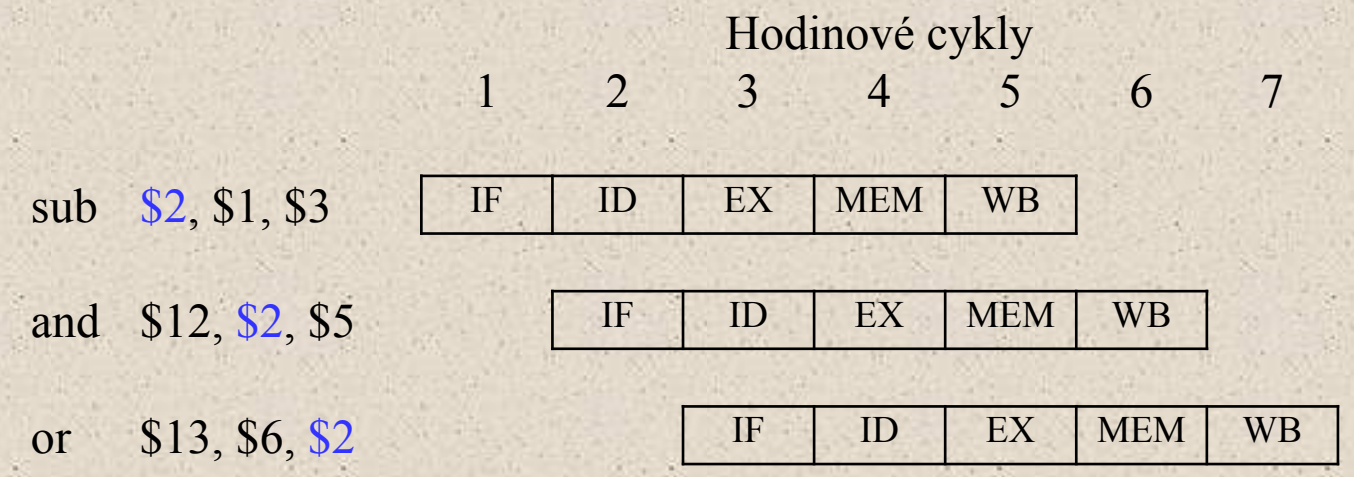
Úvod do organizace počítače

Pipelining - detaily

Detailní pohled na pipeline

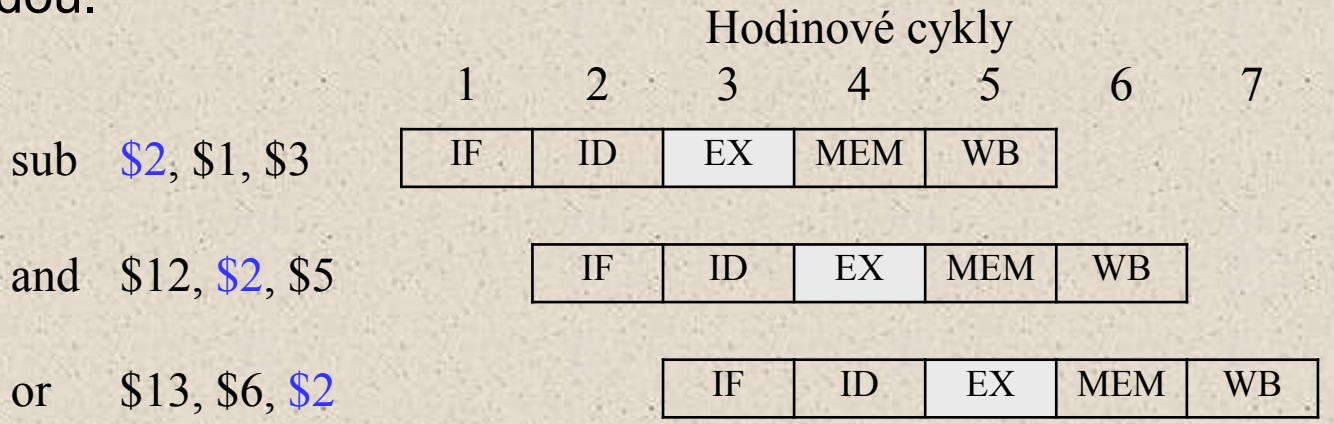
- Budeme se zabývat problémy, které způsobují datové hazardy v procesoru s pipeliningem a jak je eliminovat metodou, která se nazývá forwarding.



- 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?

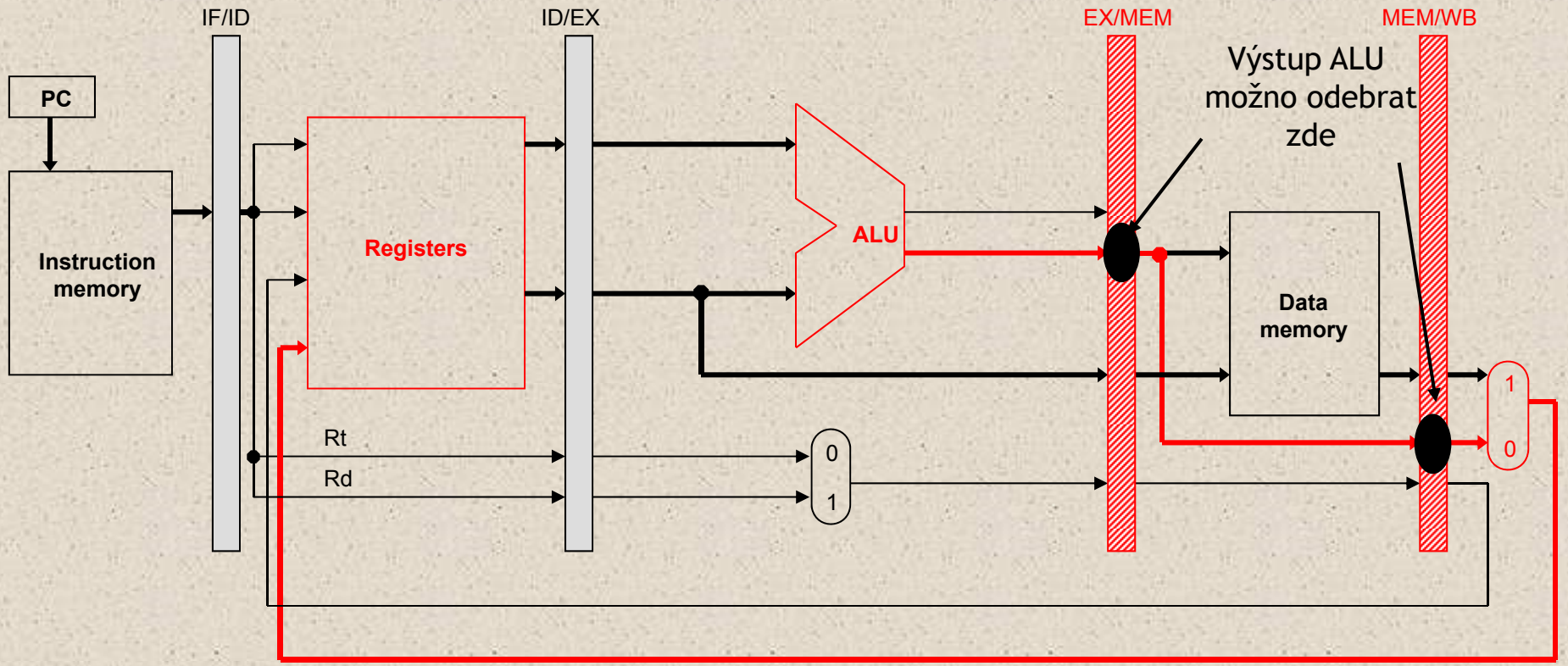
„Bypass“ registrového souboru

- 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.



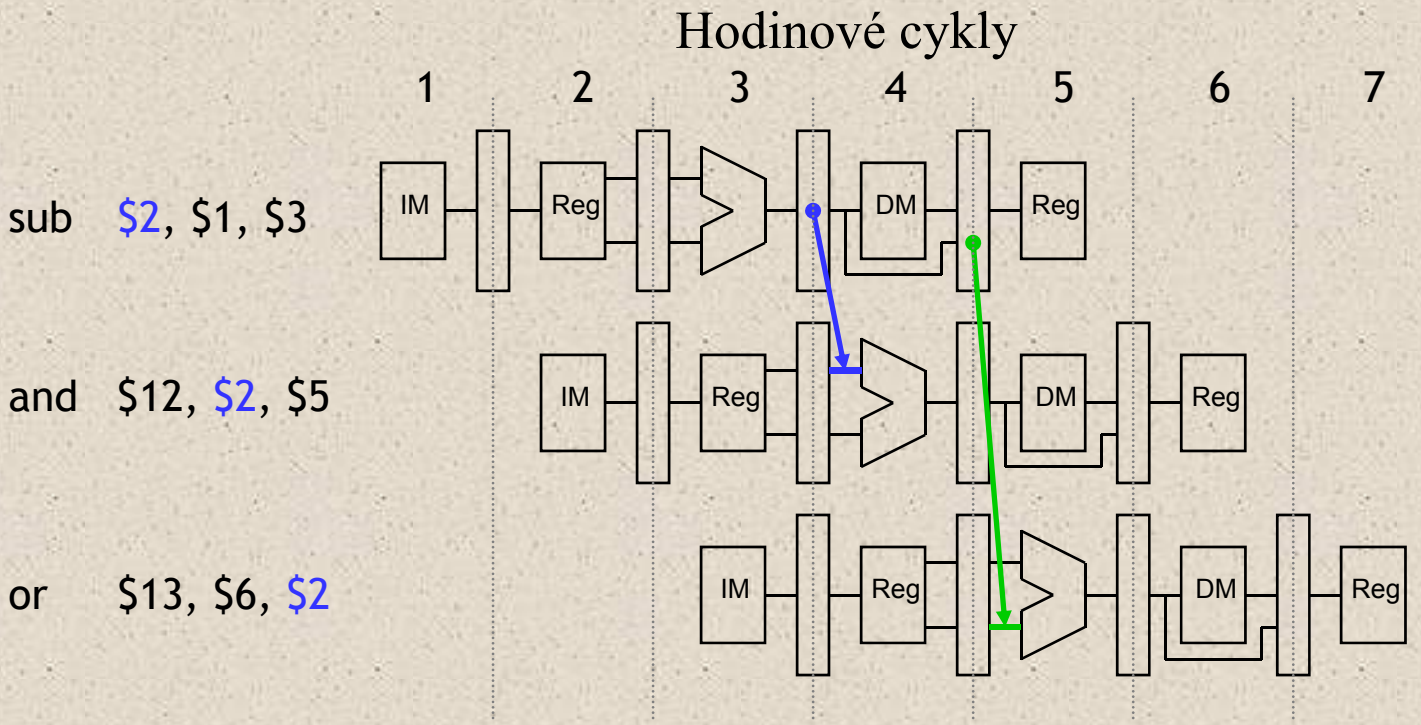
Kde odebrat výsledek ALU

- 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.



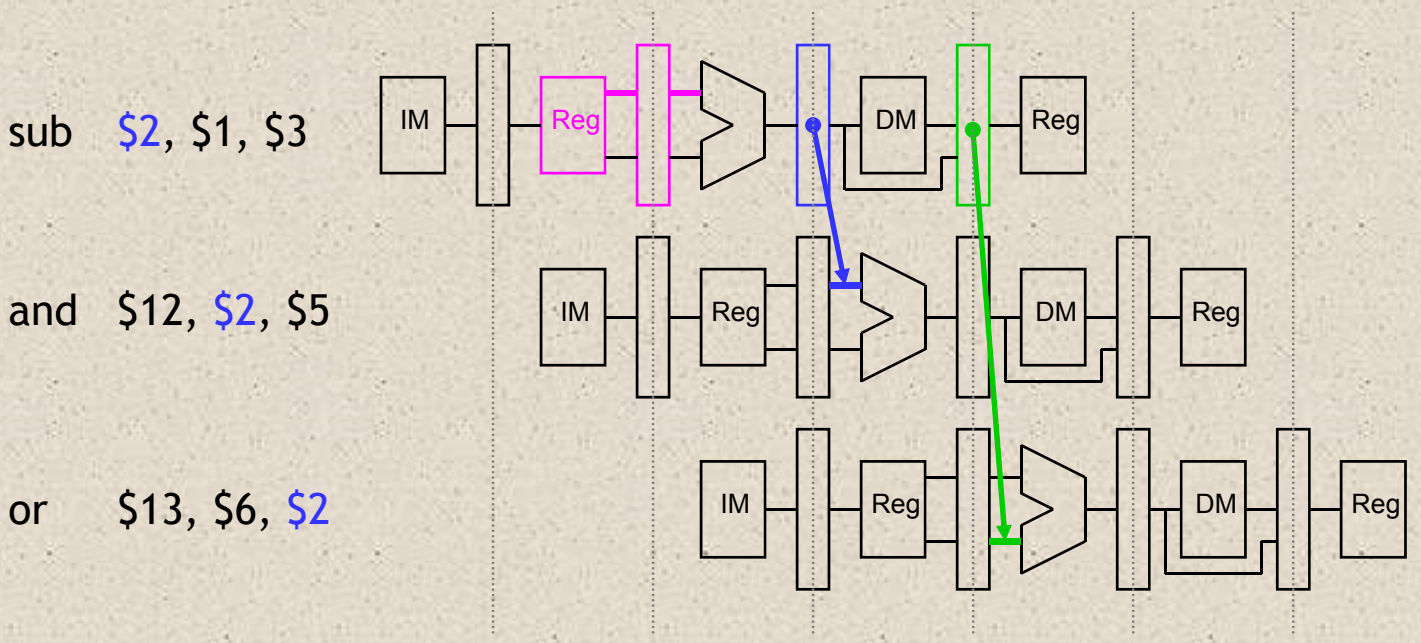
Forwarding

- 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.

