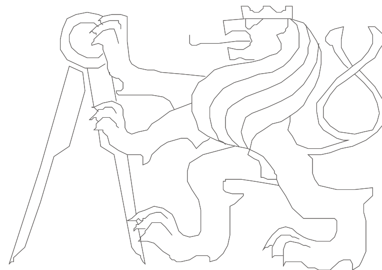
Pokročilé architektury počítačů

09

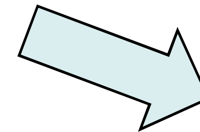
Superskalární techniky –
Tok dat z/do paměti (Memory Data Flow)
a
Procesory VLIW a EPIC



České vysoké učení technické, Fakulta elektrotechnická

Superskalární techniky – Připomeňme si…

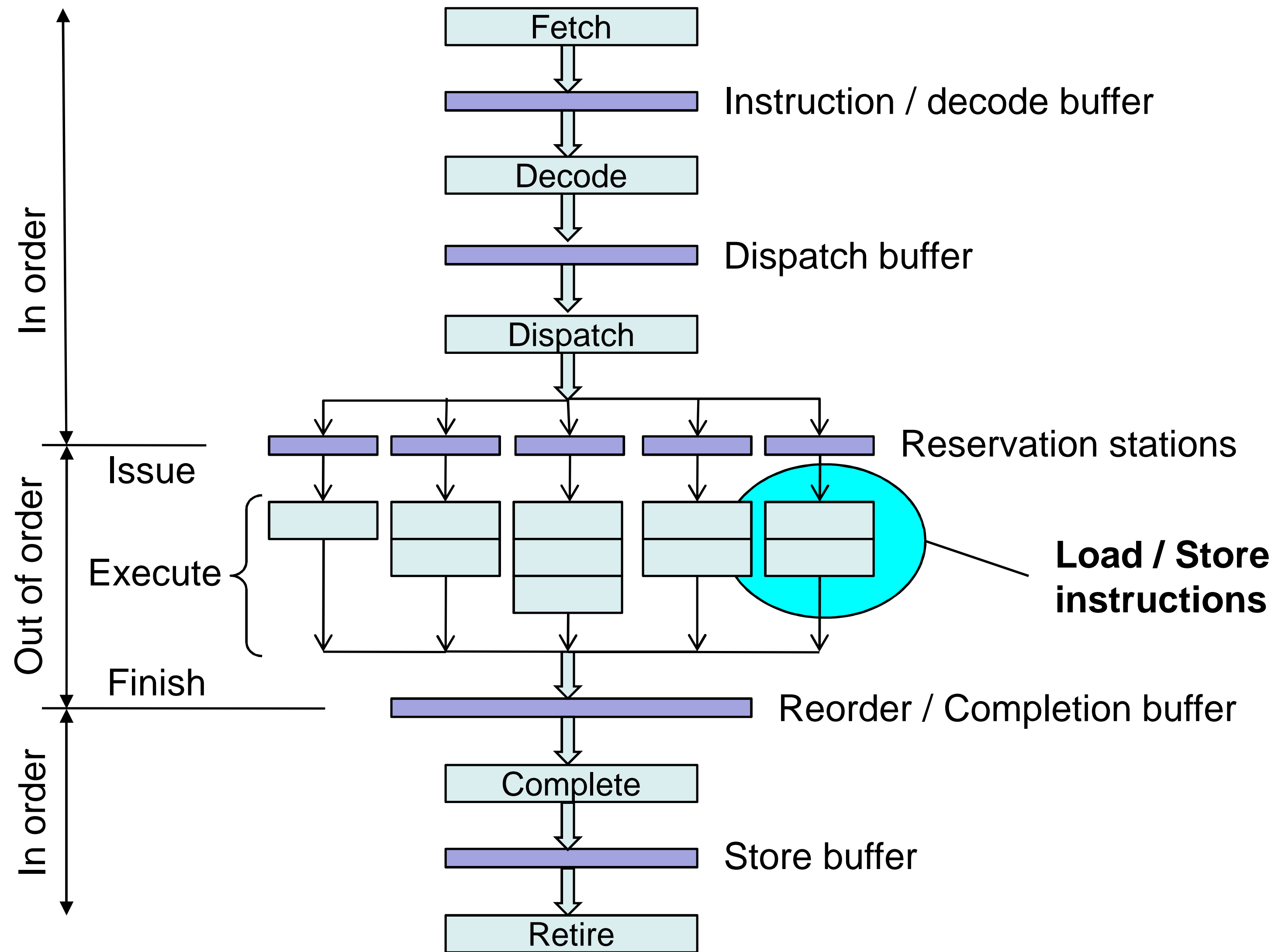
- Uvědomme si, že cílem je maximální propustnost zpracování instrukcí…
- Na **zpracování** instrukcí můžeme nahlížet jako na tok instrukcí a tok dat, přesněji:
  - tok samotných instrukcí (instruction flow)
  - tok dat mezi registry procesoru (register data flow)
  - **a tok dat z/do paměti (memory data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



téma první části dnešní přednášky

- Pokud tedy chceme maximalizovat celkový tok, musíme minimalizovat čas (penalizaci) těchto tří typů instrukcí

Superskalární organizace – **Připomeňme si..**



Co již víme..

- **Load / Store instrukce** jsou zodpovědné za přesuny dat mezi pamětí a vlastními registry procesoru
- Procesor disponuje značně omezeným počtem registrů
- Kompilátor generuje tzv. ***spill code***, kterým dočasně odkladá používaná data do paměti aby uvolnil místo v registrech – právě pomocí load/store instrukcí

Co způsobuje velkou latenci load/store instrukcí? Tři složky:

- **Generování adresy** – výpočet efektivní adresy
- **Překlad adresy** (viz virtuální paměť) – TLB hit vs. *TLB miss* (je odpovídajíci *page table* v paměti? ne? *page fault*..)
- **Samotný přístup do paměti** – viz další slajd

Přístup do paměti

- Load instrukce:
  - Data přijatá z paměti jsou zapsána buďto **do rename registru nebo reorder buffru**. V tomto bodě instrukce končí vykonávání (finishing execution). Aktualizace architekturálního registru se vykoná až bude instrukce dokončena – completed (uvolnění z reorder buffru)
- Store instrukce:
  - Instrukce končí vykonávání již po úspěšném překladu adresy. Data z registru, která mají být uložena do paměti jsou držena v reorder buffru. Zápis se provede až po dokončení instrukce, ne dřív. Proč je tomu tak?
  - **store buffer** - FIFO ; instrukce je retired když se aktualizuje paměť. Retiring – když je volná sběrnice..

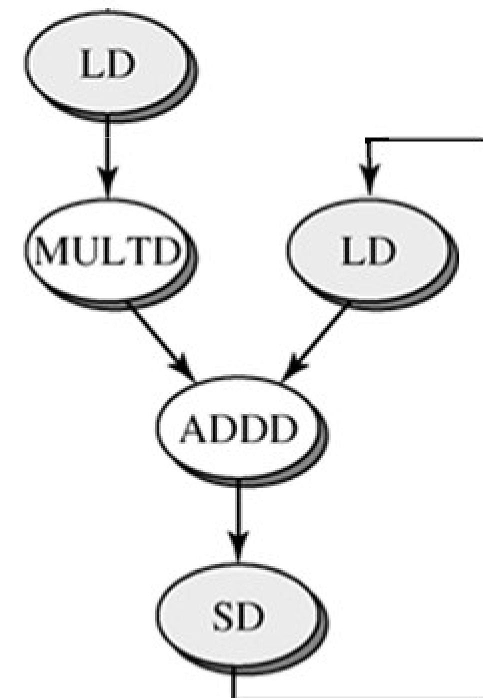
Uspořádání paměťových přístupů

- Datové závislosti – RAW, WAR, WAW – mezi load/store instrukcemi odkazujícími se na tu samou adresu
- **Total ordering** – dodržení programového pořadí všech load/store instrukcí. Je to nezbytné?

```
Y(i)=A*X(i)+Y(i)

     F0 ← LD,a
     R4 ← ADDI,Rx,#512        ;last address

Loop:
     F2 ← LD,0(Rx)            ;load X(i)
     F2 ← MULTD,F0,F2         ;A*X(i)
     F4 ← LD,0(Ry)            ;load Y(i)
     F4 ← ADDD,F2,F4          ;A*X(i)+Y(i)
     0(Ry) ← SD,F4            ;store into Y(i)
     Rx ← ADDI,Rx,#8          ;inc. index to X
     Ry ← ADDI,Ry,#8          ;inc. index to Y
     R20 ← SUB,R4,Rx          ;compute bound
     BNZ,R20,Loop             ;check if done
```

