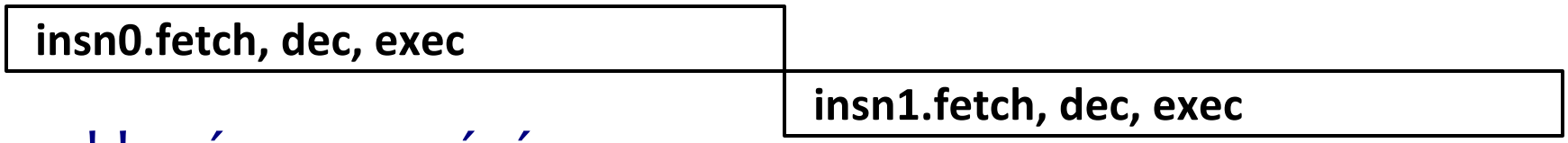
Principy počítačů a operačních systémů

Zvyšování výkonnosti procesorů

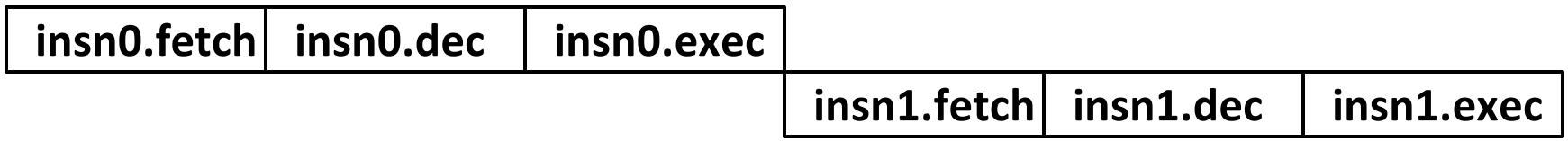
Zimní semestr 2011/2012

Co nám omezuje výkonnost procesoru?

Jednocyklové zpracování



Vícecyklové zpracování



Můžeme mít nízké CPI a krátký strojový cyklus?

- ne, pokud datová cesta vykonává instrukce po jedné

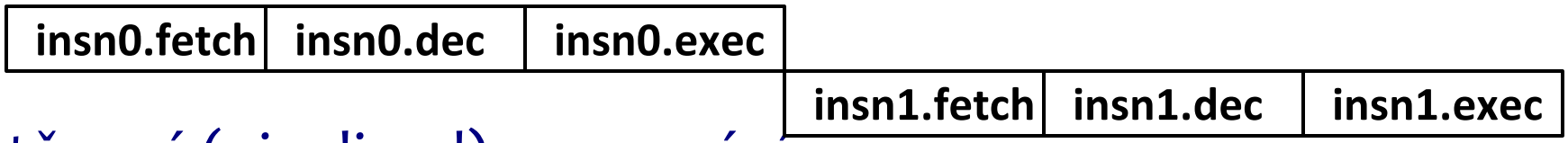
Latence vs. propustnost

- ***latenci*** nezlepšíme – instrukce nejdou (téměř) zrychlit
- zlepšíme ***propustnost*** – zajímá nás doba vykonávání celého programu
  - programy sestávají z miliard instrukcí, délka samotné instrukce není zajímavá

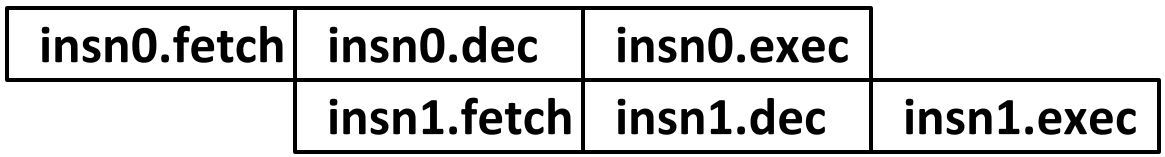
Klíčem je paralelizace zpracování instrukcí

Zřetězené zpracování instrukcí

Vícecyklové zpracování



Zřetězené (pipelined) zpracování



Zlepšuje propustnost, nikoliv dobu zpracování instrukcí

- při zahájení 2. kroku zpracování instrukce začne datová cesta zároveň vykonávat 1. krok zpracování následující instrukce
- paralelizmus na úrovni instrukčních kroků, zachovává sekvenční model
- délka zpracování jednotlivých instrukcí se nemění, ale instrukce vstupují a opouštějí datovou cestu mnohem rychleji
- obecný princip zvyšování propustnosti, analogie v mnoha oblastech

Jaký výkon můžeme očekávat?

Zrychlení při zřetězeném zpracování

Srovnání s vícecyklovým zpracováním

- zpracování ***n*** instrukcí, hodinový signál s periodou t_{clk}
- vícecyklové zpracování s ***k*** kroky na instrukci

$$T_m = n \cdot (k \cdot t_{clk})$$

- zřetězené zpracování v ***k-stupňové*** pipeline
  - první instrukce opustí pipeline po ***k*** taktech, ostatní po ***1***

$$T_p = k \cdot t_{clk} + (n-1) \cdot t_{clk}$$

- zrychlení proti vícecyklovému zpracování

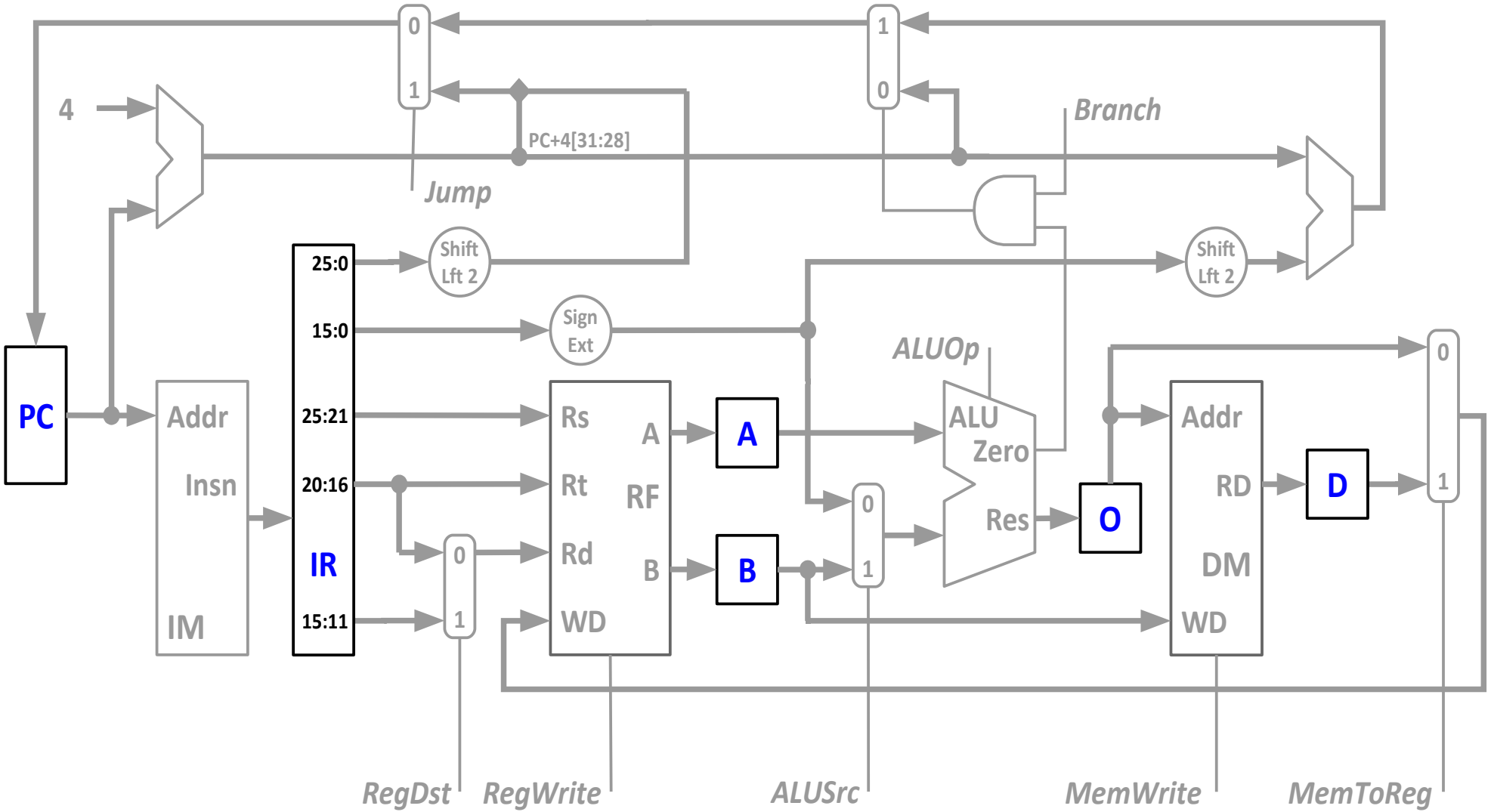
$$\frac{T_m}{T_p} = \frac{n \cdot (k \cdot t_{clk})}{k \cdot t_{clk} + (n-1) \cdot t_{clk}} = \frac{n \cdot k}{k + (n-1)} = S$$

- pro ***n >> k***

$$k + (n-1) \approx n, S \to k$$

Konstrukce datové cesty pro zřetězené zpracování

Vyjdeme z vícecyklové datové cesty



Transformace na zřetězené zpracování

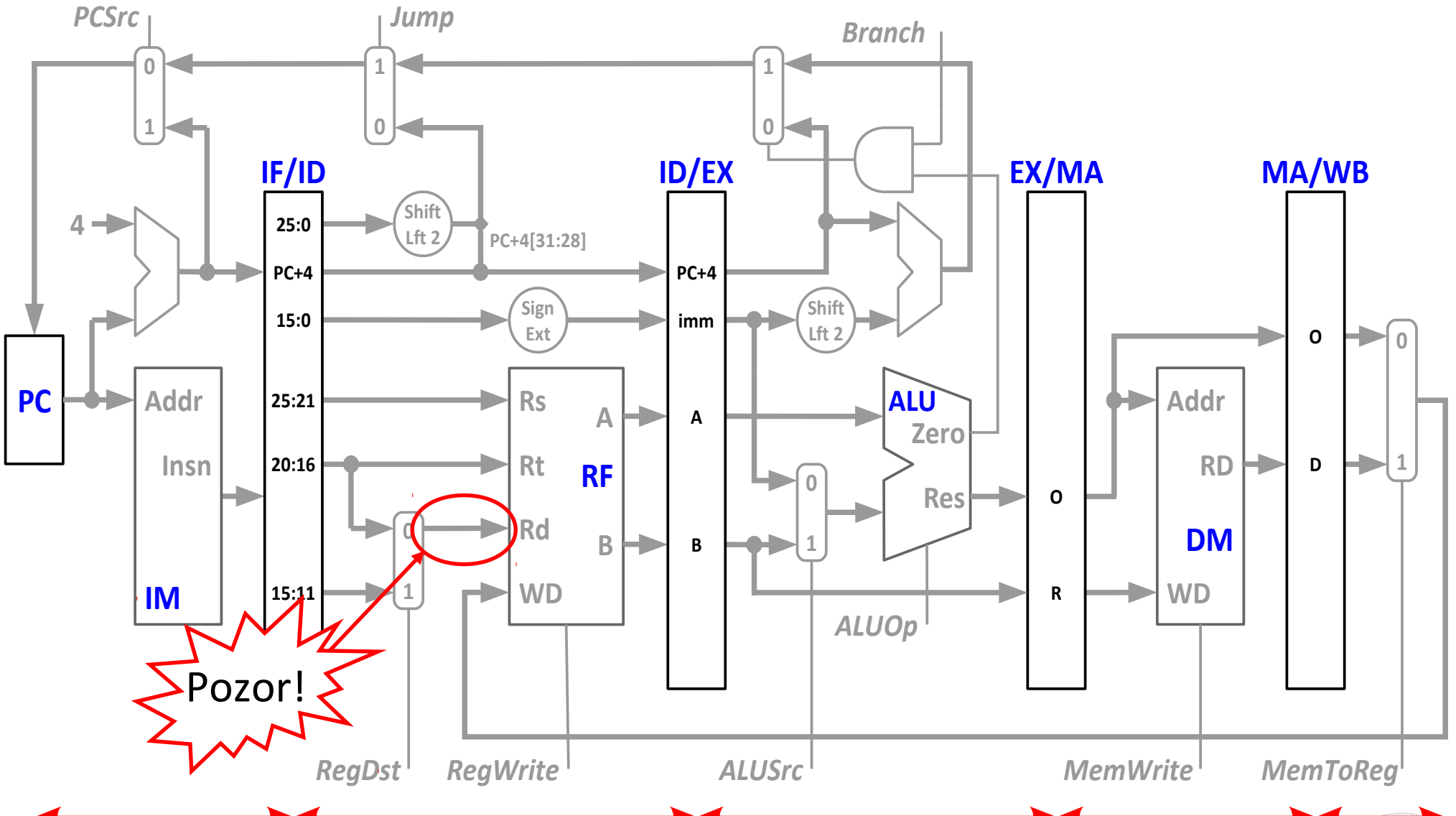
Datovou cestu rozdělíme do ***k*** úseků (stages)

- v každém úseku budeme zpracovávat jinou instrukci
- nejpomalejší úsek určuje rychlost pipeline
  - obdoba nejdelší instrukce v jednocyklové datové cestě
- výchozí ***CPI=1*** (ideální stav)
  - v každém kroku opustí pipeline jedna instrukce
  - ve skutečnosti ***CPI>1*** (pipeline se často zasekne)
  - latence instrukcí zvyšuje režii, nesnižuje propustnost

