Řetězené zpracování

INP 2019
FIT VUT v Brně

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

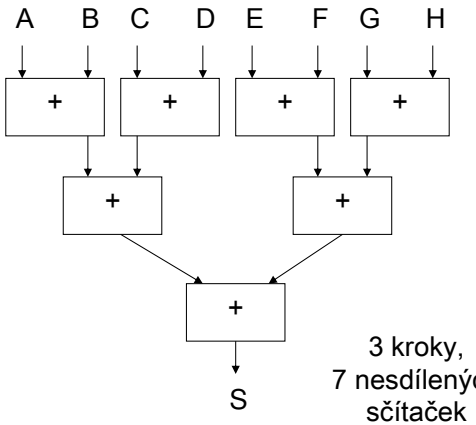
Techniky urychlování výpočtu v HW

- Lze realizovat speciální kódování dle potřeby dané úlohy
  - Příklad: kód zbytkových tříd
- Lze zvolit „optimální“ počet bitů pro danou úlohu
  - Příklad: 13-bitová ALU, pokud 13 bitů stačí pro danou úlohu (a ne právě 32 nebo 64 bitů, které nabízí procesor)
- Lze realizovat speciální výpočetní jednotky dle potřeby dané úlohy
  - Příklad: FFT, která není v běžném procesoru
- Paralelní zpracování (násobné výpočetní jednotky)
- Zřetězené zpracování – jinak též *pipelining*, překládané též jako *proudové zpracování*.
  - tato přednáška

Paralelní zpracování

Př. S = A + B + C + D + E + F + G + H

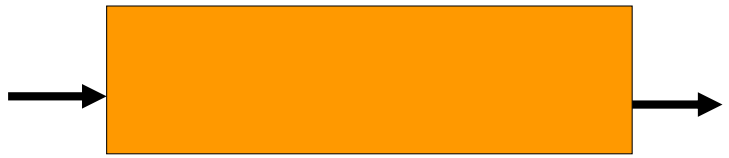
```
SW: sekvenčně

S = A + B;
S = S + C;
S = S + D;
S = S + E;
S = S + F;
S = S + G;
S = S + H;

7 kroků,
1 sdílená
sčítačka
```



3 kroky,
7 nesdílených
sčítaček

Obvod bez zřetězení

(Př. obvod=filtr, vstup=pixel, výstup=upravený pixel)



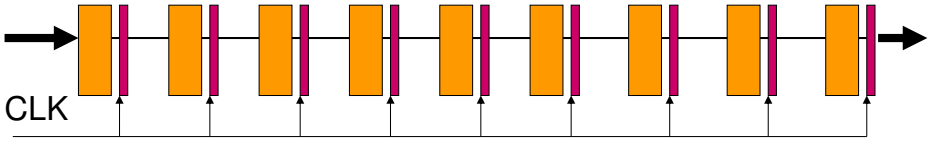
1 2 3 4 5 6

1’ 2’ 3’ 4’ 5’ 6’

Příklad: 65536 pixelů
Předpokládejme zpoždění obvodu: 90 ns
Celkem: 65536 x 90 ns = 5898240 ns = 5,9 ms

Zřetězený filtr:

rozděl původní obvod, přidej registry



1.PIXEL								
2	1							
3	2	1						
4	3	2	1					
5	4	3	2	1				
6	5	4	3	2	1			
7	6	5	4	3	2	1		
8	7	6	5	4	3	2	1	
9	8	7	6	5	4	3	2	1
10	9	8	7	6	5	4	3	2
11	10	9	8	7	6	5	4	3
12	11	10	9	8	7	6	5	4

9 pixelů je zpracováváno současně ⇒ zrychlení 9x oproti předchozímu řešení

Zřetězený filtr – 9 stupňů

- Příklad: 65536 pixelů
- Předpokládejme zpoždění stupně: 10 ns
- 1. pixel bude zpracován za 9 x 10 = 90 ns
- 2., 3. až 65536. pixel – každý za 10 ns
- Celkem: 90 ns + 65535 x 10 ns = 655440 ns = 0.655440 ms
- Zrychlení: 8.9989 krát
  - není to 9x, protože se musí naplnit zřetězená linka
  - při výpočtu jsme neuvažovali zpoždění registru