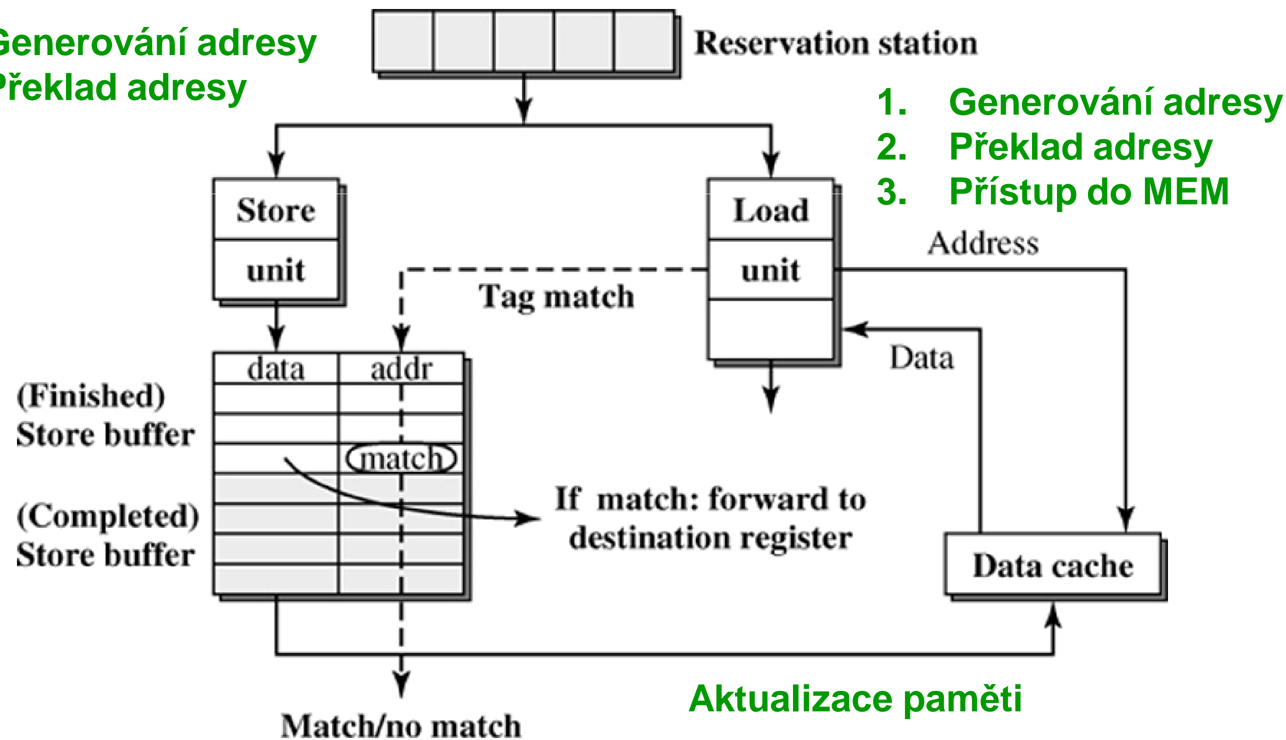


Model sekvenční konzistence

- **Podmínka sekvenční konzistence** klade jistá omezení na out-of-order vykonávání load/strore instrukcí
- Co když nastane exeption?
- Stav paměti se musí odvíjet **dle sekvenčního pořadí** load/store instrukcí
- To znamená, že store instrukce musí být vykonány v programovém pořadí, nebo přesněji, že paměť musí být aktualizována tak, jakoby store instrukce byly vykonány v programovém pořadí
- Pokud budou **store** instrukce vykonány **v programovém pořadí**, máme garantováno dodržení WAW a WAR závislostí. Zůstává dodržet RAW závislosti…
- Load instrukce – out-of-order

Load forwarding a Load bypassing

- ***Load bypassing*** umožňuje vykonat load před store, pokud jsou paměťove nezávislé. V opačném případě pak (pokud existuje):
- ***Load forwarding*** přeposílá data z instrukce store do instrukce load aby bylo zaručeno dodržení RAW závislosti



Store: dispached, issued, finished, completed, retired

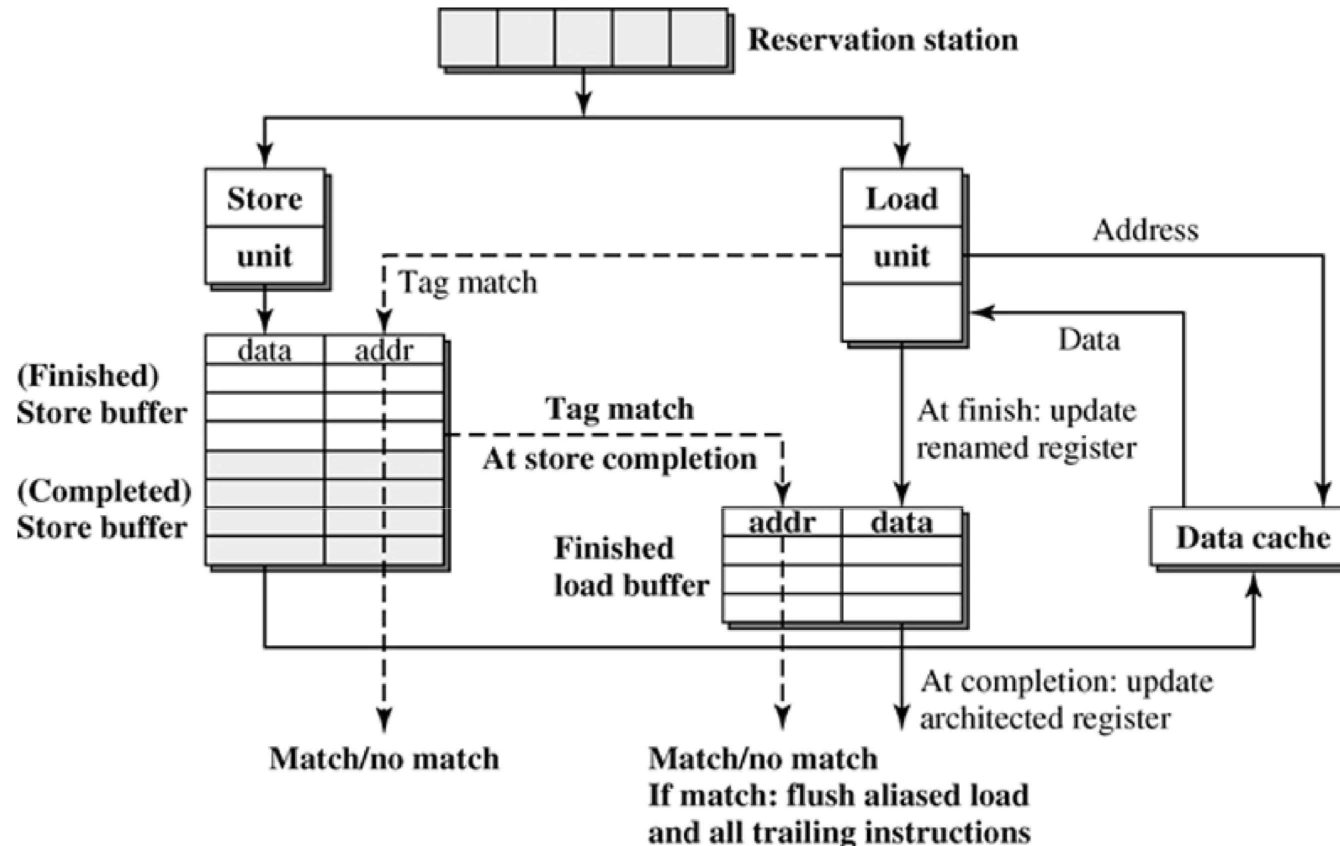
Load – if match: zahodí se přinesená data a berou se z Store buffru

Pokud neexistuje forwarding, pak se load označí za spekulativní...