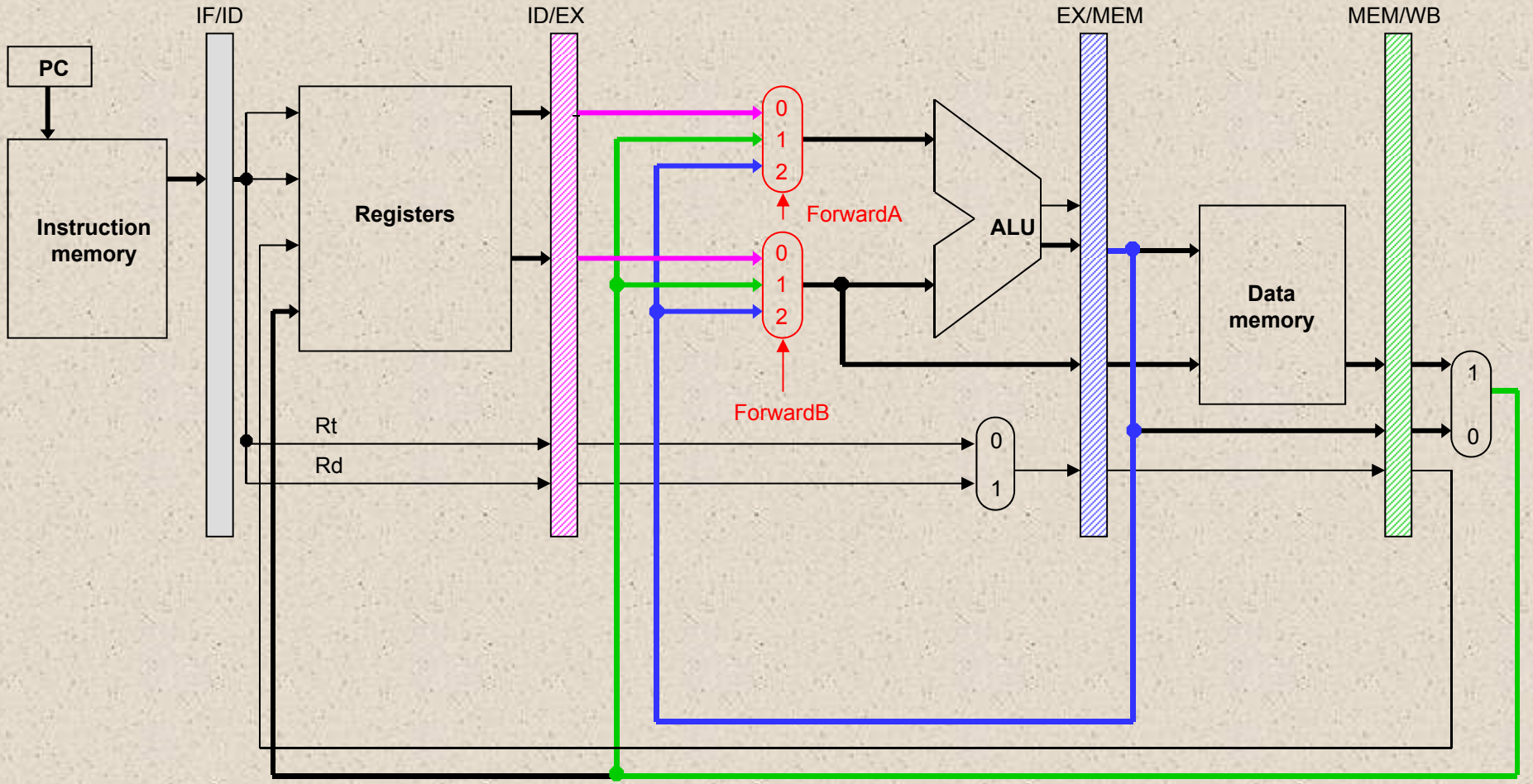


Návrh hardware pro forwarding

- 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.

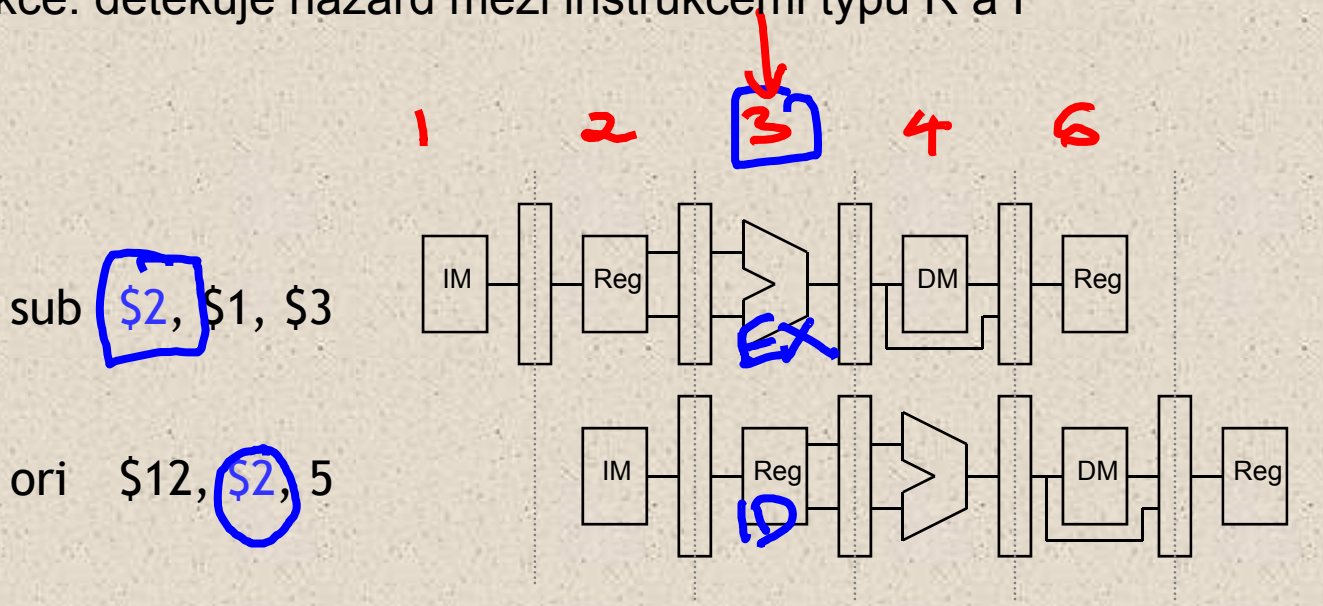


Zjednodušené datové cesty s multiplexery pro forwarding



Detekce hazardů EX/MEM

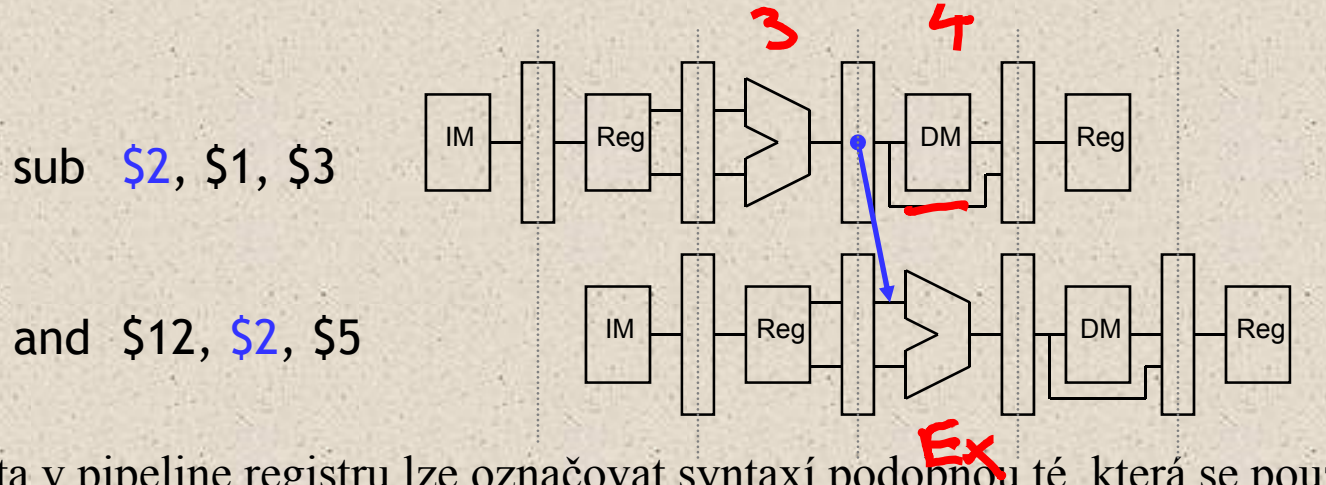
- Sekce: detekuje hazard mezi instrukcemi typu R a I



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

Eliminace datových hazardů EX/MEM

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



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

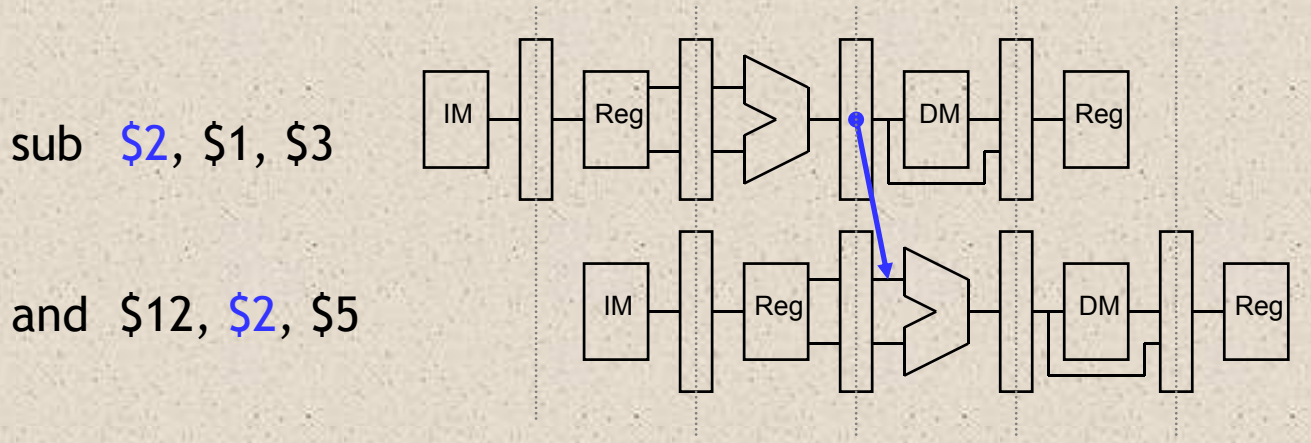
Rovnice datových hazardů EX/MEM

- 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



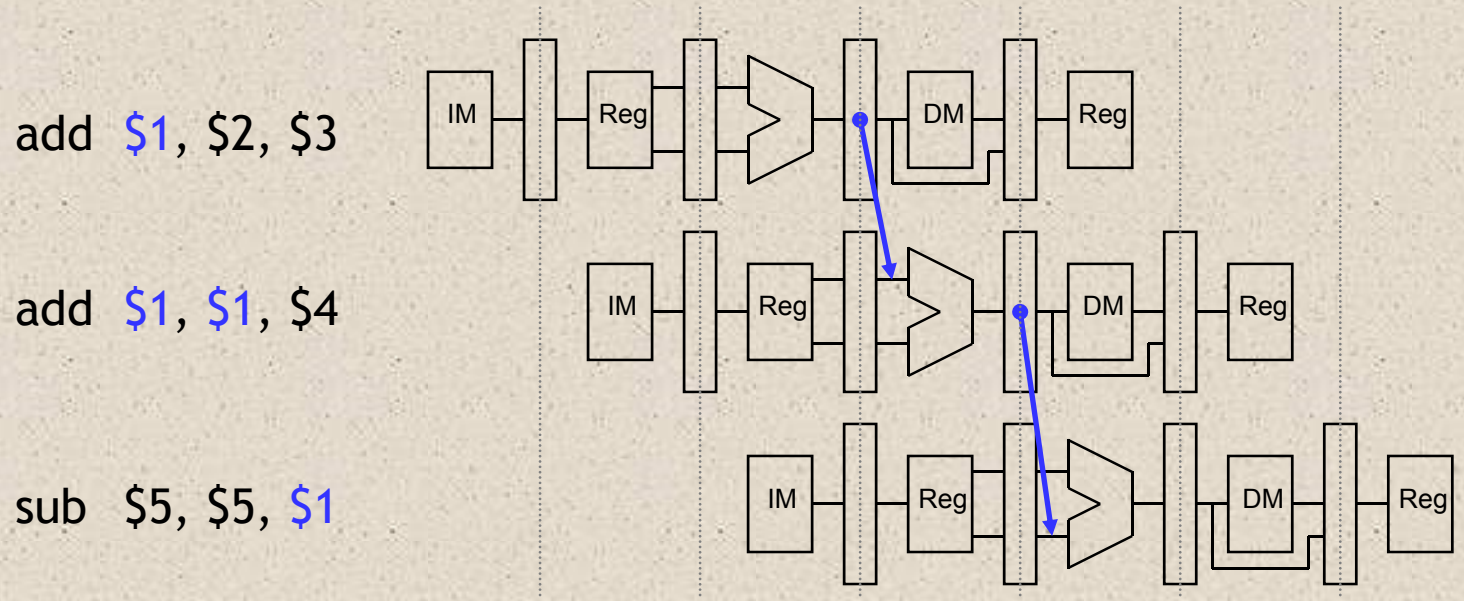
Datové hazardy MEM/WB

- 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

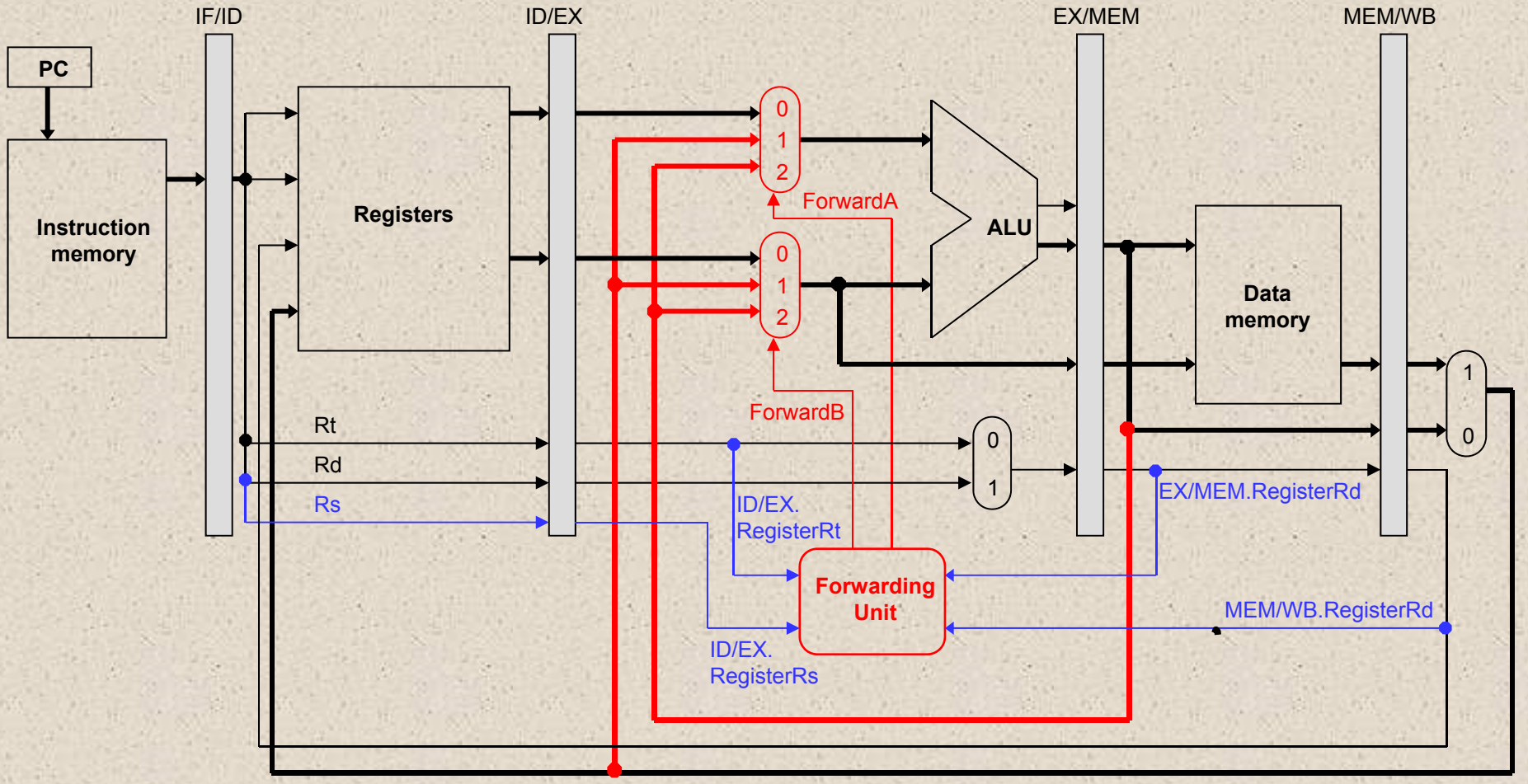
- Rovnice pro detekci a ošetření MEM/WB hazard prvého operandu ALU.

