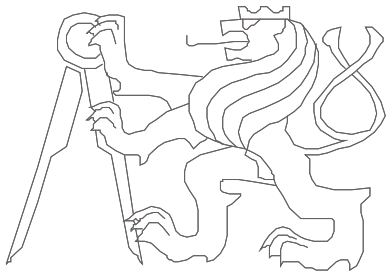
Pokročilé architektury počítačů

03

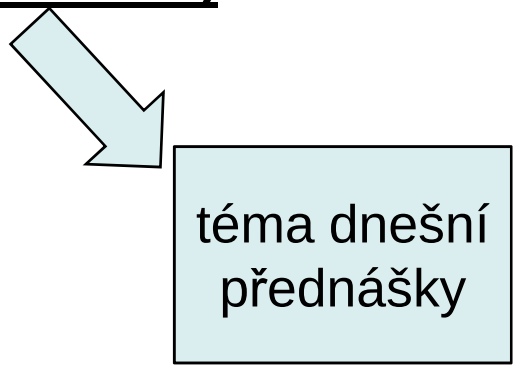
Superskalární techniky – Tok dat uvnitř procesoru z pohledu vykonávání instrukcí (Register Data Flow)



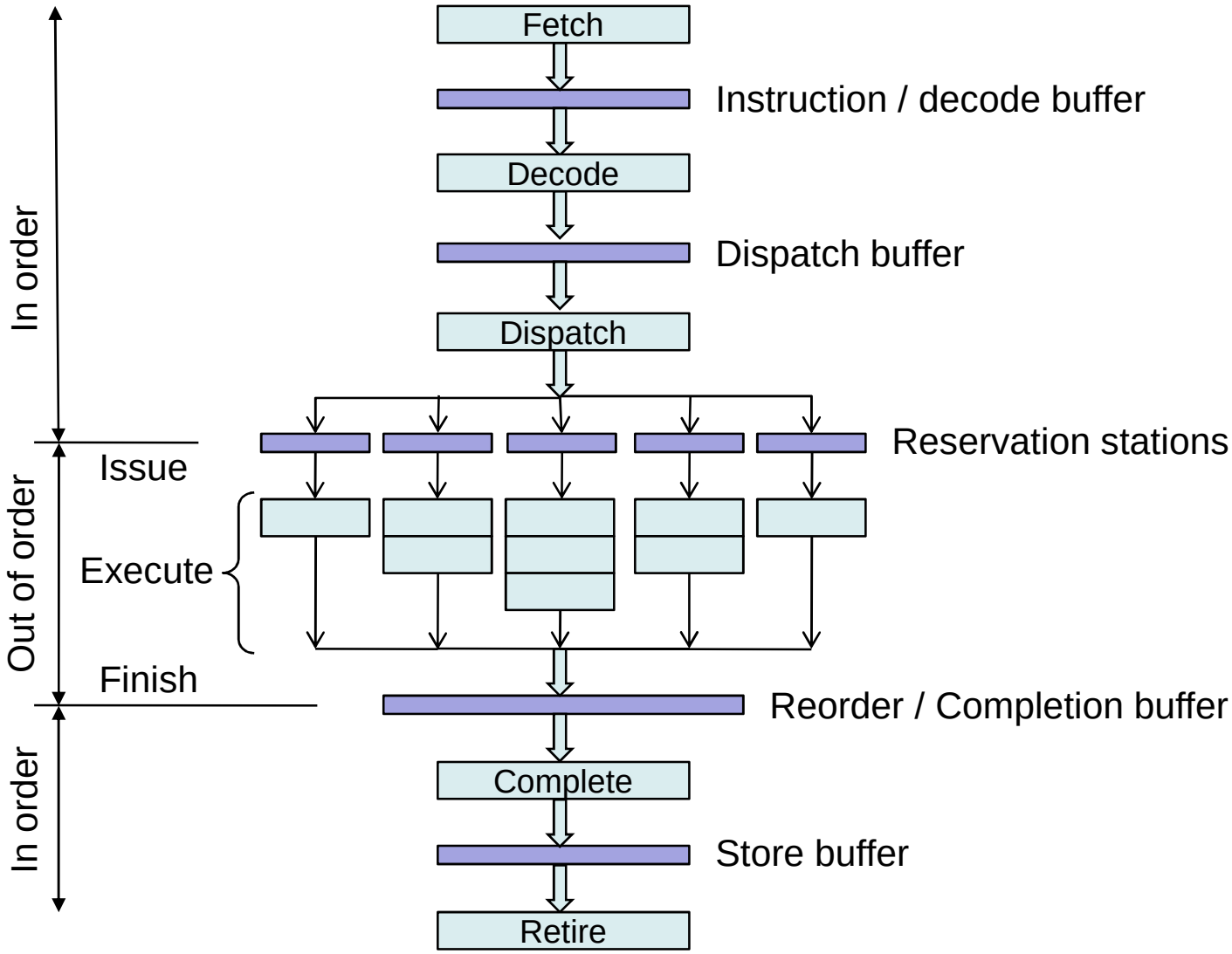
České vysoké učení technické, Fakulta elektrotechnická

Superskalární techniky – Připomeňme si…

- Uvědomme si, že cílem je maximální propustnost zpracování instrukcí…
- Na **zpracování** instrukcí můžeme nahlížet jako na tok instrukcí a tok dat, přesněji:
  - **tok dat mezi registry procesoru (register data flow)**
  - tok samotných instrukcí (instruction flow)
  - a tok dat z/do paměti (memory data flow)
- To zhruba odpovídá:
  - skokové instrukce
  - aritmeticko-logické / výpočetní instrukce
  - load/strore instrukce
- Pokud tedy chceme maximalizovat celkový tok, musíme minimalizovat čas (penalizaci) těchto tří typů instrukcí



Superskalární organizace – **Připomeňme si..**



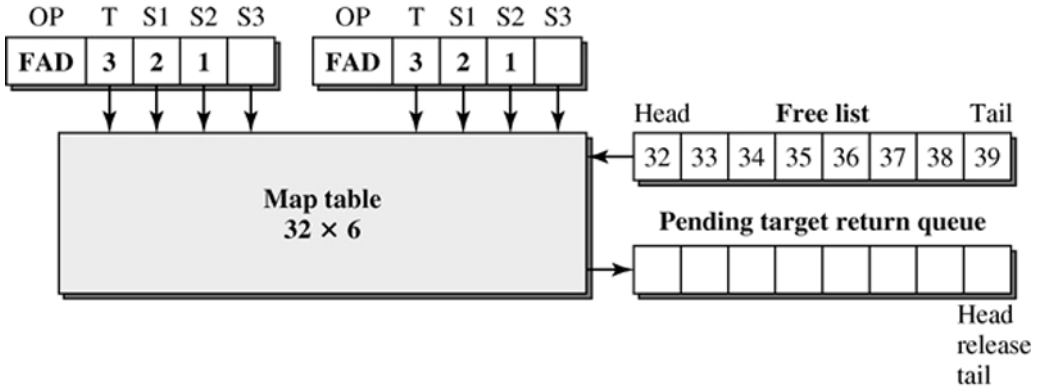
Register data flow

- V první části dnešní přednášky se zaměříme na tok dat při vykonávání **instrukcí typu registr-registr** (instrukce vykonávající operace nad registry a ukládající výsledky opět do registru)
- ***Register recycling*** – opětovné použití již použitého registru. Má dvě formy:
  - **statickou** – vytváří kompilátor ve fáze *register allocation*. Kompilátor nejdřív generuje tzv. *single-assignment code*, který předpokládá neomezený počet regitrů. V důsledku omezení ISA jsou pak v druhém kroku (register allocation) přirazeny logické registry se snahou držet co nejvíce dočasných hodnot právě v těchto registrech. Cílem je zamezit přesunu dat z/do paměti.
  - **dynamickou** – vzniká za běhu programu uvnitř daného procesoru

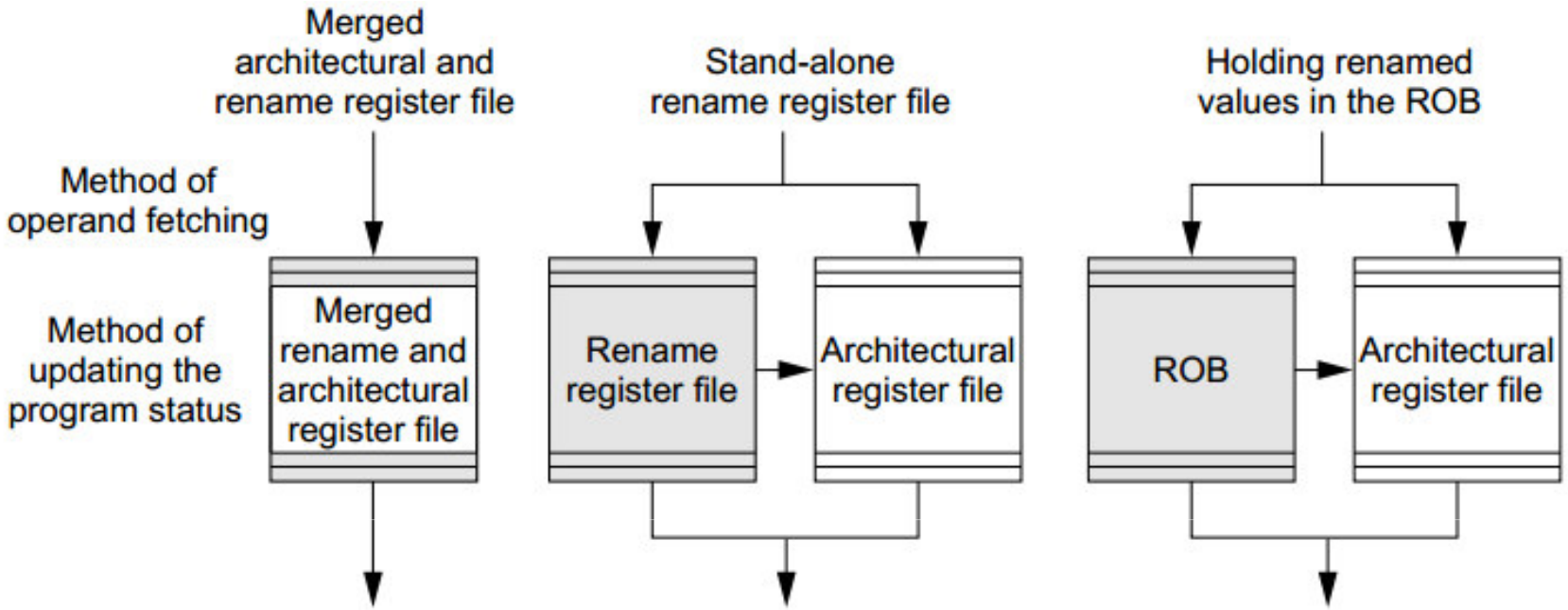
```
for(i=0; i<10; i++){
        R1 = R1+R2;
}
```

Technika přejmenování registrů v HW

- Přejmenování registrů může používat
  - dva oddělené soubory registrů – *rename register file* (RRF) jako doplnění k *architected register file* (ARF)
  - jeden společný prostor registrů – *pooled/merged/shared register file* – libovolný registr může být označen jako architekturální (definován v ISA)
    - Výhoda: Není nutno kopírovat obsah skrytého registru do architekturálního, stačí změnit příznak
    - Nevýhody: Zvýšené nároky na HW v porovnání s prvním přístupem; Při přepínání kontextu je nutno identifikovat, které registry jsou architekturální
  - Příklad: IBM RS/6000:



Technika přejmenování registrů v HW



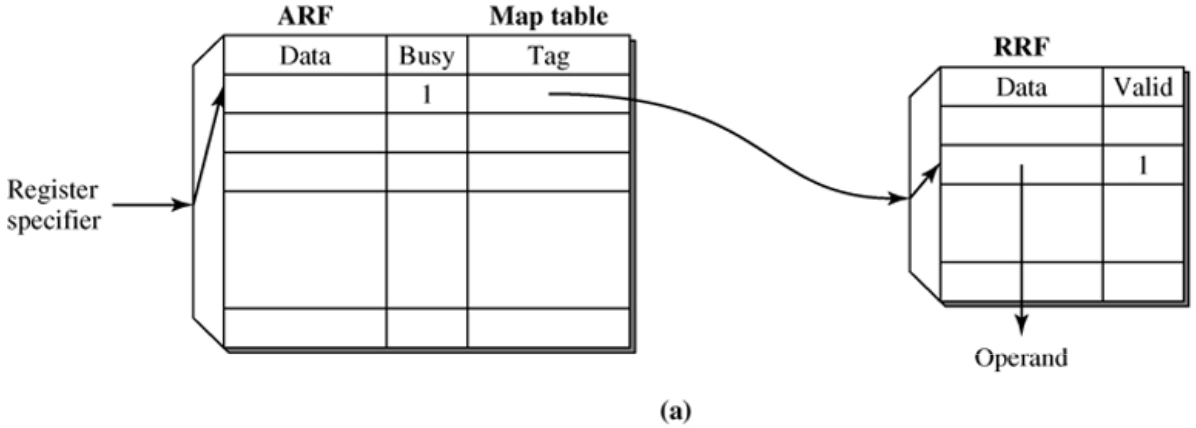
Power1 (1990)
Power2 (1993)
ES/9000 (1992)
Nx586 (1994)
PMI (Sparc64, 1995)
R10000 (1996)
R12000 (1999)
Alpha 21264 (1998)
Pentium 4 (FP) (2000)
K7 (FP) (1999)
K8 (FP) (2003)

