

Řetěžené zpracování

INP 2019
FIT VUT v Brně



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

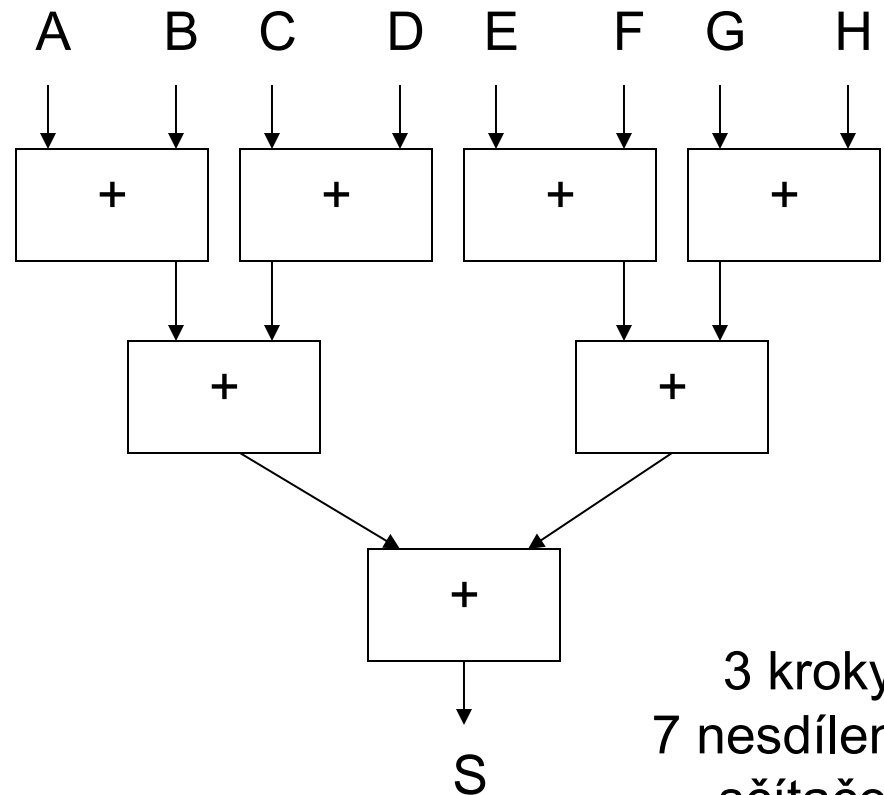
Paralelní zpracování

$$\text{Př. } S = A + B + C + D + E + F + G + H$$

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'

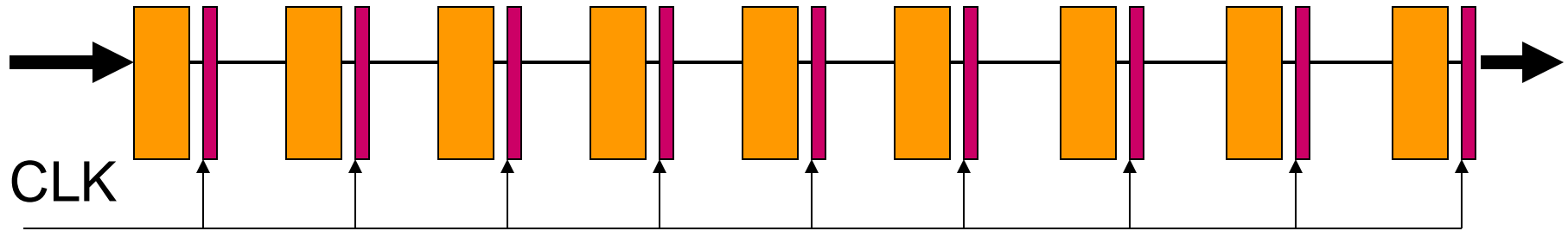
Příklad: 65536 pixelů

Předpokládejme zpoždění obvodu: 90 ns

Celkem: $65536 \times 90 \text{ ns} = 5898240 \text{ ns} = 5,9 \text{ ms}$

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 zpracovááno
současně \Rightarrow zrychlení 9x
oproti předchozímu řešení