  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.

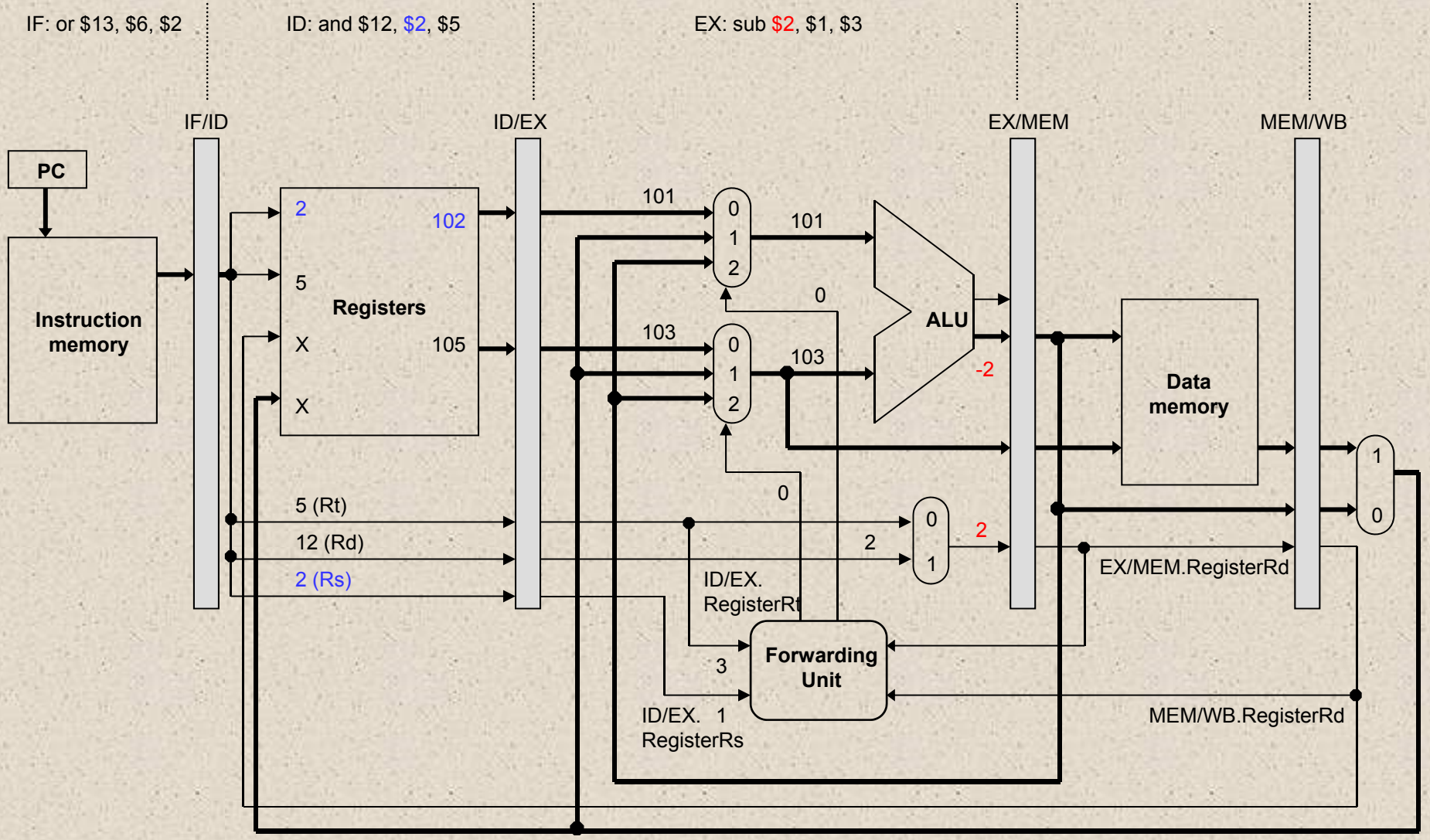
Příklad

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

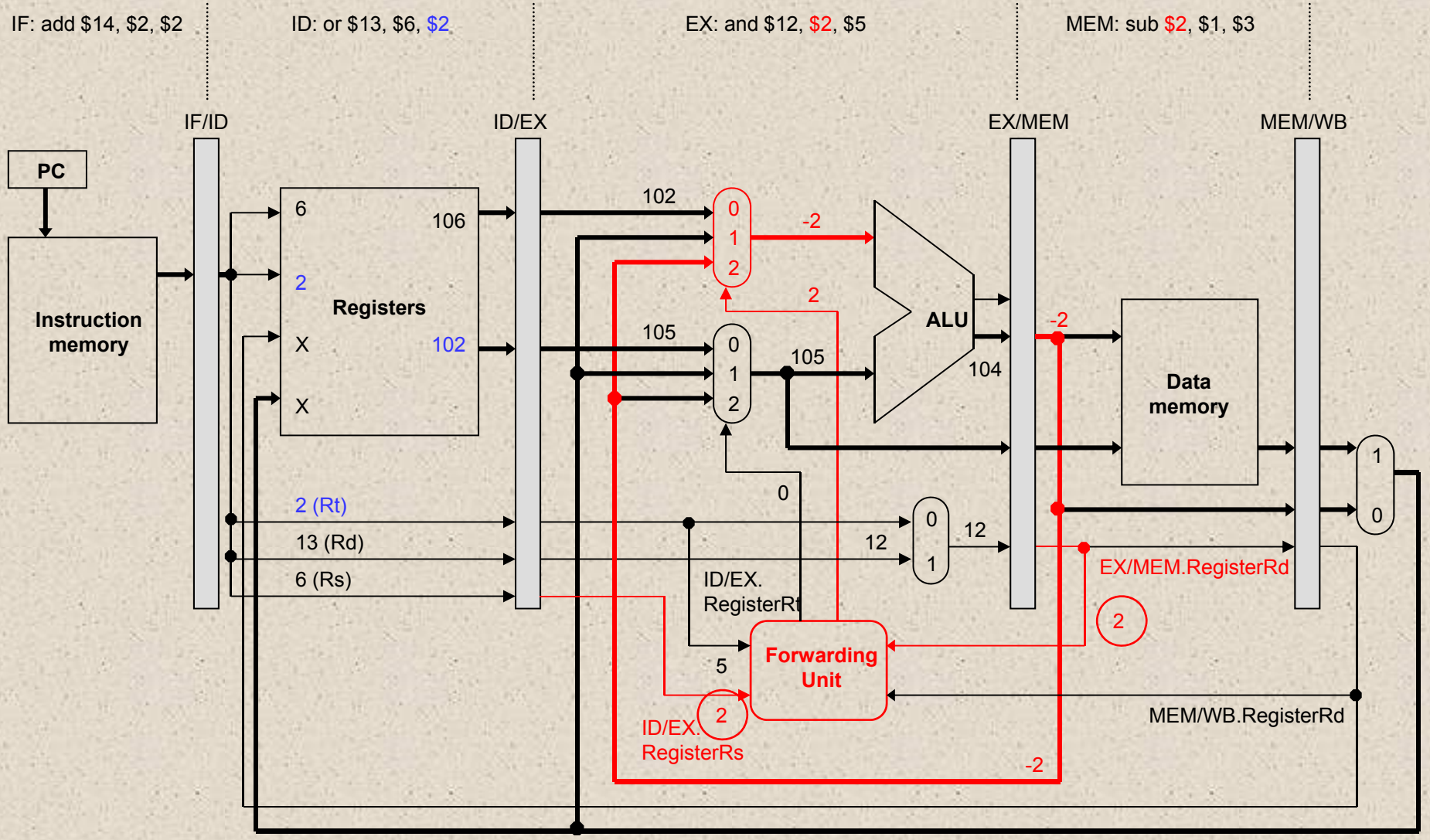
- Předpokládejme, že každý registr obsahuje svoje číslo plus 100.
  - Po provedení prvé instrukce $2 by měl obsahovat -2 (101 - 103).
  - Ostatní instrukce použijí -2 jako jeden z operandů.