Zrychlení použitím řetězení:

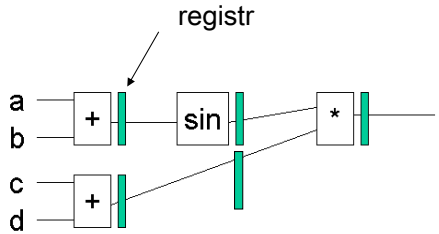
vytvoříme *k* stupňů oddělených registry

- Počet vstupů, které zpracováváme: N
- Počet stupňů: k
- Zpoždění stupně: t
- Zpoždění registru: d
- Zrychlení: $\frac{N.k.t}{(k+N-1)*(t+d)}$

Kdy má smysl zavést řetězené zpracování?
• Pokud je N dostatečně velké!
• Pokud je zpoždění stupňů přibližně stejné.
Jaký je optimální počet stupňů?
Jaká je maximální frekvence, pokud je zpoždění stupňů různé?

Zřetězení + duplikace jednotek ⇒ další urychlení

```
SW:
Vypočtěte
F[i]=(c[i]+d[i])*sin(a[i]+b[i])
pro i=1..100

for (i=1; i<=100; i++)
{
  pom = a[i]+b[i];
  pom1 = sin(pom);
  pom2 = c[i]+d[i];
  F[i] = pom1 * pom2;
}

Celkem: 100 x 4 = 400 kroků
(a to neuvažujeme test podmínky
v cyklu)
```



První výsledek za 3 kroky.
2. až 100. výsledek za 1 krok.
Celkem 102 kroků
Zrychlení 3,92 krát.

Řetězené zpracování instrukcí v procesorech

- Princip zřetězení se značně překrývá s principy procesorů typu RISC.
- Základní myšlenka: V procesorech CISC používali složité strojové instrukce (CPI >> 1) pouze špičkoví programátoři, ale standardní rutiny kompilátoru je nepoužívaly.
- Výhodnější by bylo implementovat pouze jednoduché, ale rychlé instrukce
  - Dojde ke zrychlení zpracování instrukcí a úspoře plochy na čipu.
  - Chybějící složité instrukce jsou nahrazeny podprogramy sestavenými z jednoduchých instrukcí.
- Cílem je dosáhnout parametru instrukčního souboru CPI =1.
- To lze zajistit pouze trikem: Překrýváním cyklů F, D a E.
  - Pozn.: U jednoduchého procesoru má instrukce CPI minimálně 3.

- Pozn: CPI – Cycles Per Instruction (počet cyklů nutných pro vykonání instrukce). Uvádí se u každé instrukce a také průměrná hodnota pro celý instrukční soubor.

Cykly procesoru

- von Neumannův procesor 1. generace

```
  F D E F D E F D E …
  I1     I2     I3     CPI = 3          ————————→ čas
```

- Řetězený procesor, základní, teoretický typ, ideální případ bez prostojů

```
F D E               Instr. 1
  F D E             Instr. 2
    F D E           Instr. 3
      F D E         Instr. 4
        F D E       Instr. 5         CPI = 1
      …
```