PowerPC 603 (1993)
PowerPC 604 (1995)
PowerPC 620 (1996)
Power3 (1998)
PA 8000 (1996)
PA 8200 (1997)
PA 8500 (1999)

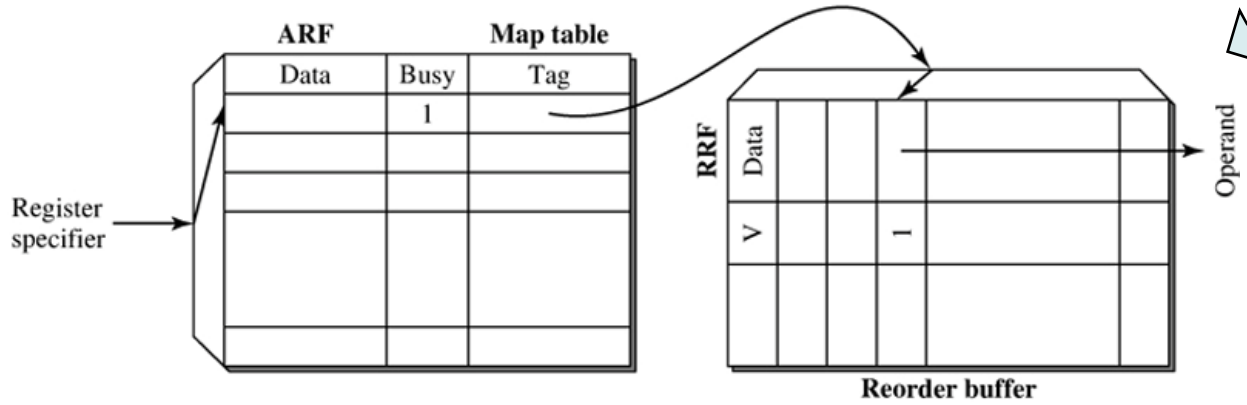
Am290000 superscalar (1995)
K5 (1995)
M1 (1995)
K6 (1997)
Pentium Pro (1995)
Pentium II (1997)
Pentium III (1999)
Pentium 4 (FX) (2000)
Pentium M (2003)
Core (2006)

Technika přejmenování registrů v HW

- Podívejme se na implementaci pokud máme dva oddělené soubory registrů – RRF (Rename RF) a ARF (Architectural RF)



(a)



(b)

RRF jako část Reorder buffru

Nevýhoda:
- plýtvání místem

Výhoda:
- ROB již obsahuje porty pro příjem dat z funkčních jednotek a pro aktualizaci ARF

Co se musí udělat?

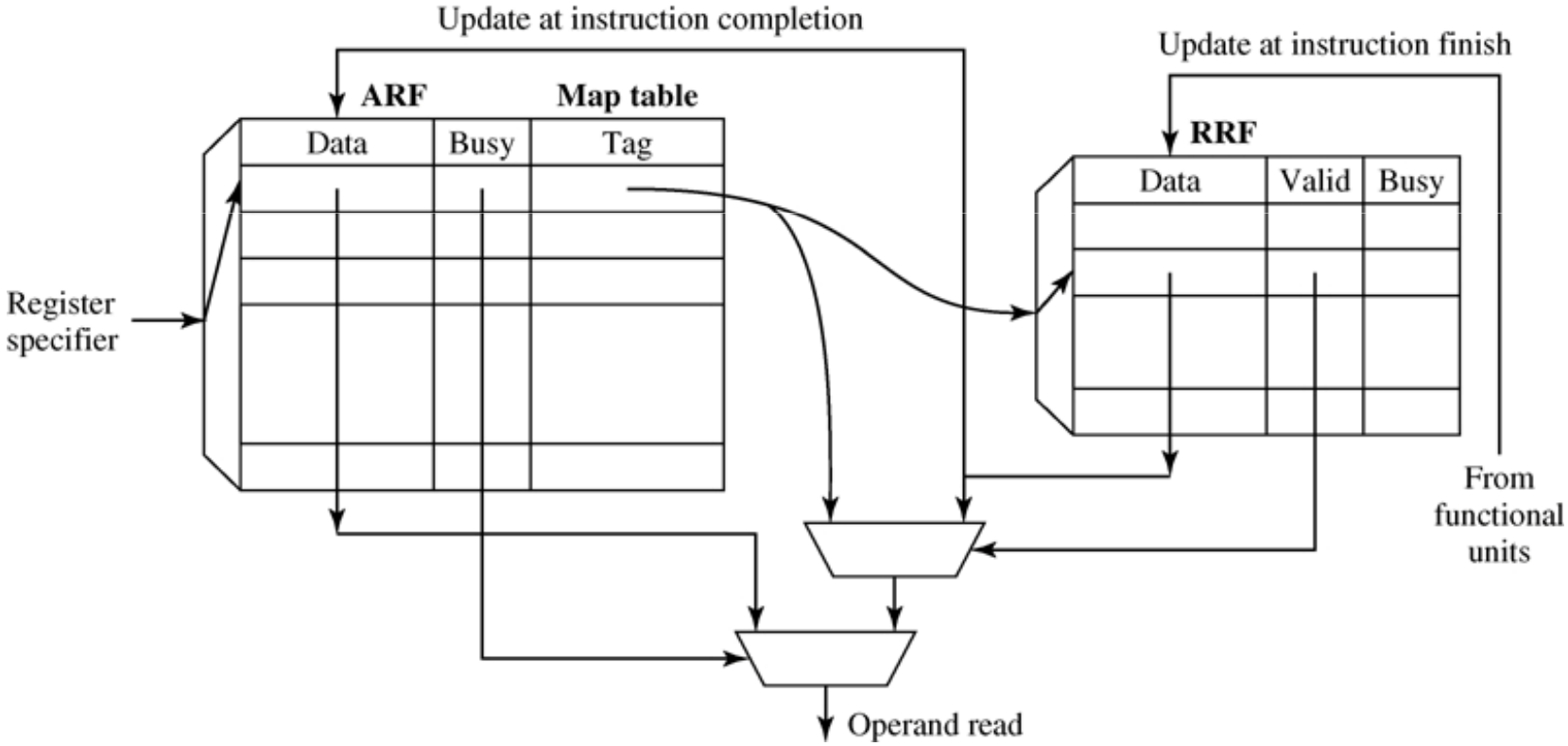
- Přejmenování registrů zahrnuje:
  1. **čtení vstupů** (typicky během dekódování nebo dispečování)
     Mohou nastat tyto případy:
     - Hodnota je v architekturálním registru – hodnotu můžeme použít
     - Hodnota není v ARF (je nastaven příznak Busy)
       - V RRF je nastaven bit Valid (instrukce dokončila vykonávání – finished, čeká však na dokončení – completion) – hodnotu můžeme použít
       - V RRF není nastaven bit Valid – hodnotu nemůžeme použít, bereme Tag
  2. **vyhrazení prostoru pro ukládání hodnot** (typicky během dekódování nebo dispečování) – musí se:
     - nastavit bit Busy v ARF i RRF (v RRF vybíráme nepoužitý registr: Busy==0)
     - přiradit Tag
     - aktualizovat Map table

     Pouze pro instrukce zapisující do registru.
  3. **aktualizaci registrů** (pokračování na dalším slajdu)

Co se musí udělat?

3. <u>aktualizaci registrů</u>

- když instrukce zapisující do registru dokončí vykonávání (execution) její výsledek je zapsán do RRF podle jejího Tagu. Později když je dokončena (completed) výsledek je zkopírován z RRF do ARF (uvolnění v RRF)



Kdo je Tomasulo?

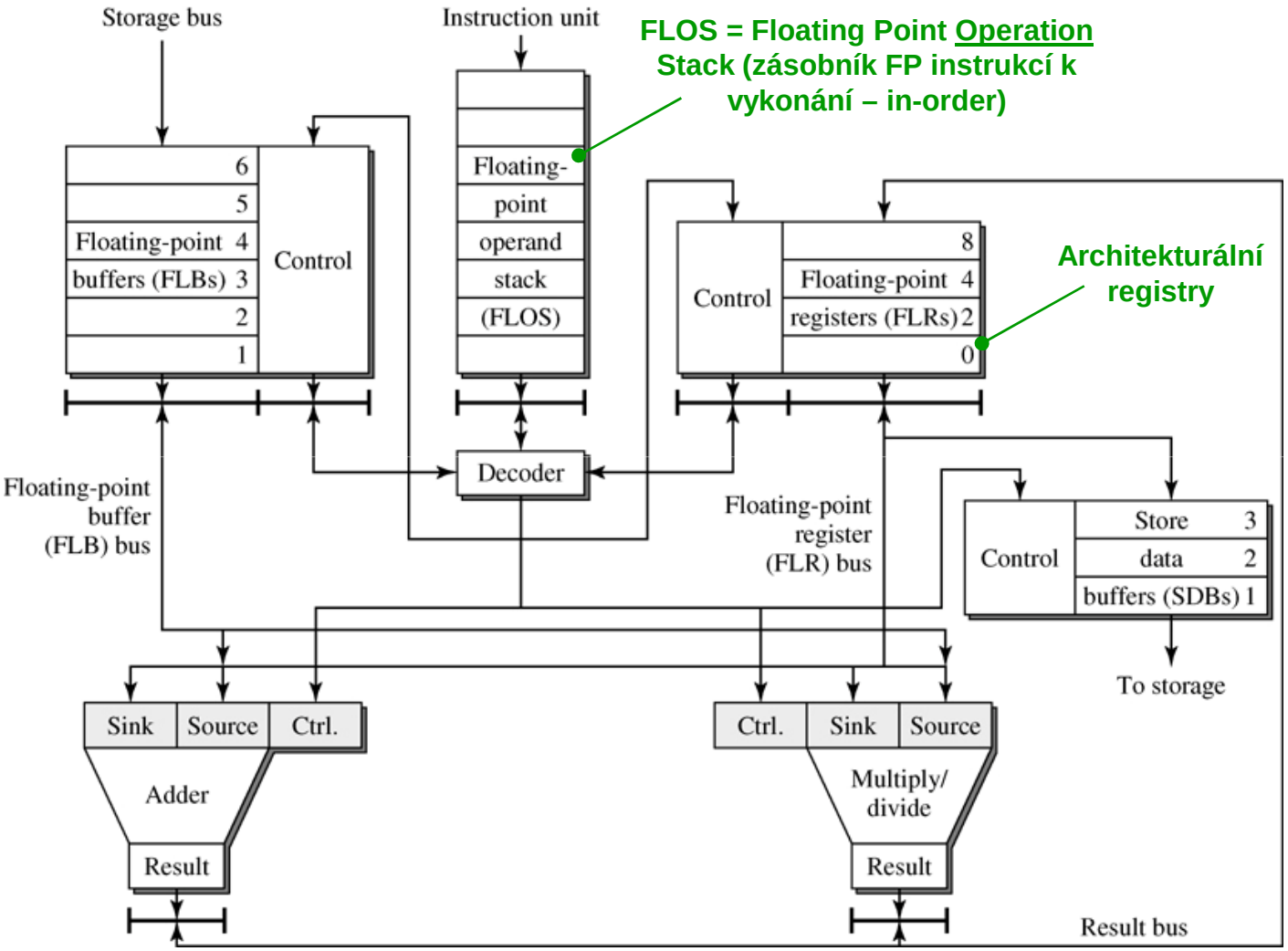
- **Robert Tomasulo**: Přednáška: Out-of-order processing-History of the IBM System/360 Model 91, University of Michigan College of Engineering, 1/30/2008.

- v 1997 obdržel cenu Eckert–Mauchly za Tomasulův algoritmus z roku 1967
- Klíčová práce: An Efficient Algorithm for Exploiting Multiple Arithmetic Units, IBM Journal of Research and Development, 11(1):25-33, January 1967.

Původní IBM System/360 – rok 1964

- Jasně rozlišoval mezi architekturou a implementací – což umožnilo vytvářet levnější (pomalejší) a dražší (rychlejší) verze
- Nejpomalejší verze měly rychlost 0.0018 do 0.034 MIPS, nejrychlejší pak 50x větší
- Na trhu velmi úspěšný
- Jeden z nejvýznamnějších počítačů v historii – **měl obrovský dopad na vývoj dalších počítačů**
- Vznikl pod vedením **Gene Amdahl**-a
- 32-bitové obecné registry, 24-bitů pro adresaci, atd.
- Zavedl některé „standardy“. Zamysleli jste se třeba nad tím:
  - Proč má **byte** fixní velikost a právě 8 bitů?
  - Proč má jedna buňka paměti (ve smyslu adresace) právě jeden byte? (vs. bit, vs. slovo)
  - odkud pochází kódování EBCDIC? (Extended Binary Coded Decimal Interchange Code)
  - co bylo před **IEEE 754**-1985 a čím byl tento standard ovlivněn?