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

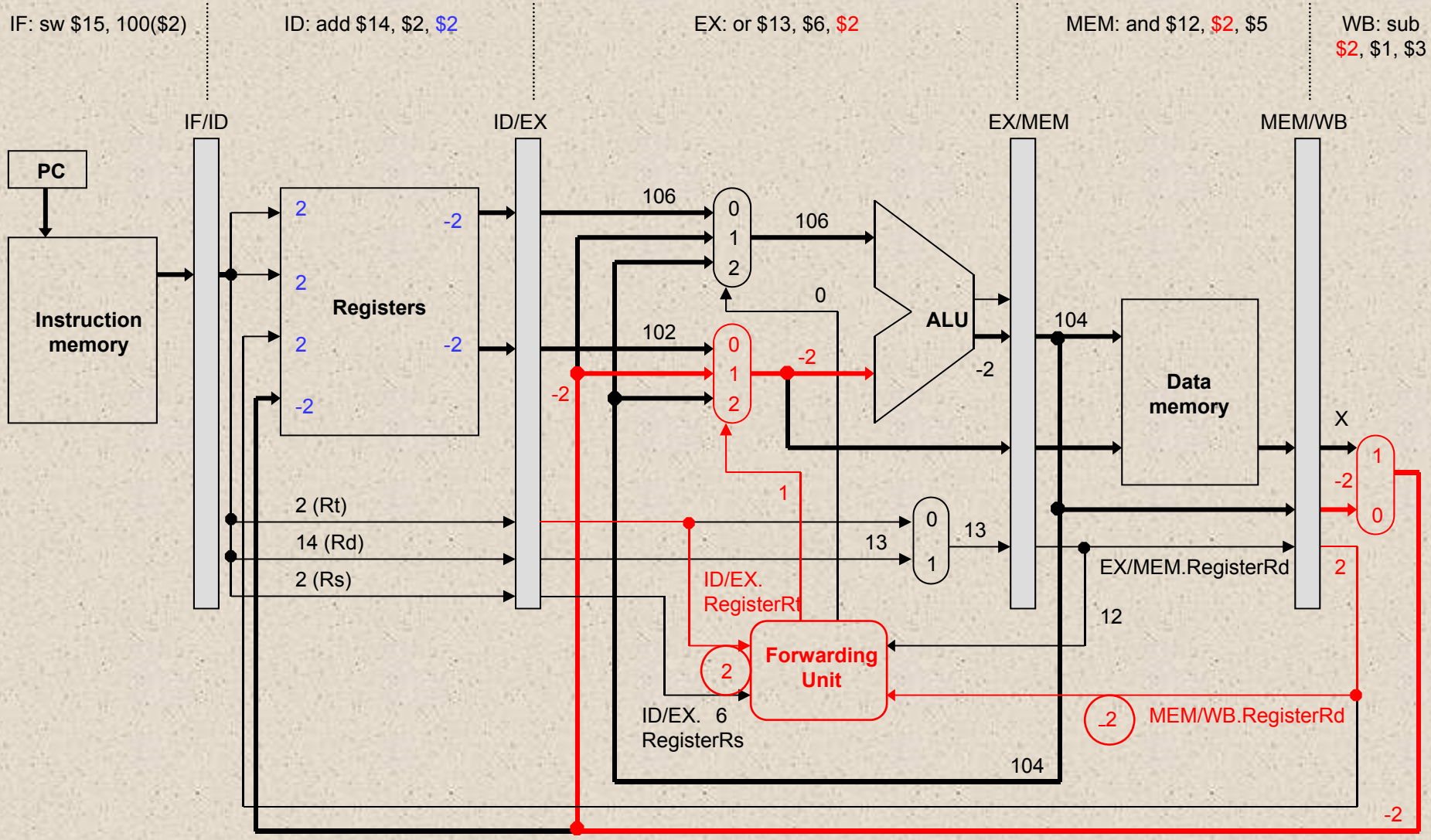
Hodinový cykl 3



Hodinový cykl 4: odběr $2 z EX/MEM



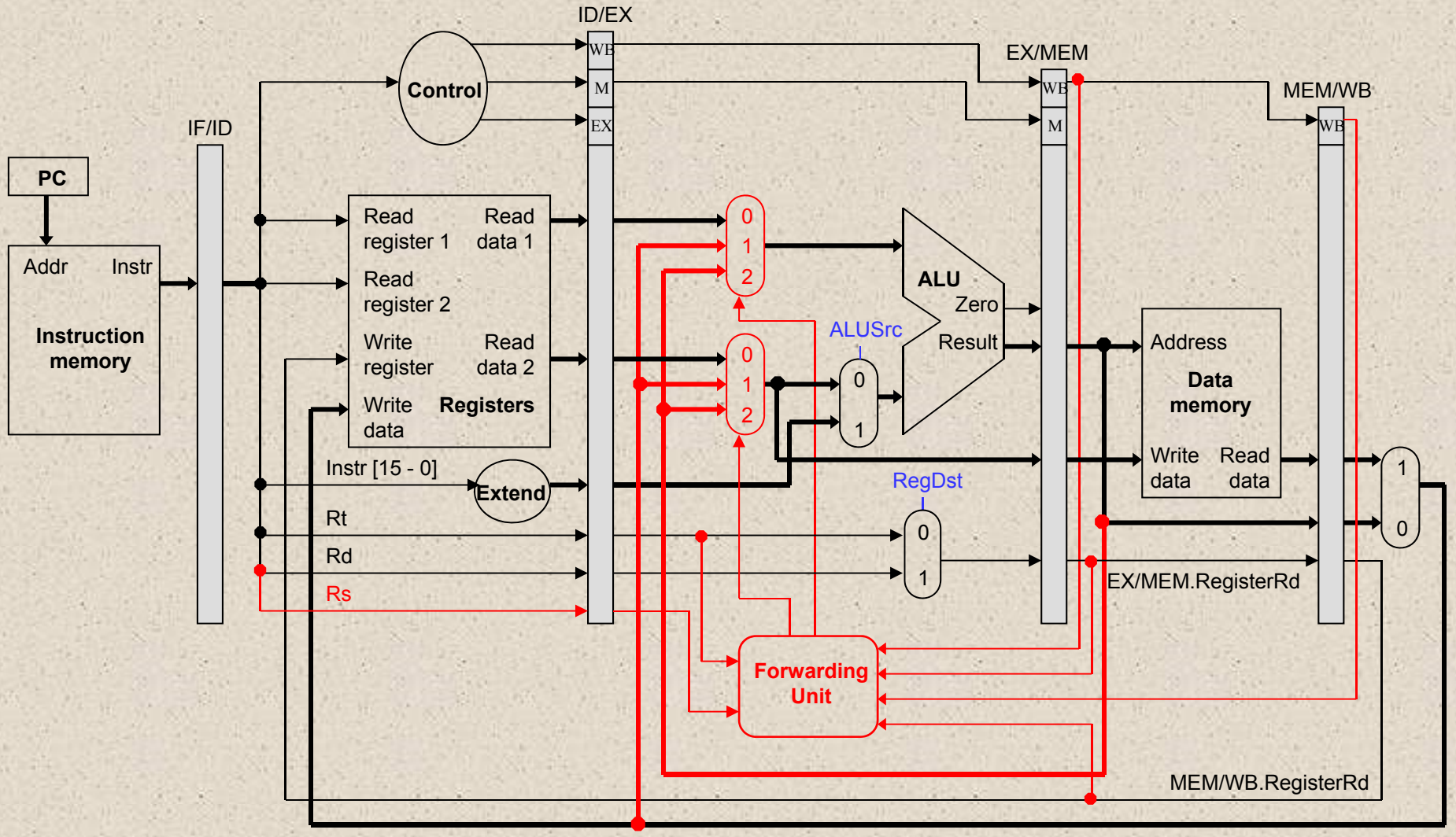
Hodinový cykl 5: odběr $2 z MEM/WB



Vzniká řada hazardů

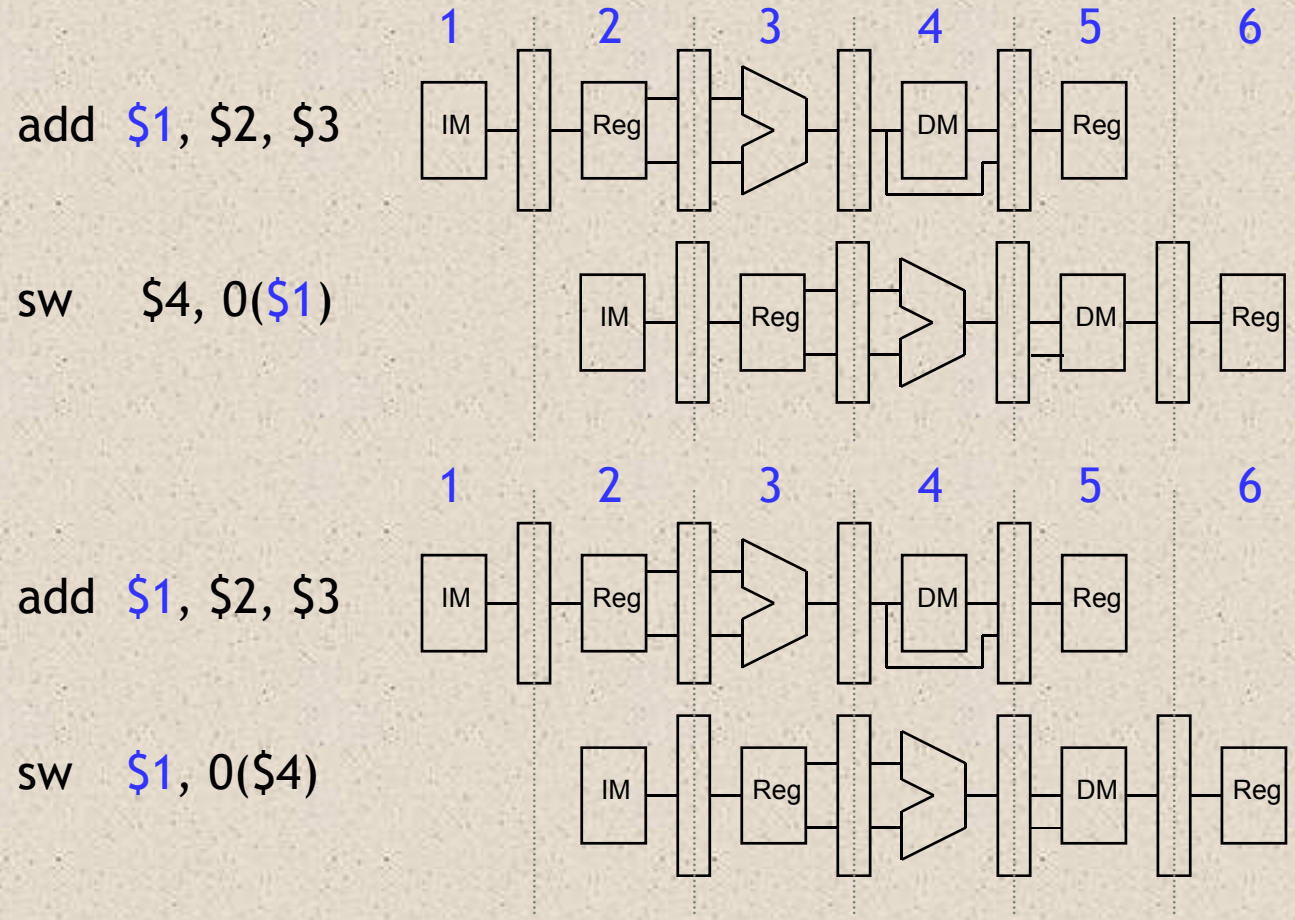
- První datový hazard nastává během cyklu 4.
  - Jednotka pro forwarding zjistí, že prvý zdrojový registr ALU pro AND je zároveň cílem instrukce SUB.
  - Správná hodnota se „forwarduje“ z registru EX/MEM. K použití nekorektní staré hodnoty v registrovém souboru nedojde.
- Druhý hazard nastává během hodinového 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

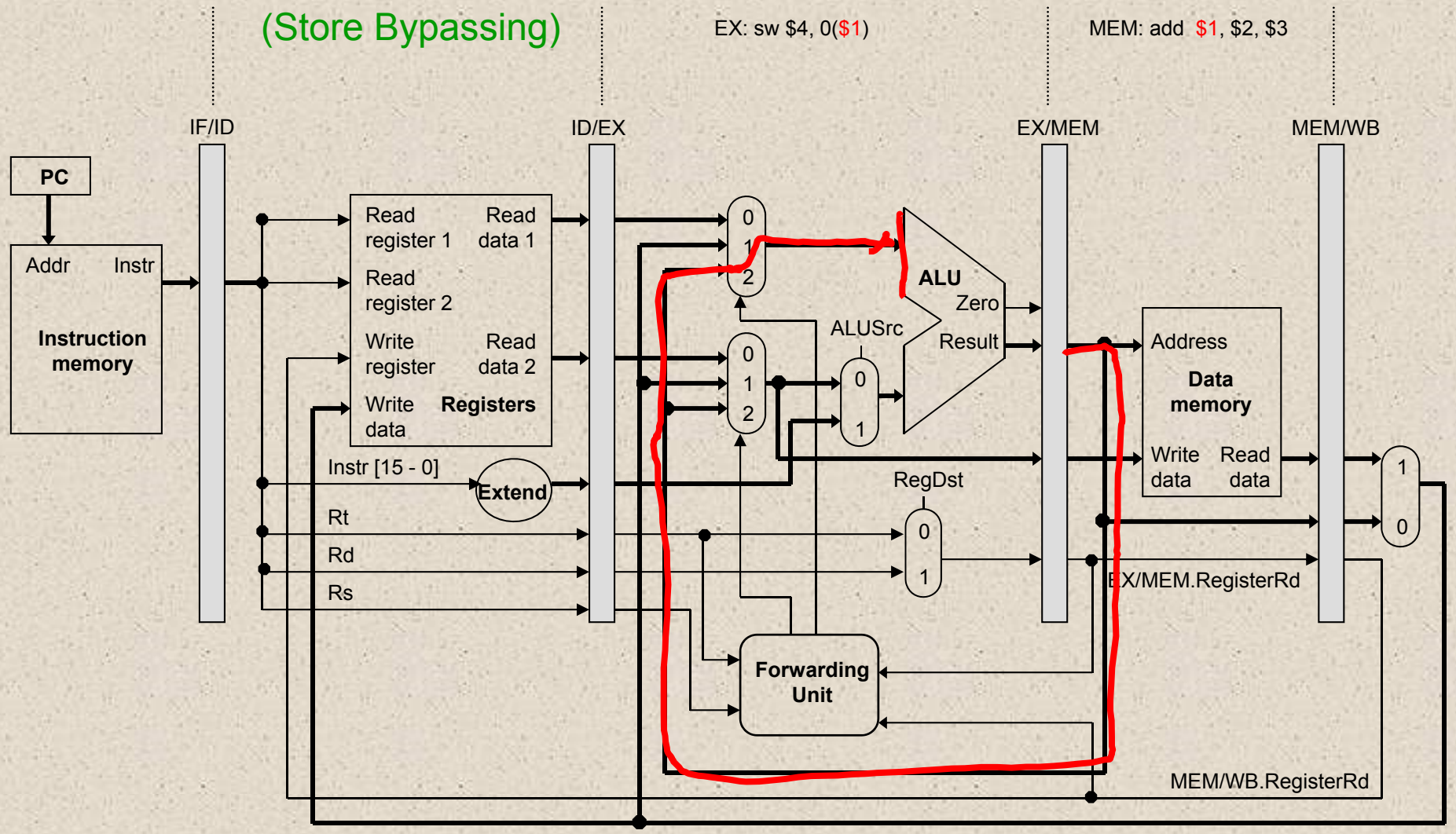


Operace zápisu?

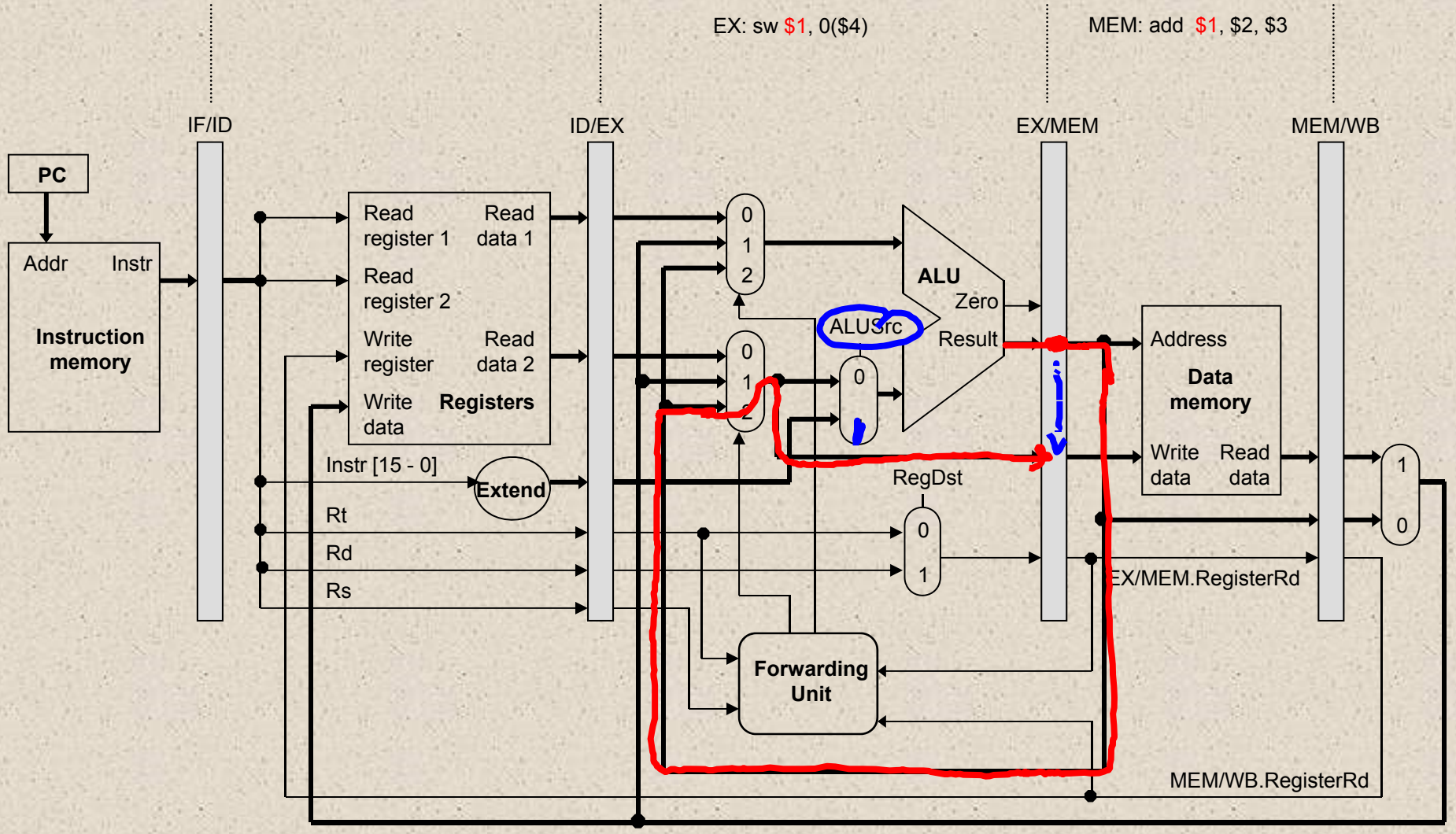
- Dva “jednoduché” případy:



Bypass při zápisu: Verze 1

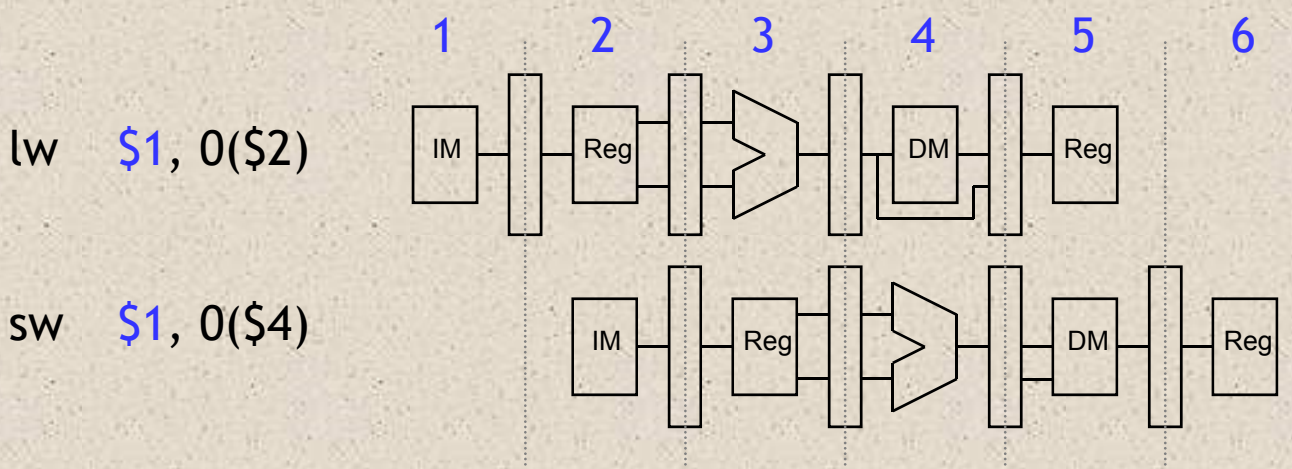


Bypass při zápisu: Verze 2



Vlastní zápis?

