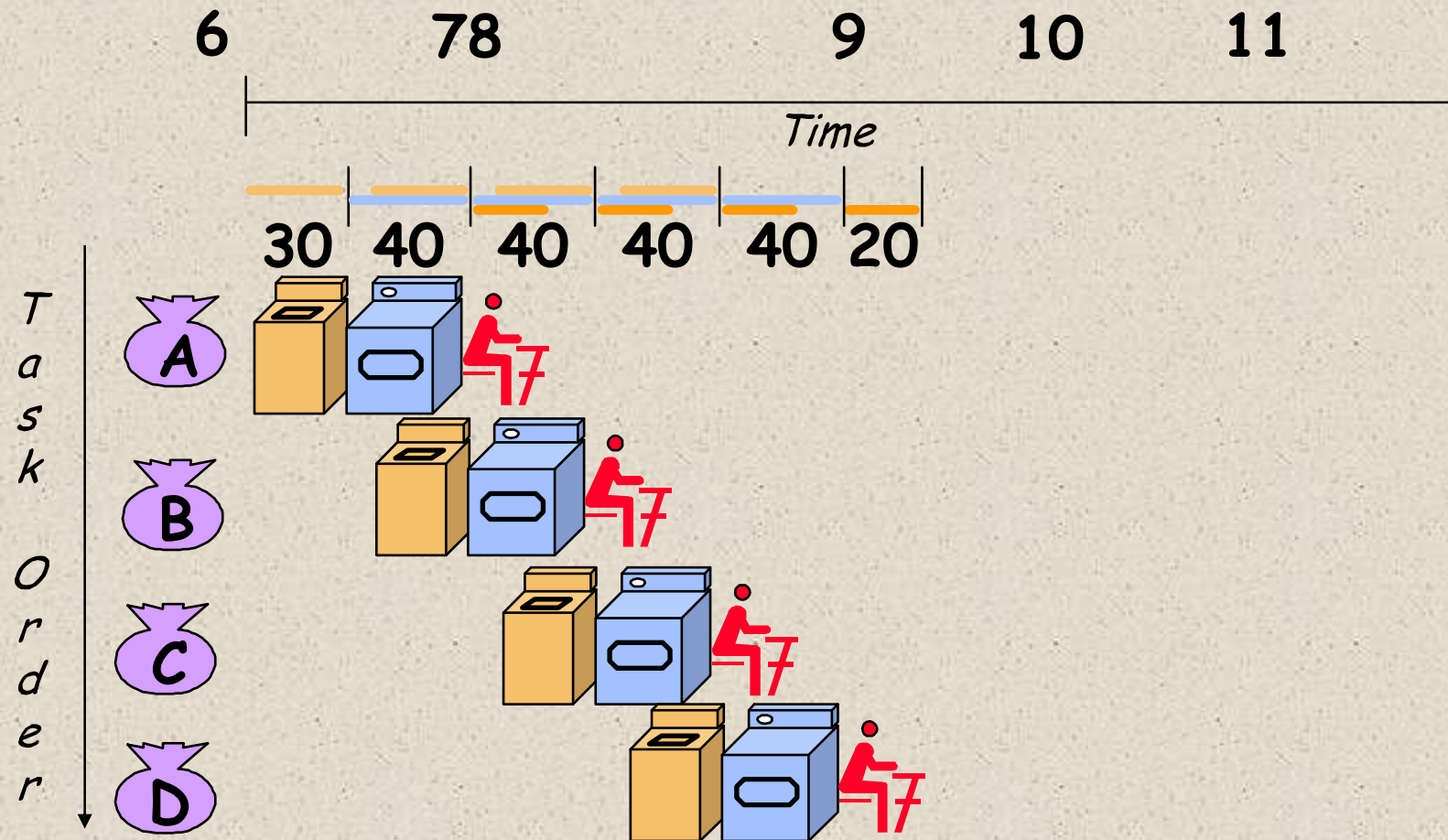
- Cílová adresa je známa až na konci třetího cyklu => POZASTAVENÍ
- **Řešení**
 - Predikce (statická a dynamická): Smyčky
 - Opoždění instrukcí větvení (branches)