Zřetězený filtr – 9 stupňů

- Příklad: 65536 pixelů
- Předpokládejme zpoždění stupně: 10 ns
- 1. pixel bude zpracován za $9 \times 10 = 90$ ns
- 2., 3. až 65536. pixel – každý za 10 ns
- Celkem: $90 \text{ ns} + 65535 \times 10 \text{ ns} = 655440 \text{ ns} = 0.655440 \text{ ms}$
- **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:
$$\frac{N \cdot k \cdot t}{(k + N - 1) \cdot (t + d)}$$

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 \Rightarrow další urychlení

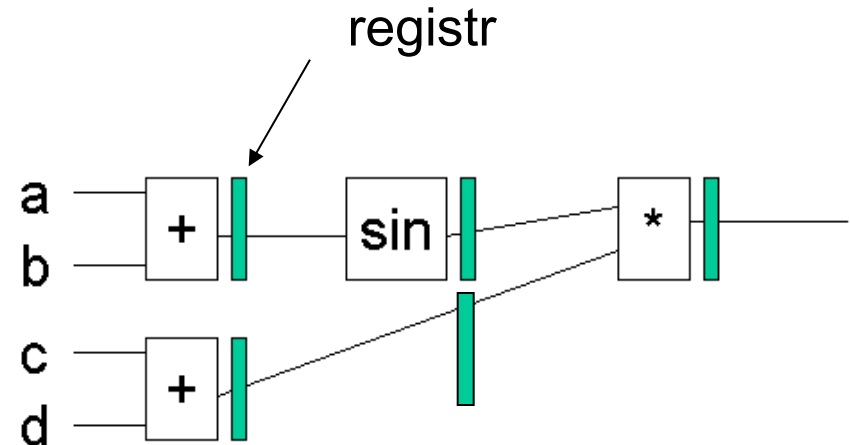
SW:

Vypočtete

```
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: $100 \times 4 = 400$ kroků
(a to neuvažujeme test podmínky
v cyklu)



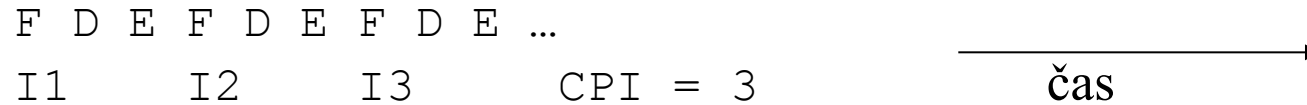
První výsledek za 3 kroky.
2. až 100. výsledek za 1 krok.
Celkem 102 kroků
Zrychlení **3,92 krát**.

Řetězené zpracování instrukcí v procesorech

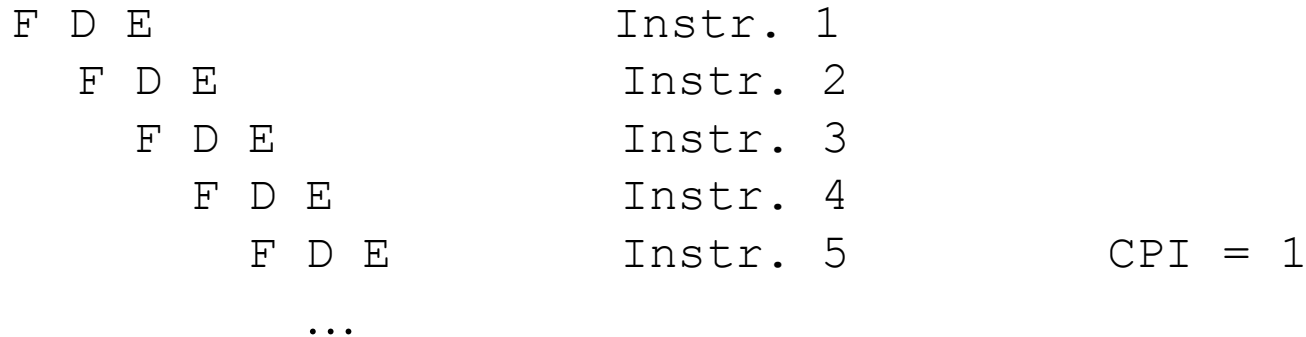
- Princip zřetězení se značně překrývá s principy procesorů typu RISC.
- Základní myšlenka: V procesorech CISC používali složité strojové instrukce ($CPI \gg 1$) pouze špičkoví programátoři, ale standardní rutiny kompilátoru je nepoužívaly.