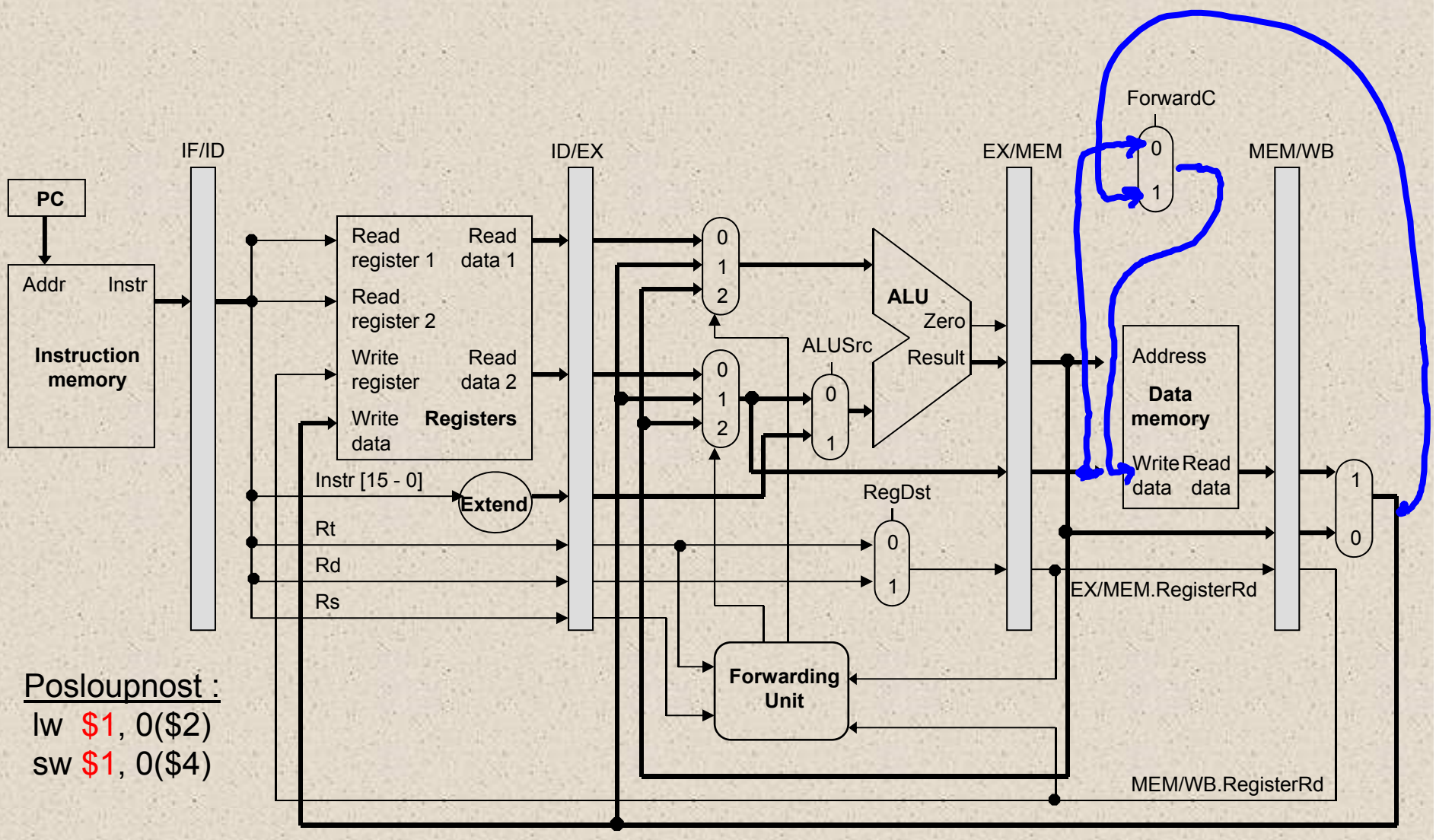
- Komplikovanější případ:



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

- Čím je třeba doplnit jednotku?

Rozšíření jednotky - Load/Store bypass



Posloupnost :

lw $1, 0($2)

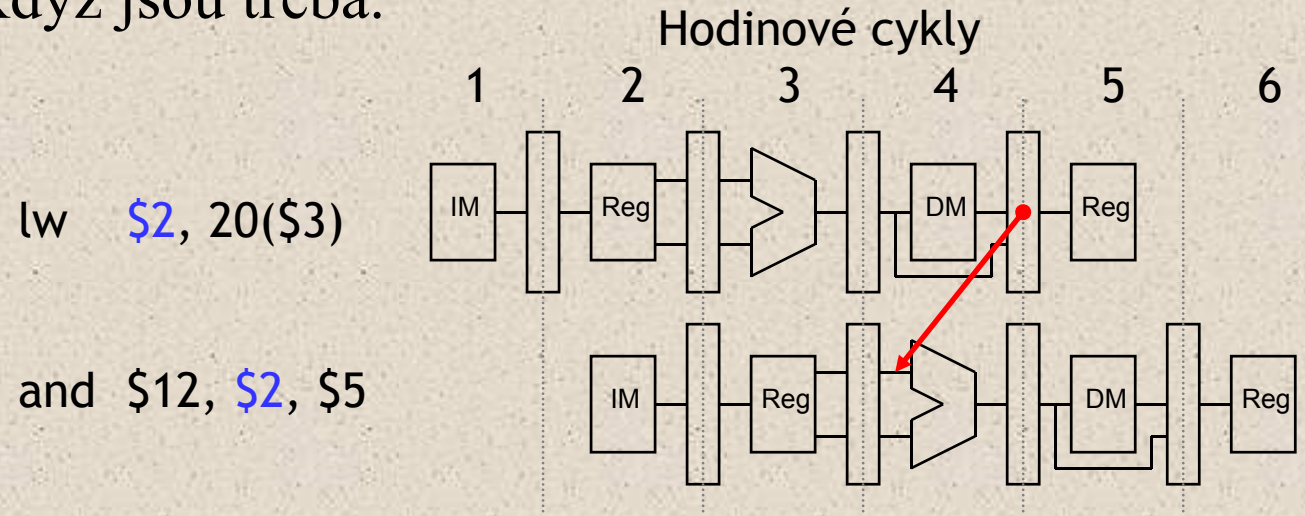
sw $1, 0($4)

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.

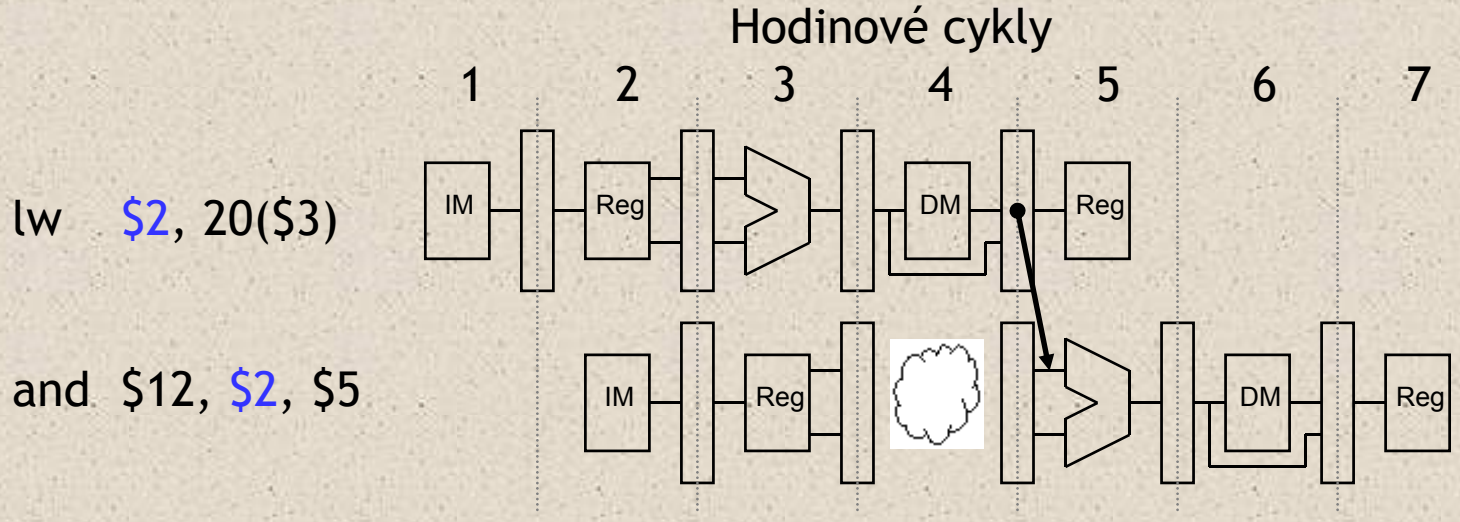
Problém se čtením

- 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.



Pozastavení

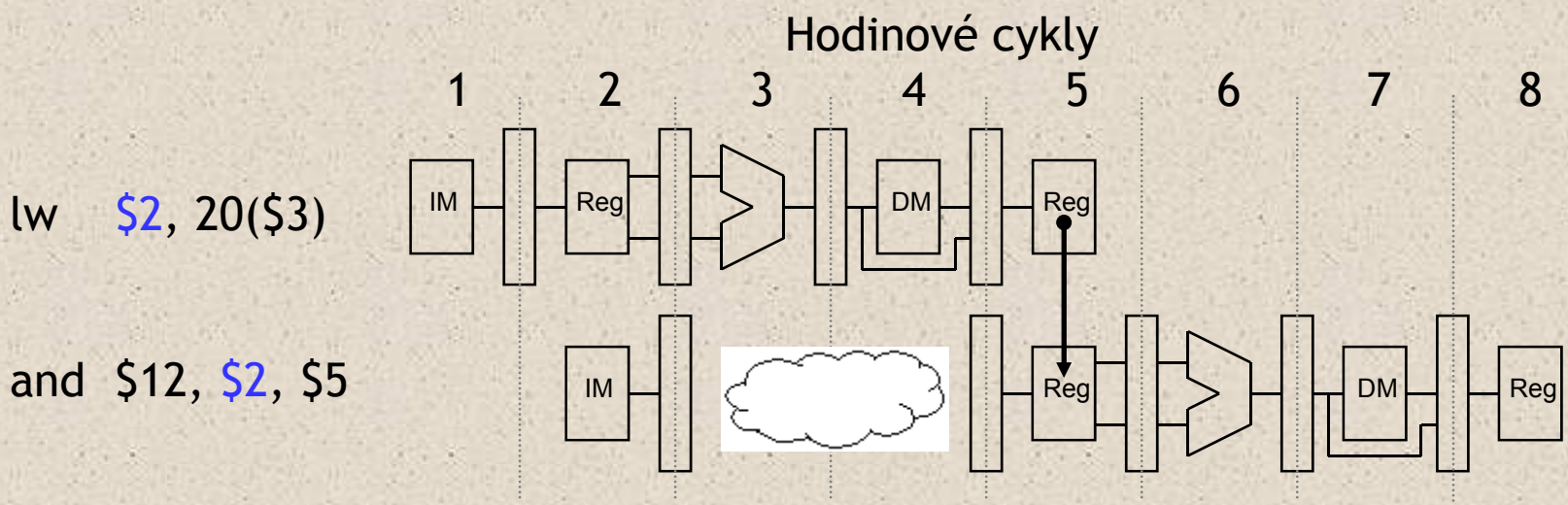
- Nejjednodušším řešením je zastavit pipeline (stall).
- Zpozdíme instrukci AND zařazením zpoždění v délce 1 cyklu do pipeline. Toto zpoždění se nazývá bublina.



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

Pozastavení a „forwarding“

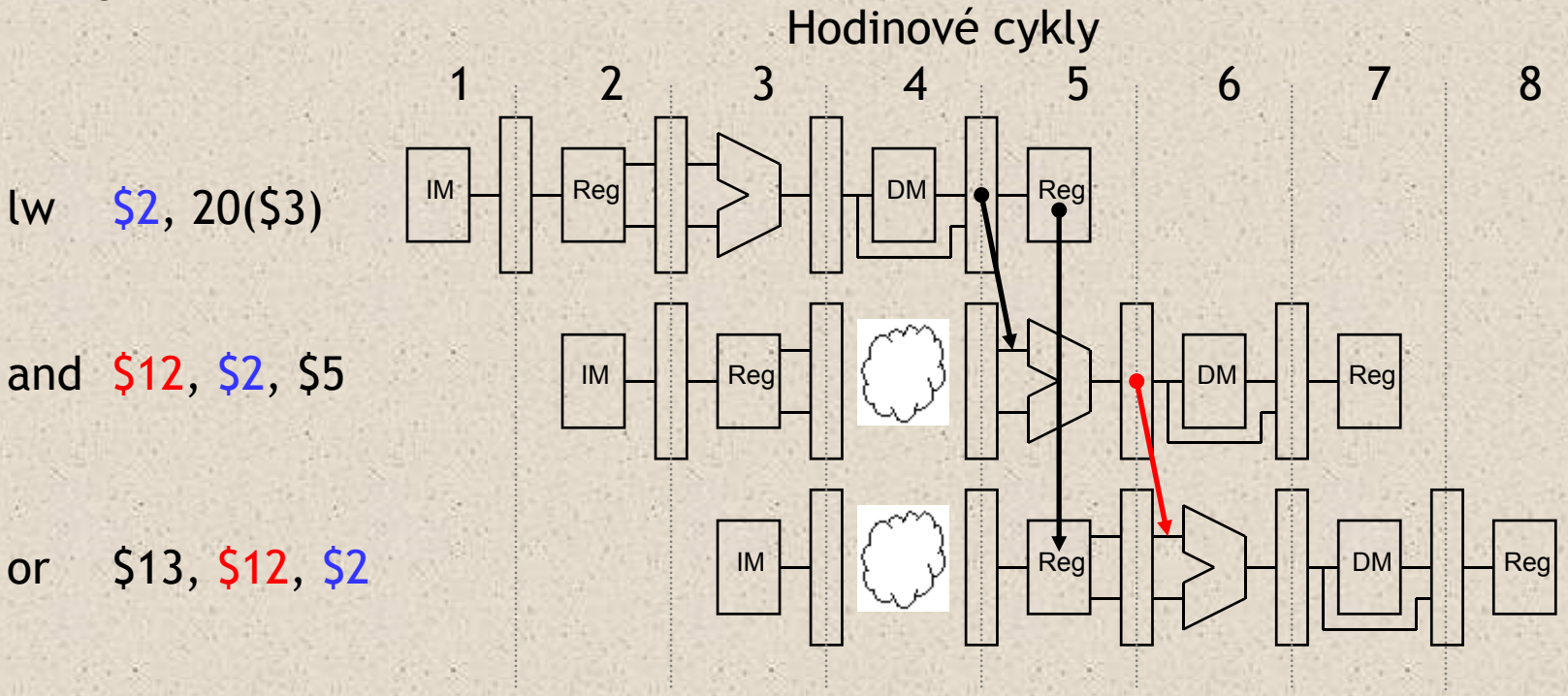
- Bez forwardingu bychom byli nuceni pozastavit na *dva* cykly a čekat na provedení zpětného zápisu instrukce LW.