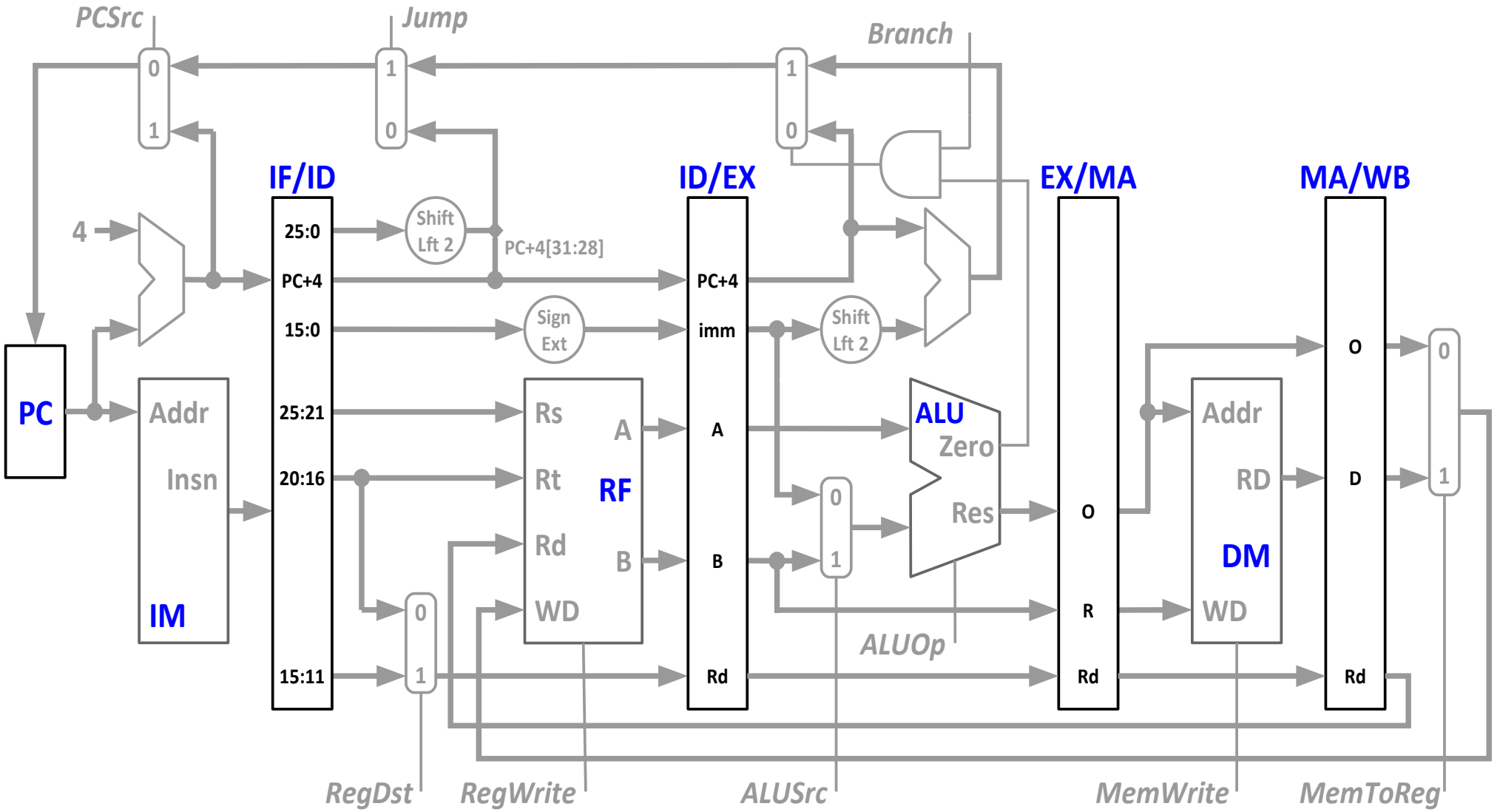
Mezi jednotlivé úseky vložíme registry (latches)

- stav instrukce, operandy, řídící signály
  - instrukce uvnitř datové cesty jsou v různých stavech
- analogie k pásu na výrobní lince

Datová cesta pro zřetězené zpracování (1. pokus)



Datová cesta pro zřetězené zpracování



Trocha terminologie

Skalární pipeline (*scalar pipeline*)

- v každém úseku se zpracovává pouze 1 instrukce
- alternativou jsou superskalární (*superscalar*)

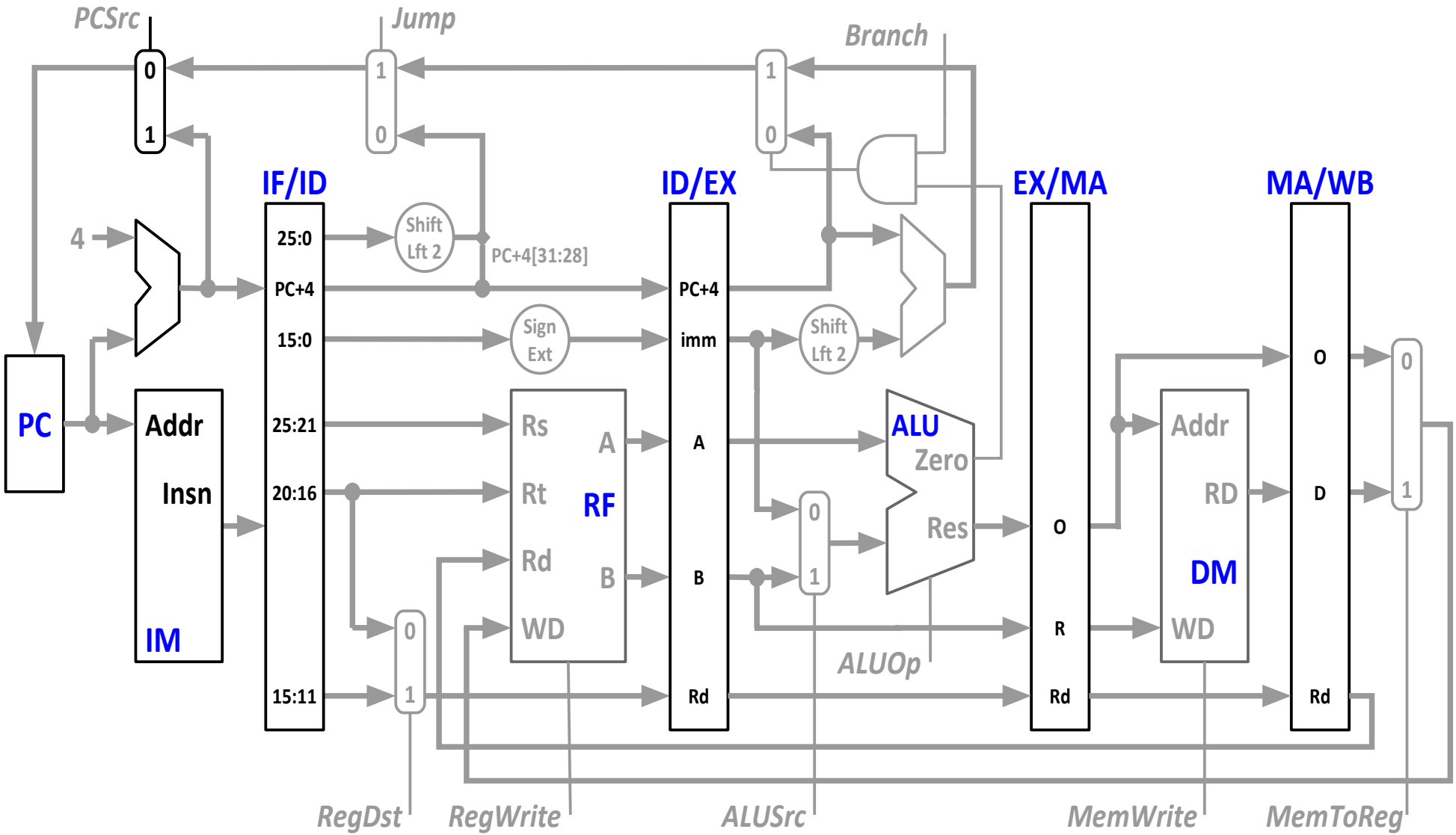
Zpracování podle pořadí (*in-order pipeline*)

- instrukce vstupují do pipeline podle pořadí v paměti
- alternativou je zpracování mimo pořadí (*out-of-order*)

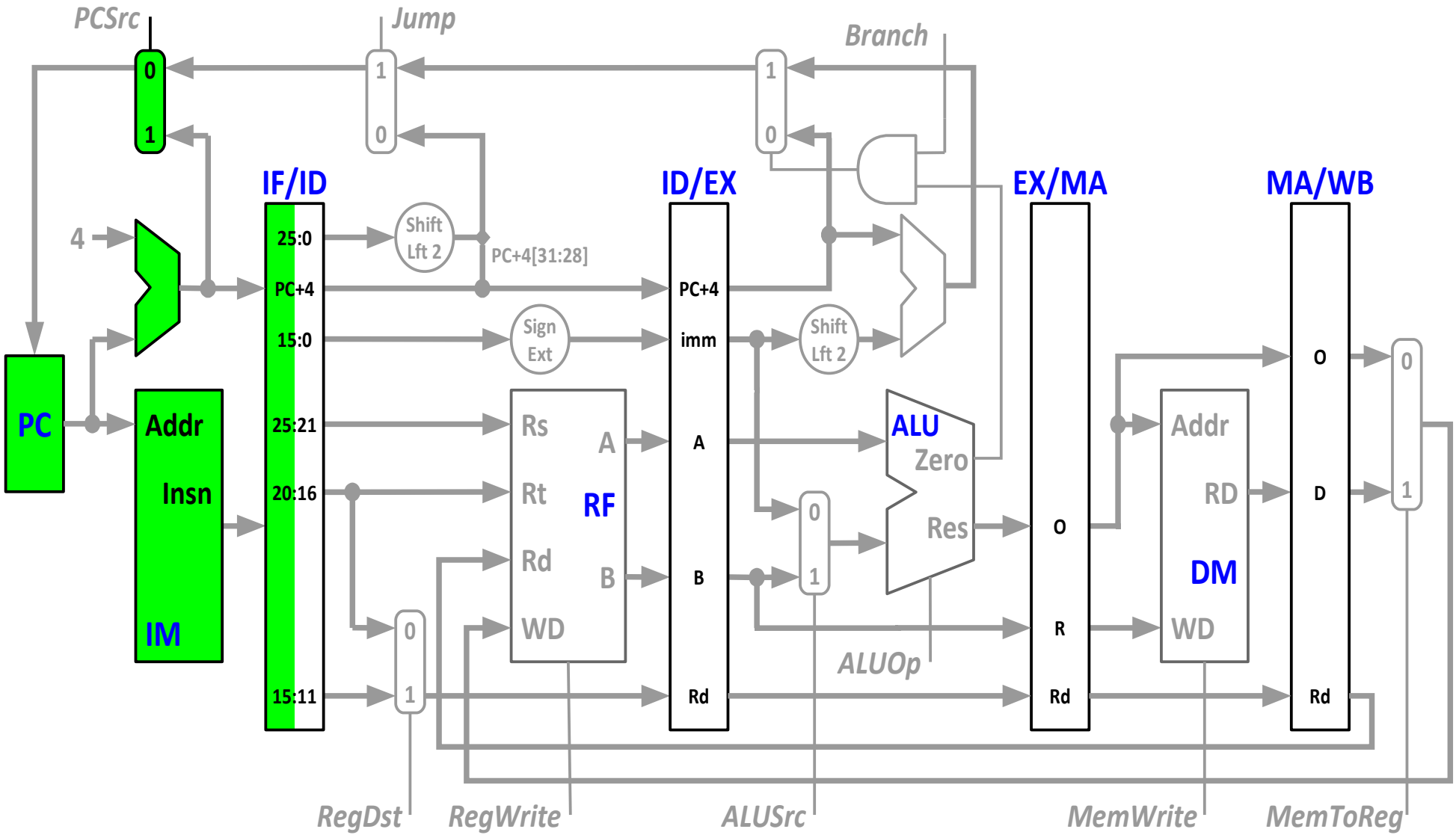
Délka pipeline (*pipeline depth*)

- počet stupňů/úseků (stages) pipeline
- 5 stupňů v případe MIPS "zrovna vyšlo"
  - trend jsou delší pipelines (> 10, Pentium 4 dokonce > 20)