- Řetězený procesor – reálná verze, ideální případ bez prostojů

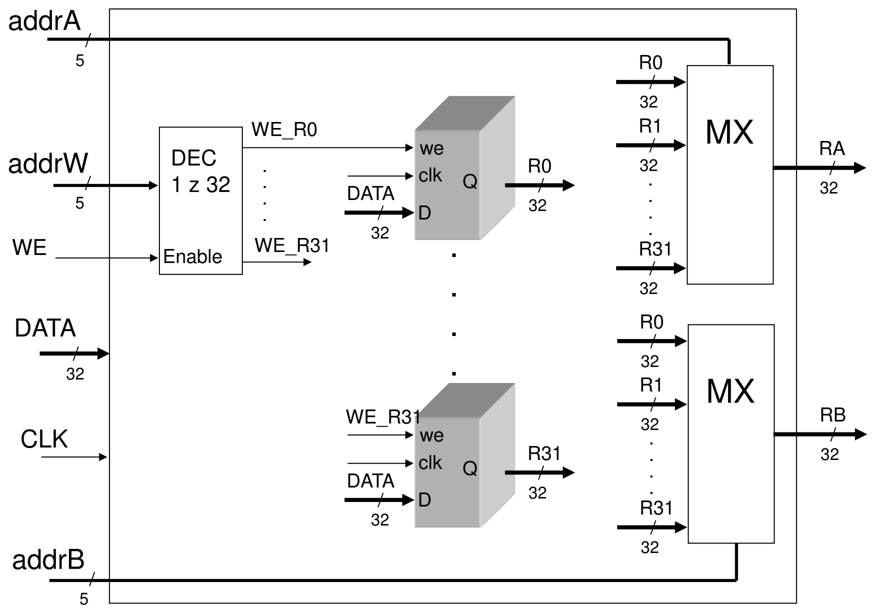
```
F D E M W           Instr. 1
  F D E M W         Instr. 2
    F D E M W       Instr. 3
      F D E M W     Instr. 4
        F D E M W   Instr. 5         CPI ≥ 1  ?
      …
```

M – memory access; W – write back

Obecné rysy architektur RISC

- Snaha o CPI = 1 (v ideálním případě)
- Všechny instrukce mají stejnou složitost
- Přístup k paměti mají pouze dvě instrukce, LOAD a STORE, ostatní pracují s registry.
- Registrová architektura (sada registrů)
- Rychlý obvodový řadič
- Oddělená paměť (cache) instrukcí a dat

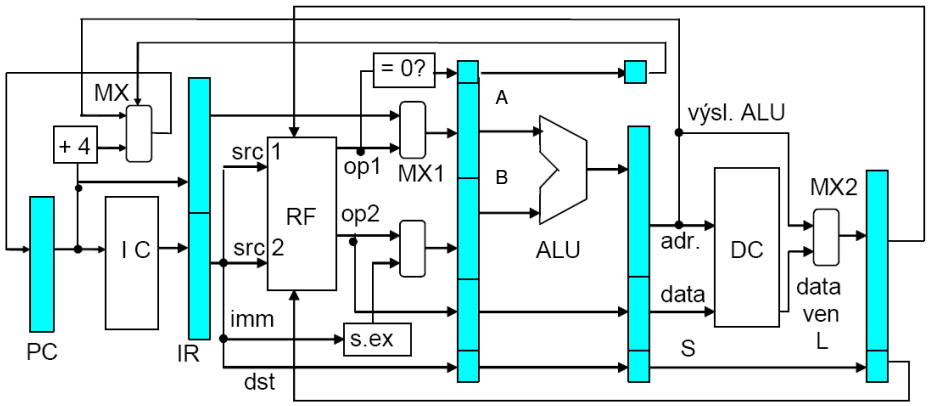
Opakování: Sada registrů (Register File) Př. 32 x 32b registr



Popis cyklů reálného řetězeného procesoru

F – instruction fetch: IR ← M [PC], NPC ← PC + 4 (načtení instrukce, zvýšení PC)
D – instruction decode + register fetch:
A ← Ri, B ← Rj (výběr operandů ze sady registrů)
Imm ← IRadr (extrakce adresy z IR)
E – provedení + cyklus výpočtu efektivní adresy
MRef. : ALUoutput ← A + Imm
I R-R : ALUoutput ← A op B
…
I Branch : ALUoutput ← PC + Imm (výpočet nové adresy)
Cond ← nastavení příznaku
M – přístup k paměti + cyklus dokončení skoku
MRef. : RegDat ← M [ALUoutput], nebo
M [ALUoutput] ← B
Branch : if (cond) PC ← ALUoutput else PC ← NPC
W – uložení výsledků
I R-R : Regs [IRadr] ← ALUoutput
I R–Imm: Regs [IRadr] ← ALUoutput
I Load: Regs [IRadr] ← Reg Dat