Load forwarding a Load bypassing

- Pokud povolíme vydávaní instrukcí (issuing) z rezervační stanice out-of-order, pak se může stát, že instrukce load může být již vykonána, ale předcházející instrukce store se kterou má RAW není ještě v Store buffru (může být vykonávána, v rezervační stanici, nebo dokonce v paměti). Navíc nemáme ani informaci o její adrese (zda vůbec existuje RAW závislost).
- Řešení?
- Budeme předpokládat, že neexistuje závislost a tento předpoklad ověříme později... => **spekulativní vykonání**
- Spekulativní vykonávání podporuje ***Finished load buffer*** (Finish load queue)

Spekulativní vykonávání



- Load instrukce je v Finished load buffru po ukončení vykonávání před dokončením
- Kdykoliv store přechází k dokončení, musí vykonat alias checking s položkami FLB. Žádný konflikt -> store je dokončena; Konflikt -> zrušení spekulace load instr.

Spekulativní vykonávání

- Proč umožnit spekulace?
- Dřívější zavedení load může spustit *cache miss*
- A to může maskovat *cache miss penalty*

- Nicméně: V případě mylné spekulace – zrušení spekulativních instukcí (od load dále) – stálo nás to čas a prostředky, které mohly být využity lépe..
- Proto: ***Dependence prediction***
  V typických programech je závislost mezi store a load dobře predikovatelná
- ***Memory dependence predictor*** pak rozhodne zda začít spekulativně vykonávat load a další instrukce

Další **způsoby redukování latence paměti**

1. Paměťová hierarchie: L1 cache, L2, L3…
   Tohle již všichni znáte. Nebudeme se tím dále zabývat.

2. Používat neblokující cache (nonblocking cache, look-up free cache)
   Tradiční přístup: Pokud je cache miss pozastavíme vykonávání dokud neobdržíme data.
   Přístup neblokující cache: Instrukci, která spustila miss dáme stranou (do missed load queue) a jdeme dál. Pochopitelně závislosti na „neobsloužené" instrukci load musí být dodrženy (stall nebo predikce hodnoty->spekulace).

3. Prefetching cache
   Budeme předjímat budoucí miss a spustíme ho. K tomu potřebujeme: *memory reference prediction table* a *prefetch queue*.

Neblokující cache

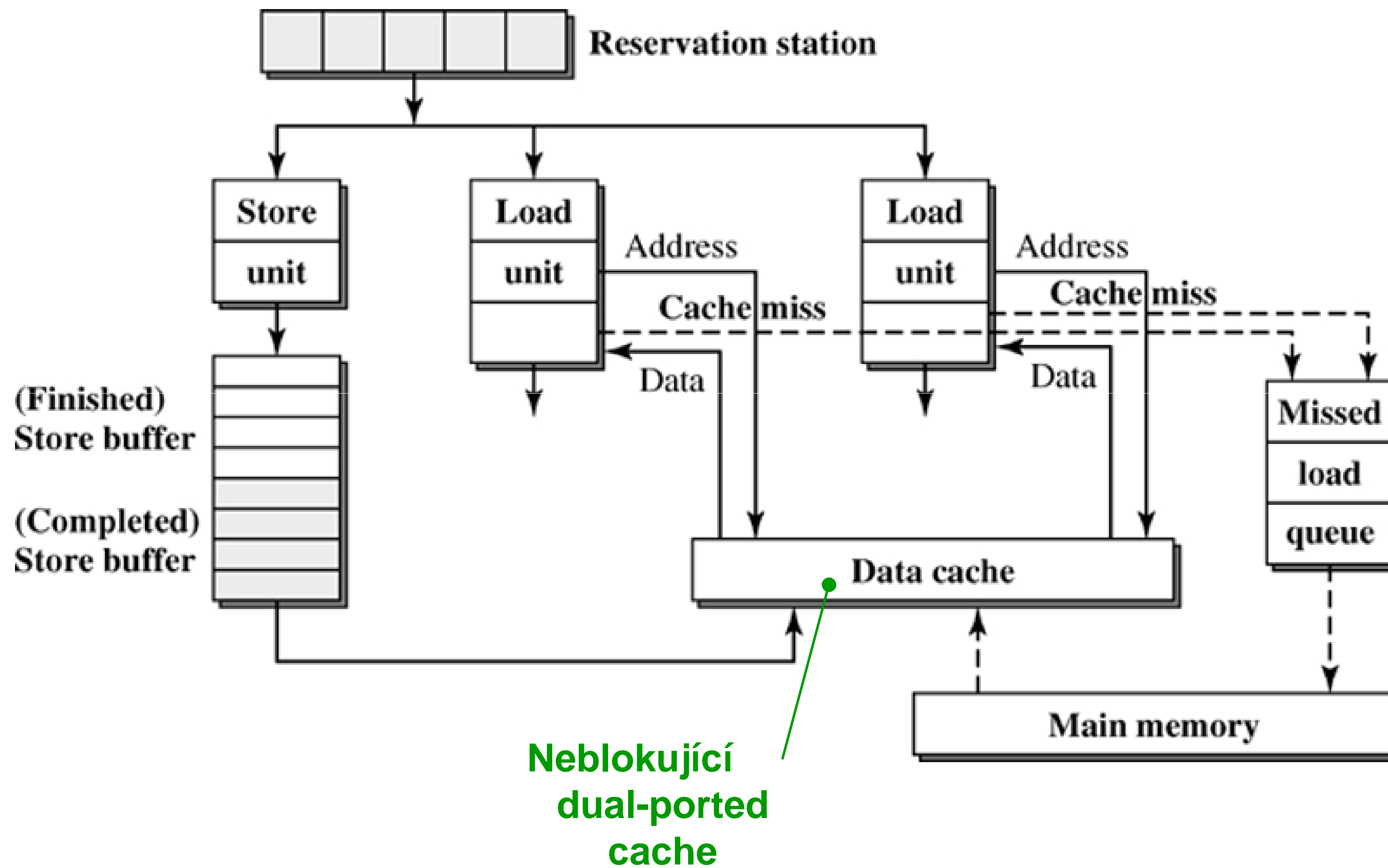
Základní myšlenka:

- Umožňuje další přístup (hit) i když nastal/obsluhuje právě jeden miss: **hit-under-miss** (další miss již způsobí stall)
- **miss-under-miss** (nebo také hit-under-multiple-misses) Příklad: Pentium Pro - 4 nevyřízené memory misses

Kdy dává smysl používat neblokující cache:

- Když procesor umí obsluhovat víc než 1 load/store (případ superskalárních procesorů)
- Když je cache společná pro víc než 1 procesor (nebo cache)