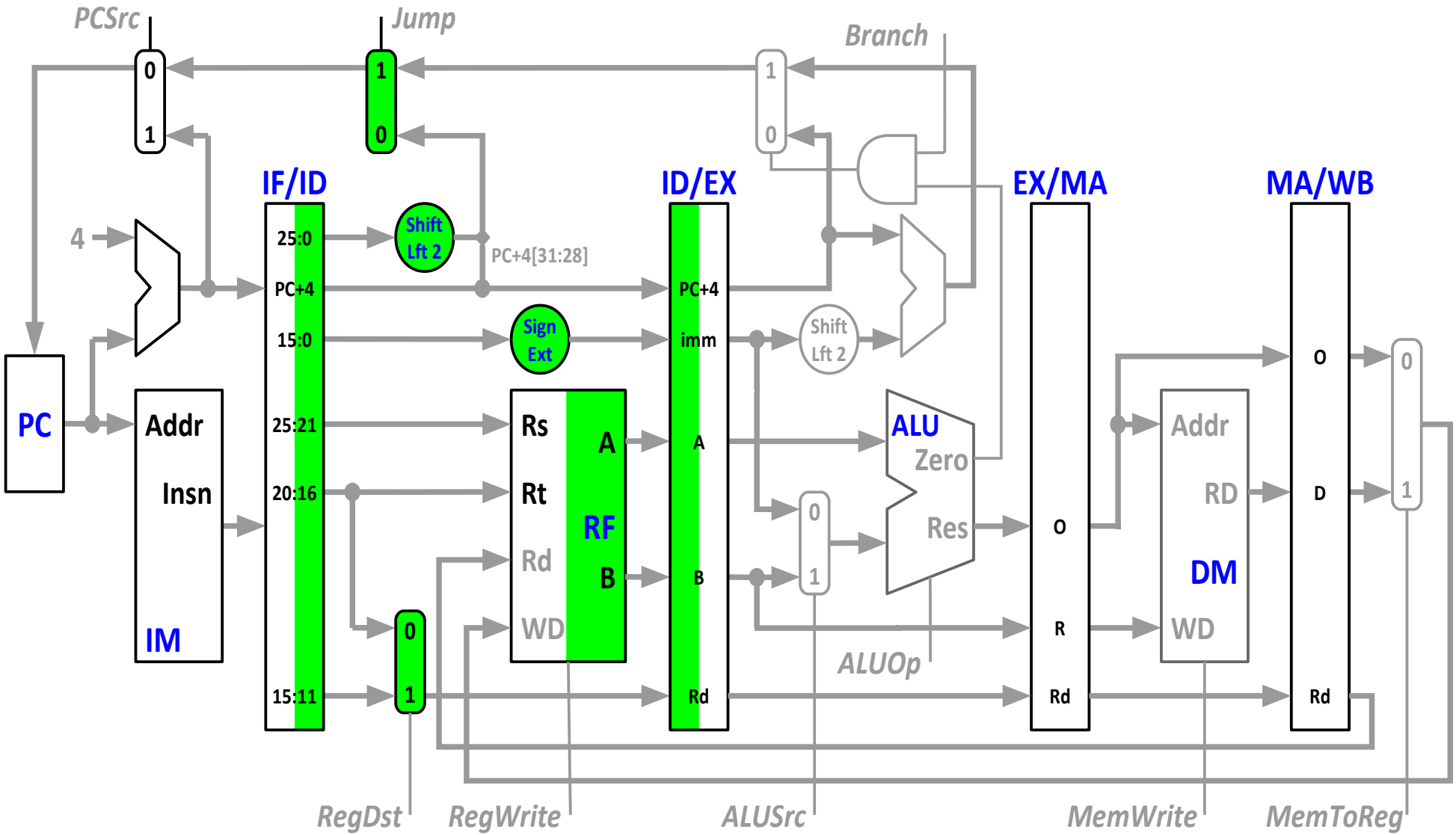
Příklad: zpracování 3 instrukcí, krok 1



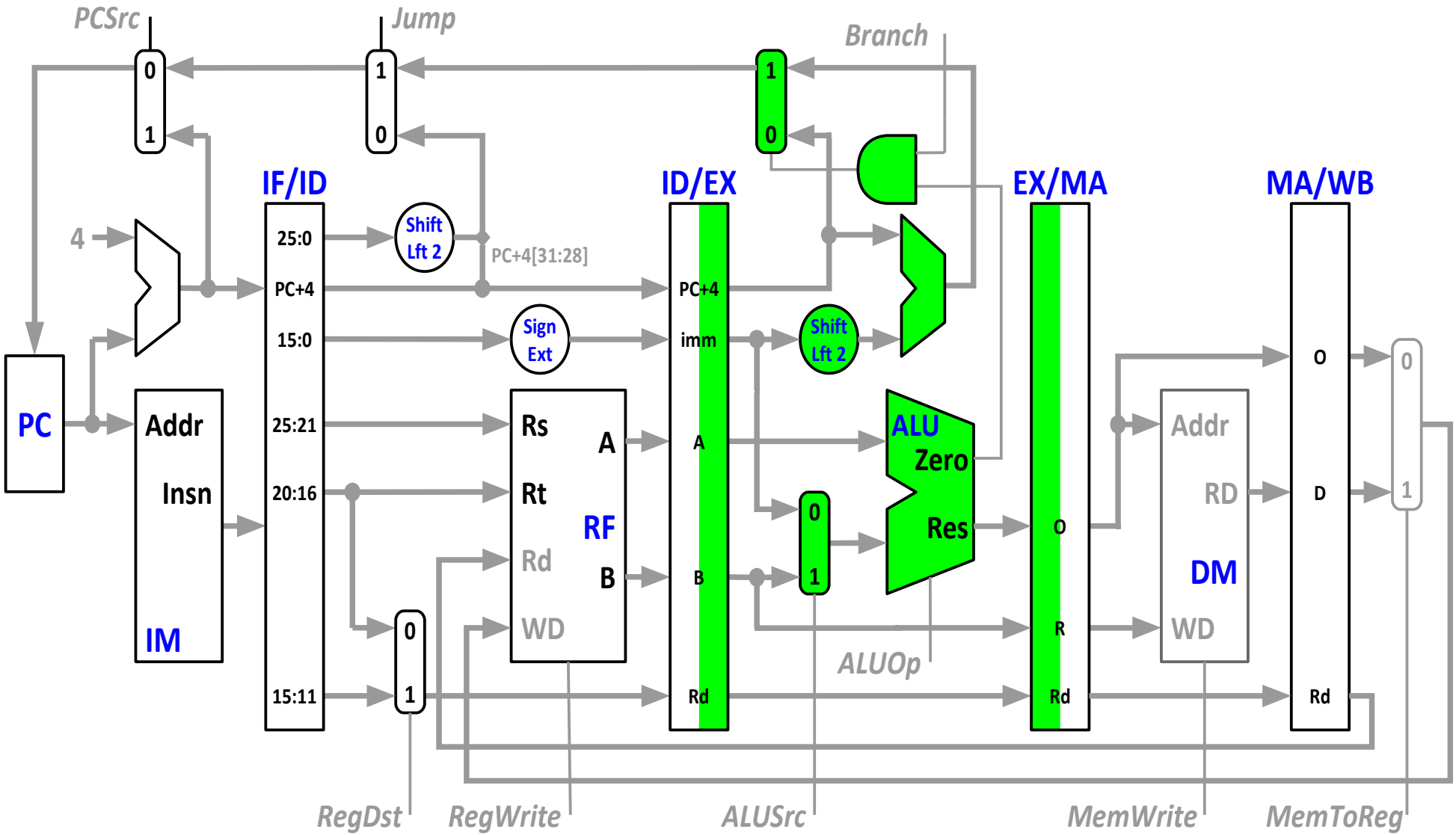
Příklad: zpracování 3 instrukcí, krok 2



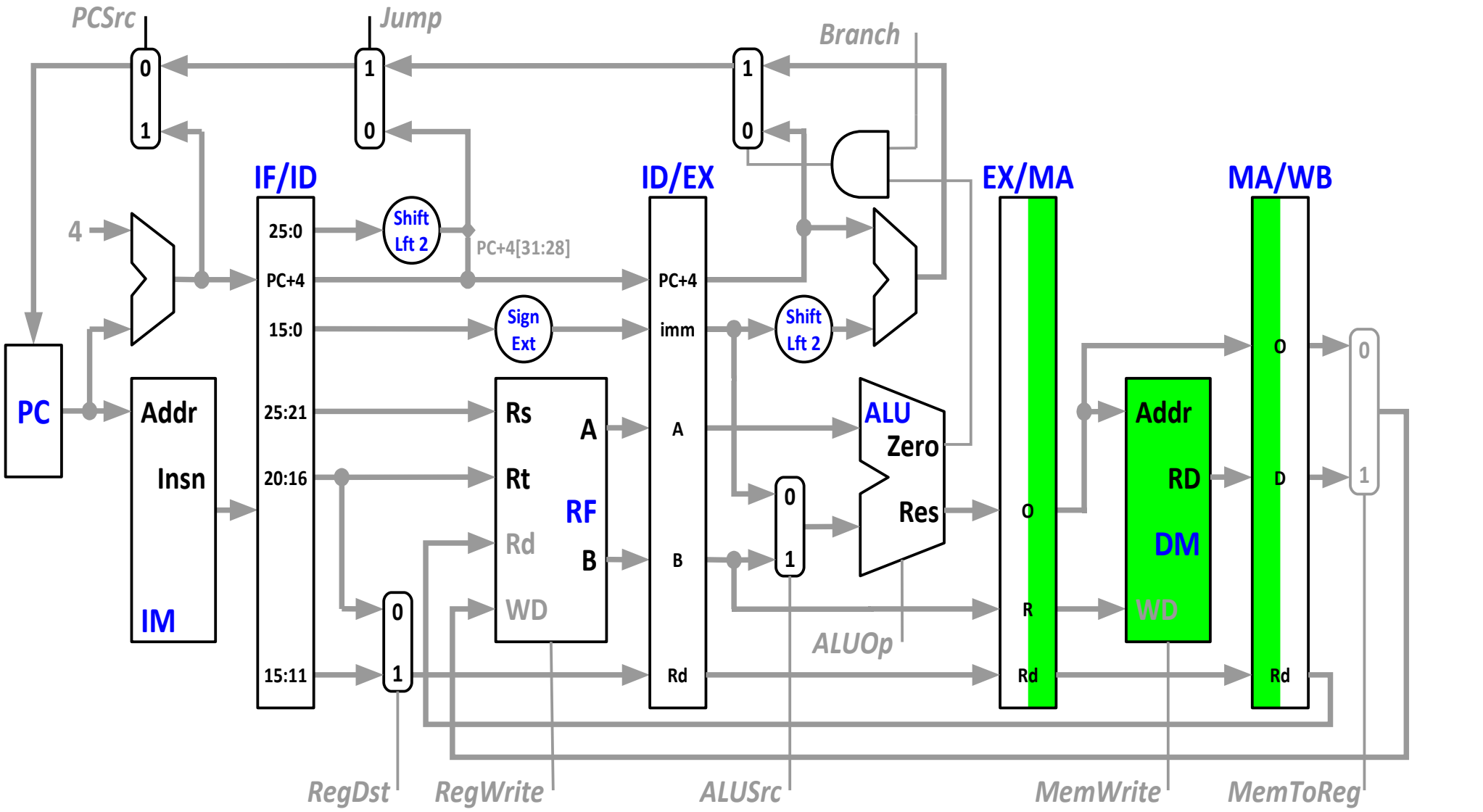
Příklad: zpracování 3 instrukcí, krok 3



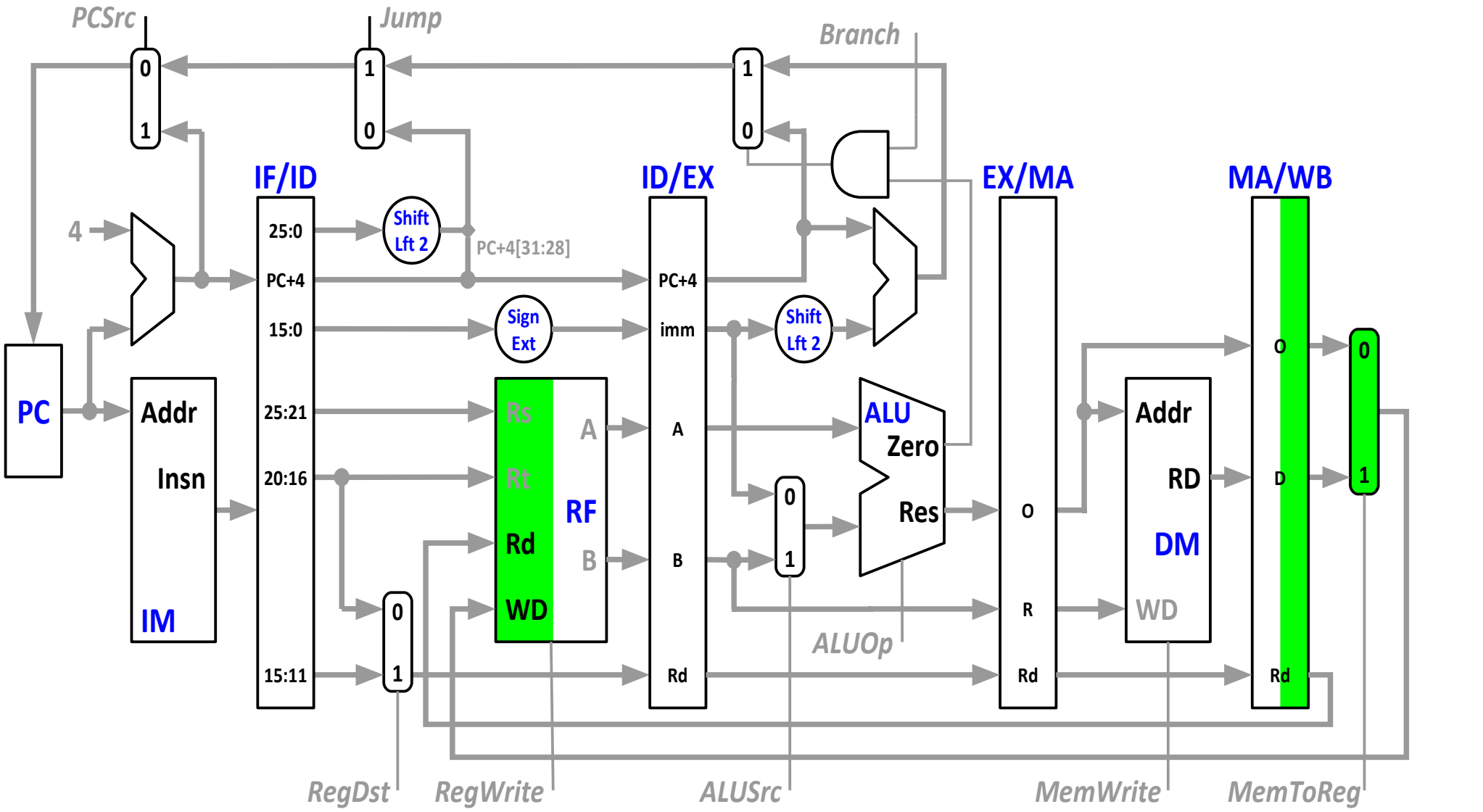
Příklad: zpracování 3 instrukcí, krok 4



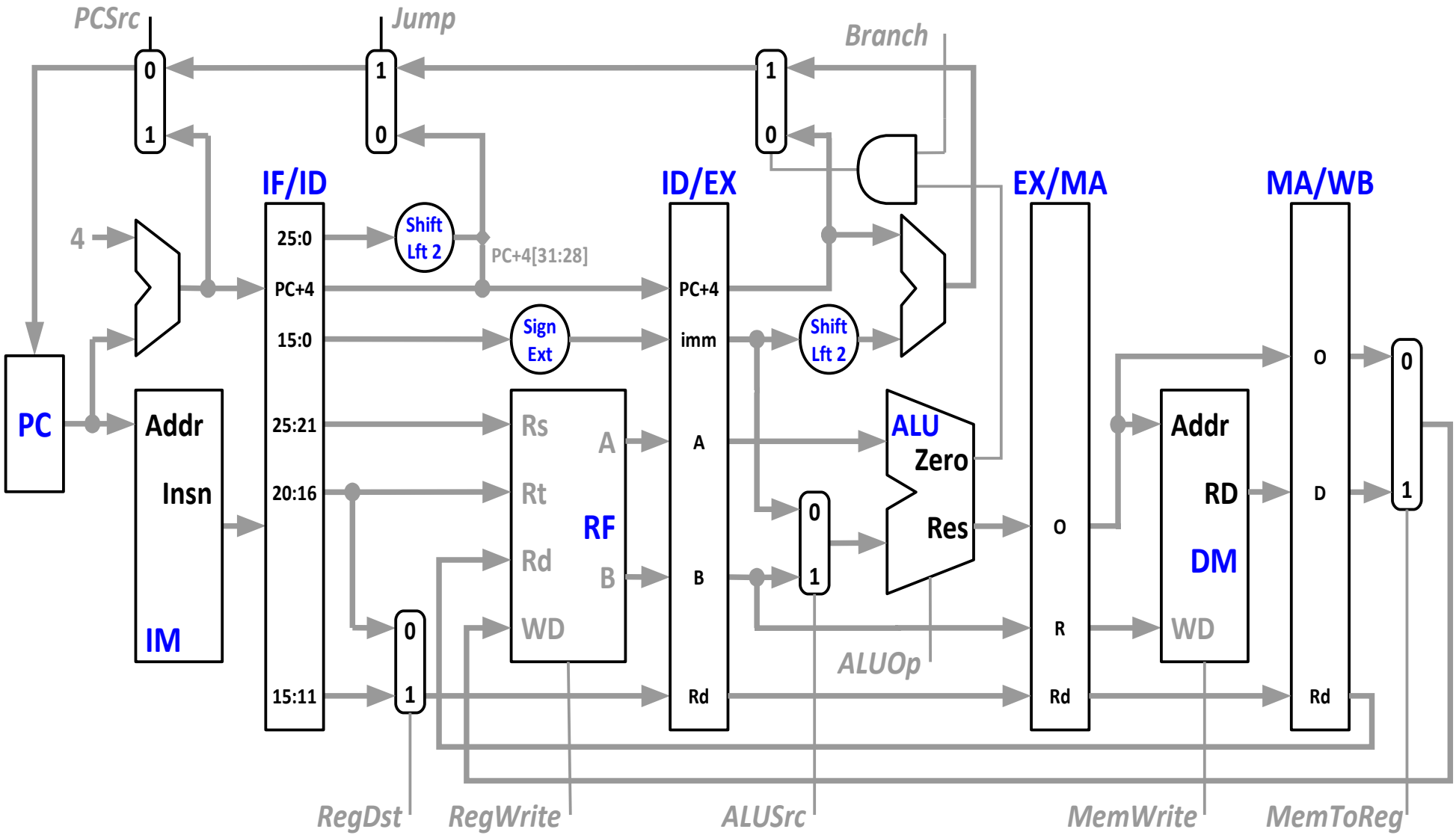
Příklad: zpracování 3 instrukcí, krok 5



Příklad: zpracování 3 instrukcí, krok 6



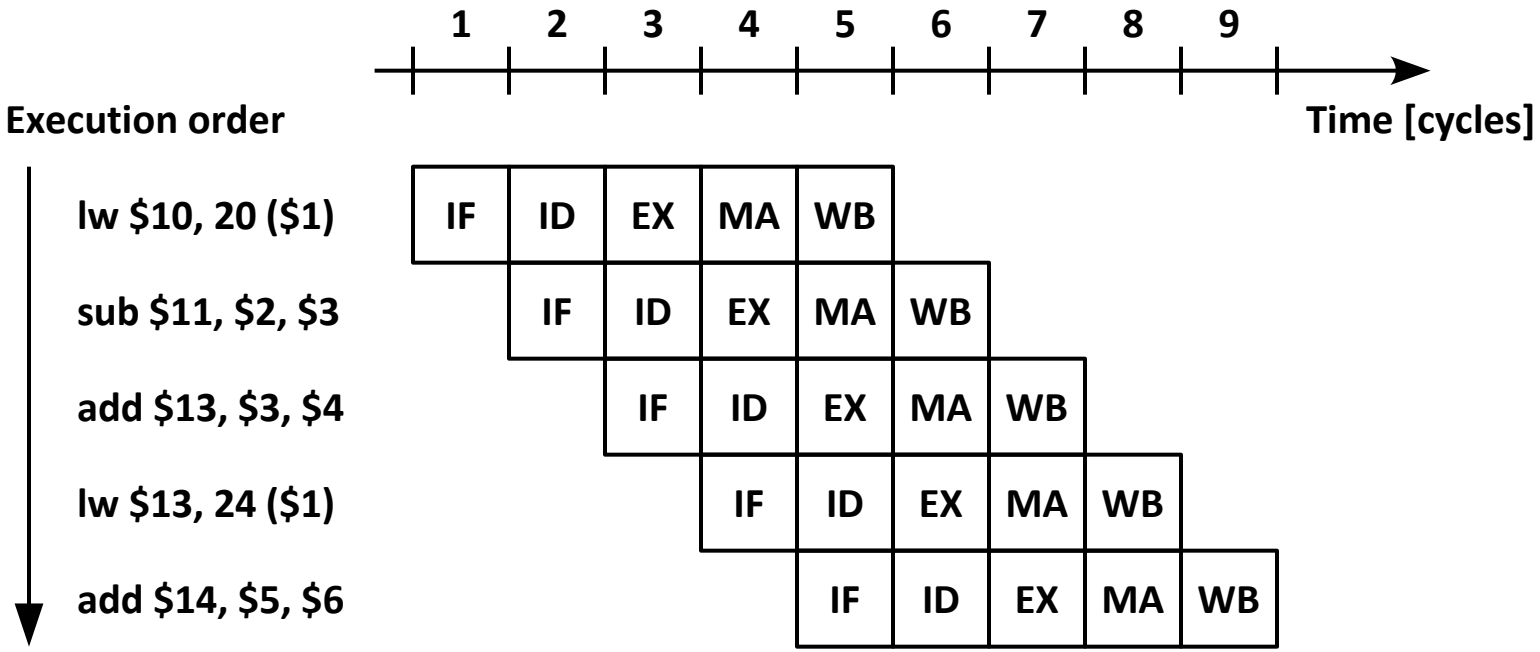
Příklad: zpracování 3 instrukcí, krok 7



Pipeline diagramy

Zjednodušená reprezentace pipeline

- všechny fáze zpracování trvají 1 cyklus
- relativní čas (v hodinových cyklech)
  - sloupec pro čas ***X*** ukazuje fázi zpracování jednotlivých instrukcí



Jak je to s řízením?

Mělo by být jako jednocyklové?

- ale signály musí být aktivovány postupně

Mělo by být jako vícecyklové?

- ale v datové cestě je více instrukcí současně

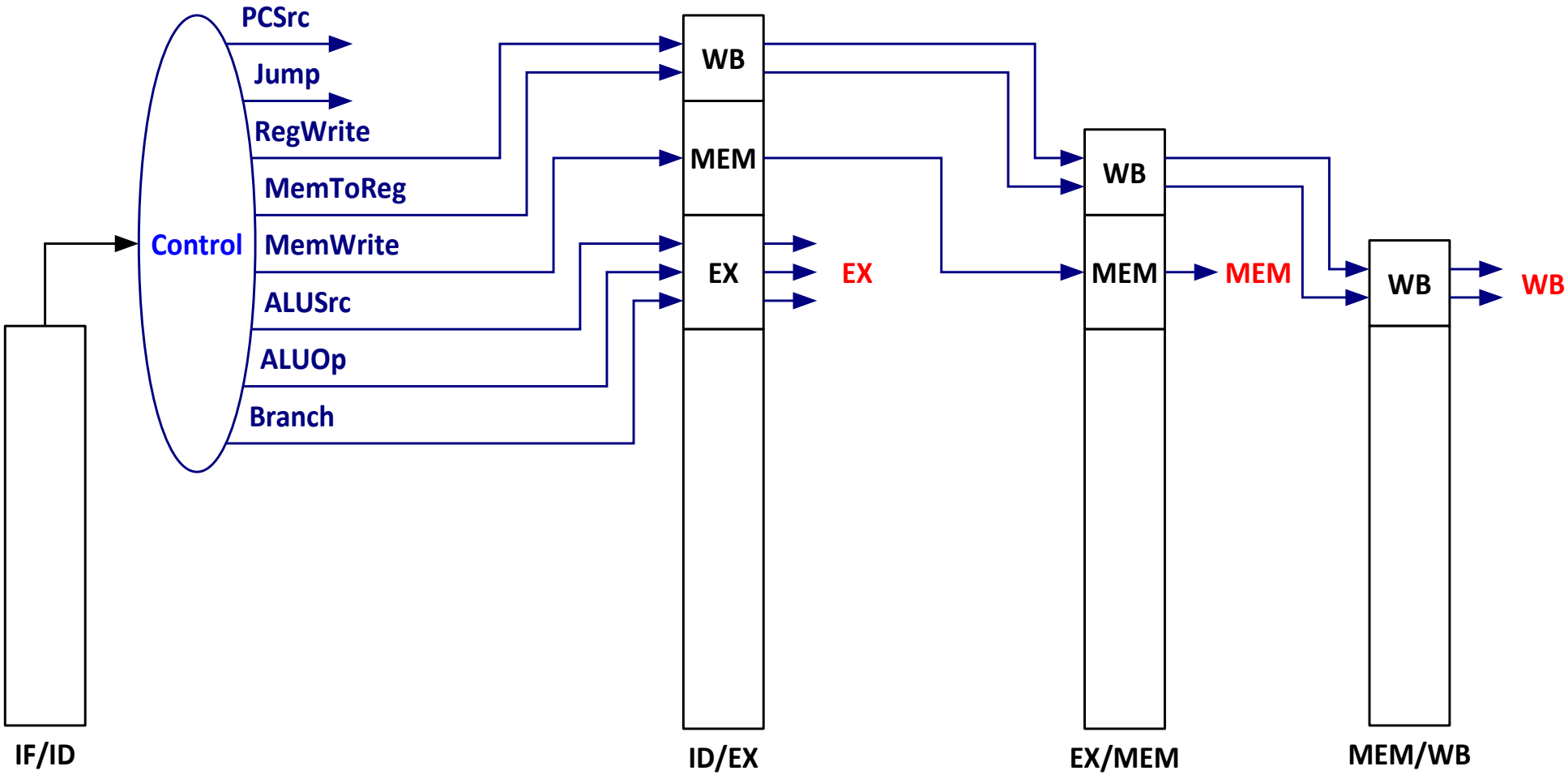
Kolik řadičů budeme potřebovat?

- jeden pro každou instrukci v pipeline?

Řešení: použijeme logiku z jednocyklového řadiče

- kombinační obvod dekódující operační kód
- ale stejně jako datovou cestu ho musíme zřetězit
  - signály zapsány do pipeline registru při načtení instrukce
  - každá instrukce si nese řídící signály "s sebou"

Zřetězené jednocyklové řízení



Výkonnost zřetězené datové cesty

Jednocyklová datová cesta

- takt = ***50ns***, ***CPI=1*** ⇒ ***50ns*** na instrukci

Vícecyklová datová cesta

- 20% branch (3t), 20% load (5t), 60% ALU (4t)
- takt = ***11ns, CPI*** ≈ ***(20% × 3) + (20% × 5) + (60% × 4) = 4***
- ***44ns*** na instrukci

Zřetězená datová cesta

- takt = ***12ns*** (cca 50ns/5 kroků + režie pipeline)
- ***CPI = 1*** (v každém cyklu je dokončena 1 instrukce)
  - ***12ns*** na instrukci? Ve skutečnosti ***CPI = 1 + režie > 1***
- ***CPI = 1.5*** ⇒ ***18ns*** na instrukci

Problémy při zřetězeném zpracování

Proč je CPI > 1?

CPI = 1 + penalizace za zpoždění

- zpoždění (*stalls*) se používají k řešení hazardů v pipeline