Sekvenční prádelna



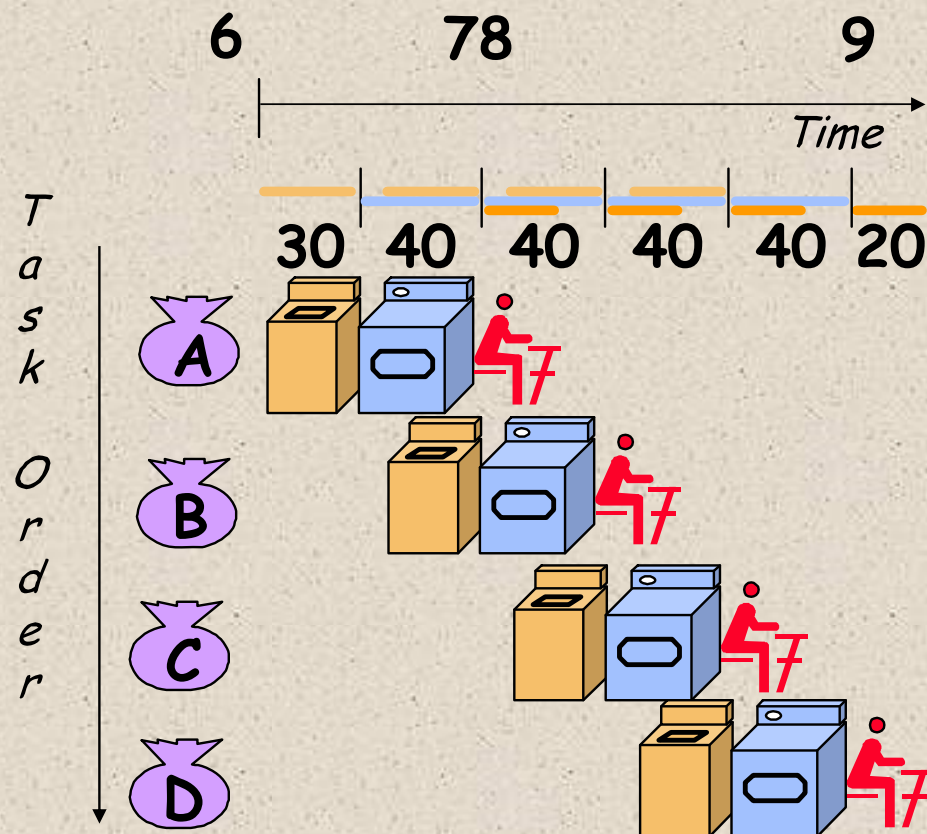
- Sekvenční praní 4 dávek trvá 6 hodin
- Jak dlouho by prali, kdyby znali pipelining?