Neblokující cache



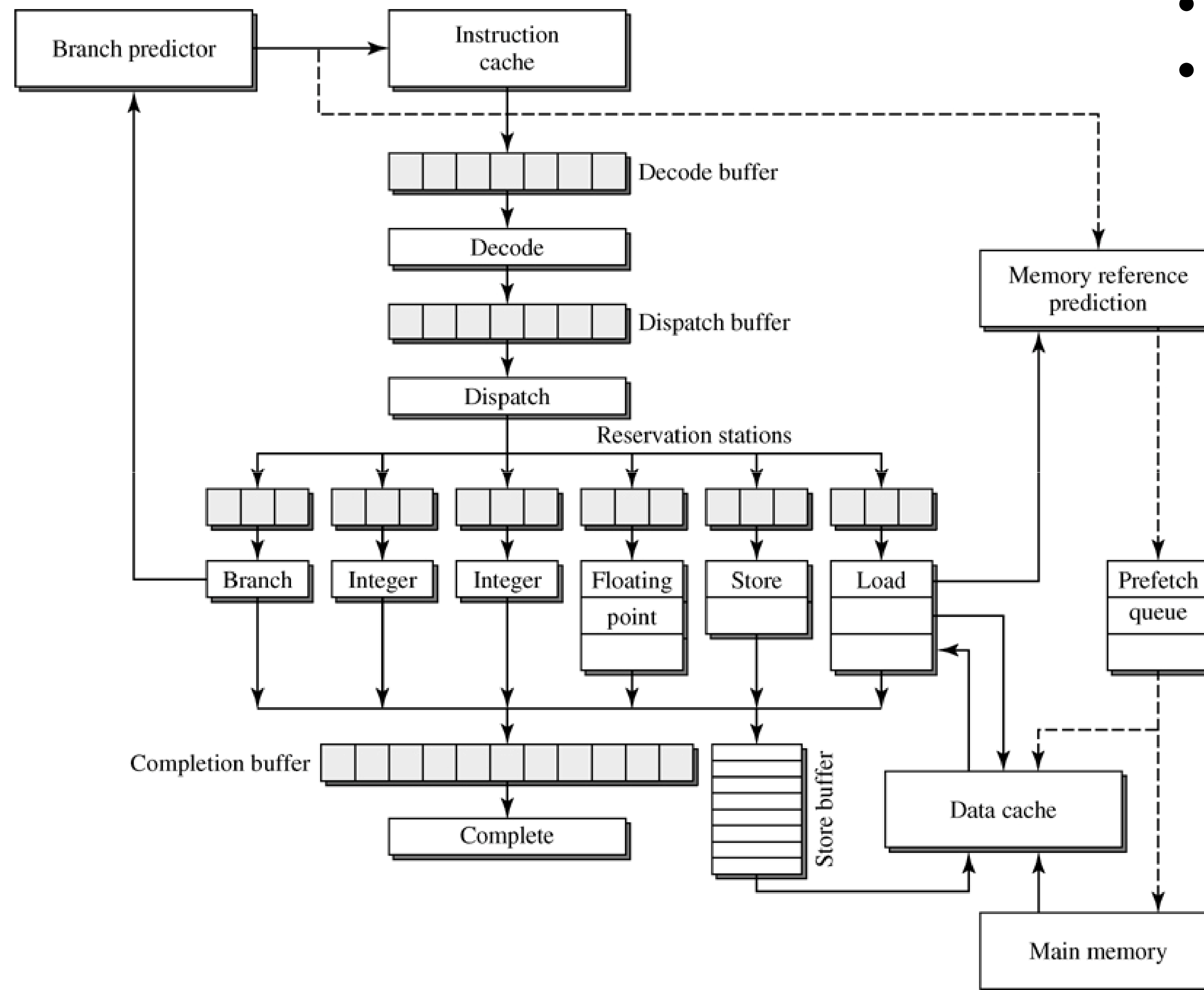
Prefetching cache / data (instruction) prefetching

- Terminologie: …záleží pouze na tom, kam dáme HW starající se o prefetching
- Myšlenka je přinést cache lines před tím, než jsou vůbec žádány
- Vzor přístupu do paměti je jiný pro instrukce a jiný pro data (instrukční cache vs. datová cache)

Může to dopadnout takto:

- užitečný předvýběr (přinesli jsme, našli jsme)
- neužitečný předvýběr (sice to v cache bylo, ale nezůstalo)
- škodící předvýber (nahradili jsme cahe line, která byla potřeba ještě potřeba – cache pollution)

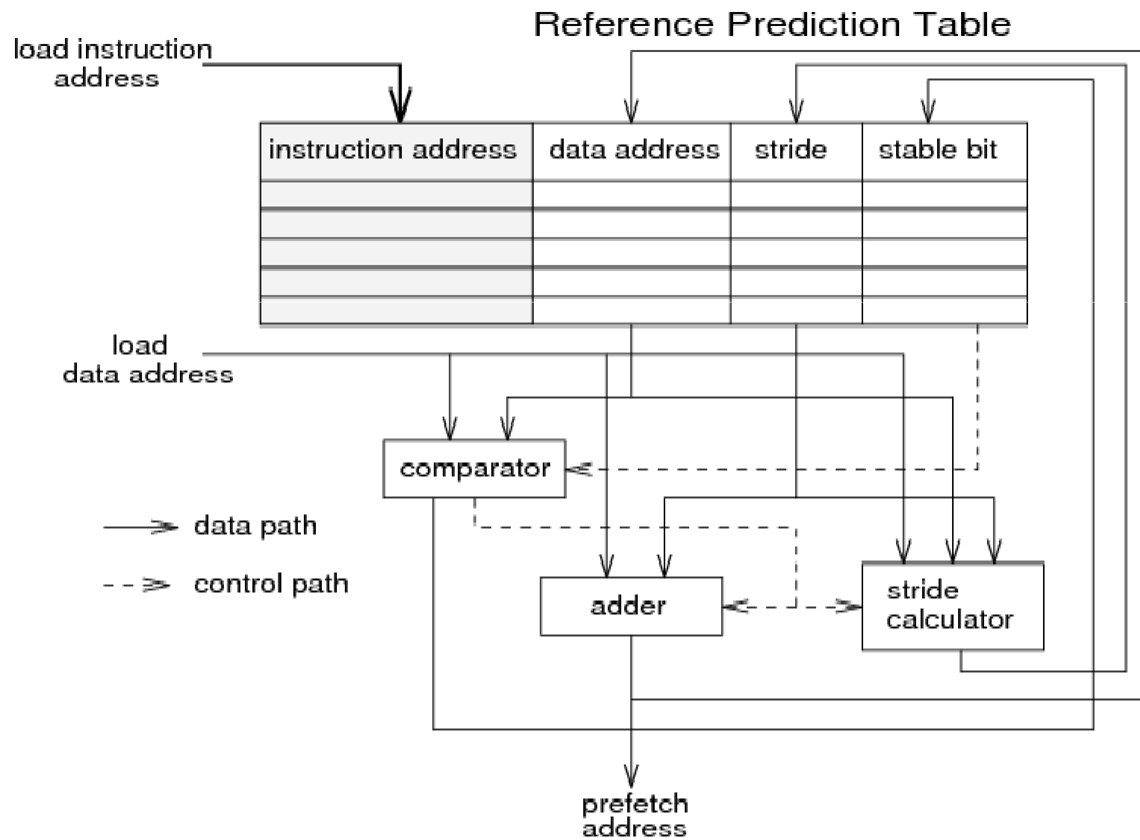
Prefetching cache / data (instruction) prefetching



- Load address prediction
- Load value prediction (minulá přednáška)

Jak zjistit odkud předvybírat? Příklad.

- Memory reference prediction



- Doteď jsme se bavili o HW technikách, které maximalizovali tok instrukcí – dynamické rozvrhování instrukcí a spekulativní vykonávání

Procesory VLIW atd.