  - ***hazard*** = situace ohrožující iluzi VN modelu
  - ***zpoždění*** = prázdný krok vložený s cílem zachovat VN iluzi

Výpočet CPI

- (frekvence výskytu zpoždění) × (počet taktů zpoždění)
- penalizace se sčítají (v in-order pipeline se nepřekrývají)

Platí stále "make common case fast"?

- velké penalizace nevadí, pokud nenastávají moc často
  - CPI = 1 + 0.01 × 10 = 1.1
- penalizace ovlivňují ideální počet stupňů pipeline

Když všechno nejde úplně hladce...

Strukturální hazard

- hardware nepodporuje danou kombinaci instrukcí
- současný přístup k prostředku z více stupňů pipeline
  - oddělení instrukční a datové paměti (ve skutečnosti cache)

Datový hazard

- instrukce nemá k dispozici data pro vykonání
- čeká se na výsledek předchozí instrukce

Řídící hazard

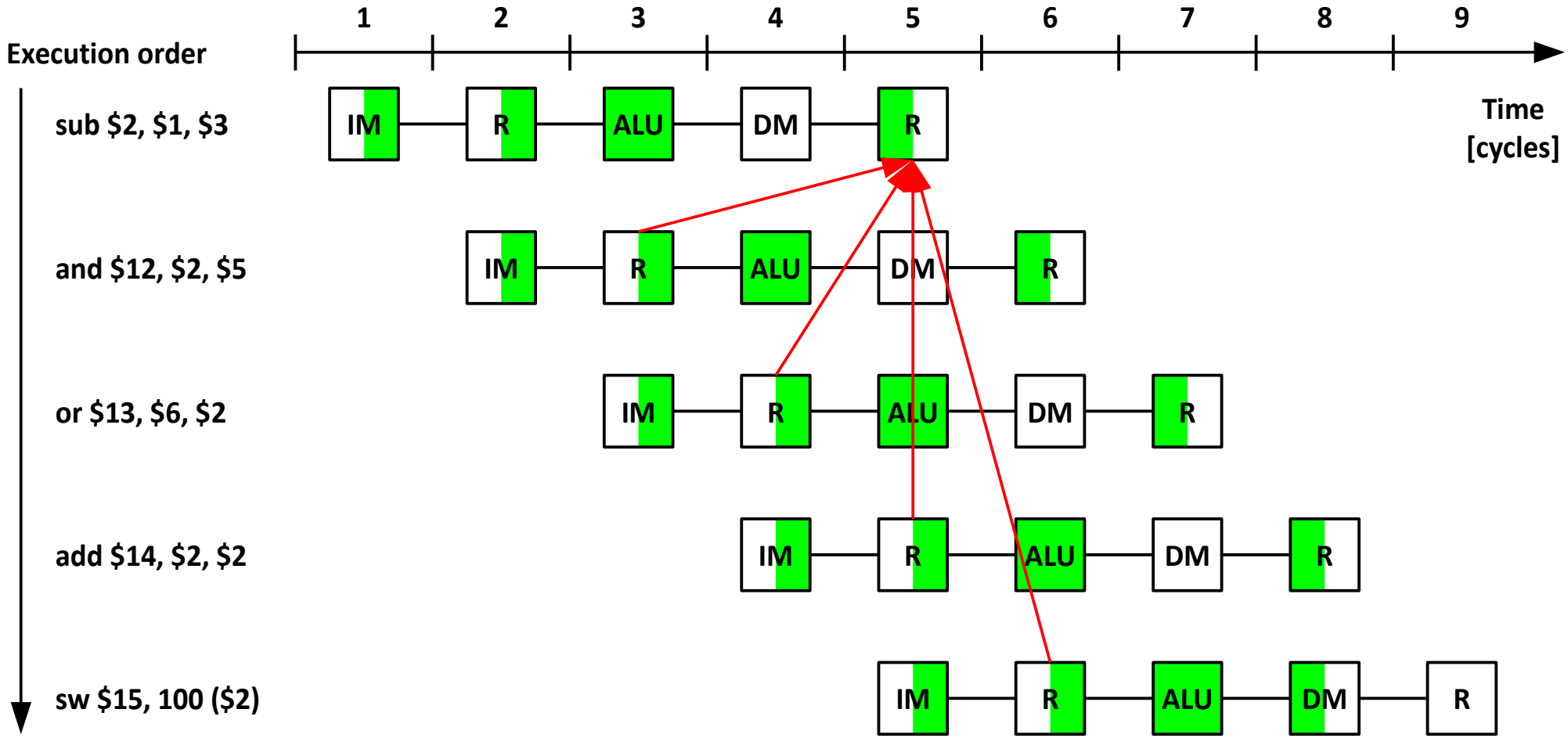
- nutno učinit rozhodnutí před vykonáním instrukce
- podmíněný skok je zpracován až ve 3. kroku
  - v té době uz pipeline zpracovává další 2 instrukce

Datový hazard

Závislosti mezi operandy instrukcí

- operand je výsledek předchozí operace
- operand je obsah paměti čtený předchozí instrukcí
- zjišťování závislostí (na papíře)
  - diagram pipeline + hrany označující závislost
  - závislosti jsou hrany vedoucí "dopředu v čase"
- zjišťování závislostí (v hardware)
  - logika vyhodnocující čísla registrů v instrukčním kódu všech instrukcí v pipeline

Datový hazard – závislosti operandů



Ošetření datových hazardů (1)

Psát programy tak, aby nenastal

- řadit instrukce tak, aby došly do pipeline až v okamžiku, kdy budou mít k dispozici data operandů

V hlavní roli překladač

- mezi dvojici závislých instrukcí vloží nezávislé instrukce
  - nezávislé = nepotřebují registr, který je předmětem závislosti
- překladač instrukce plánuje s ohledem na pipeline
  - přesouvá instrukce aby se vyhnul závislostem
  - pokud to nejde, musí vložit prázdné instrukce (***nop***)
- *software interlock*
  - *MIPS: **M**icroprocessor without **I**nterlocking **P**ipeline **S**tages*
  - komerční čistě SW řešení není vhodné – kompatibilita

Ošetření datových hazardů (2)