Původní podoba FPU počítače IBM System/360



FP instrukce:
- vnitřně se jeví jako registr-registr díky FLB a SDB

Funkční jednotky:
- nezřetězené
- Adder: 2 cykly
- Mult/Div: 3 nebo 12 cyklů

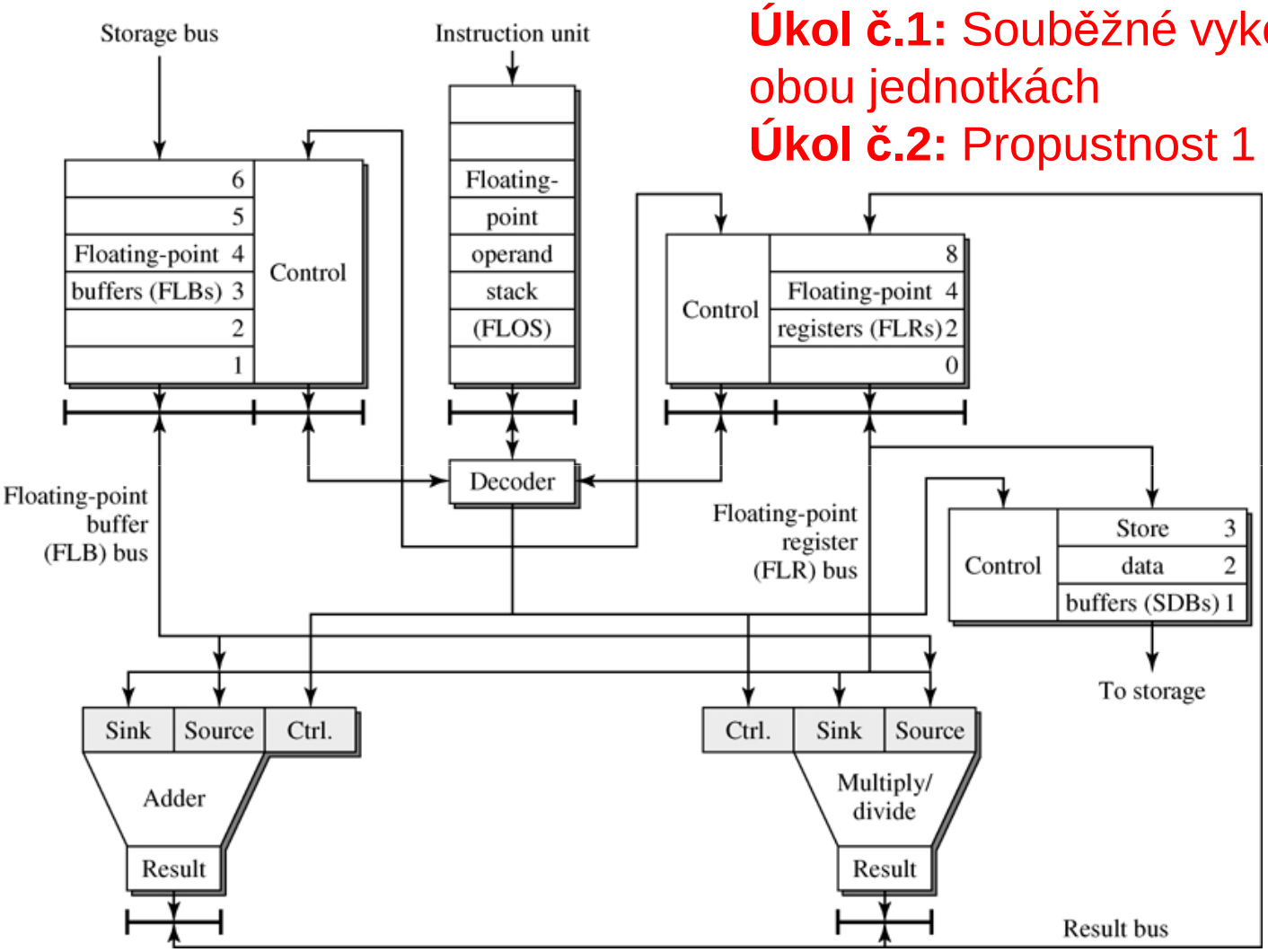
Result bus:
-pro zápis výsledků do architekturálních registrů

X = X + A;
Y = Y/X;

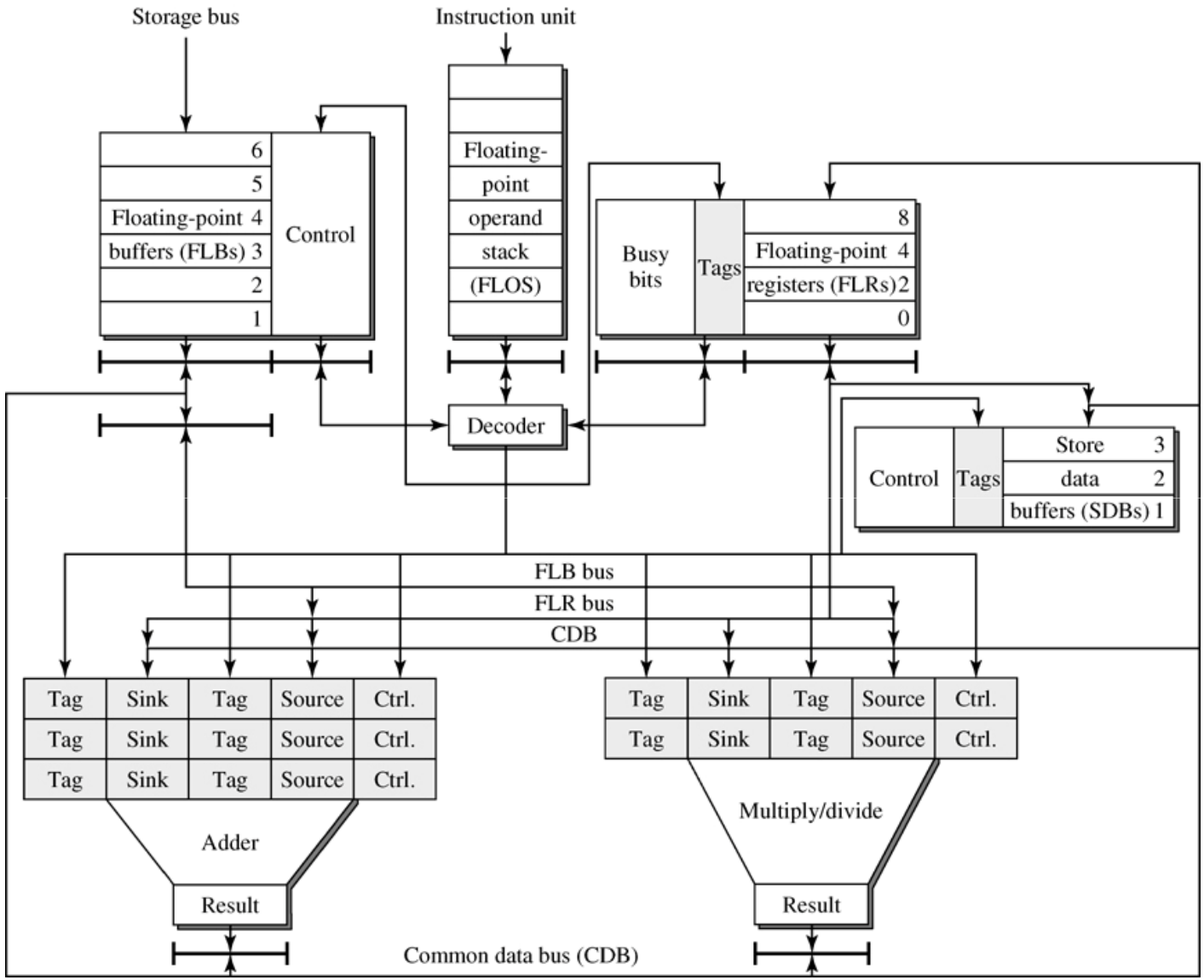
Vžijme se do situace vývojářů IBM…

Úkol č.1: Souběžné vykonávání instrukcí v obou jednotkách

Úkol č.2: Propustnost 1 instrukce/cyklus



Řešení…


Storage bus
Instruction unit
6
5
Floating-point 4
buffers (FLBs) 3
2
1
Control
Floating-
point
operand
stack
(FLOS)
Busy
bits
Tags
8
Floating-point 4
registers (FLRs) 2
0
Decoder
Control
Tags
Store 3
data 2
buffers (SDBs) 1
FLB bus
FLR bus
CDB
Tag Sink Tag Source Ctrl.
Tag Sink Tag Source Ctrl.
Tag Sink Tag Source Ctrl.
Adder
Result
Tag Sink Tag Source Ctrl.
Tag Sink Tag Source Ctrl.
Multiply/divide
Result
Common data bus (CDB)

Princip Tomasulova algoritmu

- Instrukce má dva operandy. První je zároveň i místem uložení výsledku.
- Nicméně vykonejme program:

```
w:   R4 = R0 + R5      // R0 == 6.0,   R5 == 7.8
x:   R2 = R0*R4
y:   R4 = R4 + R5
z:   R5 = R4*R2
```

Initial state of architectural registers

FLR - architektururální reg.

	Busy	Tag	Data
0			6.0
1			
2			
3			
4			
5			7.8

Cyklus 1.:

	RS #	Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
	2				
	3				
w-start		Adder			

	RS #	Tag	Sink	Tag	Source
x	4	0	6.0	1	---
	5				
		Mult / Div			

FLR - architektururální reg.

		Busy	Tag	Data
	0			6.0
	1			
x	2	yes	4	
	3			
w	4	yes	1	
	5			7.8

Instructions are dispatched in order!

First, instruction w: Read R0, Read R5 (data are valid); in R4 Set Bussy and Tag (Tag=Reservation Station Num)

Second, instruction x: Read R0 (success), Read R4 (bit Bussy is set, so Tag is used instead); in R2 set Bussy and Tag

Princip Tomasulova algoritmu

w: R4 = R0 + R5
x: R2 = R0*R4
y: R4 = R4 + R5
z: R5 = R4*R2

Cycle 2:
- instruction y: Read R4 (Tag==1 is read instead of data), Read R5 (data are valid); in R4 set Bussy and Tag (Tag=Reservation Station Num of this instruction == 2)
- instruction z: Read R4 (Tag==2), Read R2 (Tag==4); in R5 set Bussy and Tag
... on the end of cycle, Adder produces result and propagates it (together with its Tag) on the Common Data bus. Reservation Stations updates their Data (Tag match)

Cyklus 2.:

	RS #	Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
y	2	1	---	0	7.8
	3				
w-finishin		Adder			

	RS #	Tag	Sink	Tag	Source
x	4	0	6.0	1	---
z	5	2	---	4	---
		Mult / Div			

FLR - architekturální reg.

		Busy	Tag	Data
	0			6.0
	1			
x	2	yes	4	
	3			
w	4	yes	2	
z	5	yes	5	7.8

Po dokončení šíří (broadcast) výsledek spolu
s Tagem na CDB (Common Data Bus) 6.0+7.8=13.8

Cyklus 3.:

	RS #	Tag	Sink	Tag	Source
w	1				
y	2	0	13.8	0	7.8
	3				
		Adder			

	RS #	Tag	Sink	Tag	Source
x	4	0	6.0	0	13.8
z	5	2	---	4	---
		Mult / Div			

FLR - architekturální reg.

		Busy	Tag	Data
	0			6.0
	1			
x	2	yes	4	
	3			
w	4	yes	2	
z	5	yes	5	7.8

Dealokace rezervační stanice č.1

Další cykly si zkuste dodělat doma....

Princip Tomasulova algoritmu

- Pochopili jste?
- Pokud jste dávali pozor, tak jste si mohli všimnout, že Tomasulův algoritmus v této podobě nepodporuje přesné přerušení.
- V čem je problém?

- Poznámky pod čarou:
  - FLR uchovává Tag té rezervační stanice, která se stala zodpovědnou za produkci výsledku.
  - Kdykoliv se dokončí vykonání instrukce (je znám výsledek), musí se její výsledek spolu s Tagem (z které rezervační stanice instrukce pochází) šířit po CDB. Zbylé rezervační stanice i FLR musí tento Tag sledovat a případně aktualizovat svá data.

Přesné přerušení – Obecný pohled

- Přerušení (opakování pojmu)
  - je metoda pro asynchronní obsluhu vnější události/í. Procesor přeruší sekvenční sémantiku vykonávání instrukcí, přejde na obsluhu a pak se vrátí a pokračuje v činnosti předchozí.
- Výjimka (vysvětlení pojmu)
  - je přerušením (obsluhou neplánované události) vyvolaným událostí uvnitř (v procesoru). Jiné označení pro vnitřní přerušení. Patří sem ale také nedefinovaná instrukce, systémové volání, apod.
- Přesné (Precise Exception)
  - Přerušení v sekvenčním stroji je vždy přesné.