- Teď se podíváme na statické techniky, tj. kompilátorem podpořený ILP

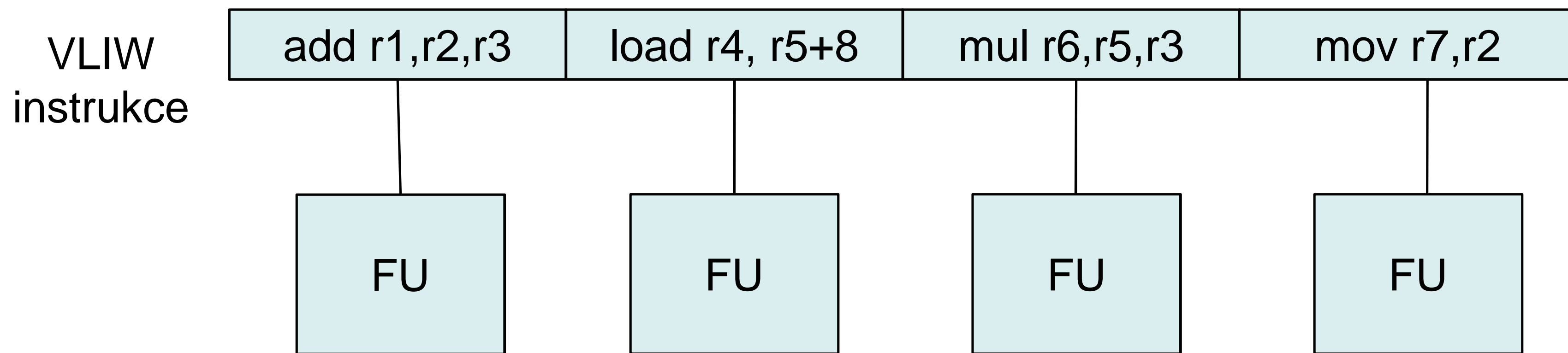
Procesory VLIW

- Very Long Instruction Word (typicky několik instrukcí 4-16)
- VLIW architektura umožňuje **paralelní zpracování** (původních několika instrukcí) **jednou novou instrukcí**.
- Paralelně zpracovatelné instrukce jsou naplánovány předem.
- Při kompilaci. (+ i -)
- VLIW je vlastně příkladem třídy MIMD.
- Klasický VLIW nepodporuje hazard detection – předpokládá nezávislost „instrukcí“ uvnitř instrukce => jednodušší HW

VLIW se zřetelem na přesné přerušení

- Sémantickou jednotkou pro akceptování přerušení zůstává instrukce (velmi dlouhá, či spíše široká)
- Pevný formát instrukce obsahuje kód několika operací, které se ale mohou provést paralelně.

VLIW - Princip



- HW vykonává všechny operace zakódované v instrukci nezávisle – jemnozrnný paralelizmus – paralelizmus na úrovni instrukcí (ILP)
- Kompilátor je zodpovědný za rozvržení instrukcí a extrakci ILP z programu

Příklad

Program pro superskalární DLX vs (V)LIW DLX

```
LF   F0,0(R1)
LF   F6,-4(R1)
LF   F10,-8(R1)
ADDF F4,F0,F2
LF   F14,-12(R1)
ADDF F8,F6,F2
LF   F18, -16(R1)
ADDF F12,F10,F2
SF   0(R1),F4
ADDF F16,F14,F2
SF   -4(R1),F8
ADDF F20,F18,F2
SF   - 8(R1),F12
SF   -12(R1),F16
SUBI  R1,R1,#20
BNEZ R1,LOOP
SF   4(R1),F20
```

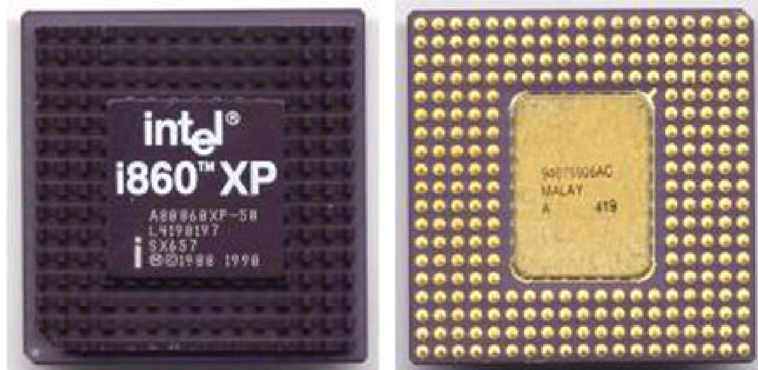
17 instructions
x 4B each
= 68B

```
LF   F0,0(R1)       NOP
LF   F6,-4(R1)      NOP
LF   F10,-8(R1)     ADDF F4,F0,F2
LF   F14,-12(R1)    ADDF F8,F6,F2
LF   F18, -16(R1)   ADDF F12,F10,F2
SF   0(R1),F4       ADDF F16,F14,F2
SF   -4(R1),F8      ADDF F20,F18,F2
SF   - 8(R1),F12    NOP
SF   -12(R1),F16    NOP
SUBI  R1,R1,#20     NOP
BNEZ R1,LOOP        NOP
SF   4(R1),F20      NOP
```

12 (long) instructions
x 8B each
= 96B

Architektura počítačů

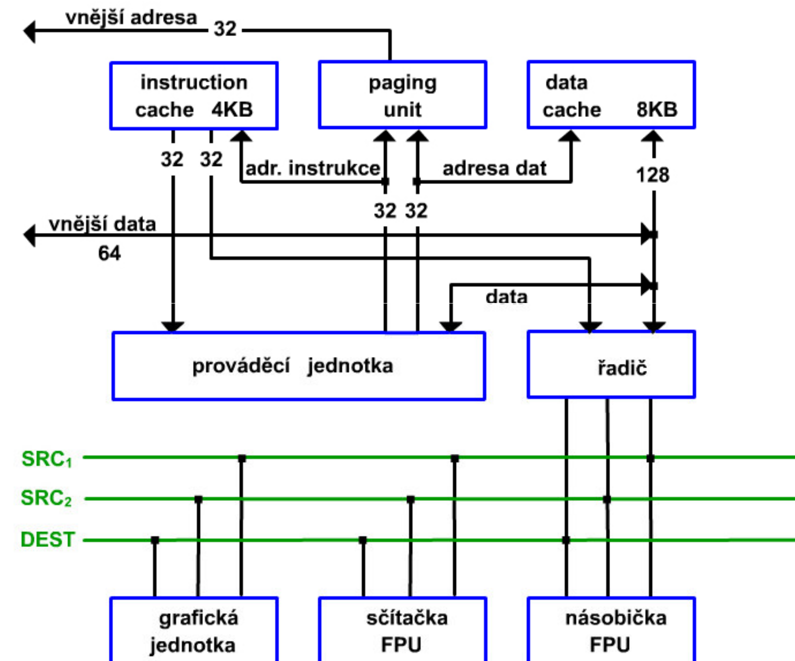
Příklad VLIWu



i860 XP
A80860XP-50
L4190197
SX657
(M)(C)1988 1990

94075906AC
MALAY
A 419

Core Frequency:	50 MHz
Board Frequency:	50 MHz
Data bus (ext.):	64 Bit
Address bus:	64 Bit
Transistors:	2,500,000
Circuit Size:	0.80 µ
Introduced:	1988
Manufactured:	week 19/1994
L1 Cache:	16+16 KB
CPU Code:	N11
Intel S-Spec:	SX657
Package Type:	Ceramic PGA-262



Další příklad VLIWu - TM3270 Media-Processor

- Navržen pro zpracování videa a zvuku, Rok 2005
- Variabilní délka od 2B do 28B

Table 1. TM3270 Architecture

Architectural feature	Quantity
Architecture	5 issue slot VLIW guarded RISC-like operations
Pipeline depth	7-12 stages
Address width	32 bits
Data width	32 bits
Register-file	Unified, 128 32-bit registers
Functional units	31
IEEE-754 floating point	yes
SIMD capabilities	1 x 32-bit, 2 x 16-bit, 4 x 8-bit
Instruction cache	64 Kbyte, 128-byte lines, 8 way set-associative, LRU replacement policy
Data cache	128 Kbyte, 128-byte lines 4 way set-associative, LRU replacement policy, Allocate-on-write miss policy