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

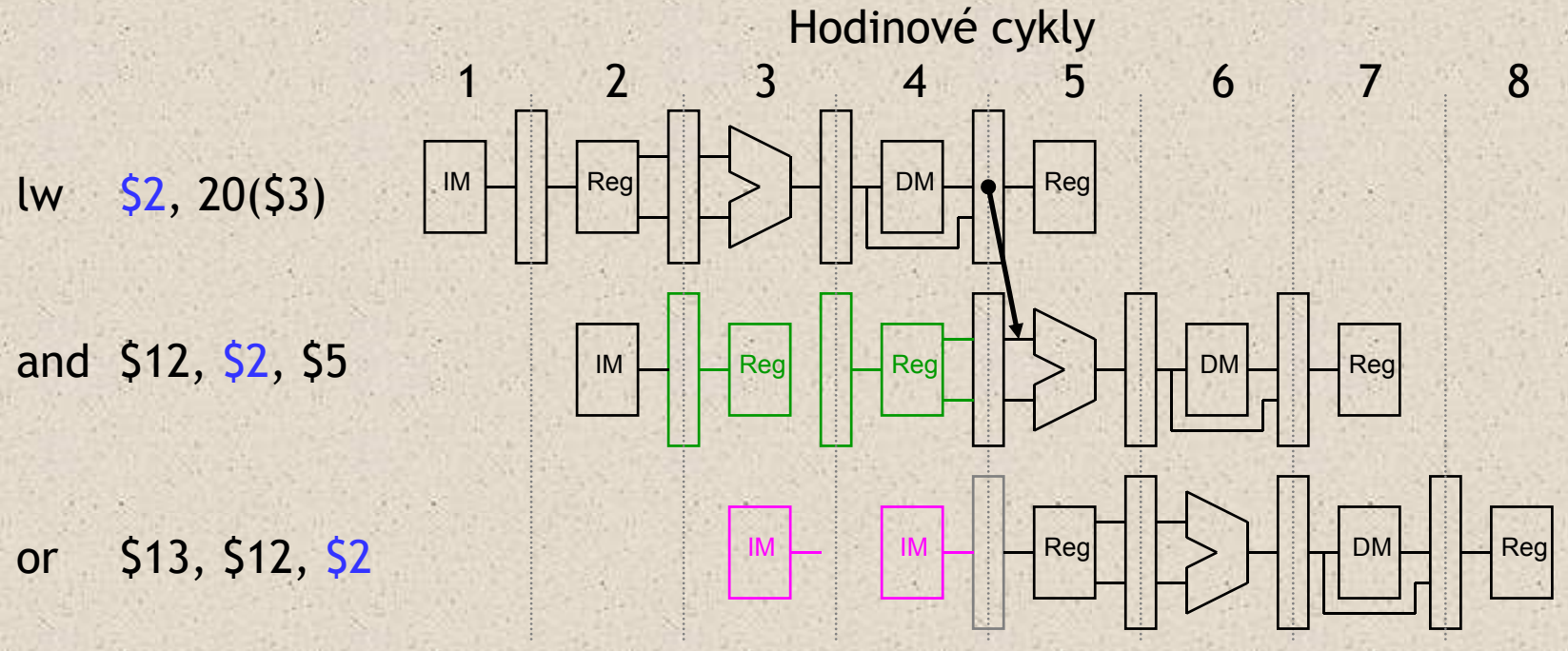
Pozastavení zdrží celou pipeline

- 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.



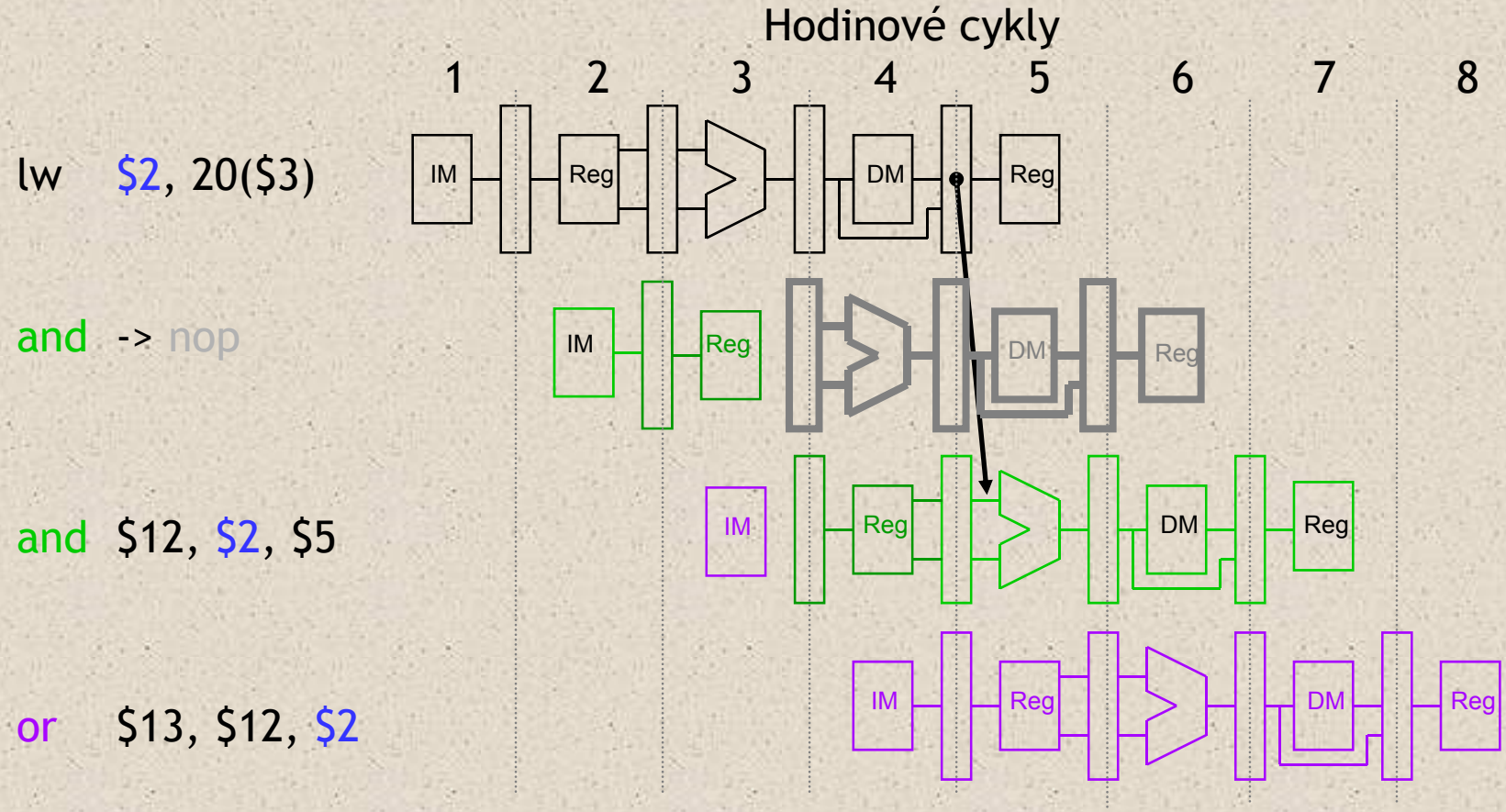
Co s registry EX, MEM, WB ?

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

Pozastavení = konverze na Nop

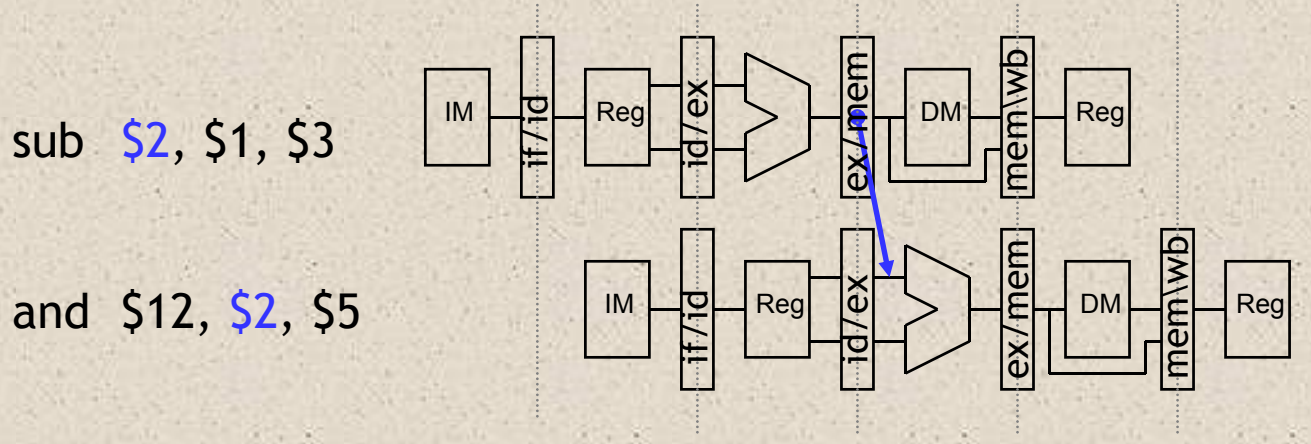


- Efekt pozastavení vlivem čtení (load stall) je vložení prázdné instrukce (nop) do pipeline

Detekce pozastavení

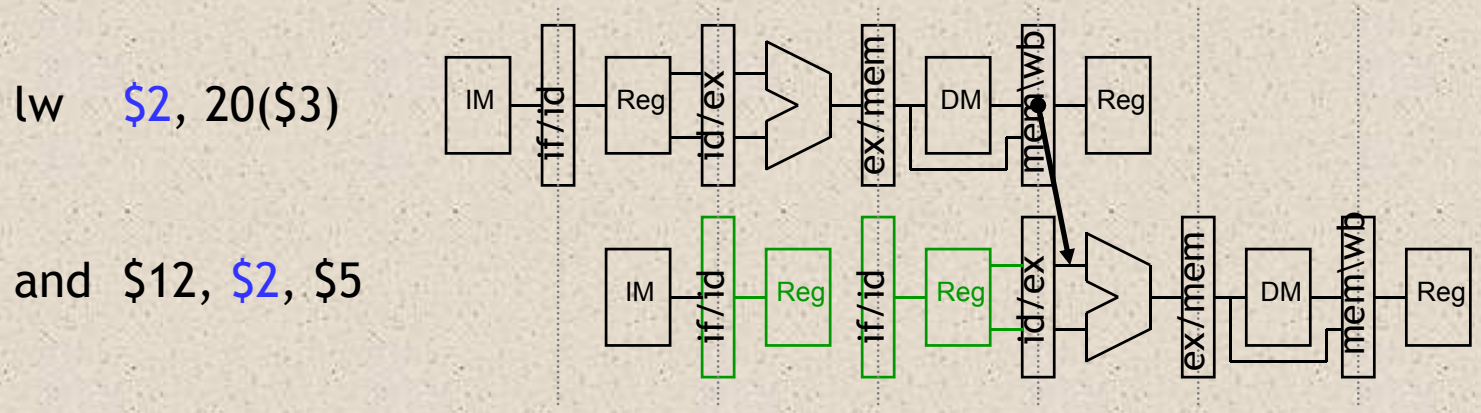
- Detekce pozastavení se podobá detekci datových hazardů. Připomeňte si rovnici detekce hazardů:

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



Detekce pozastavení, pokračování

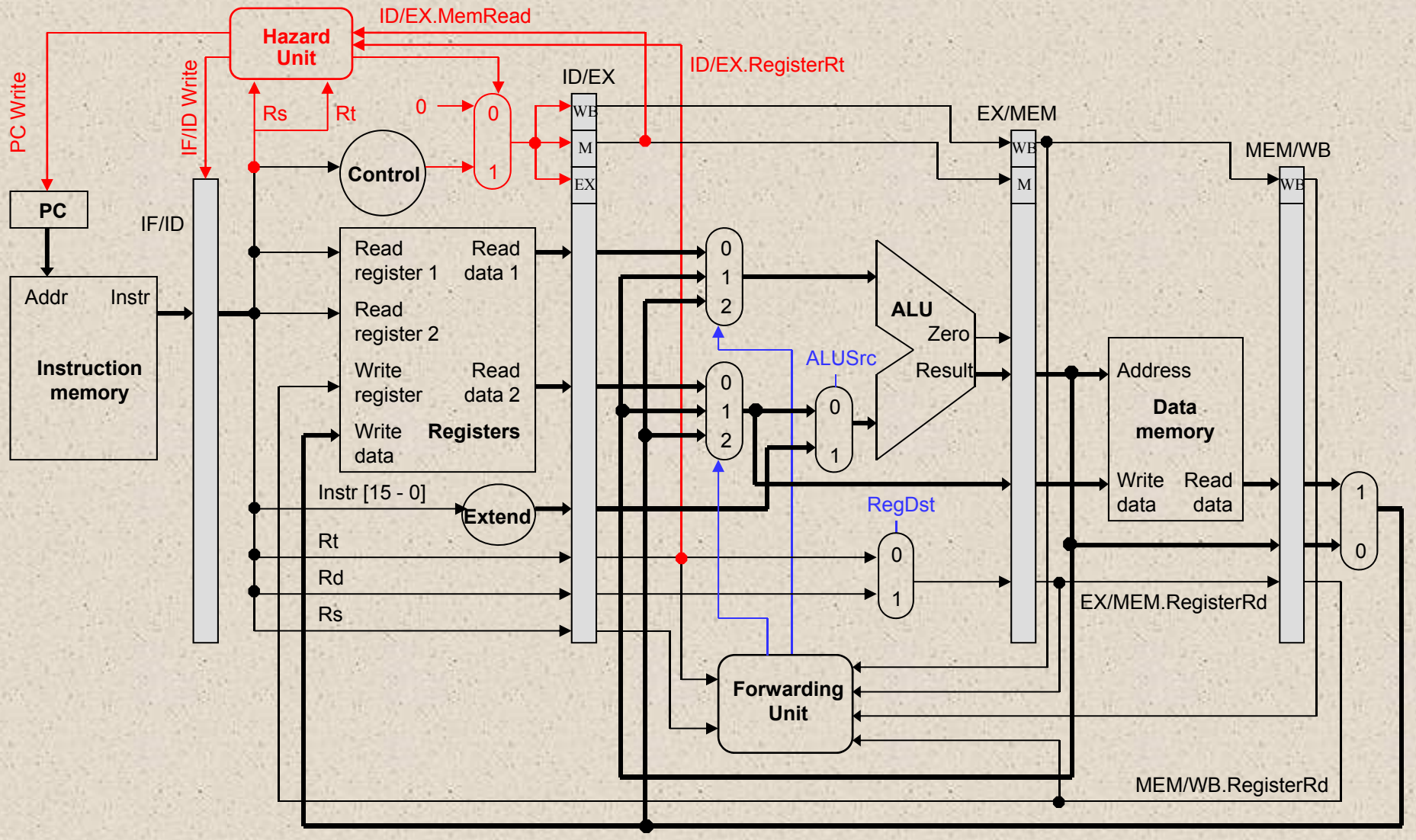
- Kdy se mají pozastavení (**stalls**) detekovat? Ve stupni EX



- Co je podmínkou pro zařazení bubliny (stall) ?