Příklady přerušení a výjimek

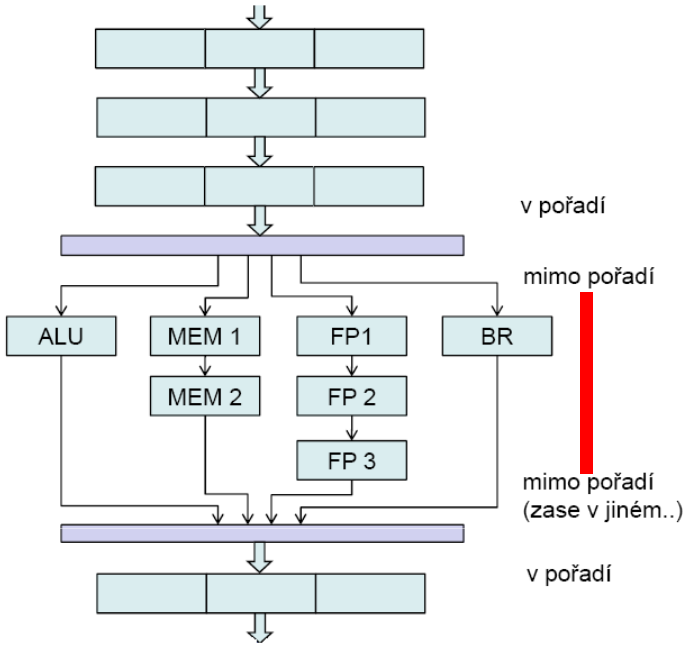
- Žádost V/V zařízení o obsluhu (I/O Device Request),
- Volání služby OS (Supervisor Call SVC),
- Trasování programu (Breakpoint),
- Přerušení AJ (aritmetické přeplnění, nenaplnění, dělení nulou,…),
- Výpadek stránky (Page Fault),
- Porušení ochrany paměti (Page Fault),
- Nezarovnaný přístup k paměti (Misaligned Memory Access)
- Nedefinovaná instrukce (Undefined Opcode),
- Chyba HW (parita, ECC, pokles napájení,…)

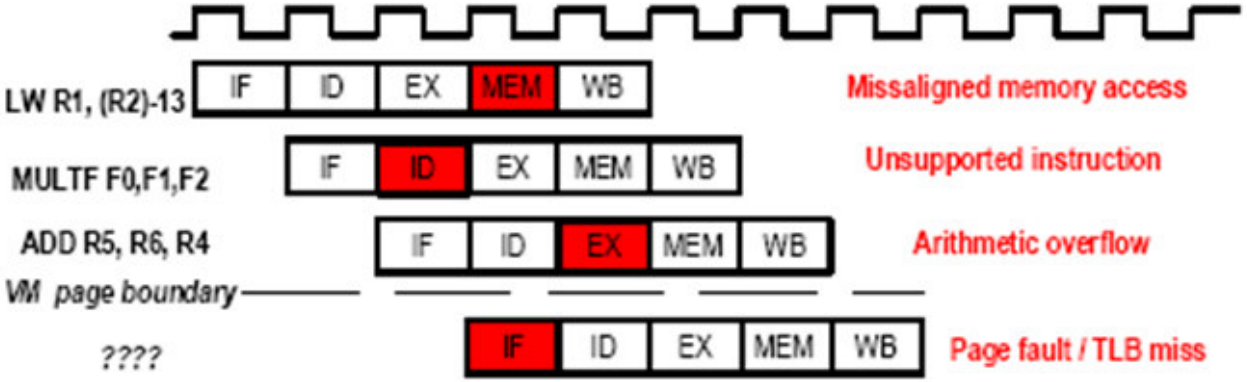
Klasifikace typů přerušení (jen pro ilustraci možných pohledů)

- Vnější vs. vnitřní, jindy se říká
- Přerušení vs. výjimka,
- Hardwarové vs. softwarové,
- Synchronní vs. asynchronní,
- Žádané uživatelem vs. vynucené,
- Maskovatelné vs. nemaskovatelné,
- V instrukci vs. mezi instrukcemi,
- Možnost návratu vs. ukončení s chybou.

Specifika přerušení u proudově pracujícího procesoru

- K přerušení může dojít v každém taktu,
- Současně může dojít k přerušení z více zdrojů,
- Pořadí výskytu přerušení nemusí odpovídat pořadí instrukcí v programu,
- Stav procesoru nemusí být v okamžiku přerušení konzistentní se sekvenční sémantikou vykonávání.





V některých taktech tu nastala 2 různá přerušení (přesněji) výjimky. Mohly nastat až dokonce 4! Řešení?

Je konkrétní přerušení přesné? Ano, pokud

- 1. procesor obslouží přerušení v programovém pořadí,
  - To nemusí odpovídat pořadí jejich objevení se v proudu (pipeline).
- 2. stav procesoru při jejich obsluze je sekvenčně konzistentní.
  - To znamená: instrukce před zdrojem přerušení musí být dokončeny a instrukce za zdrojem přerušení nesmí stav procesoru a paměti změnit.

Přirozená metoda řešení problému – Existence Commit

- Stupeň potvrzení (Commit) v proudovém zpracování.

- Stupeň potvrzení je takový stupeň proudu, že všechna přerušení mohou vzniknout nejpozději v tomto stupni.
- Po stupni potvrzení už víme, že instrukce nebude přerušena.

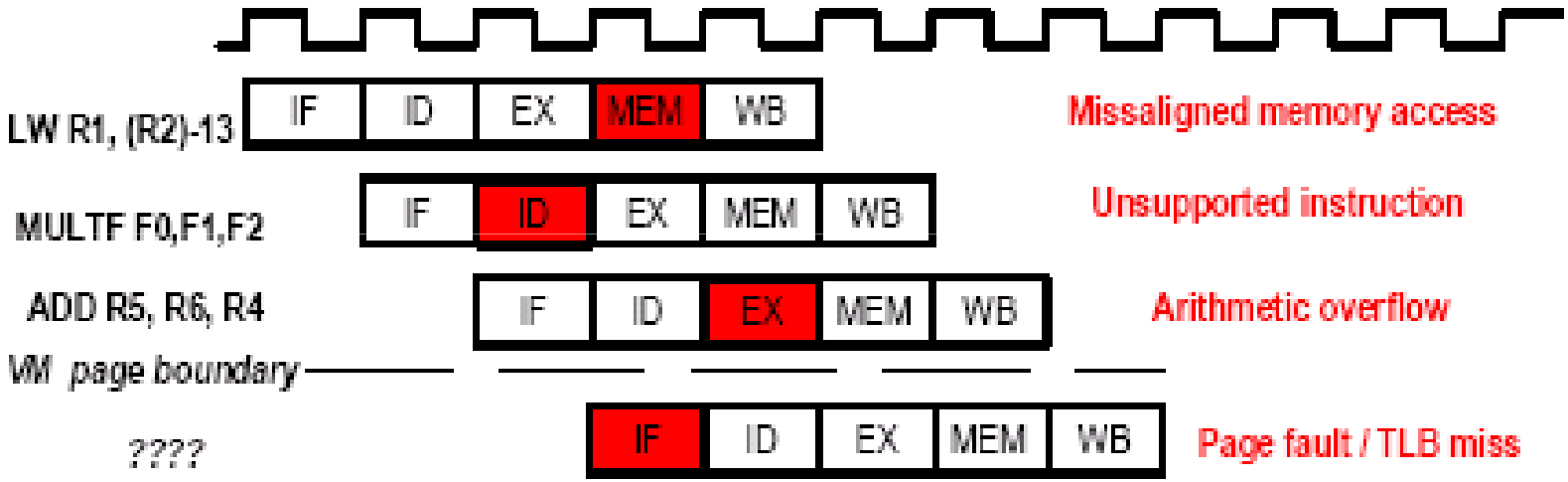
Podmínky pro přesné přerušení jsou:

- Všechny instrukce dorazí do stupně potvrzení ve správném sekvenčním pořadí.
- Instrukce nemění stav procesoru ani paměti před stupněm potvrzení.

Princip implementace přesného přerušovacího systému

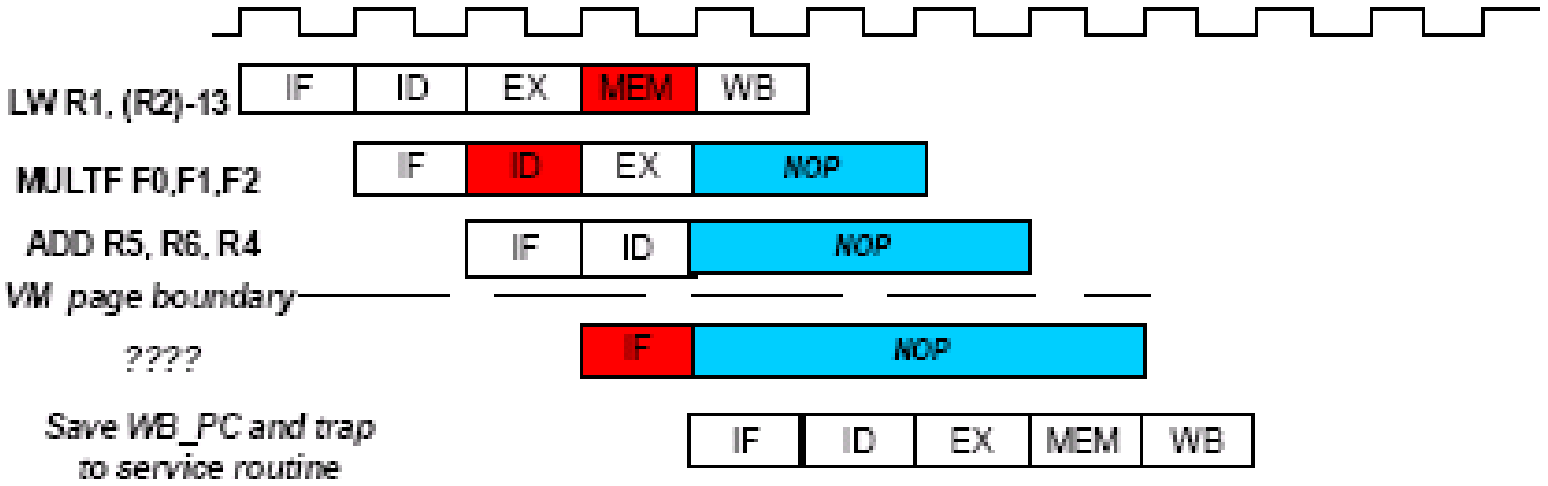
- Požadavky se akumulují ve speciálním HW a žádost společně s jejím PC postupuje v proudu až do stupně potvrzení.
- Procesor žádost testuje u každé instrukce až ve stupni potvrzení, přerušení v rámci instrukce se obsluhují v pořadí jejich vzniku.
- Obsluha začíná zrušením všech instrukcí v proudu před stupněm potvrzení a dokončí se instrukce za stupněm potvrzení.

Řešení u dříve uvedeného příkladu



Bude-li ale stupněm potvrzení (commit) u tohoto procesoru stupeň MEM ...

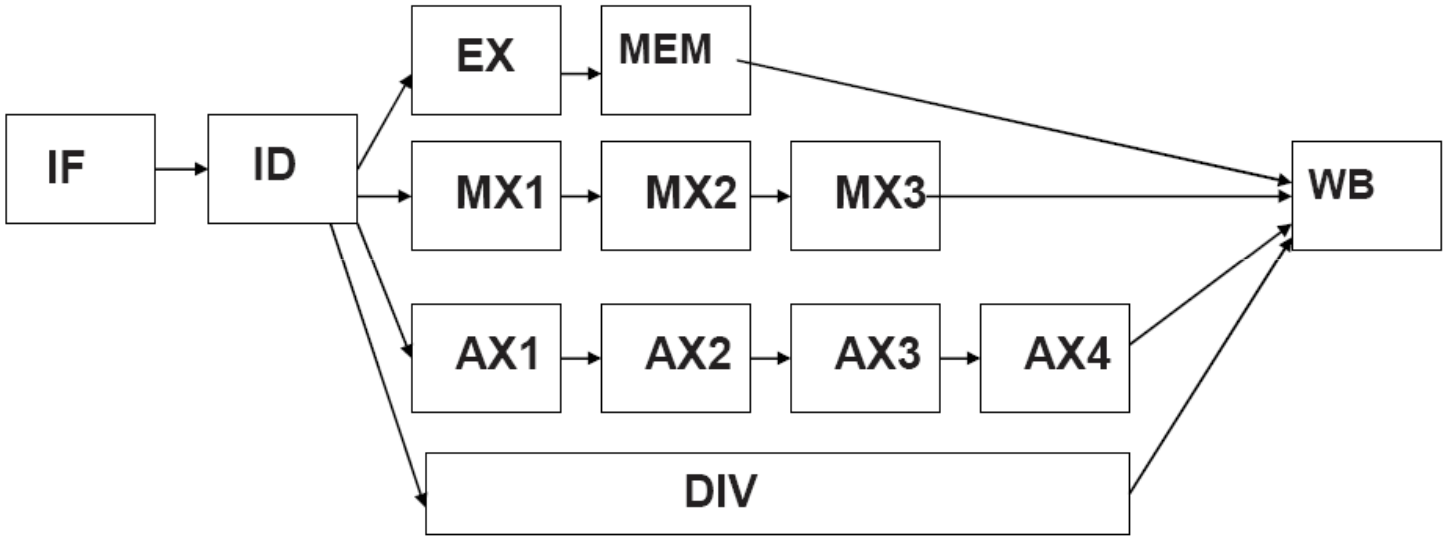
Pokračování příkladu



- Tento procesor má schopnost přesného přerušení, neboť
  - Do MEM dorazí v programovém pořadí a
  - Před stavem MEM instrukce nemění stav procesoru ani paměti.