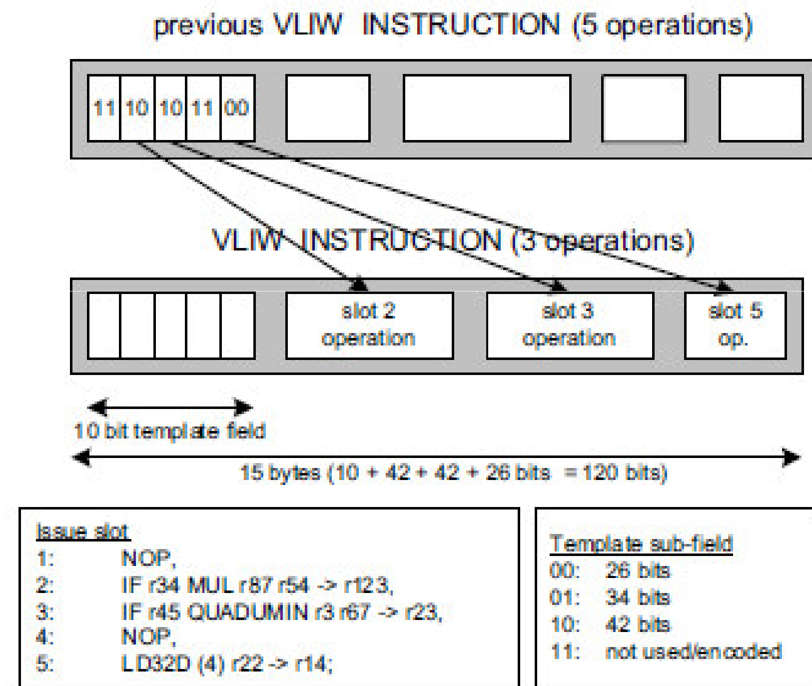


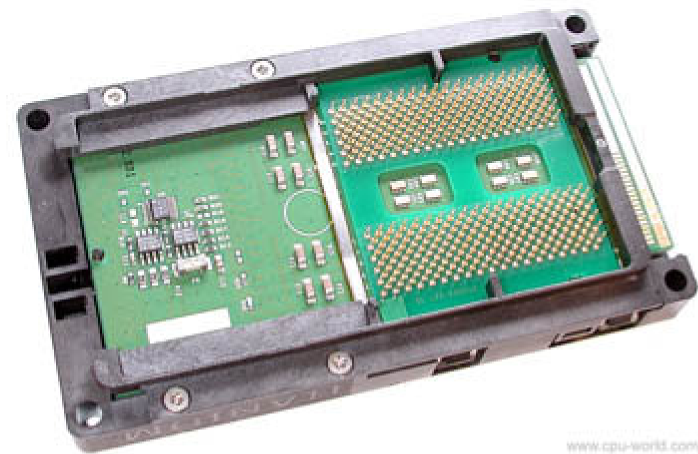
Figure 1. VLIW instruction encoding.

Moderní DSP

- Proč VLIW? Výrobce dodá zařízení (procesor) i příslušné programové vybavení…
- Superskalární zpracování
- Frekvence více než 1 Ghz
- Dvouúrovňové cache s až 8 MB
- SIMD
- VLIW – až 8 instrukcí za instrukční cyklus
- Speciální jednotky pro výpočet celé FFT

Co je to EPIC

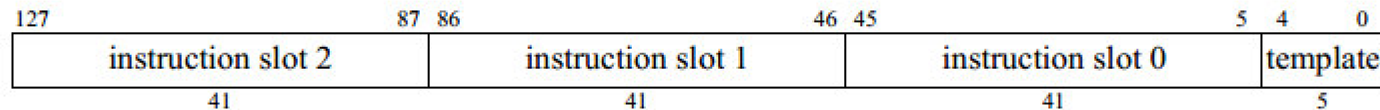
- Explicitly Parallel Instruction Computing
- Kořeny této architektury vyrůstají z VLIW
- Představitelem této architektury je **Itanium** (dřívější název **IA-64**).
- Jsou v ní implementovány dříve popsané metody jako spekulace, predikce skoků a přejmenování registrů.



Čím se liší EPIC od VLIW a co přináší

- Bundle / packet – označení pro skupinu instrukcí, které budou seskupeny dohromady
- Součásti bundle je „stop“, který indikuje, že dochází k závislosti
- SW prefetch (lfetch)
- Predikace – jistý způsob spekulace
- Spekulativní load (ld.s, ld.sa,ld.c.nc, ld.c.clr,…),
- Přesun instrukce load k dřívějšímu vykonání a pozdější checking v místě původního loadu
- Přesun instrukce load k dřívějšímu vykonání před store a checking zda nedošlo k aliasingu (ta samá adresa)

IA-64



Bundle Format

- IA-64 rozlišuje 6 typů instrukcí
- Bundle seskupuje 3 instrukce

Relationship Between Instruction Type and Execution Unit Type

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit

Predikace

- Původní kód:

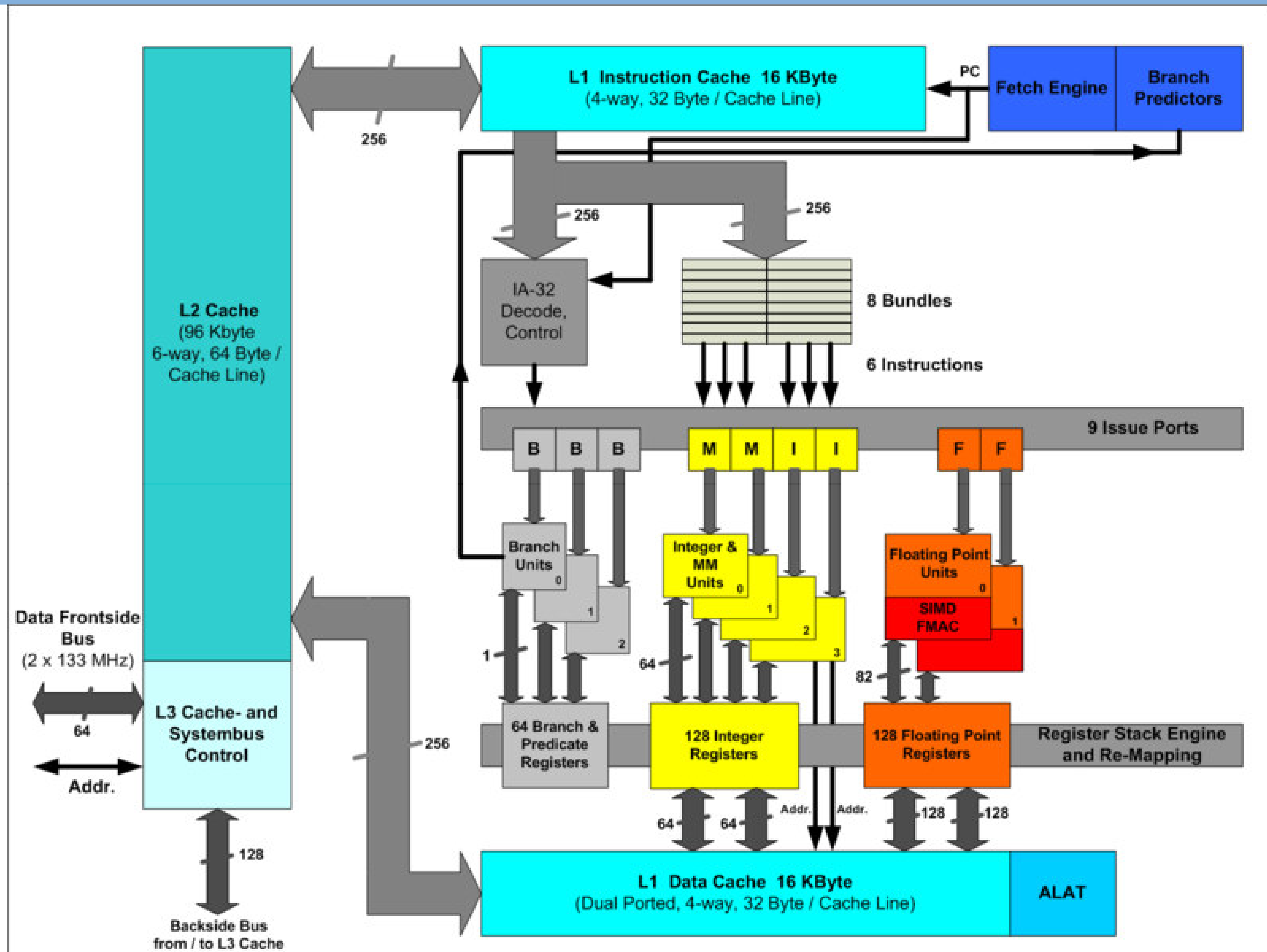
```
if (a>b) c = c + 1
else d = d * e + f
```