Prádelna v režimu pipeline



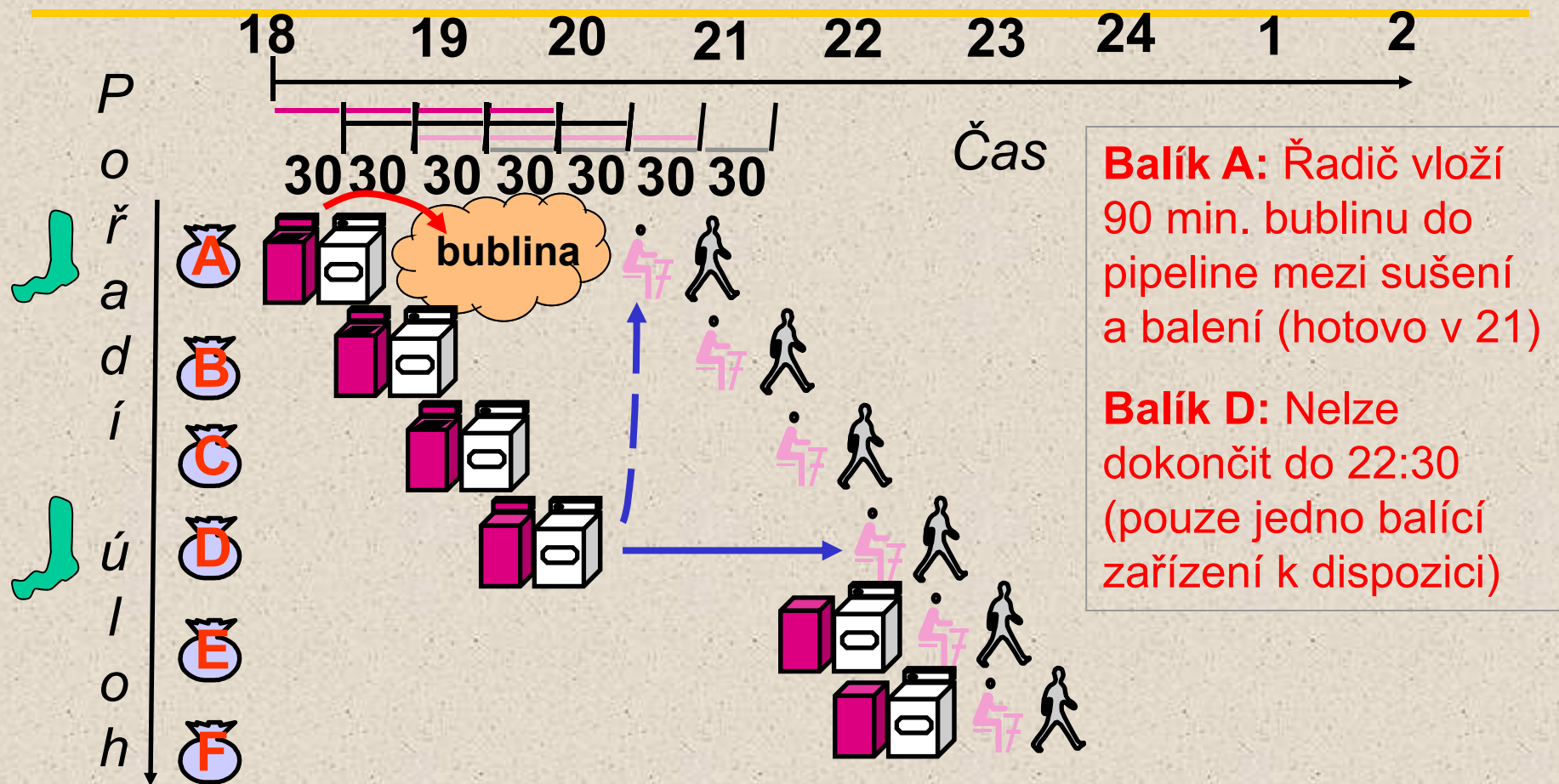
- Praní v režimu pipeline trvá 3.5 hodiny pro 4 dávky