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

Doplnění detekce hazardů do CPU



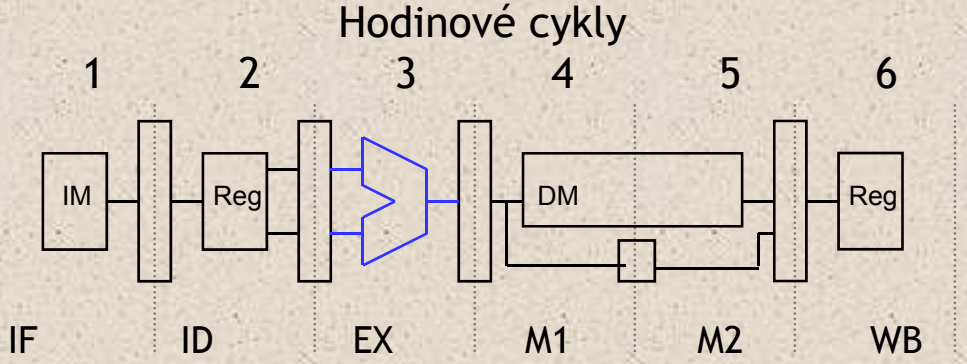
Jednotka pro detekci hazardů

- 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?



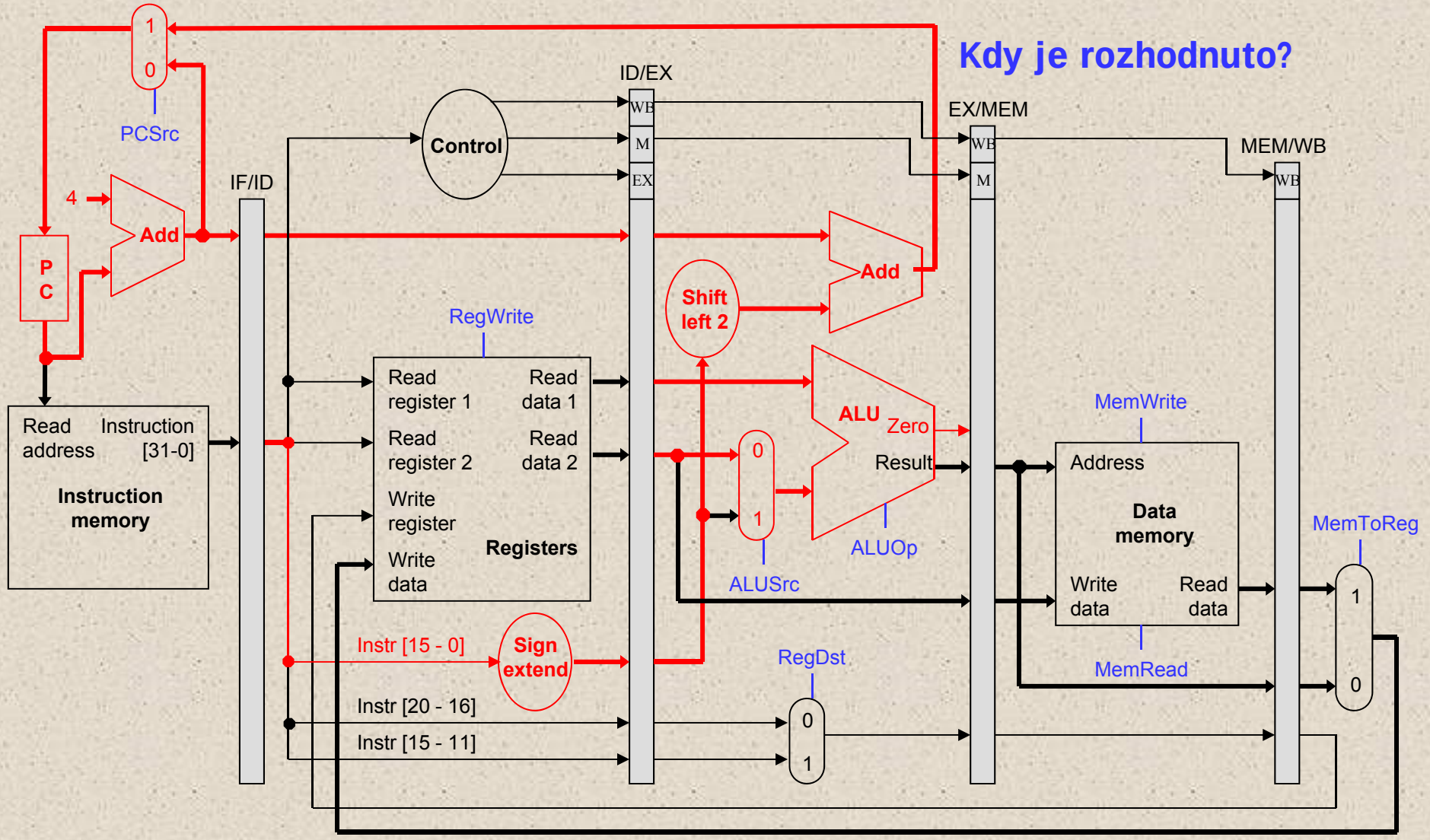
- Kolik vstupů pro bypass musí mít multiplexery ve stupni EX? (Odpověď: 4)
- Které instrukce v příkladu vyžadují pozastavení and/or bypass?

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

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

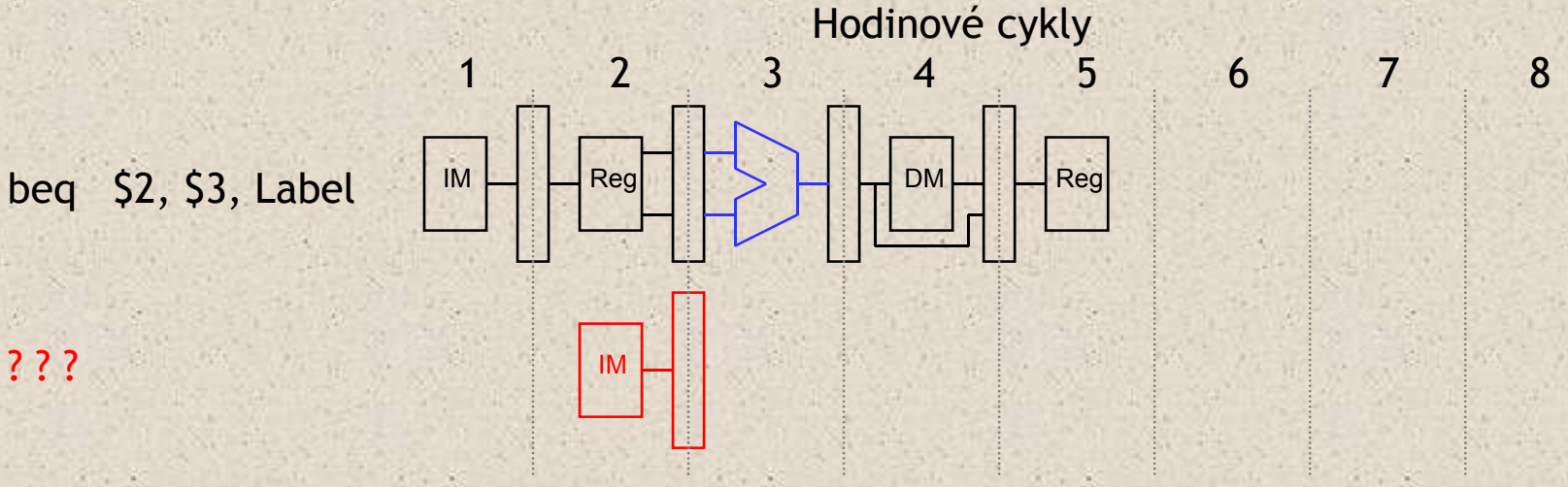
IF	ID	EX	M1	M2	WB				
	IF	ID	EX	M1	M2	WB			
		IF	ID						
				ID	EX	M1	M2	WB	

Větvení v původní jednotce s pipeliningem



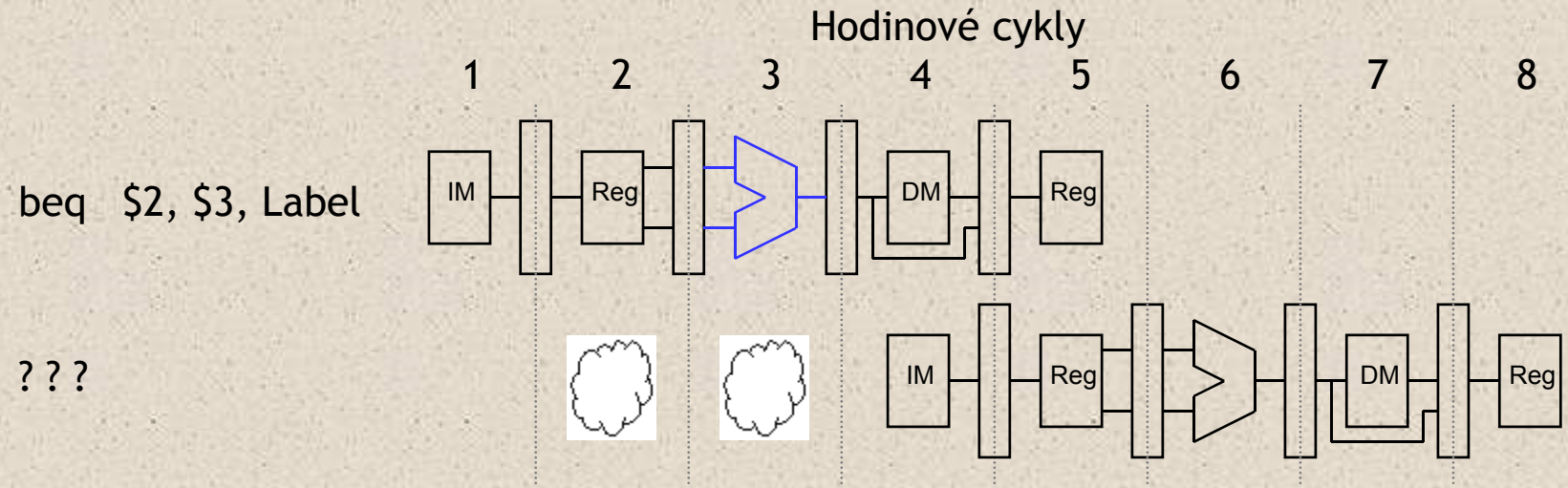
Podmíněné skoky - větvení

- 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 řídící hazard.



Pozastavení je jen jedno řešení

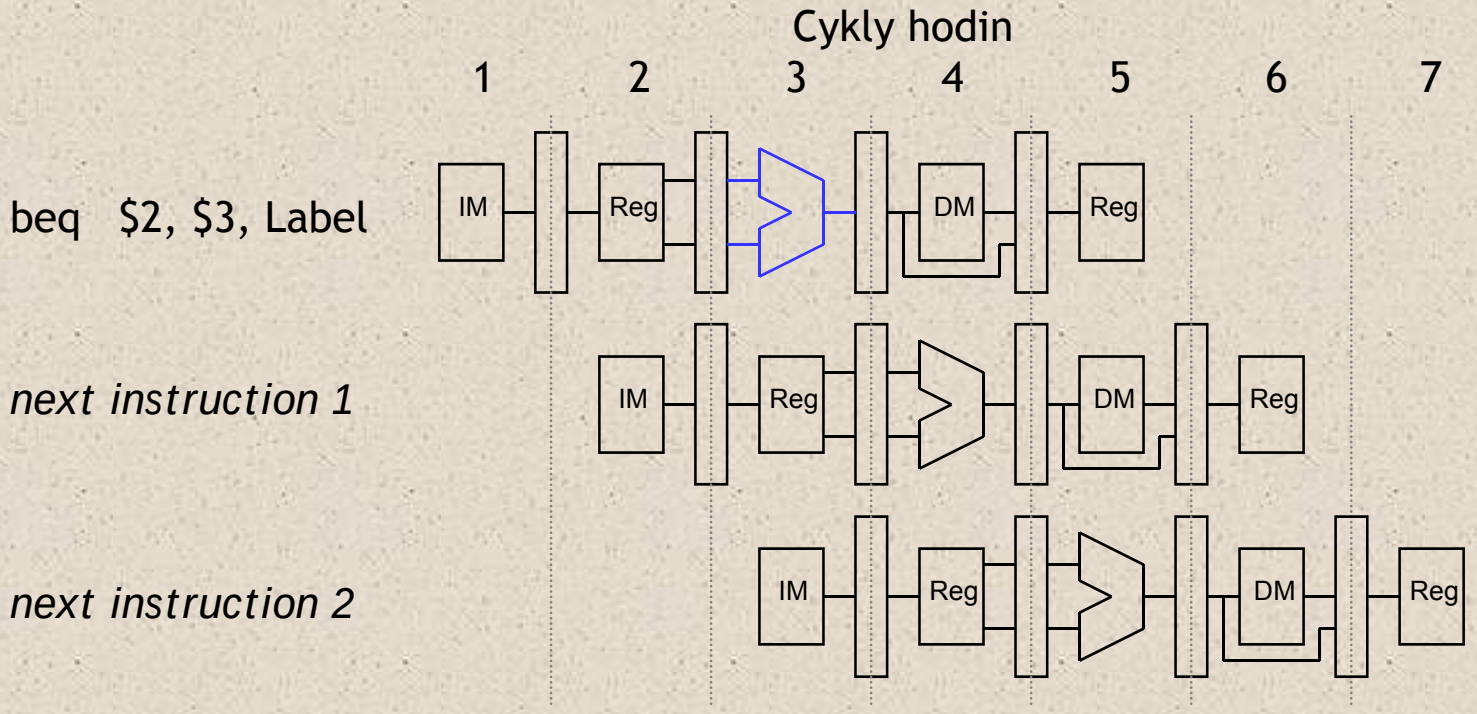
- Přesto, pozastavením lze situaci vždy řešit.