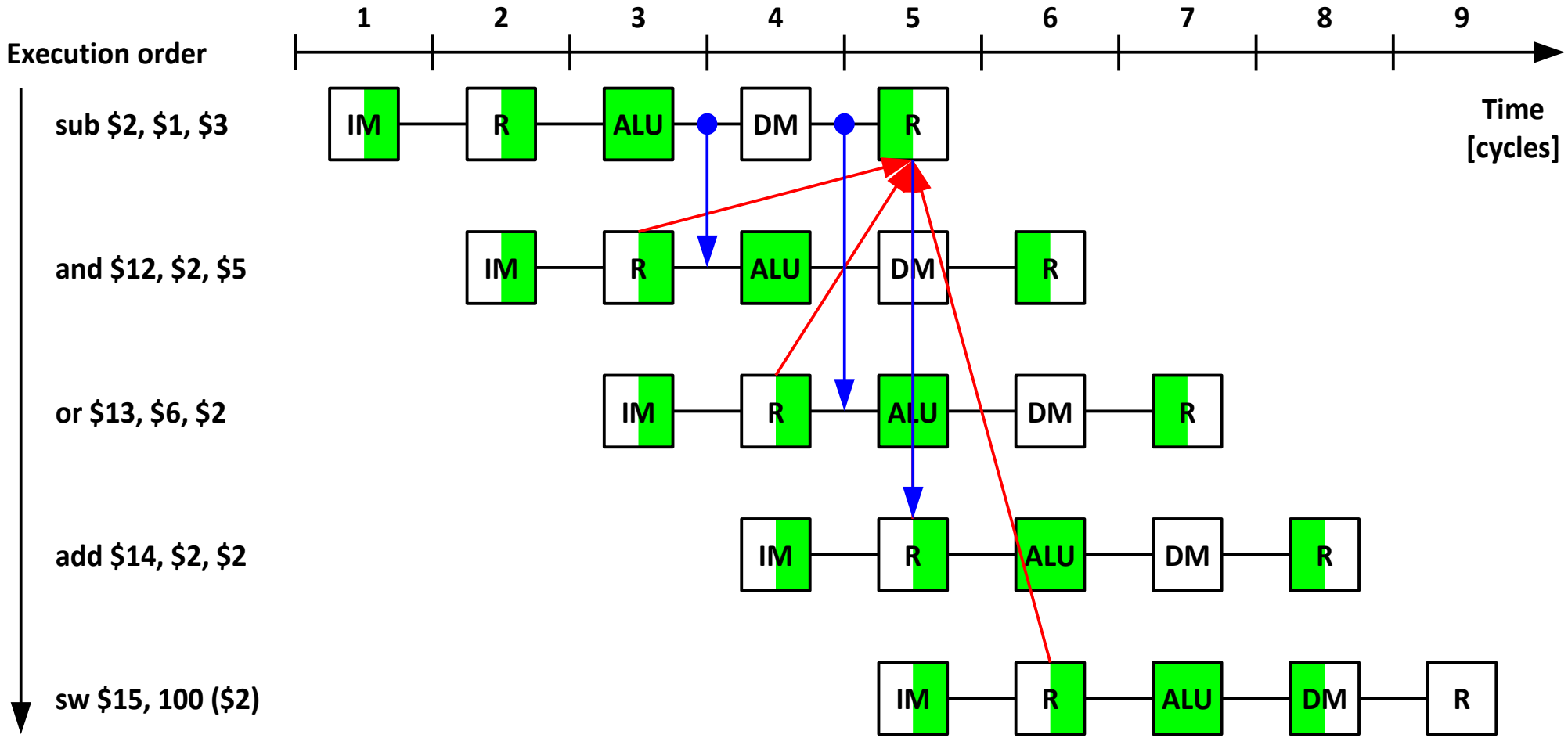
Forwarding/bypassing

- poskytnutí mezivýsledku následující instrukci
  - pokud je výsledek v některém z následujících pipeline registrů (předchozích instrukcí), je možné ho vzít z pipeline registru a nečekat na registrové pole

Forwarding unit

- zdrojový operand vykonávané instrukce je cílový operand výsledku dřívejší instrukce
  - EX/MA.RegisterRd = ID/EX.RegisterRs
  - EX/MA.RegisterRd = ID/EX.RegisterRt
  - MA/WB.RegisterRd = ID/EX.RegisterRs
  - MA/WB.RegisterRd = ID/EX.RegisterRt

Datový hazard – forwarding/bypassing



Ošetření datových hazardů (3)

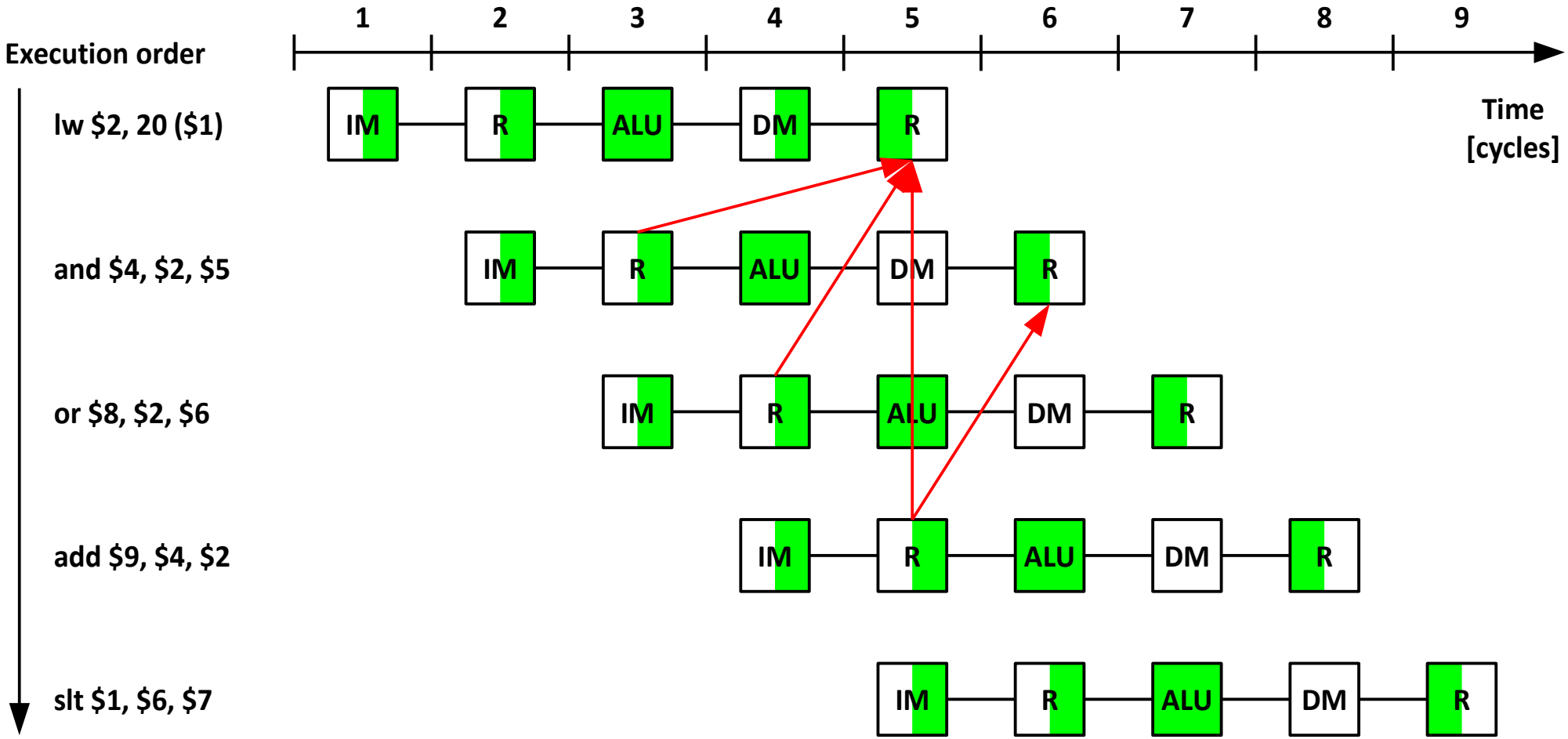
Zpoždění instrukce v pipeline

- použití operandu bezprostředně po načtení
  - load-use dependency

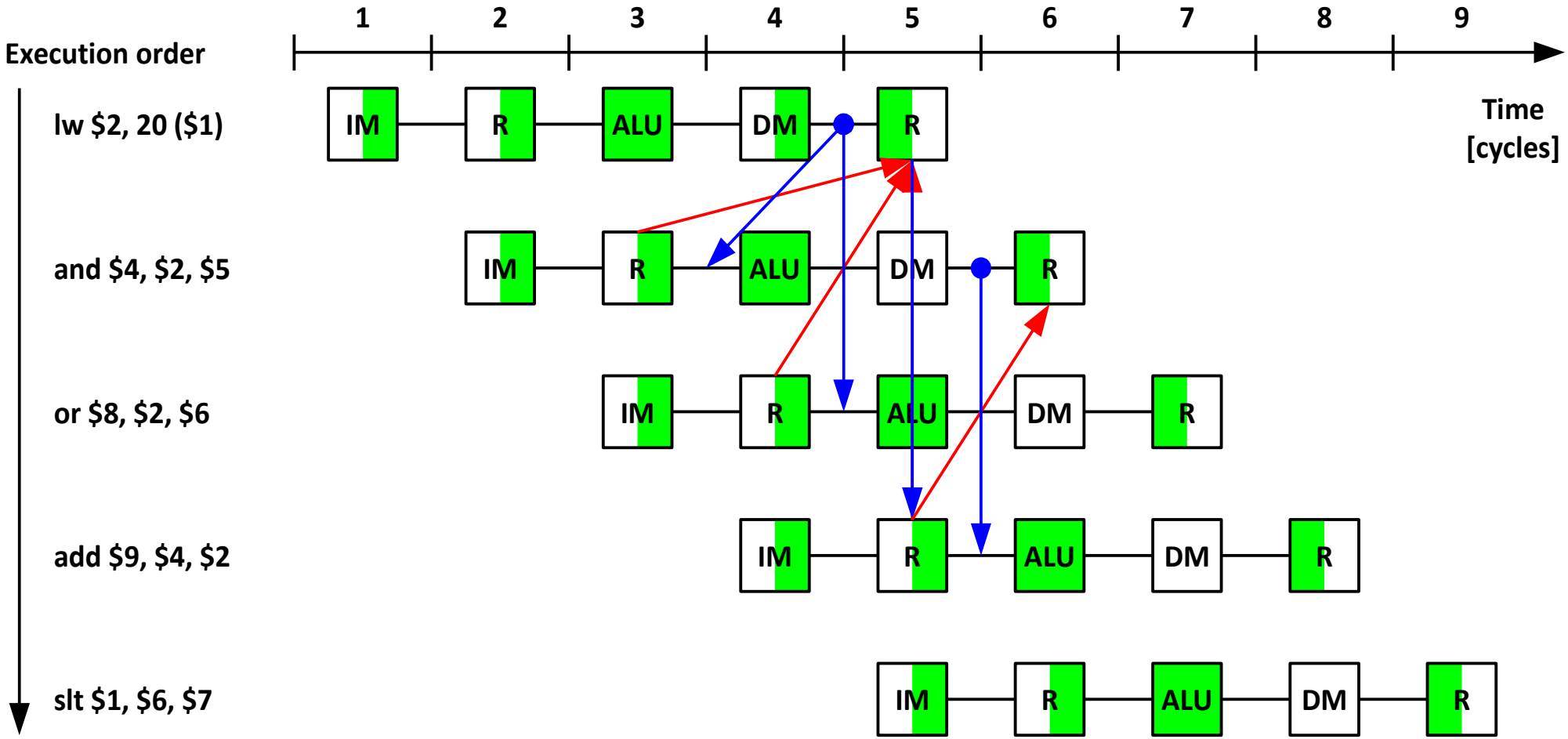
Hazard detection unit

- zdrojový operand dekódované instrukce je cílový operand dřívější instrukce čtení z paměti
  - ID/EX.MemRead and
    (ID/EX.RegisterRt = IF/ID.RegisterRs or
    ID/EX.RegisterRt = IF/ID.RegisterRt)

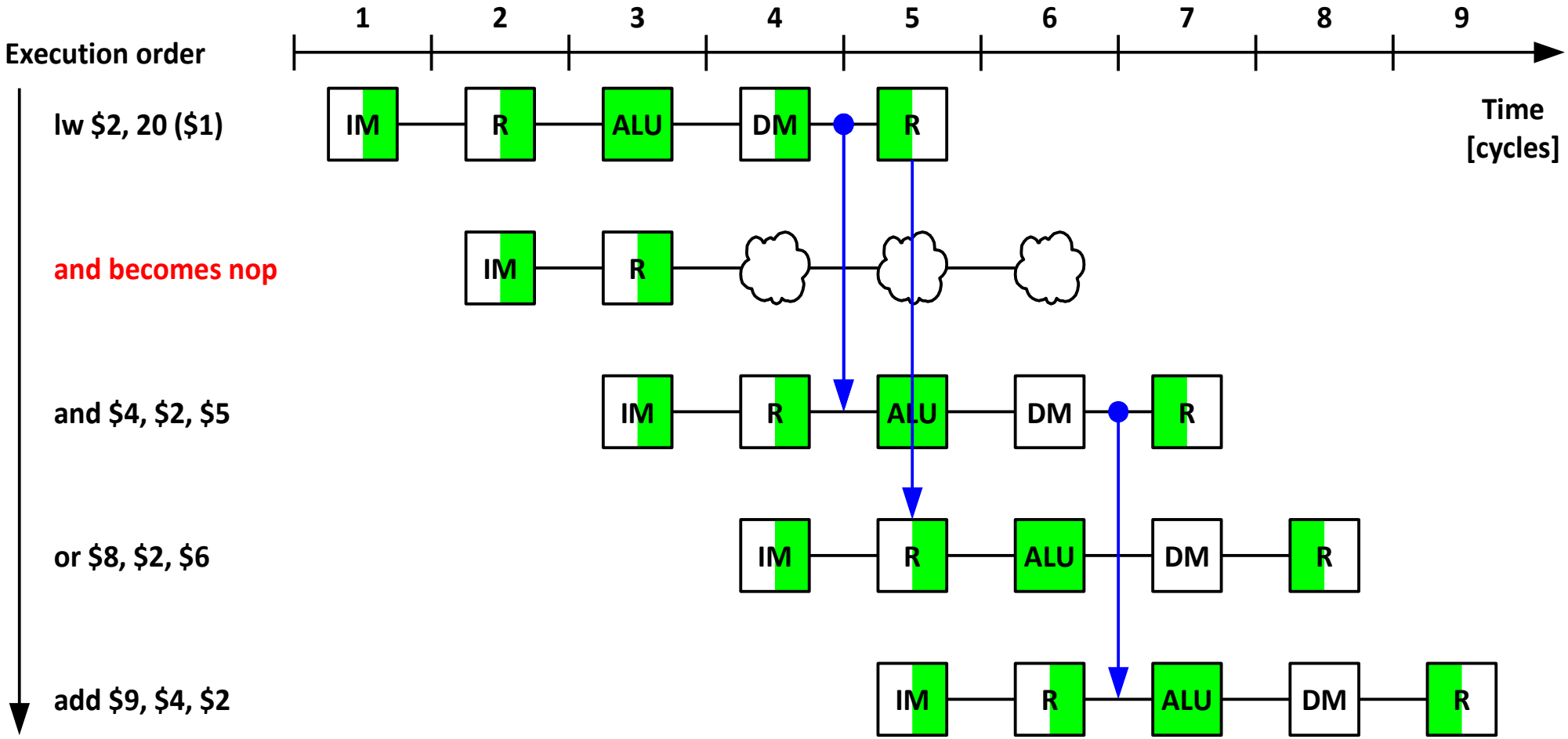
Datový hazard – load/use závislosti operandů