Lekce pipelingu



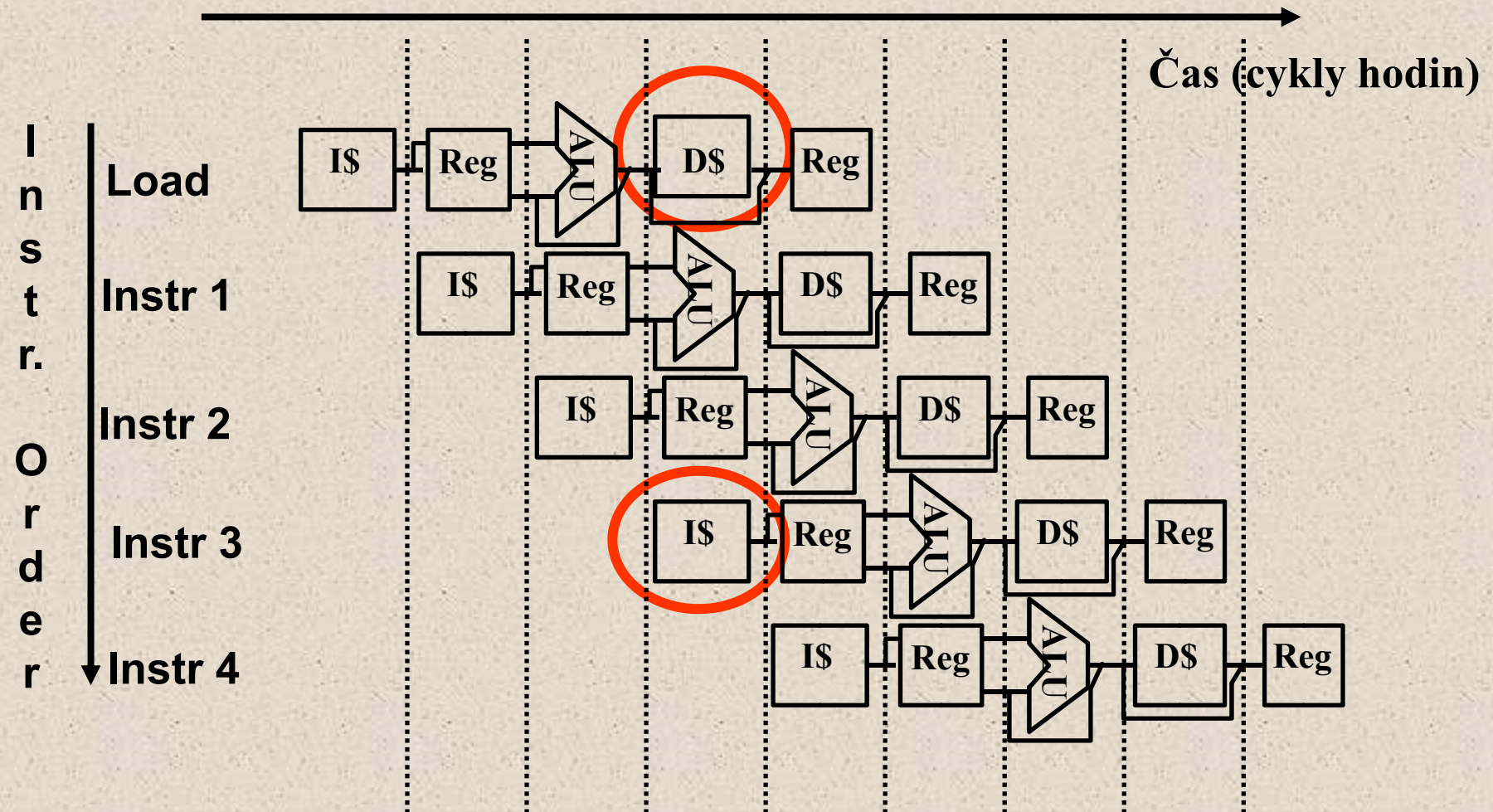
- Pipelining nezlepšuje **latenci** jednotlivých úloh, ale **propustnost** celé dávky
- Rychlost je omezena **nejpomalejším** stupněm
- **Více úloh** se zpracovává paralelně
- Potenciální urychlení = **počtu stupňů pipeline**
- Nevyvážená délka stupňů redukuje rychlost
- Čas pro **naplnění** a **doběh**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

Hazardy v pipeline (příklad)