DLX procesor – výukový RISC procesor

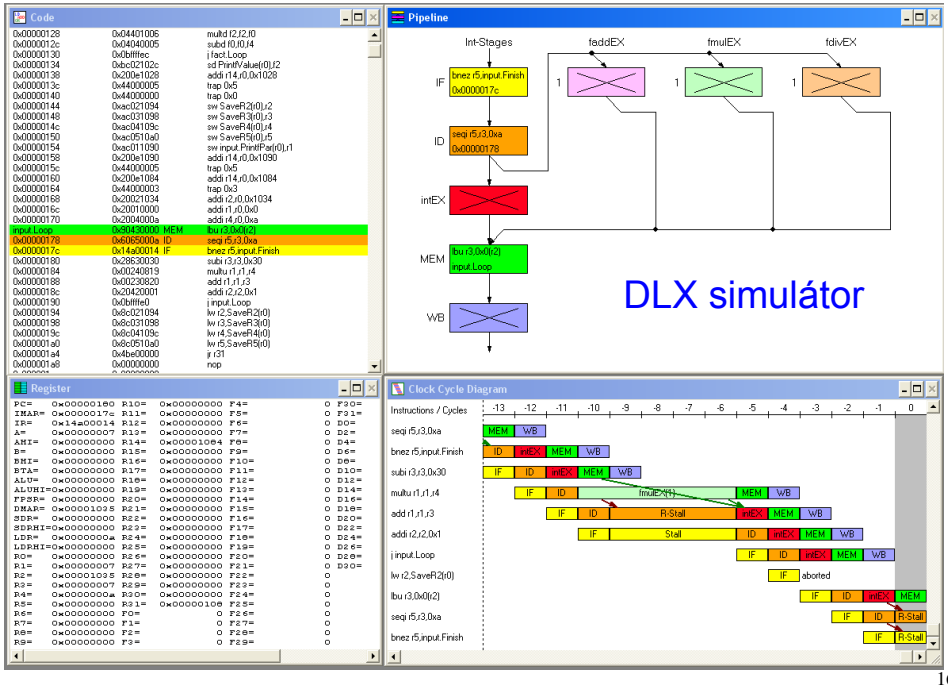


MX: multiplexor, DC: cache dat, = 0? : detektor nuly (Cond), IC: cache instrukcí, ALU: aritmeticko-logická jednotka, PC: čítač instrukcí, IR: registr instrukce, RF: soubor registrů (register file), s.ex: jednotka rozšíření znaménka (sign extension), dst: adresa cílového registru, src1 a src2: adresy zdrojových registrů, imm: přímý operand, L/S: load/store

Paměť (M) je rozdělena na paměť instrukcí (IC) a paměť dat (DC).

DLX – poznámky

- V každém taktu je celý stav zpracování instrukce (kontext) obsažen úplně a výhradně v obvodech jen jednoho stupně.
- Proto se např. index cílového registru dst (5 bitů) kopíruje z jednoho stupně do druhého, než se nakonec použije jako adresa výsledku v souboru registrů. Např. proto se také kopíruje obsah registru (op2) určený k zápisu do paměti.
- Uvedená vlastnost linky znamená, že ostatní stupně jsou volné a dají se použít pro paralelní zpracování dalších nezávislých instrukcí.
- U podmíněného skoku se adresa získá buď inkrementací (+ 4 byty u 32-bitové instrukce) nebo se stávající adresa modifikuje přičtením hodnoty získané z instrukce (26 bitů) sčítačkou v ALU.
- Přímý operand instrukce (imm) má šířku 16 bitů a v jednotce rozšíření znaménka (s.ex) se z něj vytvoří 32-bitový operand.
- Při čtení nebo zápisu do paměti D-cache může být adresa v jednom z registrů a může se k ní přičíst odstup (imm) uvedený v instrukci (16 bitů). Výpočet adresy se provádí v ALU.