Datový hazard – load/use závislosti a forwarding



Datový hazard – zpoždění pipeline



Řídící hazard

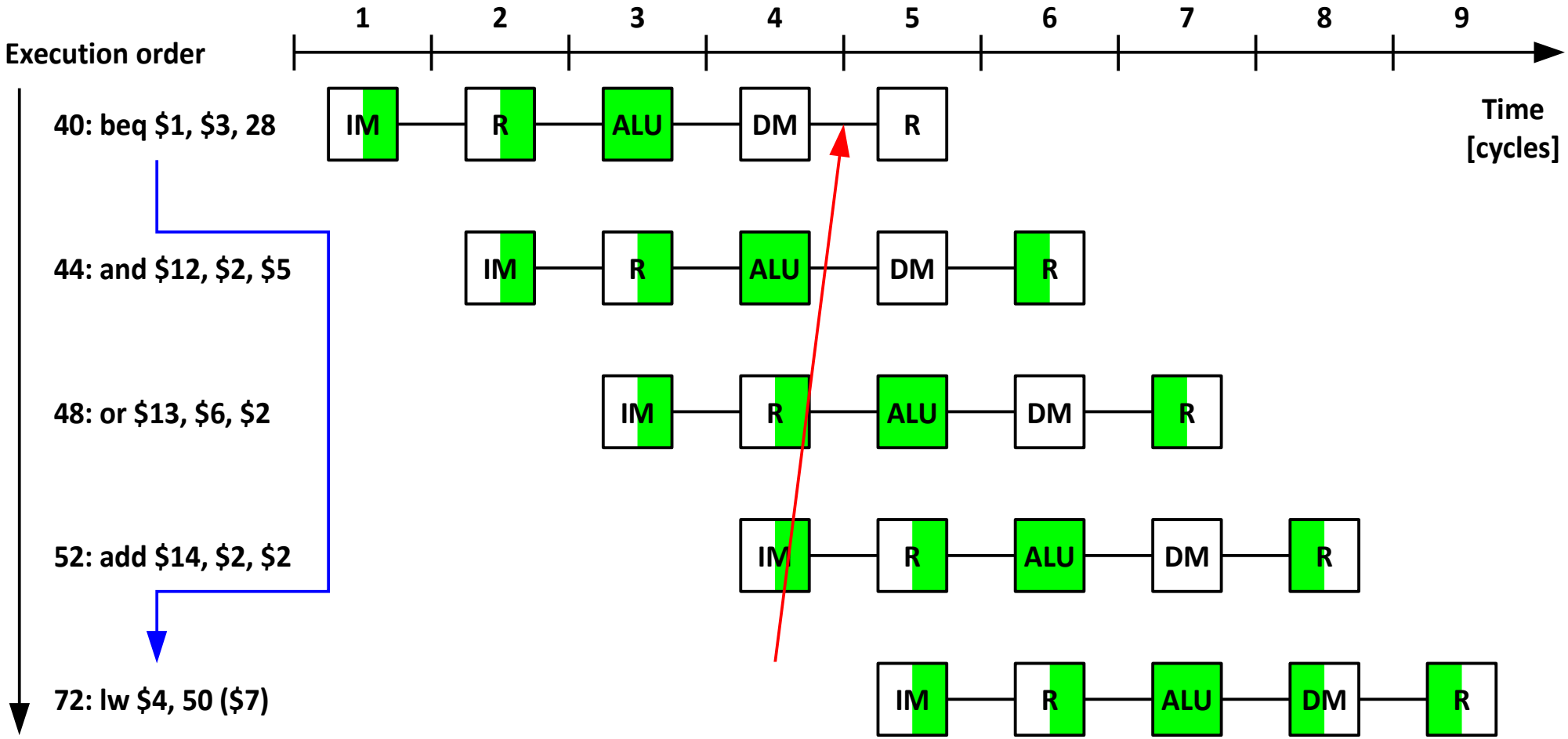
Nutno rozhodnout, odkud vzít další instrukci

- PC ovlivněn podmíněnými a nepodmíněnými skoky
  - rozhodnutí závisí na výsledku, který je k dispozici o několik taktů později, než je potřeba
- výjimky

Ošetření řídících hazardů

- forwarding nelze
  - i když je cílová adresa skoku k dispozici, neví se zda ji použít
- snaha o minimalizaci zpoždění pipeline

Řídící hazard – větvení



Ošetření řídícího hazardu

Obecně: snaha o udržení plné pipeline

- problém: odkud číst další instrukce?

Odkud tedy číst další instrukce?

- zpozdit pipeline do dokončení skoku
  - horší situace nastat nemůže → prostor pro zlepšení
- předpokládat že skok se neprovede
  - pipeline prostě čte následující instrukci
  - pokud se skok provedl, vyprázdnit pipeline
  - redukce zpoždění při provedeném skoku
- zkusit uhádnout, zda se skok provede
  - špatný odhad → penalizace jako v nejhorším případě

Jak hádat, zda se skok provede?

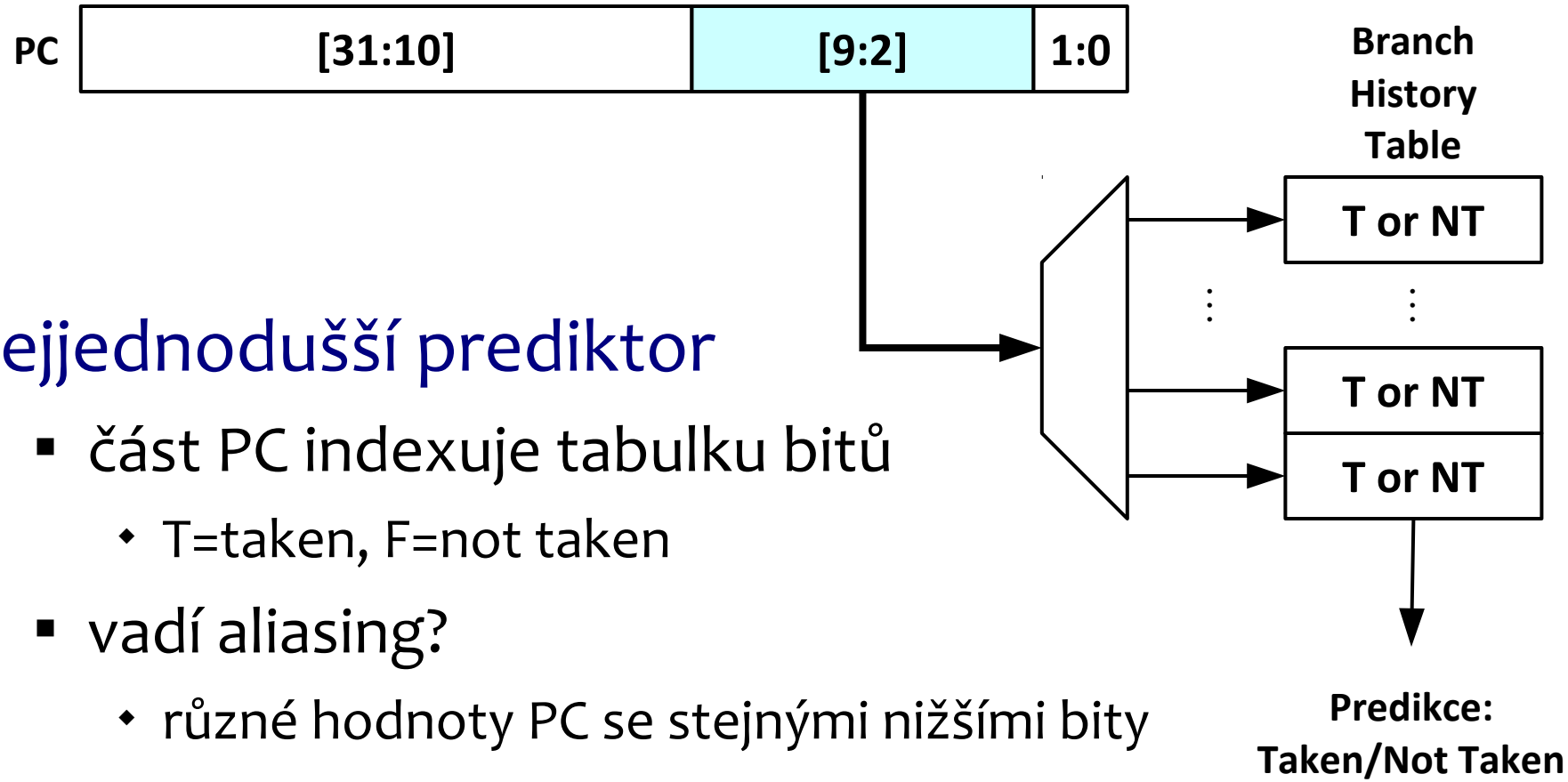
Statická predikce skoků

- predikce nezávisí na historii
- předpoklad určen HW, nebo bitem v instrukci

Dynamická predikce skoků

- predikce na základě historie
- branch prediction buffer/history table
  - predikce stejného chování jako v minulosti
  - cykly většinou skáčí na začátek → 2 špatné predikce
- 2-bitový prediktor
  - musí se 2x splést než změní "názor"

Branch History Table

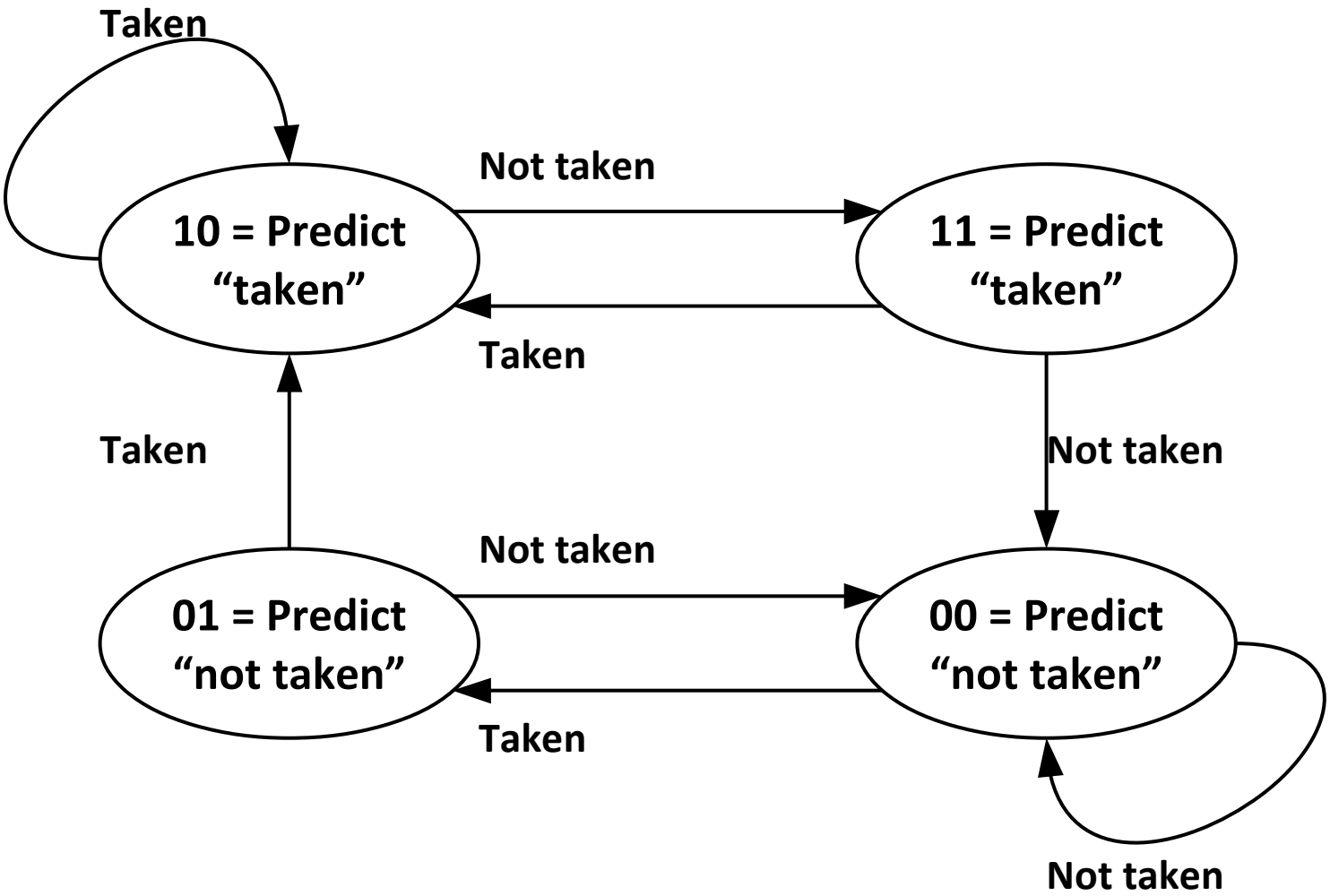


Nejjednodušší prediktor

- část PC indexuje tabulku bitů
  - T=taken, F=not taken
- vadí aliasing?
  - různé hodnoty PC se stejnými nižšími bity
- jak je to s vnořenými cykly?