Problém vyřešen?

- Ne, jsou složitější případy.
- Třeba ten, kdy nemají proudy stejnou délku.



- A tím se vracíme na začátek přednášky k Tomasulovu algoritmu…

Princip Tomasulova algoritmu – zde jsme skončili…

w: R4 = R0 + R5 // R0 == 6.0, R5 == 7.8

x: R2 = R0*R4

y: R4 = R4 + R5

z: R5 = R4*R2

	RS #	Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
y	2	1	---	0	7.8
	3				
w-finishin	Adder				

	RS #	Tag	Sink	Tag	Source
x	4	0	6.0	1	---
z	5	2	---	4	---
	Mult / Div				

		Busy	Tag	Data
FLR - architekturální reg.				
	0			6.0
	1			
x	2	yes	4	
	3			
w	4	yes	2	
	5		5	7.8

Po dokončení šíří (broadcast) výsledek spolu s Tagem na CDB (Common Data Bus) 6.0+7.8=13.8

	RS #	Tag	Sink	Tag	Source
w	1				
y	2	0	13.8	0	7.8
	3				
	Adder				

	RS #	Tag	Sink	Tag	Source
x	4	0	6.0	0	13.8
z	5	2	---	4	---
	Mult / Div				

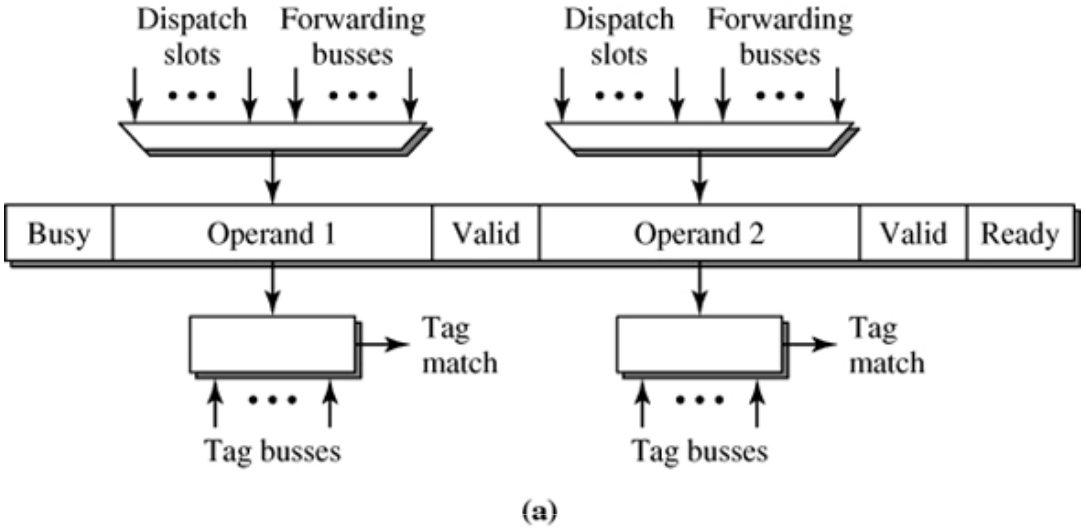
		Busy	Tag	Data
FLR - architekturální reg.				
	0			6.0
	1			
x	2	yes	4	
	3			
w	4	yes	2	
	5		5	7.8

Dealokace rezervační stanice č.1

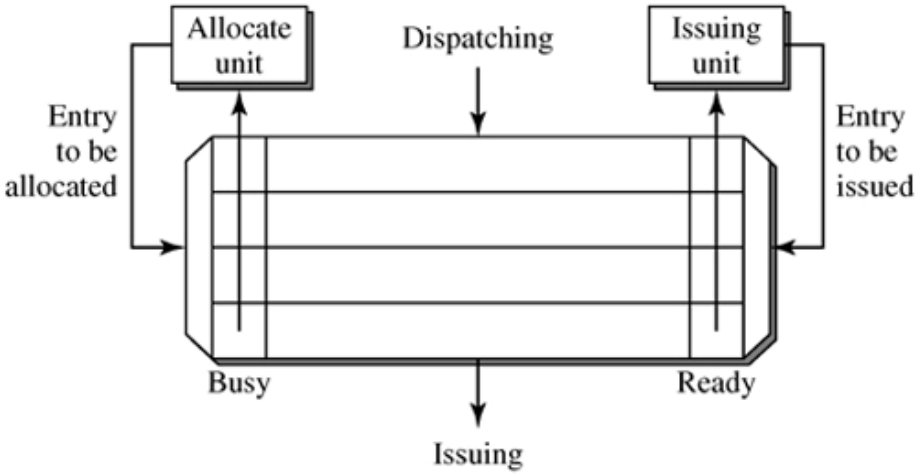
Vylepšený Tomasulův algoritmus

- **Issue / Dispatch**: Získání instrukce z čela instrukční fronty
  - Pokud je rezervační stanice k dispozici a je místo v ROB (Reorder Buffer), alokuj místo a dispečuj instrukci
  - Akce: (1)-dekódování instrukce, (2)-alokace položek RS a ROB, (3)-přejmenování registrů a čtení z RF (register file), (4)-dispečování dekódované instrukce do RS a ROB
- **Issue**: Vydání isntrukce k vykonání
  - Pokud jsou operandy připraveny, vykonej instrukci. Jinak čekej v RS a sleduj sběrnici CDB
- **Execute** (EX): Vykonání operace nad operandy
- **Write result**: Dokončení vykonávání
  - Zapiš výsledek na sběrnici CDB (&ROB) a dealokuj RS
- **Commit**: Aktualizace architekturálního registru z ROB
  - Pokud „nejstarší“ instrukce v ROB má výsledek (je vykonána), aktualizuj architekturální registr (příp. zapíš do paměti) a uvolni prostor v ROB

Jak vlastně vypadá rezervační stanice?



(a)

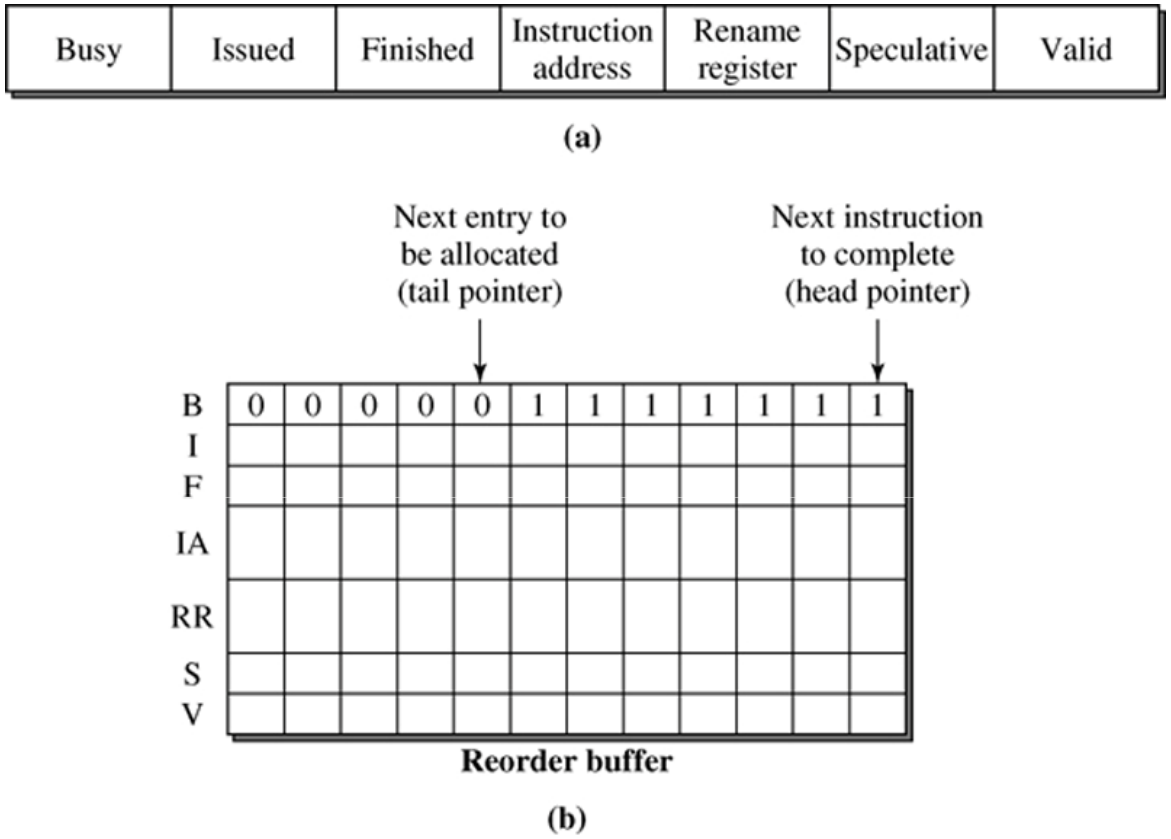


(b)

Poznámky:

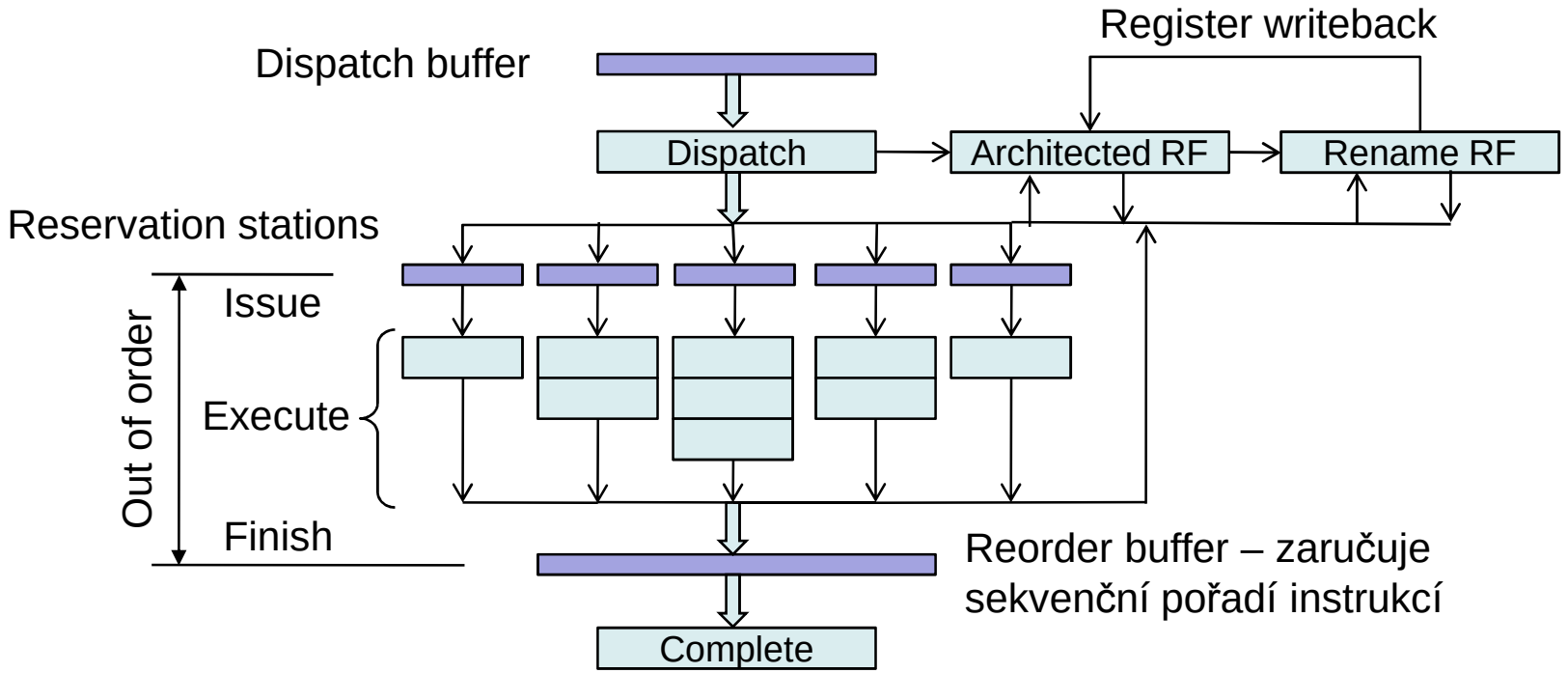
- Busy – alokace
- Valid – aktualizace operandů
- Ready – instruction wake up
- Issue – instruction select (do té doby čeká)

Jak vypadá reorder buffer?