Konflikty (hazardy) u řetězeného zpracování v procesorech, které mohou vést ke zpomalení linky

1. Strukturální – obvodová struktura neumožňuje současné provedení určitých akcí – např. současné čtení dvou hodnot z paměti nebo současné provedení dvou sčítání, pokud má procesor jednu ALU
2. Datové – když jsou zapotřebí data z předcházející instrukce, která není dokončena.
3. Řídicí – když skoková instrukce mění obsah PC, nebo jiné.

ad 1) F D E M W
F D E M W
F D E M W
F D E M W
požadavek na současné čtení instrukce a čtení operandu z paměti
Řešení: rozdělit paměť na paměť instrukcí a paměť dat
Obecné řešení: přidání výpočetních jednotek

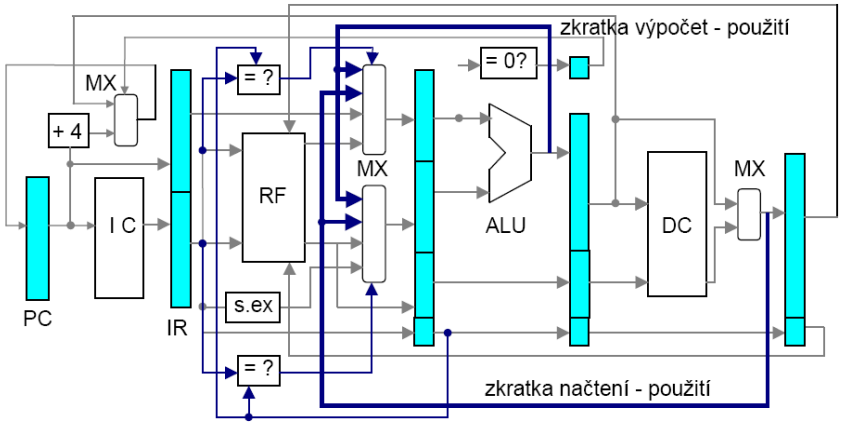
Datové konflikty

	1	2	3	4	5	6	7	8	9
ADD **R1**, R2, R3	IF	ID	EX	MEM	**WB**				
SUB R4, R5, **R1**		IF	**ID$_{sub}$**	EX	MEM	WB			
AND R6, **R1,** R7			IF	**ID$_{and}$**	EX	MEM	WB		
OR R8, **R1**, R9				IF	**ID$_{or}$**	EX	MEM	WB	
XOR R10,**R1**,R11					IF	**ID$_{xor}$**	EX	MEM	WB

Výsledek instrukce ADD bude k dispozici v 5. taktu.
Instrukce SUB, AND a OR tak provedou chybný výpočet.
Pozn: ADD R1, R2, R3 provede R1 = R2 + R3

Forwarding (bypassing)

Poskytnutí mezivýsledku dřív než bude zapsán do registru. Je to umožněno přidáním speciálních datových cest a rozšířením multiplexorů – viz příklady na obrázku. Pomáhá řešit datové konflikty.



Kdy Forwarding nefunguje?

		1	2	3	4	5	6	7	8
LW	**R1**, a	IF	ID	EX	**MEM**	**WB**			
SUB	R4, **R1**, R5		IF	ID	**EX$_{sub}$**	MEM	WB		
AND	R6, **R1** R7			IF	ID	**EX$_{and}$**	MEM	WB	
OR	R8, **R1**, R9				IF	**ID**	EX	MEM	WB

LW (load word) má data až na konci fáze MEM, ale SUB je potřebuje na začátku EX.

Je nutné přidat obvod (pipeline interlock), který hazard detekuje a pozastaví linku (fáze *stall*) , aby nebyla porušena logika programu. Někdy stačí, když překladač „přeskládá“ instrukce.

		1	2	3	4	5	6	7	8	9
LW	**R1**, a	IF	ID	EX	**MEM**	WB				
SUB	R4, **R1**, R5		IF	ID	***stall***	**EX$_{sub}$**	MEM	WB		
AND	R6, **R1** R7			IF	***stall***	ID	EX	MEM	WB	
OR	R8, **R1**, R9				***stall***	IF	ID	EX	MEM	WB

Klasifikace datových konfliktů

Mějme instrukce A a B, a platí, že A předchází B.