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 4; j++)
        // dělej něco
```

2-bitový prediktor skoků (4 stavy)



Jak omezit zpoždění skoků?

Zrychlené vyhodnocení skoku

- přesun výpočtu adresy skoku a vyhodnocení podmínky ze stupňů EX a MA do stupně ID
- podmínkou je test na rovnost (jednoduchá realizace), ale vyžaduje forwardování mezivýsledků
- zpoždění 1 cyklus při skoku

Branch target buffer

- informace o cílové adrese specifické instrukce

Zpožděný skok

- vždy se vykoná 1 instrukce po instrukci skoku

Zřetězené zpracování a výjimky

V pipeline je *k* instrukcí současně

- když nastane výjimka, která instrukce ji způsobila?
  - informace se musí propagovat v pipeline registrech
- když nastanou dvě, kterou z nich obsloužit dřív?
  - tu která patří ke starší (dříve v pořadí) instrukci

Důraz na zachování správného stavu procesoru

- data z pipeline registrů se nikam nezapisují
  - registry a paměť obsahují hodnoty před výskytem výjimky
- při obsluze je nutné vyhodit z pipeline mladší instrukce
  - k tomu se dá využít logika pro nepovedené predikce

Další metody zrychlování procesorů

Prodloužení pipeline

Trend: delší pipelines

- 486 (5 stupňů), Pentium (7 stupňů), Pentium II/III (12 stupňů), Pentium 4 (22 stupňů, *super/deep-pipelining*)
- Core/Core2 (14 stupňů)

Jaké jsou důsledky prodlužování pipeline?

- vyšší taktovací frekvence
  - ale 2x delší pipeline neumožňuje 2x zvýšit frekvenci
- zvyšuje CPI
  - zvyšuje se penalizace za neuhádnuté skoky
  - prodlužují se zpoždění pro non-bypassed hazardy
- od určité délky způsobuje pokles výkonu (ale od jaké?)
  - 1GHz Pentium 4 bylo pomalejší než 800MHz Pentium III

Výkonnostní limit skalární pipeline

Skalární (*scalar, single issue*) pipeline

- v každém stupni pouze 1 instrukce
- výkonnost omezena ***CPI=IPC=1*** (*Flynn bottleneck*)
  - nelze dosáhnout (hazardy)
- výnosy z delších pipelines rychle klesají
  - deep/super-pipelining (hazardy + režie)

Superskalární (*superscalar, multiple issue*) pipeline

- v každém stupni (2, 3, 4, ...) instrukce
  - dnešní procesory typicky 4 (Intel Core 2, AMD Opteron)
- snaha o využití paralelismu na úrovni instrukcí
  - *instruction level parallelism (ILP)*
  - nezávislé instrukce se dají vykonávat současně

Paralelizmus na úrovni instrukcí

Překladač plánuje kód aby zabránil zdržením

- dokonce pro single-issue procesory (load/use závislost)
- ještě těžší pro multiple-issue

Kolik ILP se dá v programech "najít"?

- velmi závisí na typu programu
  - příklad: kopírování paměti – při rozbalení smyčky dostaneme velké množství nezávislých instrukcí
- v běžných programech ho zdaleka tolik není
- překladač musí ILP najít a využít

Typická superskalární pipeline

Čtení instrukcí

- celý blok cache (16, 32, nebo 64B), 4-16 instrukcí
- predikce jednoho skoku v každém cyklu

Paralelní dekódování instrukcí

- nutno detekovat hazardy a závislosti

Víceportové registrové pole registrů

- plocha, latence, spotřeba, cena, složitost

Více výkonných jednotek

- sčítačky jsou jednoduché, horší je forwarding logika

Přístup do paměti

- jedno čtení v každém cyklu stačí tak na 2-issue

Static multiple issue

Issue packet

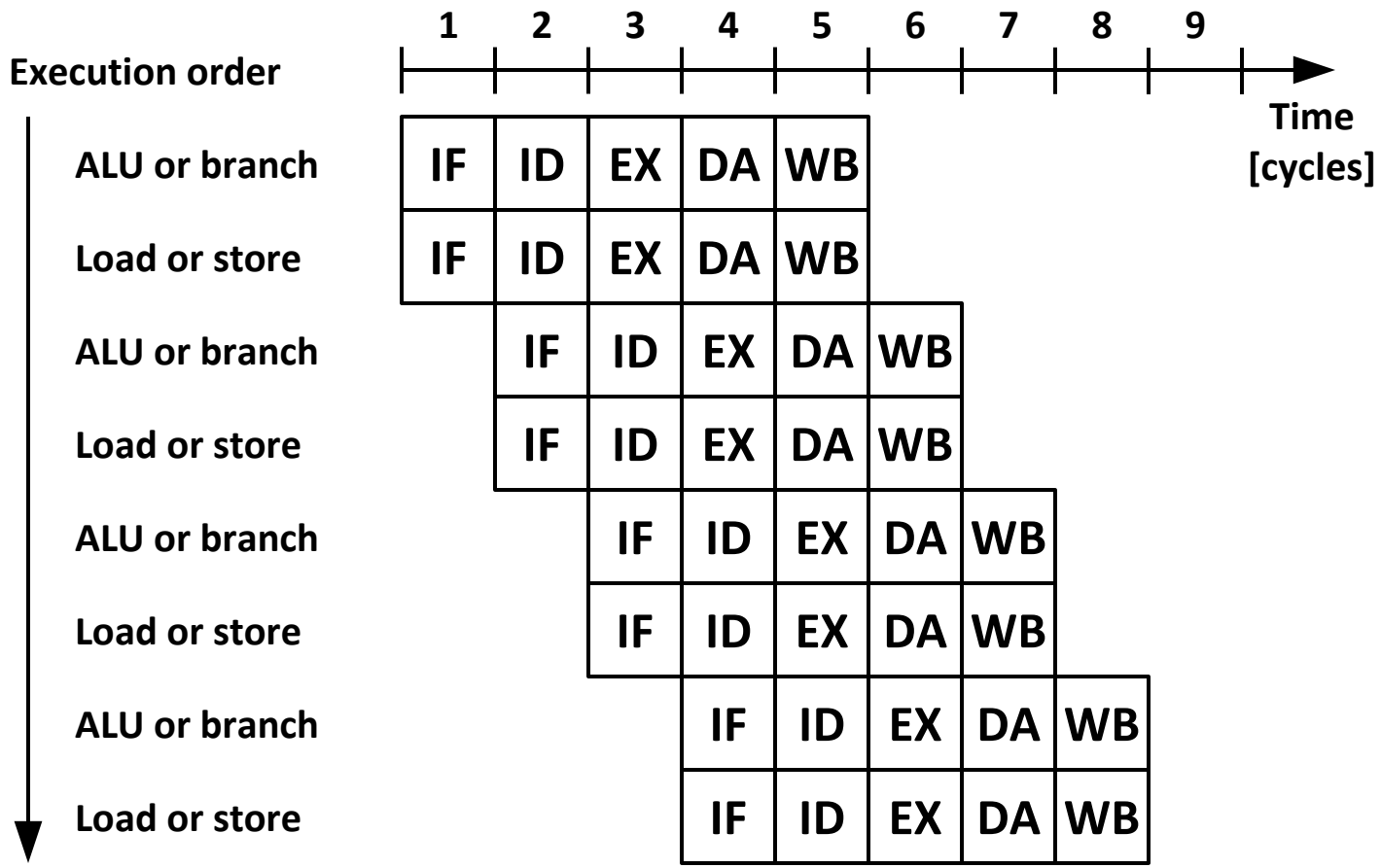
- sada instrukcí, které se mají provést zároveň
  - jednolivým instrukcím odpovídají sloty issue paketu
- sloty v issue paketu nejsou ortogonální
  - Very Long Instruction Word (VLIW)
  - Explicit Parallel Instruction Computer (EPIC)

Výkon velmi závisí na překladači

- plánování instrukcí/plnění instrukčních slotů
- omezení/odstranění datových hazardů
  - většinu/všechny důsledky hazardů řeší překladač
- statická predikce skoků a predikace instrukcí

Příklad: multiple issue MIPS ISA (1)

Vykonávání instrukcí v pipeline



Příklad: multiple issue MIPS ISA (2)

Změny oproti single issue

- načítání 64-bit instrukcí, zarovnání na 8 bajtů
  - nevyužitý slot obsahuje “nop”
- registrové pole pro přístup z obou instrukcí
- samostatná sčítačka pro výpočet adresy v paměti

Nevýhody

- vyšší latence při použití výsledků
  - složitejší plánování instrukcí překladačem
- prostoje v důsledku hazardů jsou “dražší”

Příklad: architektura IA-64 (1)

Hlavní rysy IA-64

- mnoho registrů
  - 128 GP, 128 FP, 8 branch, 64 condition
  - registrová okna s podporou přetečení do paměti
- instruction bundle
  - svazek instrukcí vykonávaný současně
  - pevný formát, explicitní závislosti
- podpora spekulace a eliminace větvení
  - lepší využití instrukčního paralelizmu

Příklad: architektura IA-64 (2)

Zajímavé vlastnosti

- instruction group
  - skupina instrukcí bez datových závislostí
  - skupiny odděleny speciálním indikátorem ***stop***
- struktura instrukčního svazku
  - 5 bitů template (použité výkonné jednotky)
  - 3 x 41 bitů instrukce
- predikace instrukcí (predication)
  - většina instrukcí může záviset na podmínkovém registru
  - 6 bitů výběr 1 z 64 podmínkových registrů

Dynamic multiple issue

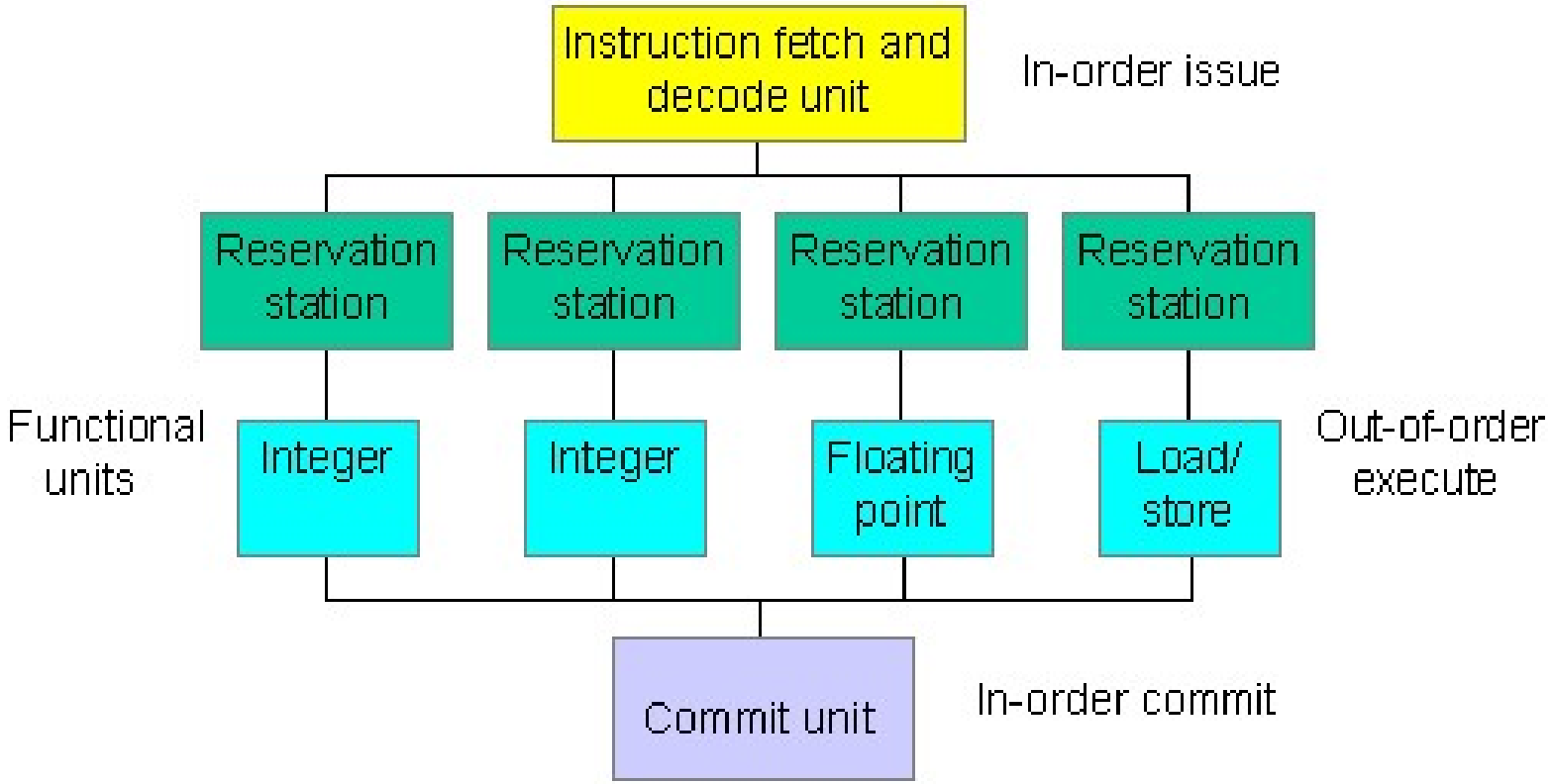
Dynamické plánování instrukcí v pipeline

- procesor vybírá instrukce ke zpracování
  - cílem je najít a využít ILP a eliminovat hazardy a prostoje v důsledku závislostí
- instrukce jsou zpracovávány mimo pořadí
  - *out-of-order execution (OOE)*
  - navenek se "tváří" jako VN
- překladač se procesoru snaží plánování usnadnit

Spekulativní provádění instrukcí

- občas se vykonávají i instrukce, u nichž není jisté, jestli mají správné operandy nebo jestli se výsledek použije

Typické uspořádání



Příklad: out-of-order execution

```
01          LOAD R2,A
02          ADD  R1,R2,R3
03          BPOS R1,LAB1   (Taken)
04          LOAD R4,B
05          BNEG R4,LAB2
06 LAB1:    LOAD R4,C
07          ADD  R5,R4,R3
08 LAB2:    SUB  R5,R7,R0
09          BPOS R5,LAB3   (NOT Taken)
10          ADD  R5,R0,R3
```

```
01          LOAD R2,A
06 LAB1:    LOAD R4,C
08 LAB2:    SUB  R5,R7,R0
10          ADD  R5,R0,R3
```

```
02          ADD  R1,R2,R3
07          ADD  R5,R4,R3
09          BPOS R5,LAB3   (NOT Taken)
```

```
03          BPOS R1,LAB1   (Taken)
```

- procesor funguje jako by byl řízen daty

Datové závislosti při přerovnání instrukcí

Kolize mezi jmény registrů v instrukcích

- True data dependency – RAW (Read after Write)
  - výstup instrukce je použit jako vstup následující
- Output dependency – WAW (Write after Write)
  - dvě instrukce zapisují na stejné místo
  - výsledek musí odpovídat pozdější instrukci
- Anti-dependency – WAR (Write after Read)
  - zatímco jedna instrukce zpracovává data, další instrukce tato data změní

- WAW a WAR lze vyřešit "přejmenováním" registrů
  - procesor má fyzicky více registrů, než definuje architektura

Příklad: eliminace WAW

Out-of-order kód z pohledu procesoru

```
MOVE r3, r7
ADD r3, r4, r5
MOVE r1, r3
...
```

Out-of-order kód po přejmenování registrů