- obshuje všechny *in flight* instrukce.. (byly dispečovány, ne dokončeny)
- kruhová fronta..
- jeho součástí může být i RRF (viz slide 7)

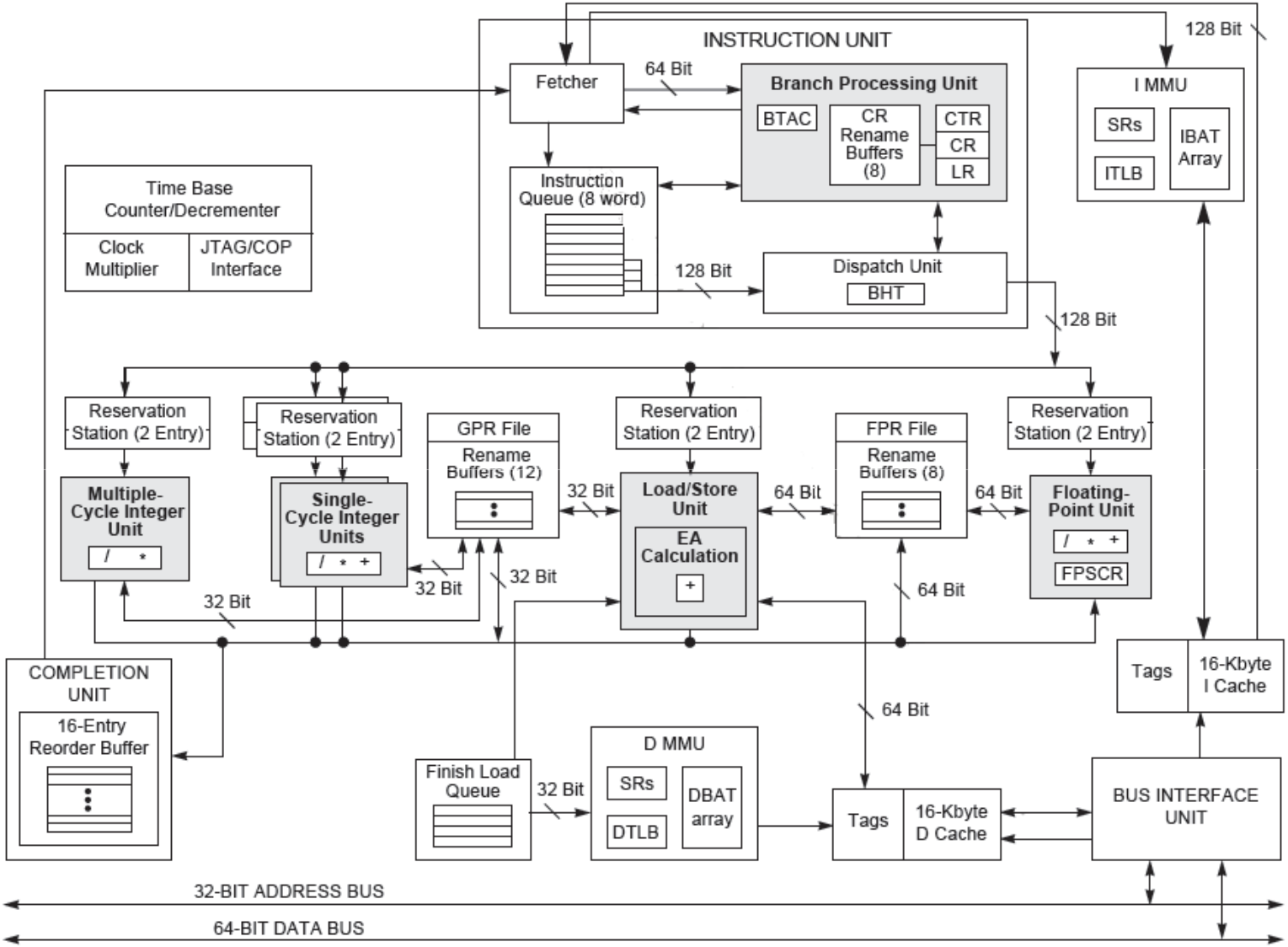
Vylepšený Tomasulův algoritmus



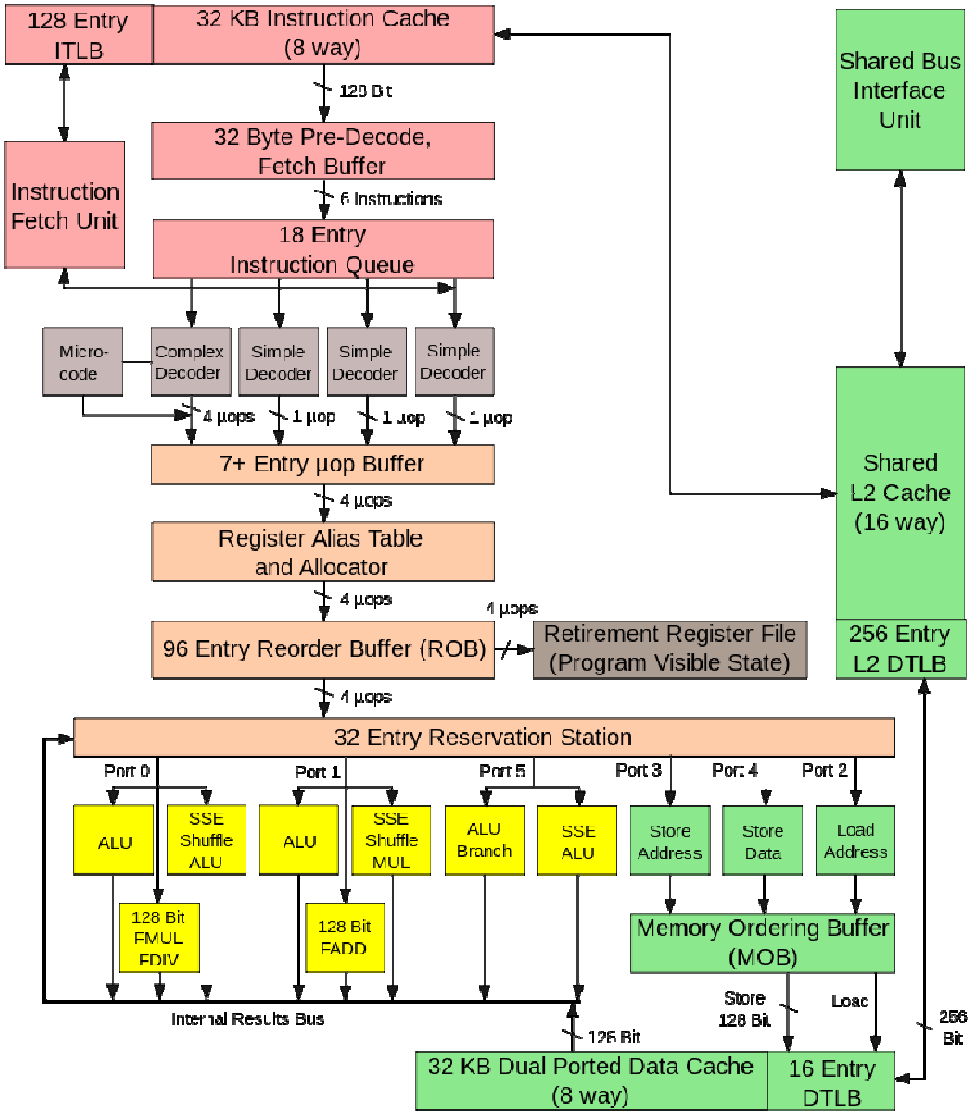
Nezapomínejme, že existují 3 verze (pro zajištění přejmenování registrů):

- Merged register file (kterýkoliv registr může být architekturální)
- Architected register file + Rename register file, přičemž RRF:
  - může být samostatný
  - může být součástí ROB (aneb návrat na slajdy 4 až 8)

PowerPC 604… Píše se rok 1994



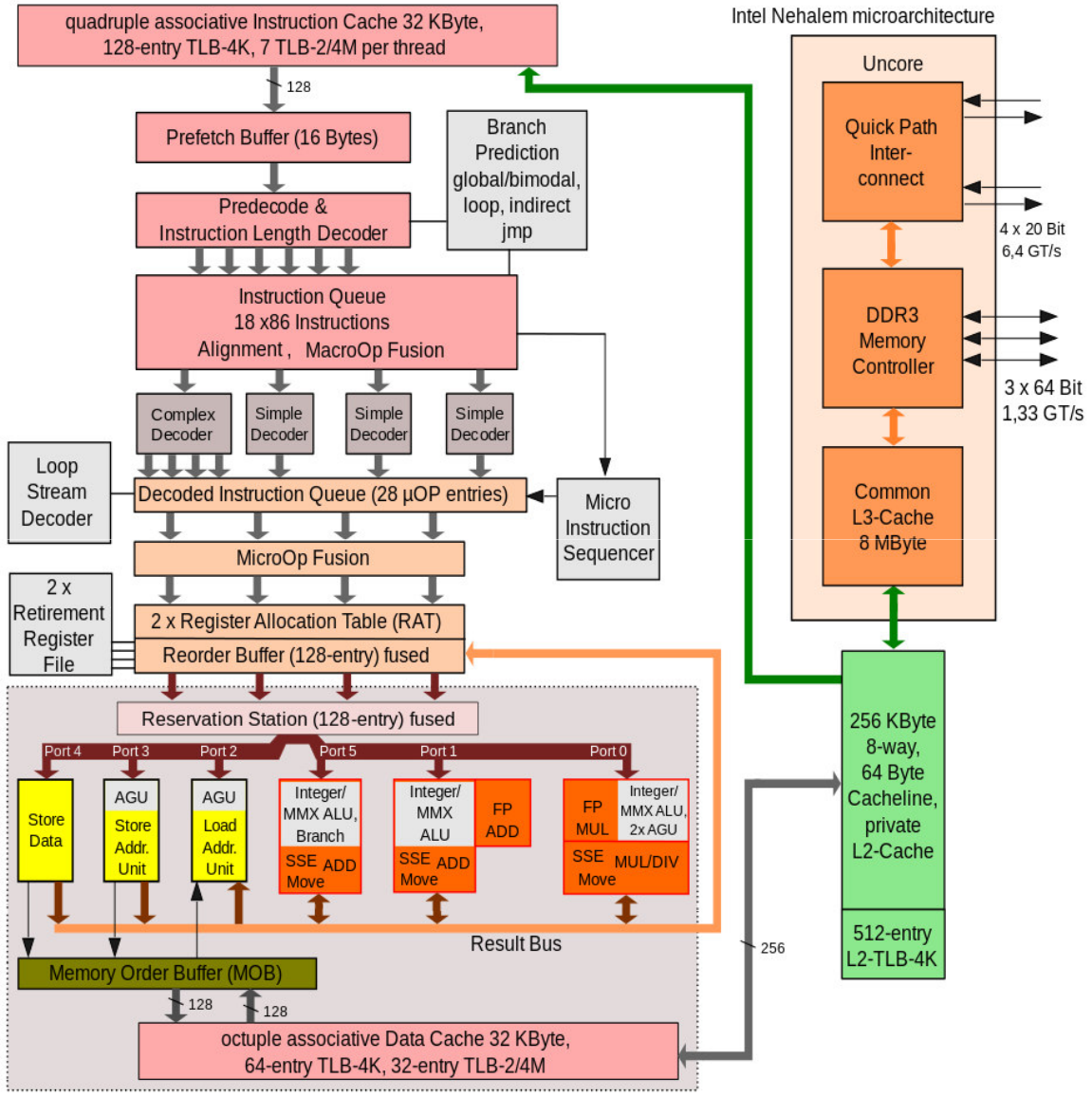
Intel **Core** microarchitecture… Rok 2006