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

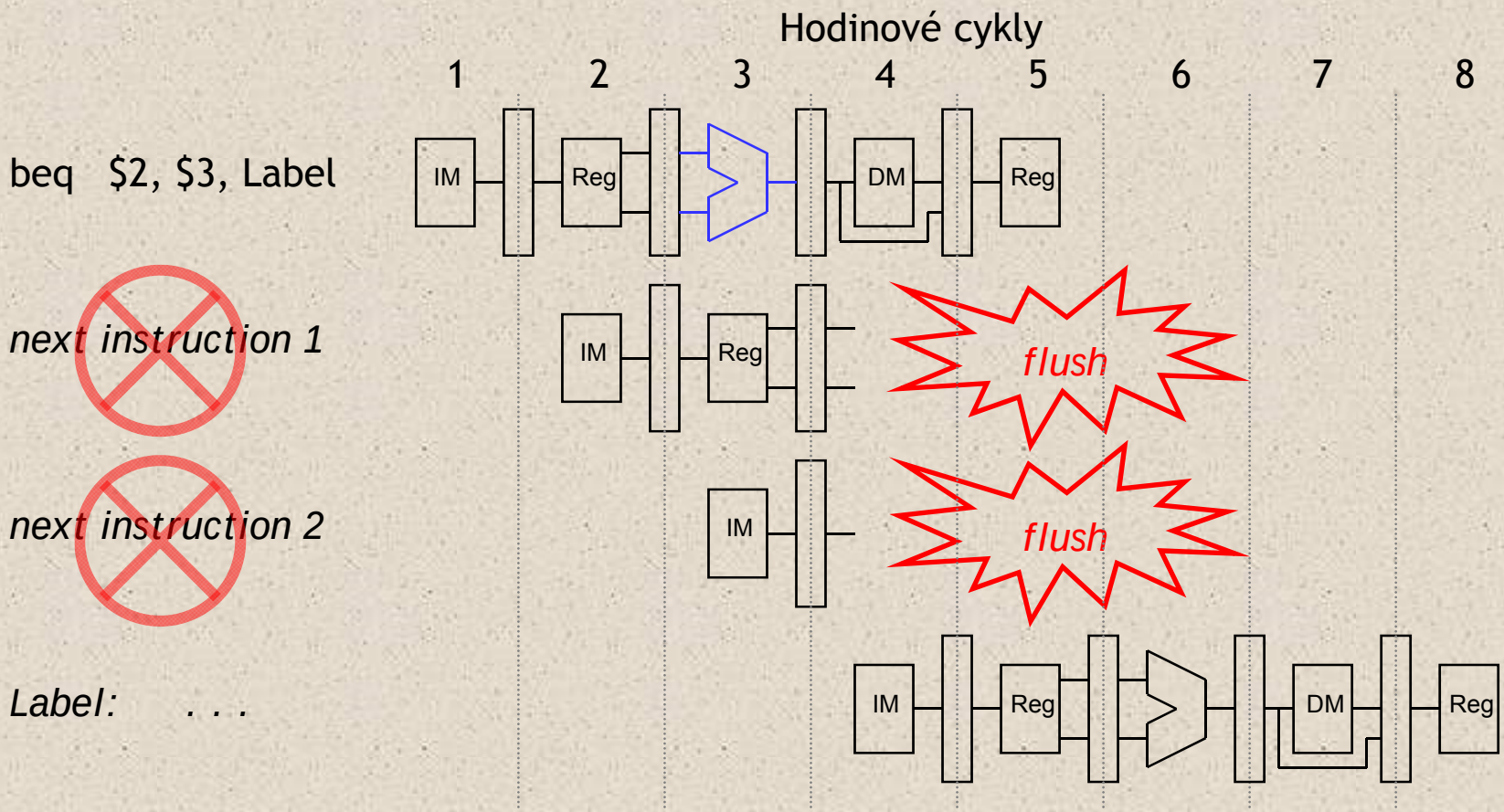
Predikce skoků

- 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).



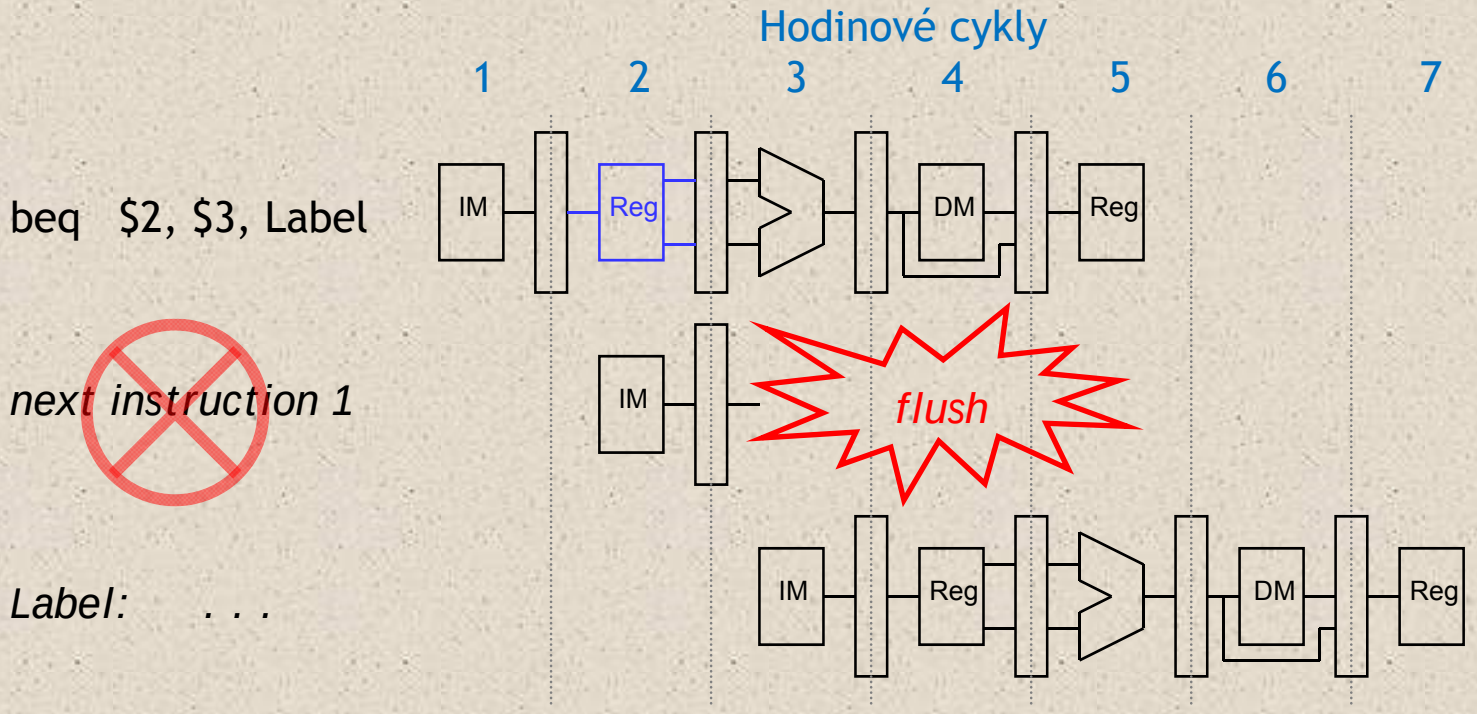
Zisk a ztráta výkonu

- 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.

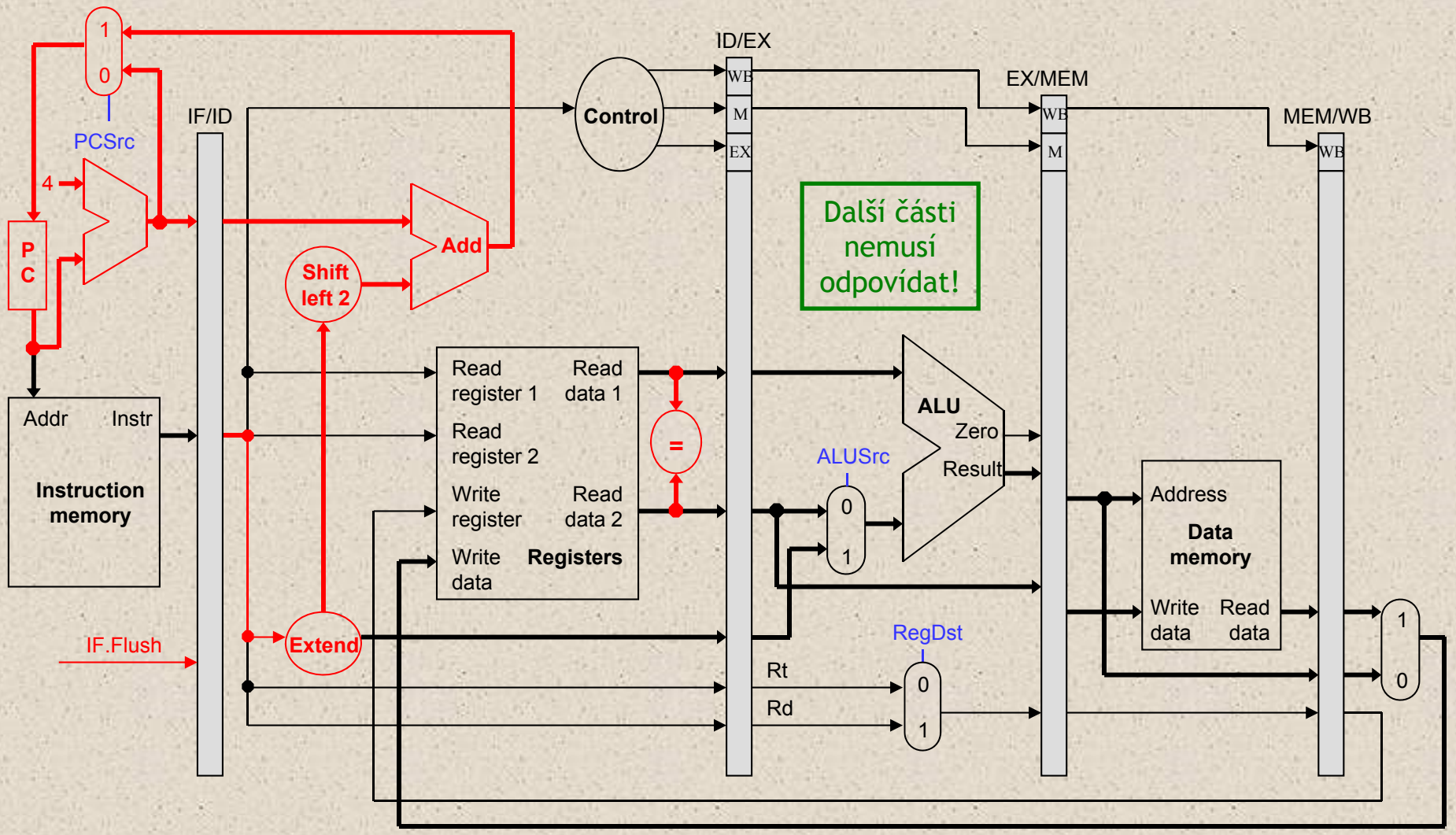
- Potom při nezdařené predikci stačí „zahazovat“ jenom jednu instrukci.



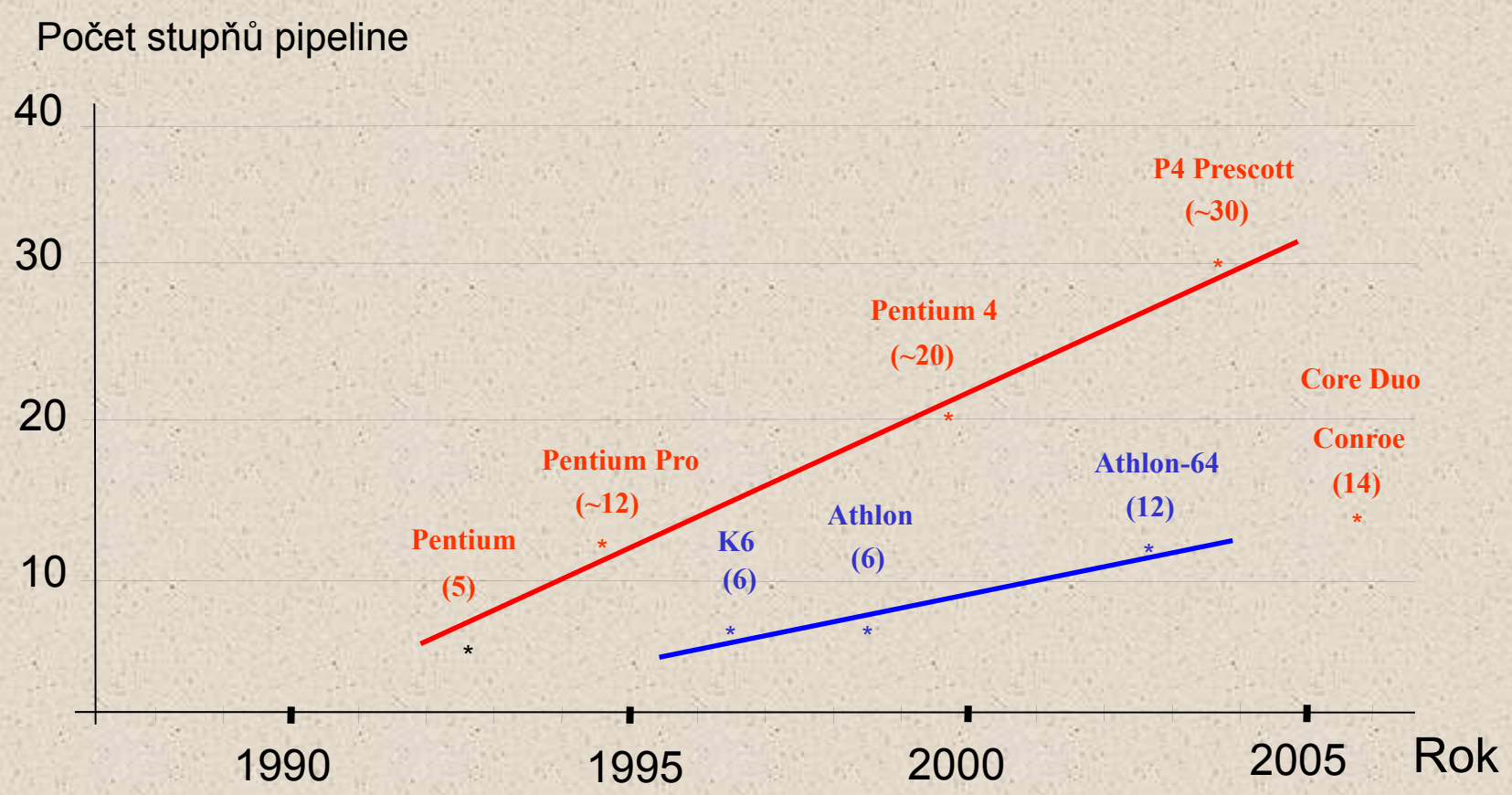
Implementace „odhazování“ (flush)

- 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.