RAW – Read After Write
B se pokouší číst zdroj před tím, než A do něj zapsala. B přečte starou hodnotu. Řešení: forwarding.

WAW – Write After Write
B se pokouší zapsat operand dřív, než je proveden zápis instrukcí A. Zápis je tak proveden v chybném pořadí. V DLX nenastane, protože je povoleno zapisovat jen ve fázi WB.

WAR – Write After Read
B se pokouší zapsat operand dřív než je přečten A. A tak získá novější hodnotu, což je chyba. Nenastane v DLX.

RAR – není hazard :-)

Řídicí konflikty

ad 3)

BC	F	D	E	M	W						// BC…podmíněný skok
BC+1		F	-	-	F	D	E	M	W		
BC+2						F	D	E	M	W	
BC+3							F	D	E	M	W

atd. linka pozastavena na 3 takty, pokud se má skočit.

Nevím, kam skočit! Adresa bude známa až po fázi M na řádku BC.
Je třeba co nejrychleji zjistit adresu následující instrukce.

Řešení:
1. Zpožděný skok, tedy vložit jiné, užitečné instrukce, což znamená přeskládat instrukce programu (kompilátor).
2. Nelze-li takové užitečné instrukce najít, vložit NOPy.
3. Zavést specializované obvody (prediktor skoku a paměť cílové adresy skoku - BTB), které odhadnou, zda provést/neprovést skok a adresu skoku.

Zpožděný skok

Původní kód:

```
        JC L1
        XOR R5, R5, R3
        JMP l2
L1:     AND R5, R5, R3
L2:     LD R6, adr1
        LD R7, adr2
        MOV R8, R7
        ST R1
```

Optimalizovaný kód:

```
        JC L1
        LD R6, adr1
        LD R7, adr2
        MOV R8, R7
        XOR R5, R5, R3
        JMP l2
L1:     AND R5, R5, R3
L2:     ST R1
```

Jaká bude následující adresa po JC?
Tři vyznačené instrukce nejsou datově závislé na okolním kódu a musí se vždy vykonat. Je tedy možné je přesunout a tím získat čas na zjištění následující adresy po JC. Nedojde k pozastavení linky.

Řídicí konflikty a prediktory skoku

Experimentálně bylo zjištěno, že četnost skoků v programech je značná, kolem 20% u univerzálních programů a 5-10% u vědecko-technických výpočtů. Celkově se zhruba 5/6 (83%) skoků provede.

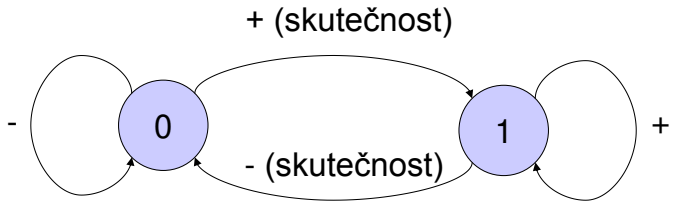
U všech skoků je třeba zrychlit zjištění cílové adresy – k tomu se používá malá paměť cache pro uložení cílových adres (BTB – Branch Target Buffer), která se postupně naplňuje a aktualizuje. Pro nepodmíněné skoky a pro správně predikované podmíněné skoky se zřetězená linka nepozastaví.

U podmíněných skoků jde ještě navíc o vyhodnocení podmínky. O splnění podmínky je možné spekulovat - jde o tzv. predikci skoků, která může být statická (kompilátor nastaví bit predikce) nebo dynamická pomocí speciálního HW.

Predikuje se podle předchozích směrů skoku. Historii můžeme popsat binárním řetězcem několika bitů, které představují stavy predikčního sekvenčního obvodu (obvykle + značí skok a – značí neskok).