Intel Core 2 Architecture

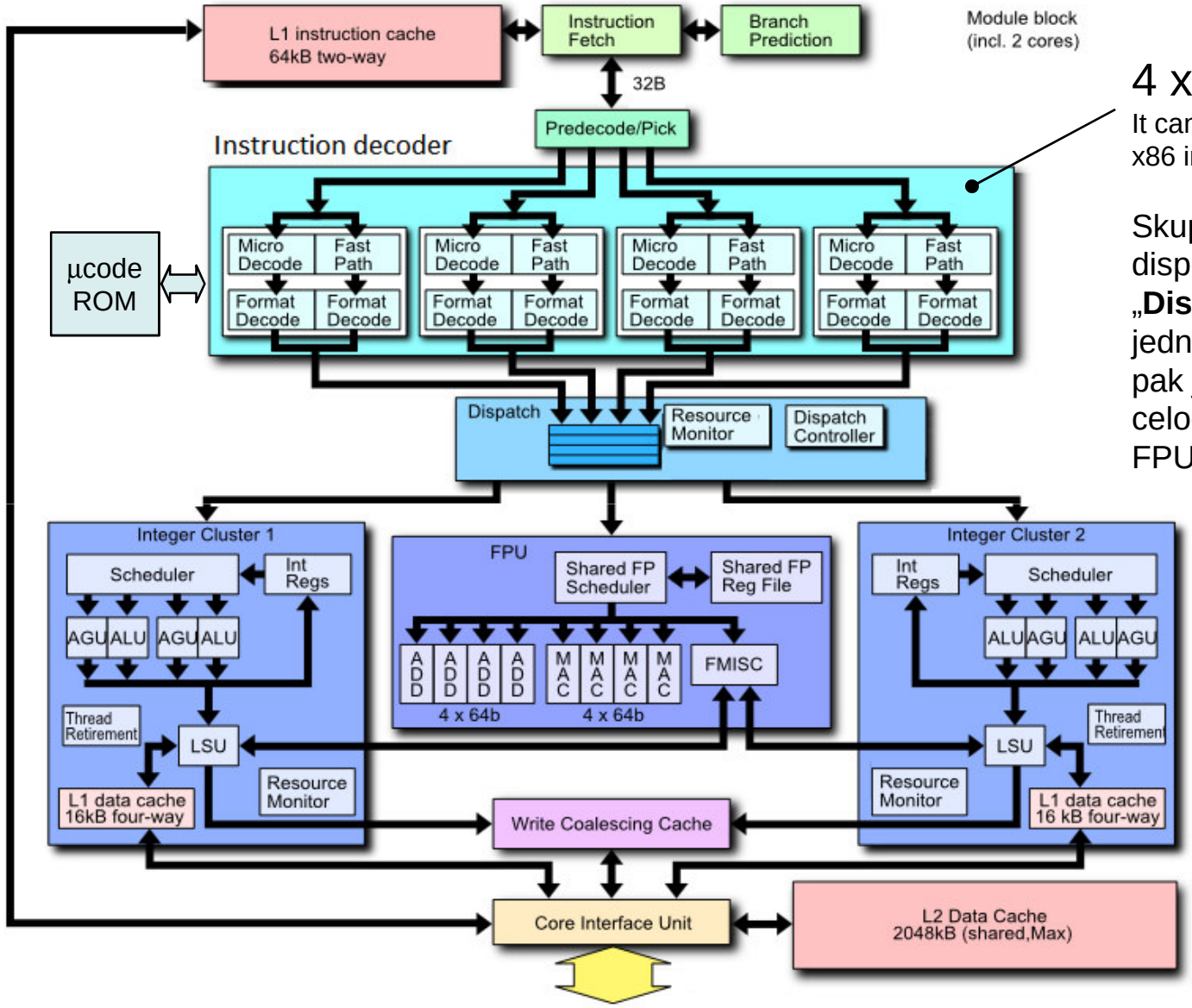
http://en.wikipedia.org/wiki/Intel_Core_(microarchitecture)

Intel **Nehalem** (Core i7) - 2008



- http://en.wikipedia.org/wiki/File:Intel_Nehalem_arch.svg

AMD **Bulldozer** 15h (FX, Opteron) - 2011



4 x86 decoders…

It can fetch and decode up to four x86 instructions per clock.

Skupina µoperací, které jsou dispečovány spolu -> „**Dispatch group**“ – vždy patří jednomu vláknu a celá grupa pak jde do jednoho ze dvou celočíselných jáder nebo do FPU. Proč je tomu tak???

- http://en.wikipedia.org/wiki/File:AMD_Bulldozer_block_diagram_(CPU_core_bloack).PNG

Co je důležité si uvědomit…

- „Všechny“ dnešní procesory implementují nějakou formu přejmenování registrů k odstranění antizávislostí a výstupních závislostí.
- Přejmenování se typicky realizuje během dekódování, co může znamenat nutnost přejmenovávání u všech instrukcí ve **fetch grupě** (mohou být závislosti nejenom mezi již vydanými, zpracovávanými a zpracovanými instrukcemi, ale i mezi instrukcemi ve fetch grupě navzájem)
- Přejmenování registrů odstraňuje WAW, WAR závislosti a tím umožňuje tyto instrukce vykonat paralelně (na rozdíl od techniky **scoreboarding**, která umožňuje paralelně vykonávat pouze nezávislé instrukce. Scoreboarding neodstraňuje, pouze sleduje závislosti (stall)).

Jěště dodejme, že...

- Během Dispatchingu musí být alokován prostror v rezervační stanici i v reorder buffru pro každou instrukci

- Což takhle zkombinovat rezervační stanice a reorder buffer dohromady?
  -> instrukční okno (instruction window)

- V tomto případě jsou instrukce dispečovány do instrukčního okna, položky kterého sledují sběrnice (Tag) pro své operandy, atd. atd.
- Velikost instrukčního okna udává počet instrukcí, které mohou být vykonány paralelně – maximální DOF (stupeň paralelizmu)

„Další“ řešení?

- Idea 1: Klidně aktualizuj architekturální registry když instrukce končí vykonávání (execution), ale v případě exception vrať změny zpět. K tomu potřebujeme nějakou paměť → **History Buffer (HB)**
- History Buffer:
  - Při dekódováni je rezervována položka v History Bufferu
  - když instrukce končí vykonávání (finishing execution) uloží hodnoty z arch. reg do HB
  - Pokud je instrukce nejstarší a žádná výjimka/přerušení nenastalo, je možné položku HB zahodit
  - Pokud je instrukce nejstarší a je potřeba obsloužit výjimka/přerušení, hodnoty z HB jsou postupně přepsány do arch. reg.

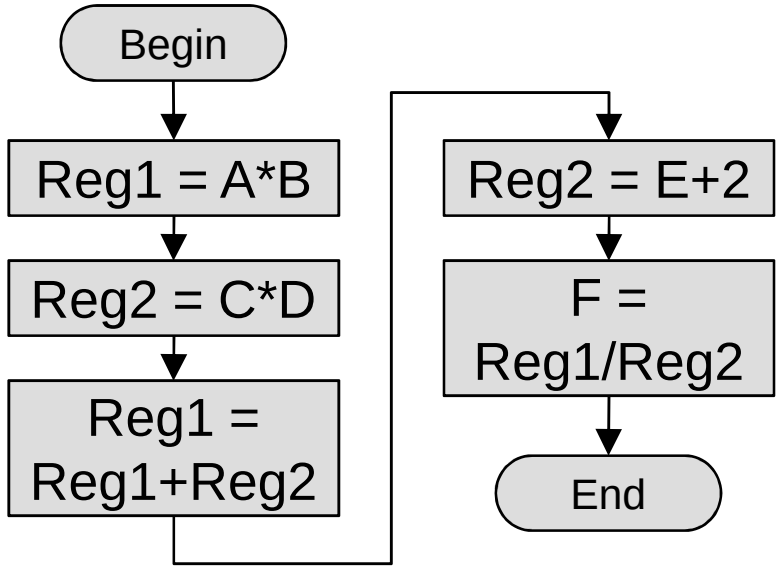
- Idea 2: Architekturální registry aktualizuj v programovém pořadí. Nicméně, pro výpočty používej jiné „neviditelné“ registry (které budou aktualizovány hned jak instrukce končí vykonávání) → **Future File (FF)**
- Future File:
  - Future file pro rychlý přístup k posledním hodnotám
  - Architectural file pro obnovení stavu cpu

A jaký je rozdíl mezi Future File a Rename Register File ? (slides 6 and 7 of this lecture)
FF má tolik položek kolik jich má ARF

Pamatujete si první přednášku?

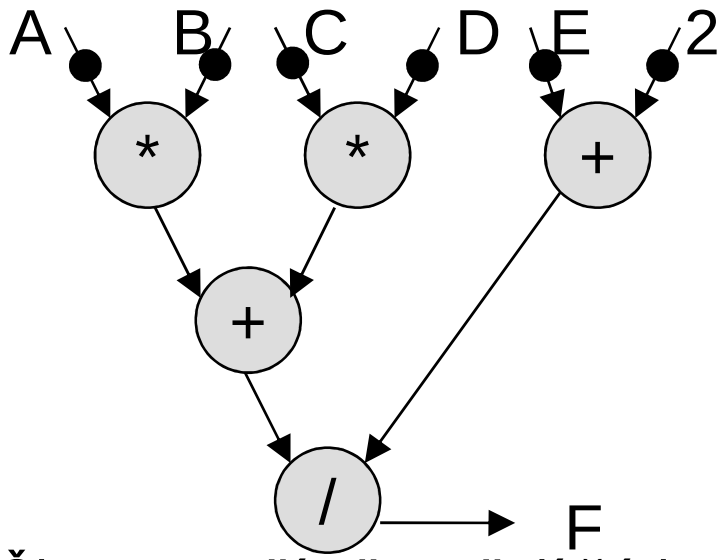
Uvažujme: F = (A*B + C*D) / (E+2)

Control Flow přístup:



Činnost počítače je určena sekvencí instrukcí…

Data Flow přístup:



Činnost počítače určují žádosti o výsledek (demand driven), nebo přítomnost operandů (data driven)..

Pamatujete si první přednášku?

- Instrukce se vykonávají sekvenčně tak, jak jsou zapsány, resp. jak to určuje čítač instrukcí (program counter).
  Explicitní přenosy řízení se realizují skokovými instrukcemi…
  → Imperativní programování
- Případné vykonávaní instrukcí (z důvodu lepšího využití HW) v jiném pořadí nesmí dávat jiný výsledek!

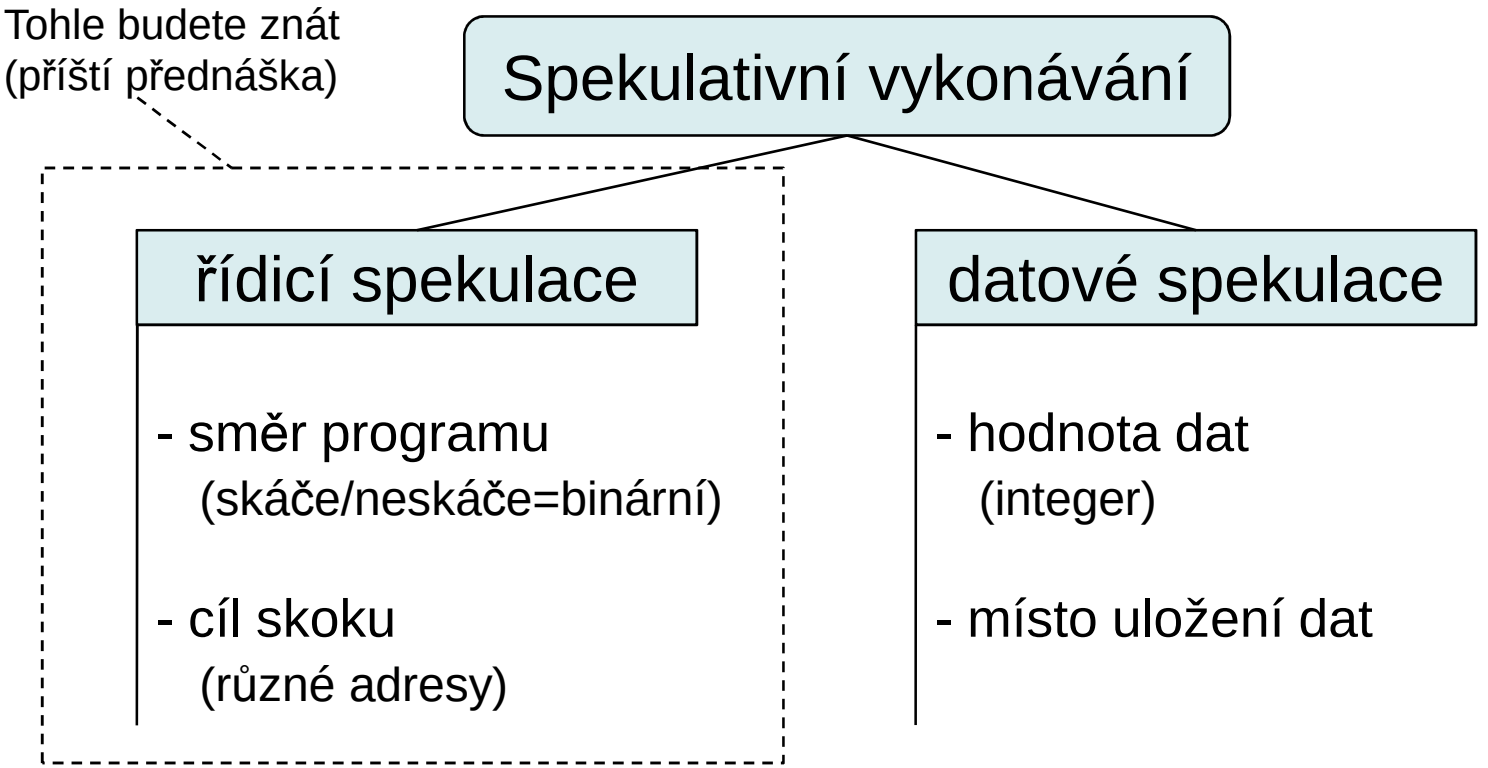
- Instrukce v DF počítači se realizuje jako šablona, kterou tvoří operátor, příjemce operandu a určení výsledku. Instrukce se vykoná vždy, když jsou dispozici potřebné operandy…
- Pořadí vykonávaní instrukcí je dáno datovými závislostmi a dostupností prostředků
  → Nezávislost instrukce od umístění v programu

Data flow limit

- Vykonávání instrukcí mimo pořadí tedy představuje realizaci principu Data Flow počítače uvnitř procesoru.
- V tomto případě se nepracuje s celým programem, ale pouze s velmi omezenou množinou instrukcí.
- Pokud bychom měli dostatek funkčních jednotek, můžeme dosáhnout maximální stupeň paralelizmu nad těmito instrukcemi (všechny WAW a WAR závislosti budou odstraněny, zůstanou pouze RAW závislosti).