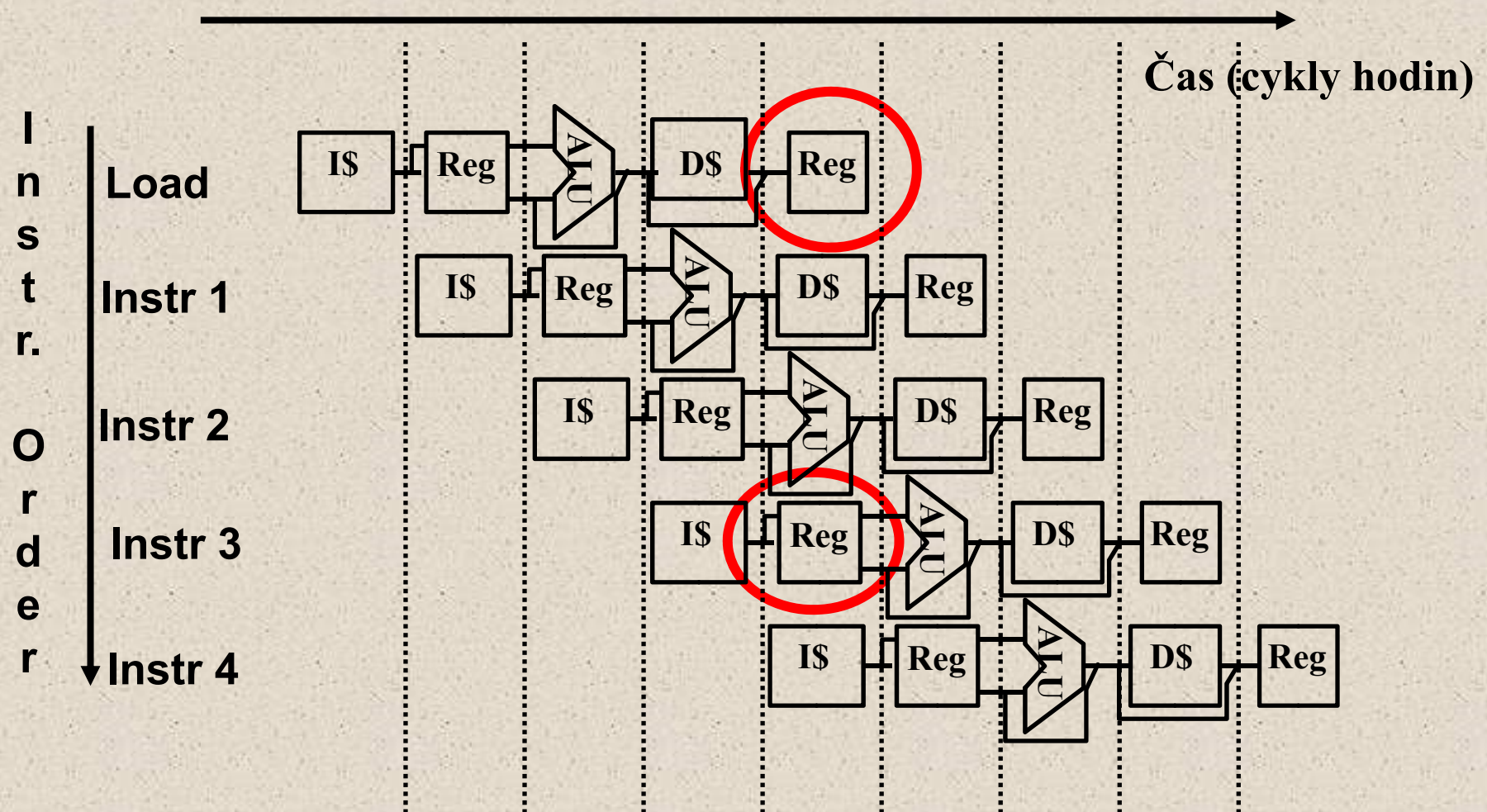
- Případ **zelených ponožek**: jedna v **A** druhá v **D**
- **D** závisí na **A** → pozastavení dokud je balička obsazena

Strukturní hazard 1: Jediná paměť



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 $\frac{1}{2}$ 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)