Nejjednodušší prediktor: Jednobitový prediktor

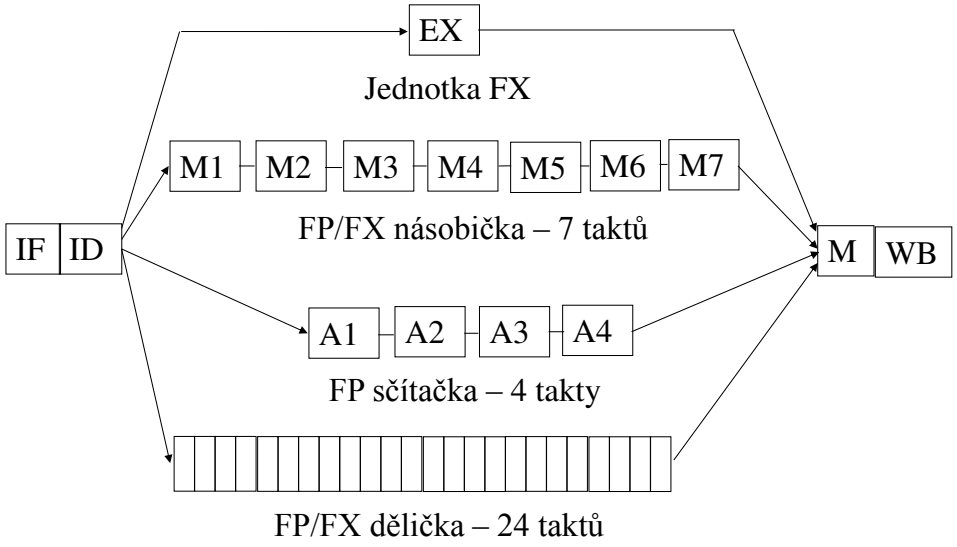
Používá jen 1 bit historie, tj. zda se skok v předchozím průchodu provedl či ne, a předpovídá stejné chování v příštím průchodu.



stav 0 = skok minule neproveden, předpověď je neskoč (-)
stav 1 = skok minule proveden, předpověď je skoč (+)
Pokud je předpověď chybná, přejdi do druhého stavu.

Při chybné predikci musí být nesprávně prováděná větev programu ukončena a musí začít načítání instrukcí ze správné větve. Je třeba aktualizovat BTB. Stojí to několik taktů řetězené linky.

Rozšíření pipeline: FX a FP operace



Příklad a potenciální problém

Mějme tento úsek programu: (.D znamená dvojnásobnou přesnost)

```
DIV.D    F0, F2, F4
ADD.D    F10, F10, F8
SUB.D    F12, F12, F14
```

Na první pohled zde nejsou problémy, protože mezi instrukcemi nejsou datové závislosti. Ovšem podle předcházející řetězené struktury budou instrukce ADD a SUB dokončeny dříve, než dříve zahájená instrukce DIV (out-of-order completion).
Pokud však nastane v době dokončování DIV výjimka, instrukce ADD a SUB se již nesmějí provést - tak jak by se stalo *na klasické neřetězené výpočetní struktuře*.
Řešení: nutno zavést koncept precizního přerušení (viz pokročilý kurz o procesorech)

Poznámky k zřetězenému zpracování

- Zřetězené zpracování přináší urychlení výpočtu nejen v procesorech, ale i v jiných číslicových obvodech (např. pro zpracování obrazu, bioinformatických dat apod.).
- Pokud použijeme zřetězené zpracování v procesoru, musíme dodat řadu podpůrných obvodů a řešit řadu nových problémů.
- Podrobnější analýza problémů zřetězeného zpracování v procesorech bude provedena v magisterském kurzu Architektura procesorů.
- Kromě řetězení se používají i další koncepty, které vedly ke vzniku nových kategorií procesorů:
  - superskalární architektury
  - VLIW procesory
  - vektorové procesory
  - multivláknové procesory
  - atd.

Literatura

- Drábek, V. Výstavba počítačů. Skriptum VUT, 1995
- Dvořák, V., Drábek, V.: Architektura procesorů. Studijní opora. FIT VUT v Brně 2006
- Win DLX simulator
  - http://electro.fisica.unlp.edu.ar/arq/downloads/Software/WinDLX/windlx.html
- Prabhu G. M.: Computer architecture tutorial
  - http://www.cs.iastate.edu/~prabhu/Tutorial/title.html