- Nový kód:

```
pT, pF = compare(a>b)
if (pT) c = c + 1
if (pF) d = d * e + f
```

- Predikát pT se nastaví pokud bude podmínka true. Predikát pF je komplement k pT
- Tím se řídicí závislost konvertovala na datovou
- Další výhodou je možnost zařadit instrukce pro paralelní vykonání

EPIC



Tento obrázek pochází z http://en.wikipedia.org/wiki/File:Itanium_arch.png

EPIC

- Od architektury x86 (i x86-64) se dramaticky liší.
- Je založena na explicitní ILP, o paralelizaci se rozhoduje při překladu.
- Nepotřebuje ale speciální HW pro zajištění odstranění hazardů.
- Další podrobnosti? Referát.

Využití datového paralelismu

Využití datového paralelismu, SIMD

- Single Instruction, Multiple Data, jedna ze tříd klasické Flynovy taxonomie.
- První široce užívanou implementací SIMD architektury bylo pro hry určené MMX (*MultiMedia eXtensions)* rozšíření pro x86.

Vektorové instrukce v ISA

- Do této skupiny patří i další rozšíření:
- 3DNow! Od AMD,
- SSE a další verze SSE2, SSE3, SSE4 od Intelu.

SSE instrukce - příklad

- SSE3 instrukce **ADDSUBPD** (*Add-Subtract-Packed-Double*)
  Input: { A0, A1 }, { B0, B1 }
  Output: { A0 – B0, A1 + B1 }
- SSE3 instrukce **HADDPS** (*Horizontal-Add-Packed-Single*)
  Input: { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
  Output: { A0 + A1, A2 + A3, B0 + B1, B2 + B3 }
- SSE4 Instrukce **MPSADBW**: 8 součtů
  $|x_0-y_0|+|x_1-y_1|+|x_2-y_2|+|x_3-y_3|$,
  $|x_0-y_1|+|x_1-y_2|+|x_2-y_3|+|x_3-y_4|,\ldots,$
  $|x_0-y_7|+|x_1-y_8|+|x_2-y_9|+|x_3-y_{10}|$
  Důležité pro HighDefinition (HD) kodeky
- SSE4 Instrukce **DPPS**: Skalární součin Array of Structs

Putting it all together

Loop

```
for (i=0; i<1000; i++)
      x[i] = x[i] + s;
```

```
Loop:   LD      F0,0(R1)    ;F0=vector element
        NOP
        ADDD    F4,F0,F2    ;add scalar from F2
        SD      0(R1),F4    ;store result
        SUBI    R1,R1,8     ;decrement pointer 8bytes (DW)
        BNEZ    R1,Loop     ;branch R1!=zero
        NOP                 ;delayed branch slot
```

Smyčka, 7 cyklů

Loop unrolling

```
1 Loop: LD     F0,0(R1)
2       LD     F6,-8(R1)
3       LD     F10,-16(R1)
4       LD     F14,-24(R1)
5       ADDD   F4,F0,F2
6       ADDD   F8,F6,F2
7       ADDD   F12,F10,F2
8       ADDD   F16,F14,F2
9       SD     0(R1),F4
10      SD     -8(R1),F8
11      SD     -16(R1),F12
12      SUBI   R1,R1,32
13      BNEZ   R1,LOOP
14      SD     8(R1),F16      ; 8-32 = -24
```

- Odstraněny NOP
- Vhodné i pro CPU nepodporující přejmenování registrů (Registry přejmenoval kompilátor.)
- Minimalizuje stall pro skalární procesor

Smyčka rozbalena 4x, 14 cyklů

Vykonání na superskalárním procesoru

- Umíte si představit Tomasulův algoritmus v akci?

Iteration no.	*Instructions*	*Issues*	*Executes*	*Writes result*
		clock-cycle number		
1	**LD F0,0(R1)**	**1**	**2**	**4**
1	**ADDD F4,F0,F2**	**1**	**5**	**8**
1	**SD 0(R1),F4**	**2**	**9**	
1	**SUBI R1,R1,#8**	**3**	**4**	**5**
1	**BNEZ R1,LOOP**	**4**	**5**	
2	**LD F0,0(R1)**	**5**	**6**	**8**
2	**ADDD F4,F0,F2**	**5**	**9**	**12**
2	**SD 0(R1),F4**	**6**	**13**	
2	**SUBI R1,R1,#8**	**7**	**8**	**9**
2	**BNEZ R1,LOOP**	**8**	**9**	

4 cykly na iteraci, NOP?

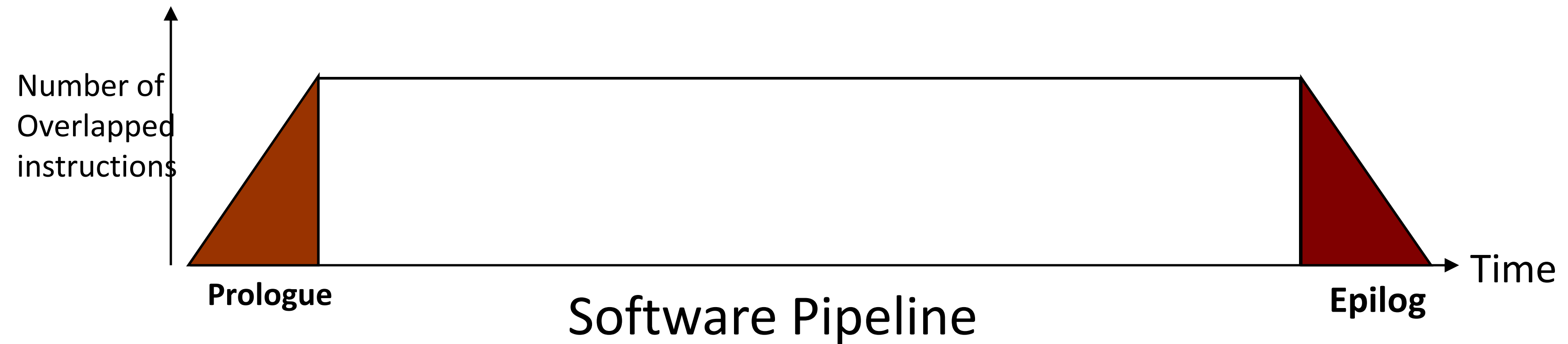
Loop unrolling ve VLIW-u

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD -0(R1),F28				BNEZ R1,LOOP	9