- Výhodnější by bylo implementovat pouze jednoduché, ale rychlé 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

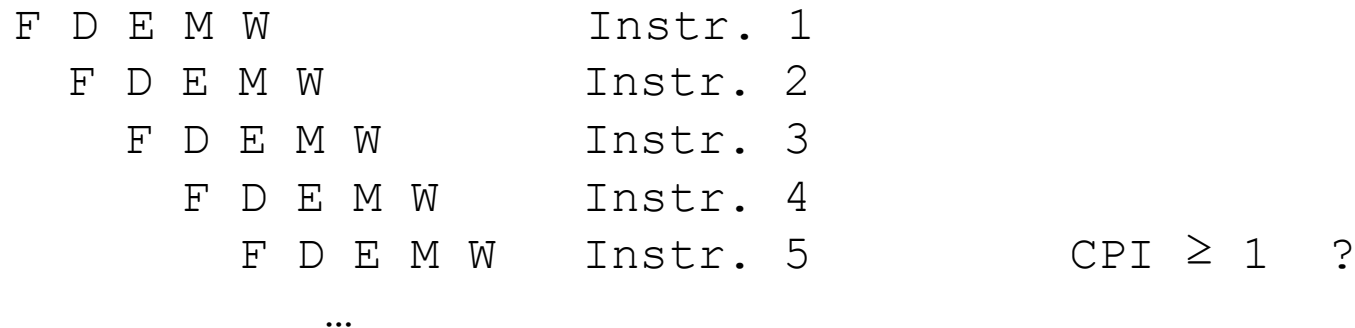
- von Neumannův procesor 1. generace



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



- Řetězený procesor – reálná verze, ideální případ bez prostojů



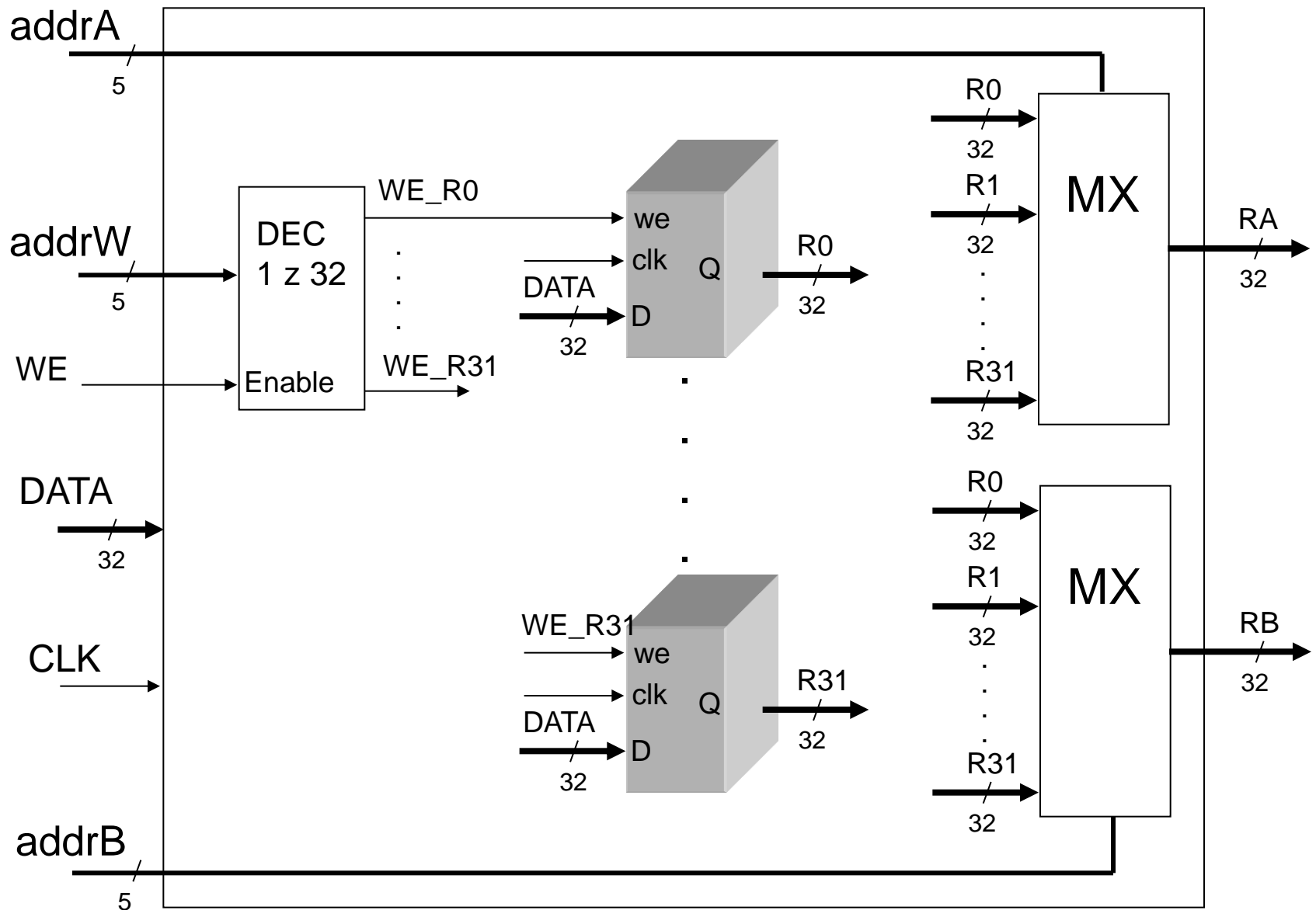
M – memory access; W – write back

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**: $IR \leftarrow M[PC]$, $NPC \leftarrow PC + 4$ (načtení instrukce, zvýšení PC)