- Co je **Data flow limit**?
- Navrhněte způsob dalšího zrychlení !!!

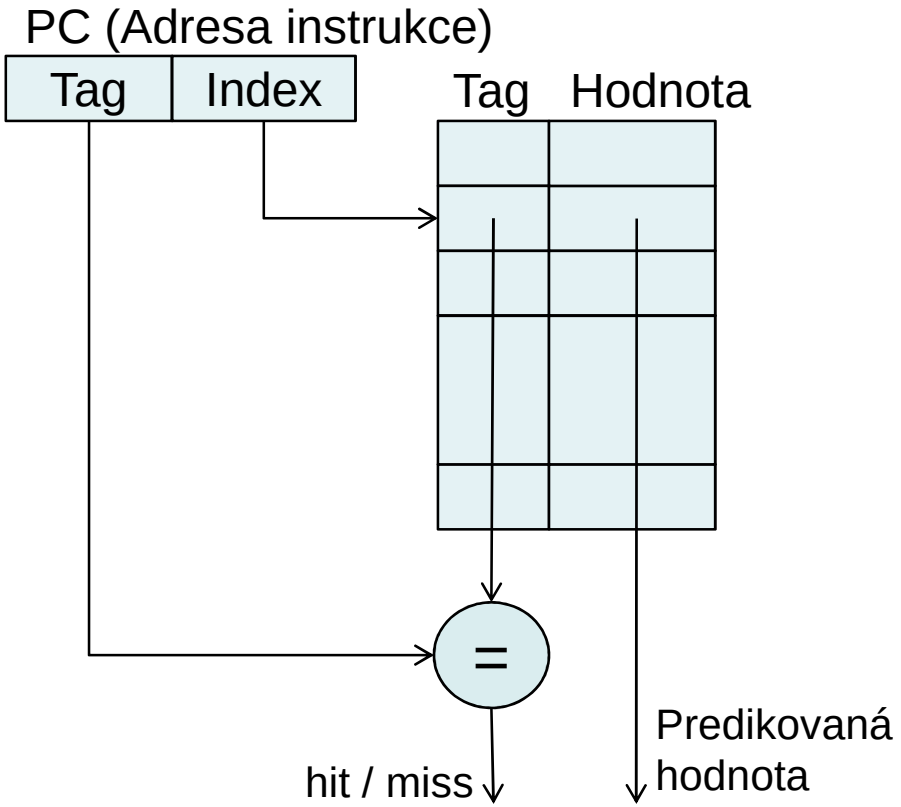
Spekulativní vykonávání – Co se určuje?

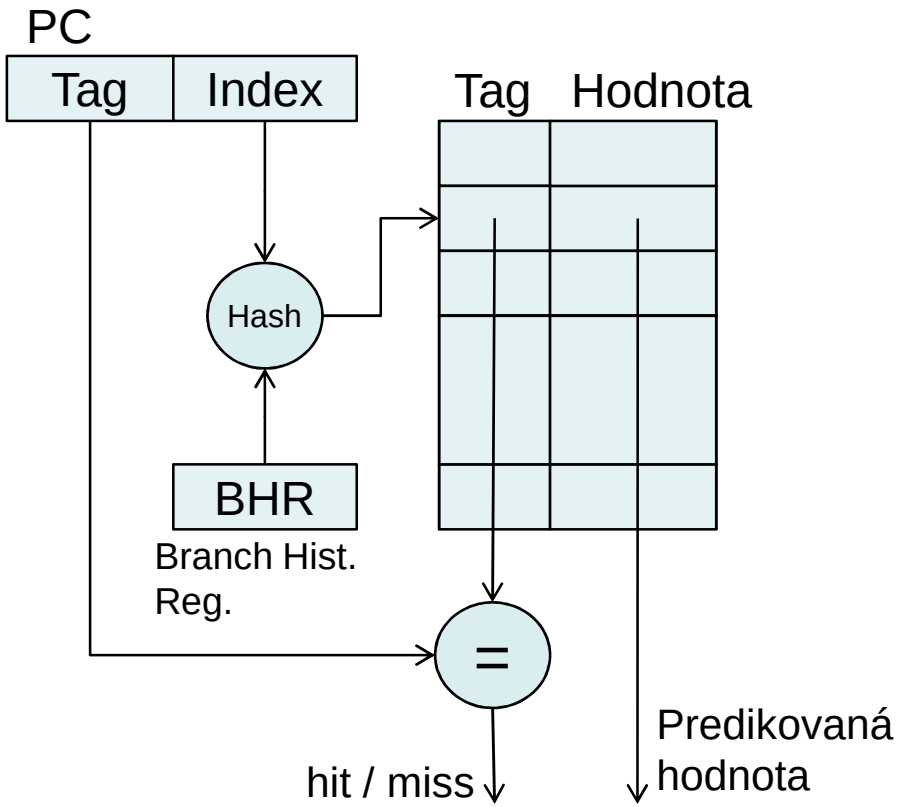


- Co je **Data flow limit**?
- Pokud tedy chceme dosáhnout další zrychlení, můžeme se „pokusit odstranit“ RAW závislosti…

Datové spekulace nad hodnotou

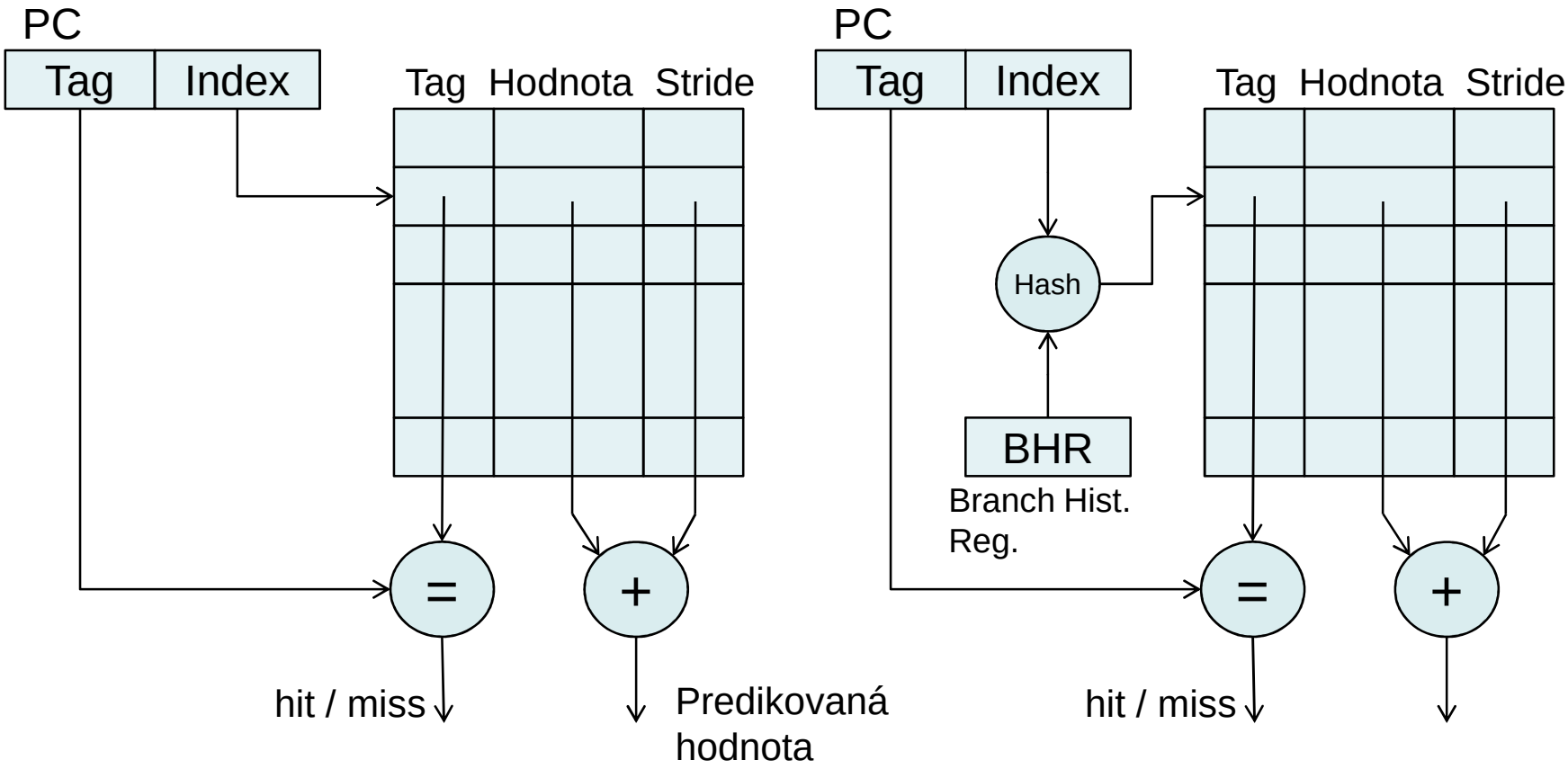
- Last-value predictor. Motivace: n=scanf(„%d“,&x), atd.





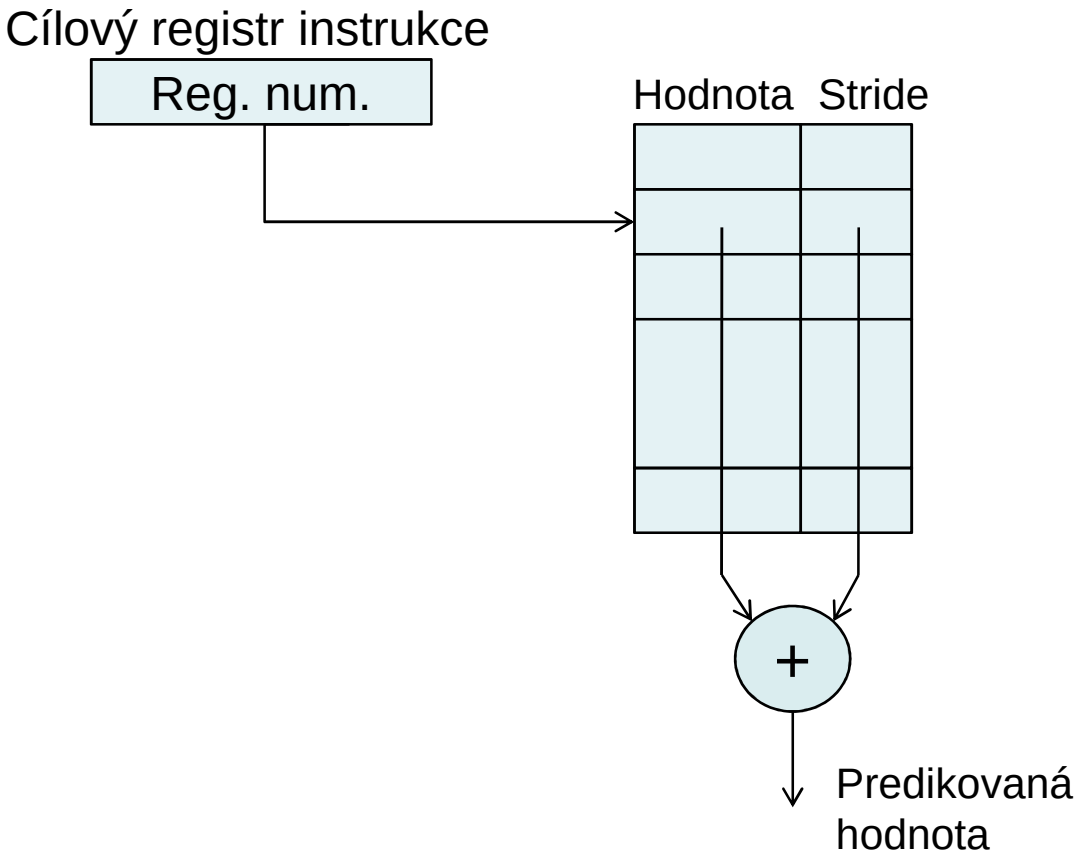
Datové spekulace nad hodnotou

- Stride predictor. Motivace: for(i=0;i<100;i++), p=malloc(16), atd.



Datové spekulace nad hodnotou

- Register-file predictor



Co dál?

- Existuje i jiná možnost odstranění WAW a WAR závislostí?
- Existuje i jiná možnost vyspořádání se s RAW závislostí?

- Ano… „mažeme i uzly a nejenom hrany“

Zdroje:

1. D. Sima: The design space of register renaming techniques http://www.eecs.umich.edu/eecs/courses/eecs470/papers/RegisterRenaming_Sima.pdf
2. http://en.wikipedia.org/wiki/IBM_System/360
3. Onur Mutlu: Computer Architecture 15-740/18-740, Lecture 9: More on Precise Exceptions. Carnegie Mellon University. 2011
4. **Shen, J.P., Lipasti, M.H.: Modern Processor Design : Fundamentals of Superscalar Processors, First Edition, New York, McGraw-Hill Inc., 2005**
5. http://www.dia.eui.upm.es/Asignatu/Arq_par/articulos/AMDBulldozer.pdf
6. http://www.cs.cmu.edu/afs/cs/academic/class/15213-f01/docs/mpr-p6.pdf