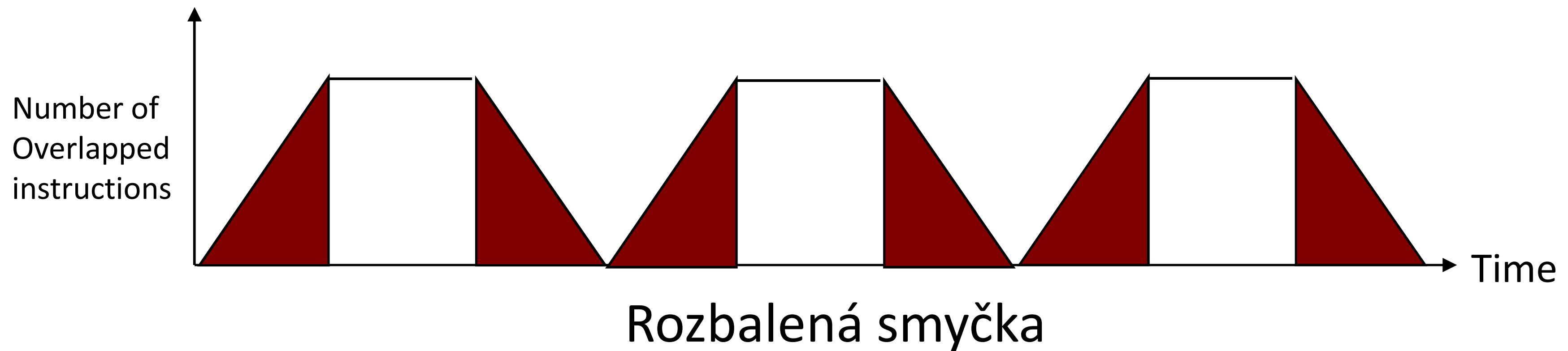
1 řádek – 1 instrukce

Smyčka rozbalena 7x, 9 cyklů

Software Pipeline



Software Pipeline



Rozbalená smyčka

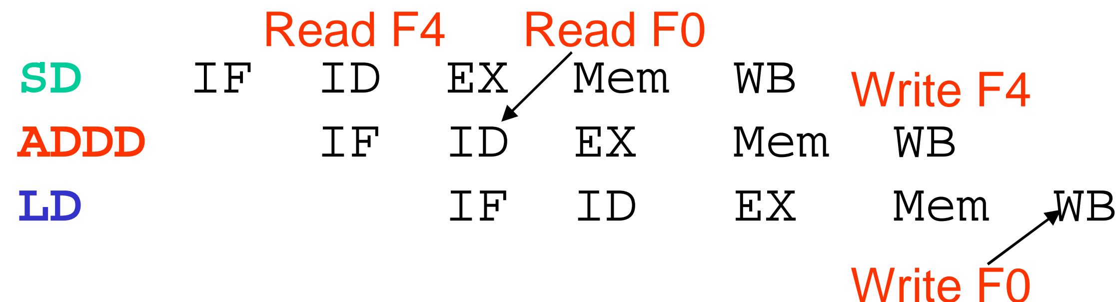
SW Pipelining - Příklad

Before: Unrolled 3 times

```
1   LD    F0,0(R1)
2   ADDD  F4,F0,F2
3   SD    0(R1),F4
4   LD    F6,-8(R1)
5   ADDD  F8,F6,F2
6   SD    -8(R1),F8
7   LD    F10,-16(R1)
8   ADDD  F12,F10,F2
9   SD    -16(R1),F12
10  SUBI  R1,R1,#24
11  BNEZ  R1,LOOP
```

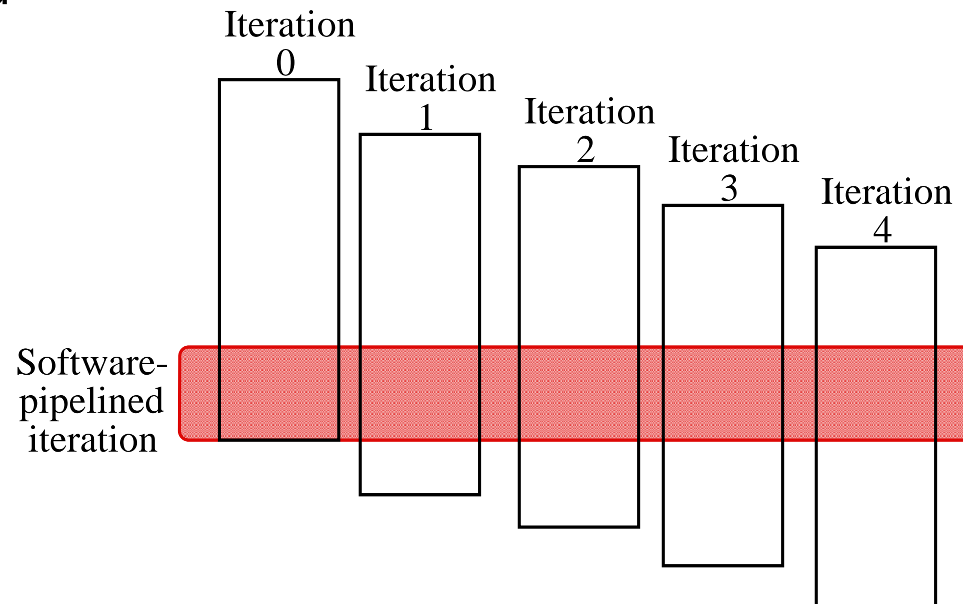
After: Software Pipelined

```
    LD    F0,0(R1)
    ADDD  F4,F0,F2
    LD    F0,-8(R1)
1   SD    0(R1),F4;     Stores M[i]
2   ADDD  F4,F0,F2;     Adds to M[i-1]
3   LD    F0,-16(R1);   loads M[i-2]
4   SUBI  R1,R1,#8
5   BNEZ  R1,LOOP
    SD    0(R1),F4
    ADDD  F4,F0,F2
    SD    -8(R1),F4
```



SW Pipelining – Symbolic Loop Unrolling

- Pokud jsou **jednotlivé iterace cyklu nezávislé** (instrukce uvnitř cyklu mohou být závislé), můžeme dosáhnout zvýšení ILP tím, že seskupíme instrukce z různých iterací
- SW pipelining reorganizuje smyčky (≈ Tomasulovu algoritmu nad rozbalenou smyčkou)
- Dosáhneme největší střední stupeň paralelizmu pouze při malém nárůstu kódu



Příklad odstranění závislosti mezi iteracemi

- Závislost mezi iteracemi způsobuje B

OLD:

```
for (i=1; i<=100; i=i+1) {
        A[i+1] = A[i] + B[i];     /* S1 */
        B[i+1] = C[i] + D[i];} /* S2 */
```

NEW:

```
A[1] = A[1] + B[1];
for   (i=1; i<=99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] =  + A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Použité zdroje:

1. **Shen, J.P., Lipasti, M.H.: Modern Processor Design : Fundamentals of Superscalar Processors, First Edition, New York, McGraw-Hill Inc., 2005**
2. Sunil Kim, Alexander V. Veidenbaum: Second level cache organization for data prefetching. 1996
3. David A. Patterson: Lecture 5: VLIW, Software Pipelining, and Limits to ILP. Computer Science 252, Fall 1996.
4. Ioannis Papaefstathiou: Advanced Computer Architecture - Chapter 4. Advanced Pipelining. CS 590.25 Easter 2003.
5. van de Waerdt, J.-W.; Vassiliadis, S.; et al. "The TM3270 media-processor,"*Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on* , vol., no., pp.12 pp.,342, 16-16 Nov. 2005
6. http://www.csee.umbc.edu/portal/help/architecture/aig.pdf
7. http://www.cs.cmu.edu/afs/cs/academic/class/15740-f03/public/doc/discussions/uniprocessors/ia64/mpr_ia64_isa_may99.pdf