D – **instruction decode + register fetch**:

$A \leftarrow Ri$, $B \leftarrow Rj$ (výběr operandů ze sady registrů)

$Imm \leftarrow IR_{adr}$ (extrakce adresy z IR)

E – **provedení + cyklus výpočtu efektivní adresy**

MRef. : $ALU_{output} \leftarrow A + Imm$

I R-R : $ALU_{output} \leftarrow A \text{ op } B$

...

I Branch : $ALU_{output} \leftarrow PC + Imm$ (výpočet nové adresy)

Cond \leftarrow nastavení příznaku

M – **přístup k paměti + cyklus dokončení skoku**

MRef. : $RegDat \leftarrow M[ALU_{output}]$, nebo

$M[ALU_{output}] \leftarrow B$

Branch : if (cond) $PC \leftarrow ALU_{output}$ else $PC \leftarrow NPC$

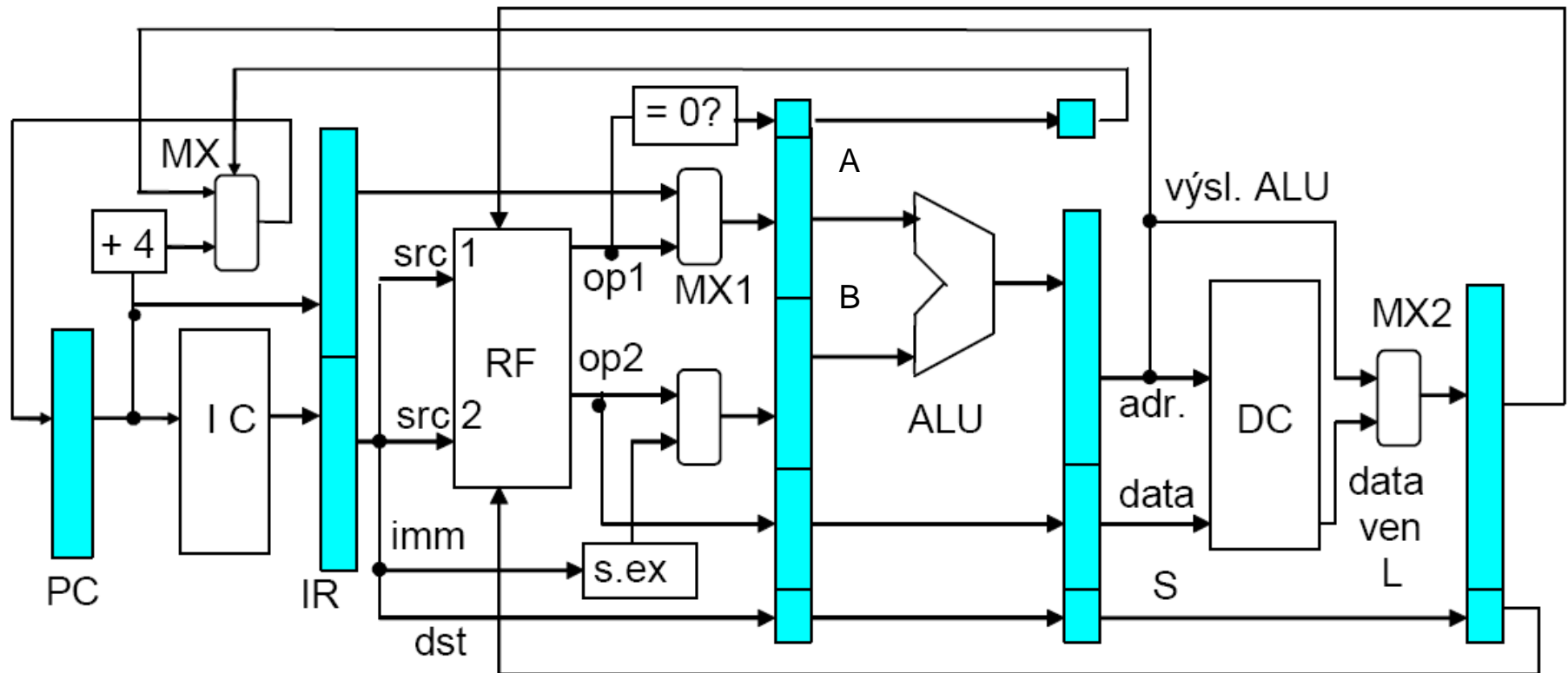
W – **uložení výsledků**

I R-R : $Regs[IR_{adr}] \leftarrow ALU_{output}$

I R-Imm: $Regs[IR_{adr}] \leftarrow ALU_{output}$

I Load: $Regs[IR_{adr}] \leftarrow RegDat$

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).

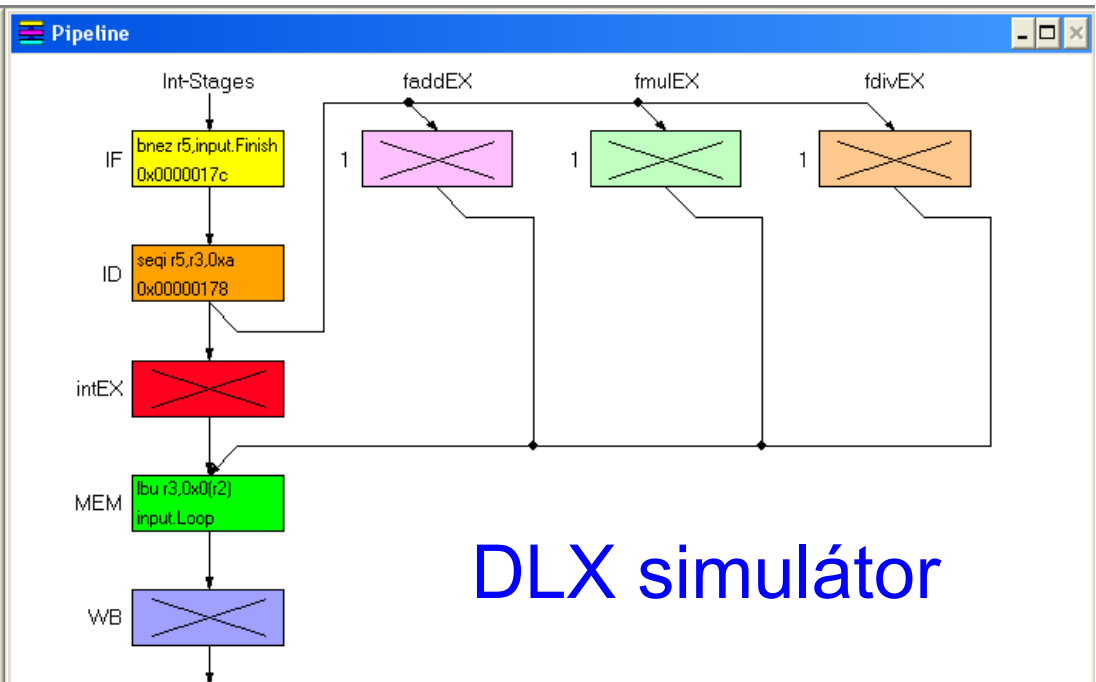
DLX – poznámky

- 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.

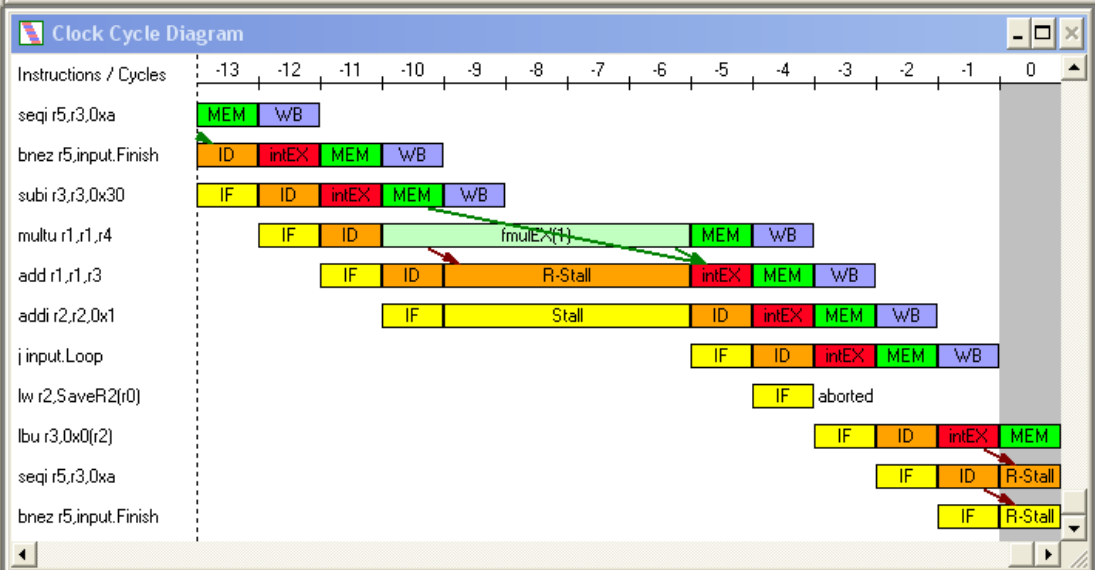
```