```
MOVE fr3, r7
ADD fr8, r4, r5
MOVE r1, fr8
...
```

Zpracování výjimek

Ještě horší, než skalární pipeline...

- při výskytu výjimky musí procesor "zastavit" v místě určeném instrukcí, která výjimku vyvolala
  - při zpracování mimo pořadí ale nemusela být na řadě
- následující instrukce nesmí ovlivnit stav stroje
- nesmí existovat nezpracované starší instrukce
- výjimky způsobené staršími instrukcemi jsou vyřízeny

Precizní přerušení/výjimka

- vždy spojena se správnou instrukcí

Spekulativní provádění instrukcí

Odhadnutí vlastností/výsledku instrukce

- umožňuje zahájit zpracování závislých instrukcí
- nutno zajistit vždy korektní výsledek

Spekulace při překladu

- speciální kód pro opravu chybných spekulací

Spekulace v procesoru

- spekulativní výsledky kumulovány v procesoru

Problém s výjimkami

- nevyvolávat dokud jsou spekulativní

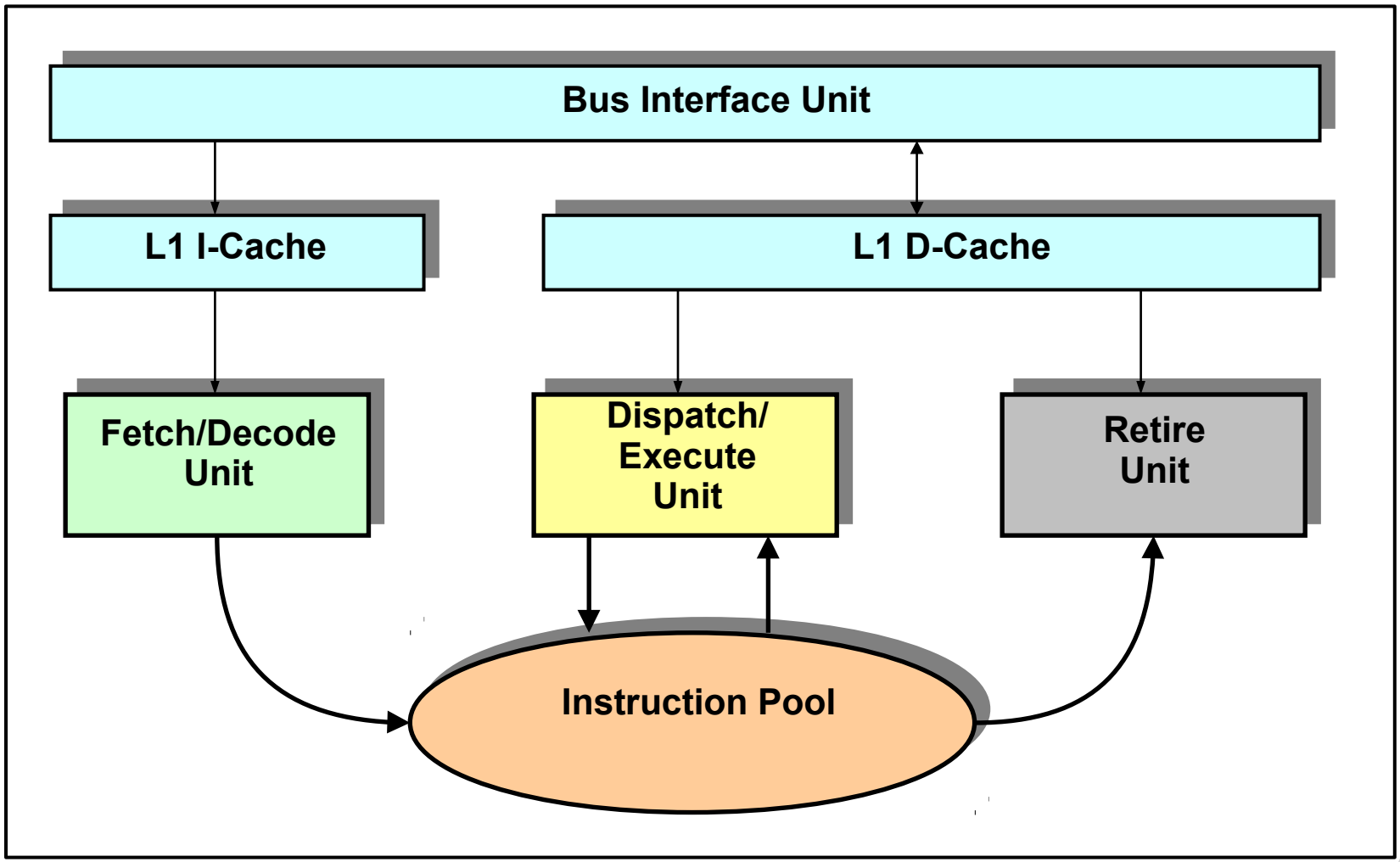
Příklad: architektura IA-32 (1)

Intel Pentium Pro až Pentium 4

- instrukční sada CISC interpretována mikrooperacemi na jádře post-RISC
- instrukce rozkládány na mikroinstrukce
- pipeline provádí mikroinstrukce
- superskalární, spekulativní provádění instrukcí

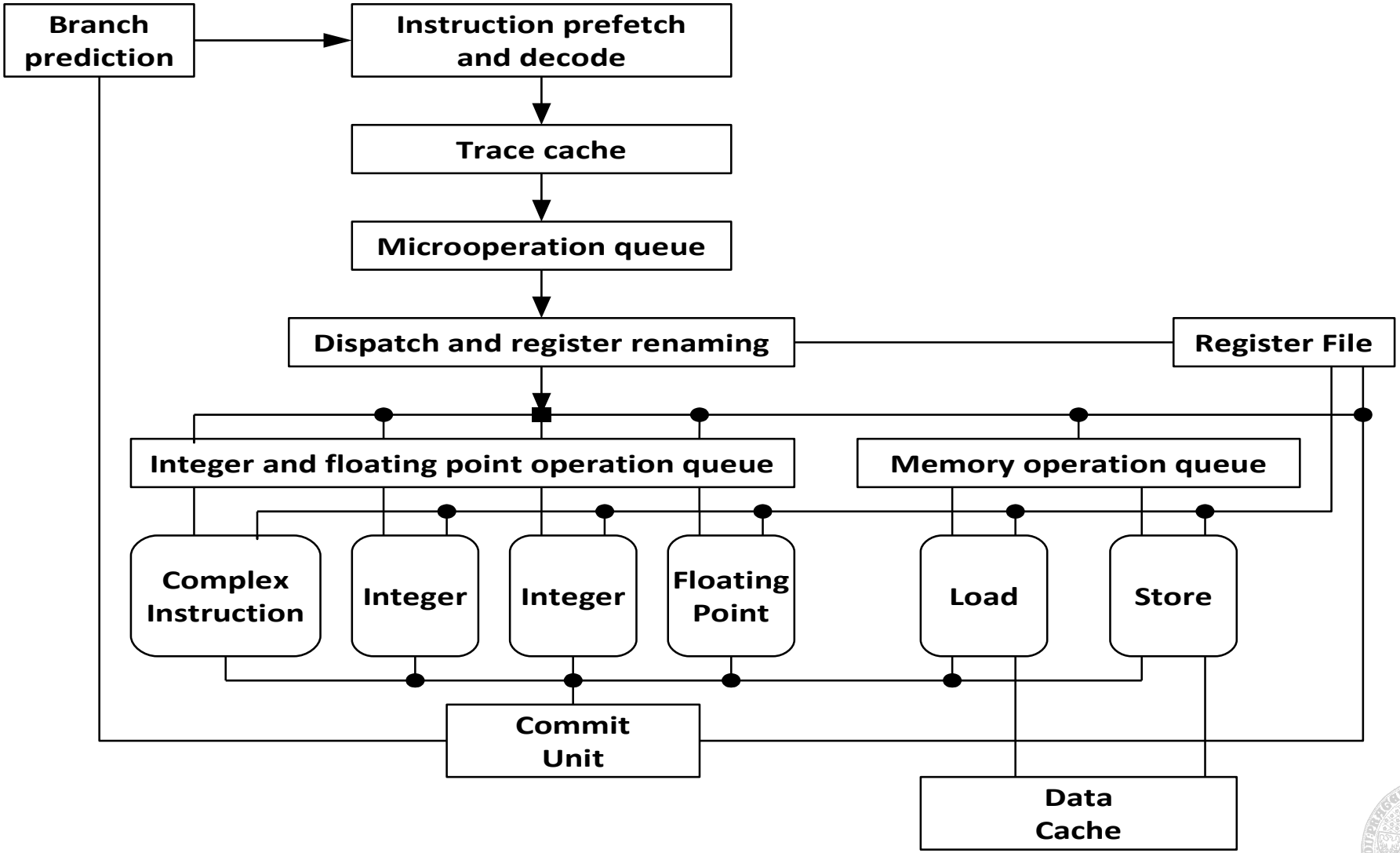
Příklad: architektura IA-32 (2)

Základní struktura



Příklad: architektura IA-32 (3)

Intel Pentium 4



Příklad: architektura IA-32 (4)

Intel Pentium 4 vs. Pentium III

- dvojnásobná délka pipeline (cca 20 stupňů vs. 10)
- více funkčních jednotek (7 vs. 5)
- podpora více rozpracovaných operací (126 vs. 40)
- trace cache (urychluje dekódování instrukce)
- lepší prediktor skoků (4K položek vs. 512)
- vylepšený paměťový subsystém

RISC vs. CISC

Další způsoby zvyšování výkonnosti

Návrh ISA pro pipelining, multiple-issue, atd.

- důraz kladen na návrh "dobře zpracovatelné" instrukční sady, která umožní realizovat uvedené techniky zvyšování výkonu

Simultánní multithreading

- do pipeline se zavádějí instrukce různých vláken, které na sobě skoro jistě nebudou závislé

Návrh ISA pro výkonné procesory (1)

Statistika využití instrukcí (IBM/360)

skupina	zastoupení
přesuny dat	45,28%
řízení	28,73%
aritmetika	10,75%
porovnávání	5,92%
logické operace	3,91%
posuny, rotace	2,93%
bitové operace	2,05%
I/O a ostatní	0,43%

Návrh ISA pro výkonné procesory (2)

Další pozorování

- 56% konstant je v rozsahu ±15 (5 bitů)
- 98% konstant je v rozsahu ±511 (10 bitů)
- 95% podprogramů potřebuje pro předání parametrů méně než 24 bytů

Výzkum DEC: v typickém programu bylo

- použito 58% instrukční sady
- pro 98% instrukcí stačilo pouhých 15% firmware

Návrh ISA pro výkonné procesory (3)

Změna priorit při návrhu ISA

- původní cíl
  - rozsáhlé instrukční sady, složité instrukce
  - překlenutí sémantické mezery mezi assemblerem a vyšším programovacím jazykem
- nový cíl
  - snaha o jednoduché instrukce → rychlé provádění
  - rychlejší paměti → složité instrukce nejsou klíčové
  - optimalizující kompilátory vyšších jazyků

Vznik procesorů typu RISC (1)

Reduced Instruction Set Computer

- omezená a jednoduchá instrukční sada
- velké množství obecných registrů
- důraz na optimalizaci pipeline

Výzkumné systémy

- RISC-1 (1981, Berkeley University)
- MIPS (1982, Stanford University)
- IBM-801

Vznik procesorů typu RISC (2)

Charakteristické rysy RISC procesorů

- instrukční sada
  - pevný formát instrukce, velké množství registrů
  - malý počet a jednoduché instrukce a adresovací režimy
- provádění instrukcí
  - load/store architektura, operace registr/registr
  - zřetězené zpracování instrukcí, 1 instrukce na cyklus
  - zvláštní zpracování skoků
  - obvodový řadič
- silná závislost na překladači

Typické RISC procesory

Desktop/server

- Alpha (DEC)
- PA-RISC (HP)
- PowerPC (IBM + Motorola)
- MIPS (Silicon Graphics)
- SPARC (Sun Microsystems)

Embedded systems

- ARM (Advanced RISC Machines)
- Thumb (Advanced RISC Machines)
- SuperH (Hitachi)
- M32R (Mitsubishi)
- MIPS16 (Silicon Graphics)

Procesory typu CISC

Complex Instruction Set Computer

- zachycuje původní trend vývoje
  - přesun složitosti do HW pro snažší programování
  - méně instrukcí pro daný úkol znamená méně přístupů do (pomalé a drahé) paměti
  - implementace pomocí mikrokódu se dá snadno změnit
- zpětné označení jako alternativa k RISC

Konvergence CISC a RISC

Důsledek vývoje technologie i znalostí

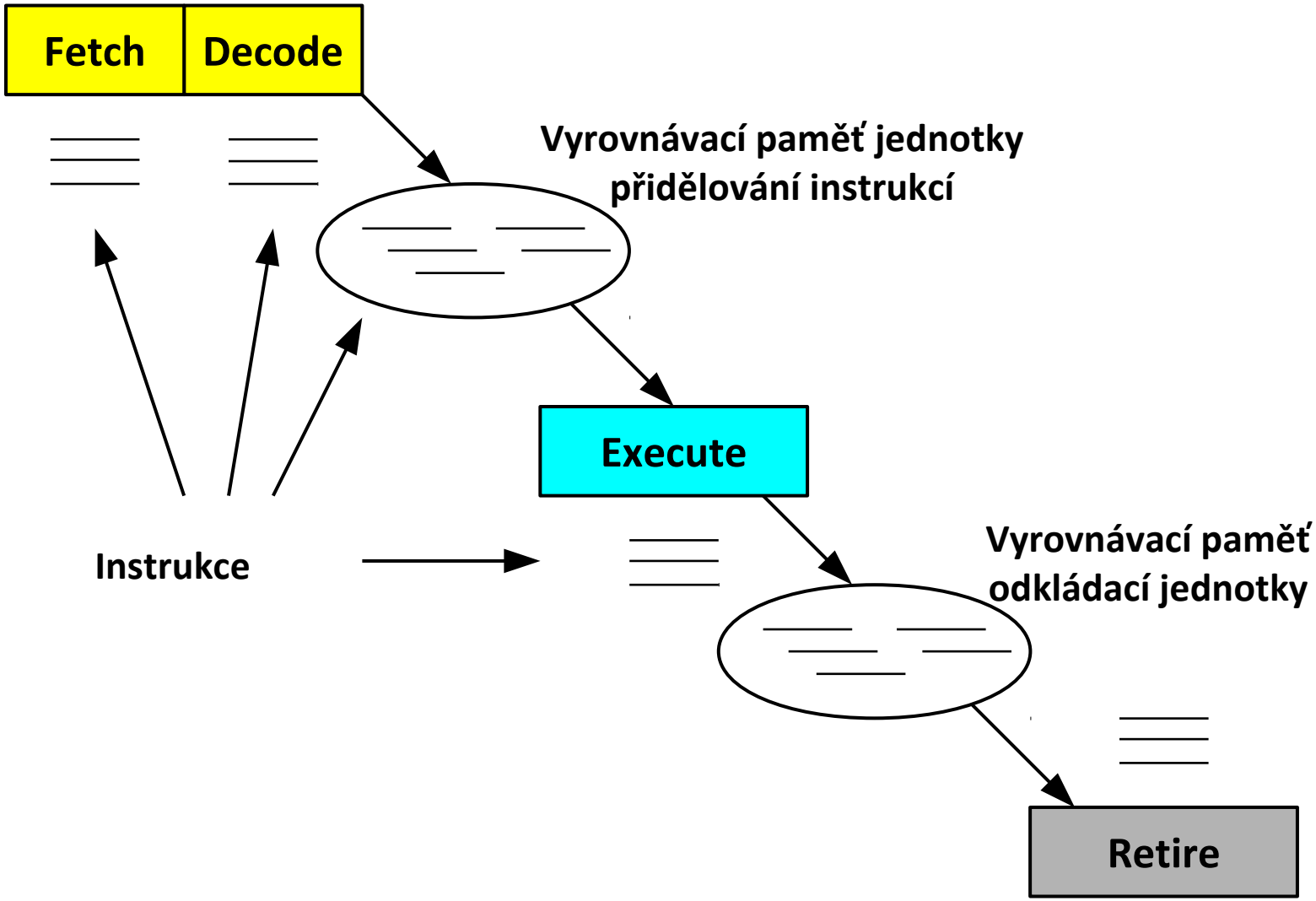
- mnohé techniky jsou používány v obou typech
  - CISC – schopné vykonávat v 1 taktu více instrukcí,
  - RISC – vyplnění komplikovanějšími instrukcemi
- vznik „post-RISC“ procesorů
  - kombinace obou přístupů s novými metodami
  - zachovává charakteristické vlastnosti RISC

Post-RISC procesory

Odlišnosti od superskalárních RISCů

- přidání ne-RISCových instrukcí (pro zvýšení výkonu)
- agresivní přerovnávání instrukcí při zpracování
  - out-of-order/speculative execution
  - odklon od závislosti na kompilátoru
- nové uspořádání
  - nové jednotky, větší míra paralelismu
- výkon RISC vs. post-RISC
  - RISC: dán stupněm paralelismu
  - post-RISC: dán počtem najednou dokončených instrukcí

Zřetězené zpracování v post-RISC



Přechod RISC – Post-RISC

Company	Old Processor	SpecInt92	New Processor	SpecInt92	% Increase Clock Rate	% Increase SpecInt92
DEC	21064a 300 MHz	220	21164 333 MHz	400	11%	82%
HP	PA-7150 125MHz	136	PA-8000 133MHz	360	6%	164%
IBM	PPC-601 80MHz	91	PPC-604 133MHz	176	66%	93%
Intel	Pentium 166MHz	198	P6 200MHz	320	20%	62%
Sun	HyperSparc 125MHz	131	UltraSPARC 140MHz	200	12%	52%
MIPS	R-4400 200MHz	141	R-10000 200MHz	300	0%	113%