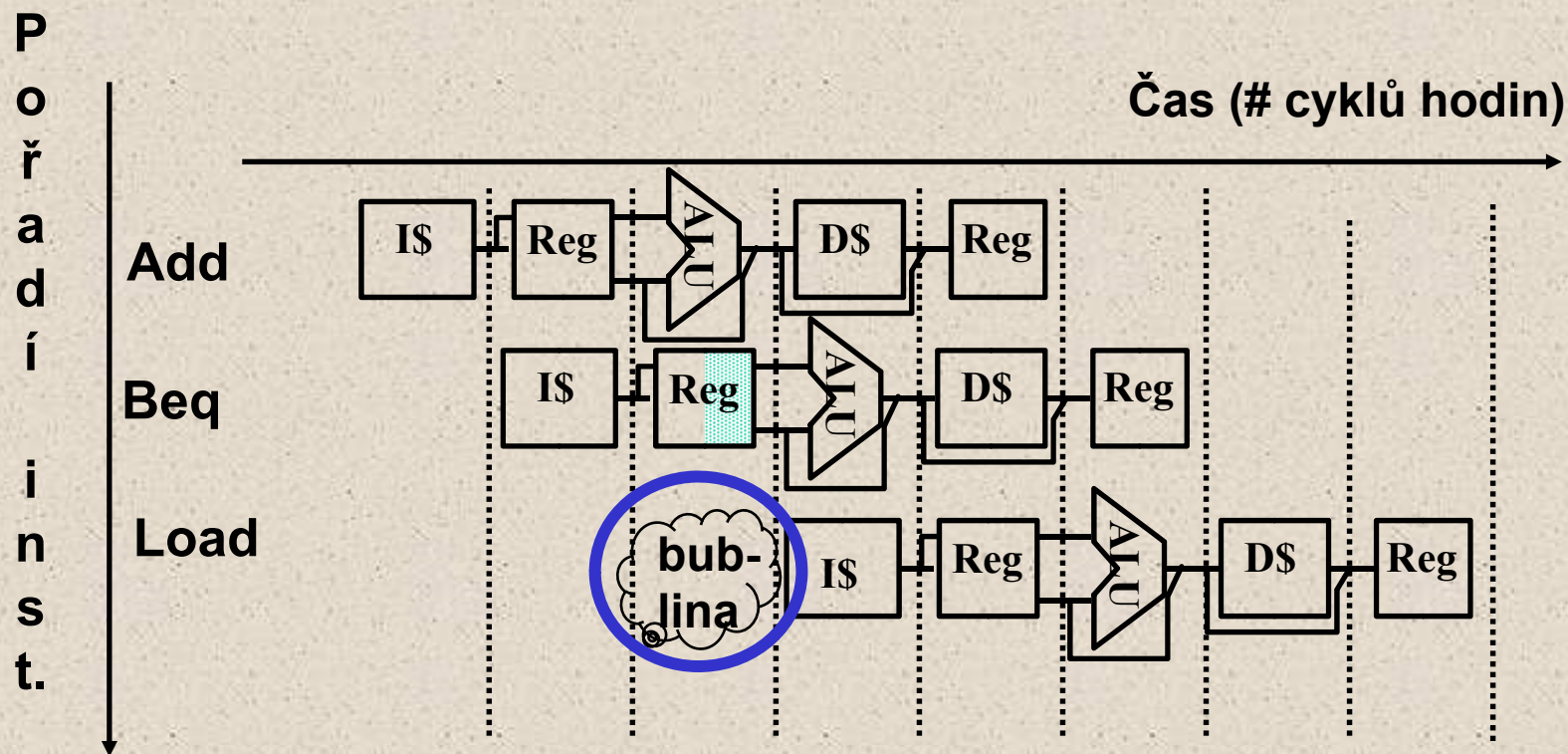
- Hardware ve stupni ALU, podporující větvení
 - *Vždy* se načtou dvě další instrukce ležící za skokem, ať se skok koná nebo nikoliv
- Jaké by bylo nejvhodnější provádění skoků:
 - jestliže se skok nebude konat, neztrácet čas a pokračovat normálně ve výpočtu
 - jestliže se skok koná, neprovádět žádnou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