Code
0x00000128      0x04401006      multd f2,f2,f0
0x0000012c      0x04040005      subd f0,f0,f4
0x00000130      0x0bffffec      j fact.Loop
0x00000134      0x0c02102c      sd PrintValue(r0),f2
0x00000138      0x200e1028      addi r14,r0,0x1028
0x0000013c      0x44000005      trap 0x5
0x00000140      0x44000000      trap 0x0
0x00000144      0xac021094      sw SaveR2(r0),r2
0x00000148      0xac031098      sw SaveR3(r0),r3
0x0000014c      0xac04109c      sw SaveR4(r0),r4
0x00000150      0xac0510a0      sw SaveR5(r0),r5
0x00000154      0xac011090      sw input.PrintPar(r0),r1
0x00000158      0x200e1090      addi r14,r0,0x1090
0x0000015c      0x44000005      trap 0x5
0x00000160      0x200e1084      addi r14,r0,0x1084
0x00000164      0x44000003      trap 0x3
0x00000168      0x20021034      addi r2,r0,0x1034
0x0000016c      0x20011000      addi r1,r0,0x0
0x00000170      0x2004000a      addi r4,r0,0xa
input.Loop      0x90430000 MEM      lbu r3,0x0(r2)
0x00000178      0x6065000a ID      seqi r5,r3,0xa
0x0000017c      0x14a00014 IF      bnez r5,input.Finish
0x00000180      0x28630030      subi r3,r3,0x30
0x00000184      0x00240819      multu r1,r1,r4
0x00000188      0x00230820      add r1,r1,r3
0x0000018c      0x20420001      addi r2,r2,0x1
0x00000190      0x0bffffe0      j input.Loop
0x00000194      0x8c021094      lw r2,SaveR2(r0)
0x00000198      0x8c031098      lw r3,SaveR3(r0)
0x0000019c      0x8c04109c      lw r4,SaveR4(r0)
0x000001a0      0x8c0510a0      lw r5,SaveR5(r0)
0x000001a4      0x4be00000      jr r31
0x000001a8      0x00000000      nop

```



Register			
PC=	0x00000180	R10=	0x00000000
IMAR=	0x0000017c	R11=	0x00000000
IR=	0x14a00014	R12=	0x00000000
A=	0x00000007	R13=	0x00000000
AHI=	0x00000000	R14=	0x00001084
B=	0x00000000	R15=	0x00000000
BHI=	0x00000000	R16=	0x00000000
BTA=	0x00000000	R17=	0x00000000
ALU=	0x00000000	R18=	0x00000000
ALUHI=	0x00000000	R19=	0x00000000
FP3R=	0x00000000	R20=	0x00000000
DMAR=	0x00001035	R21=	0x00000000
SDR=	0x00000000	R22=	0x00000000
SDRHI=	0x00000000	R23=	0x00000000
LDR=	0x0000000a	R24=	0x00000000
LDRHI=	0x00000000	R25=	0x00000000
RO=	0x00000000	R26=	0x00000000
R1=	0x00000007	R27=	0x00000000
R2=	0x00001035	R28=	0x00000000
R3=	0x00000007	R29=	0x00000000
R4=	0x0000000a	R30=	0x00000000
R5=	0x00000000	R31=	0x00000108
R6=	0x00000000	F0=	0
R7=	0x00000000	F1=	0
R8=	0x00000000	F2=	0
R9=	0x00000000	F3=	0
		F4=	0
		F5=	0
		F6=	0
		F7=	0
		F8=	0
		F9=	0
		F10=	0
		F11=	0
		F12=	0
		F13=	0
		F14=	0
		F15=	0
		F16=	0
		F17=	0
		F18=	0
		F19=	0
		F20=	0
		F21=	0
		F22=	0
		F23=	0
		F24=	0
		F25=	0
		F26=	0
		F27=	0
		F28=	0
		F29=	0



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

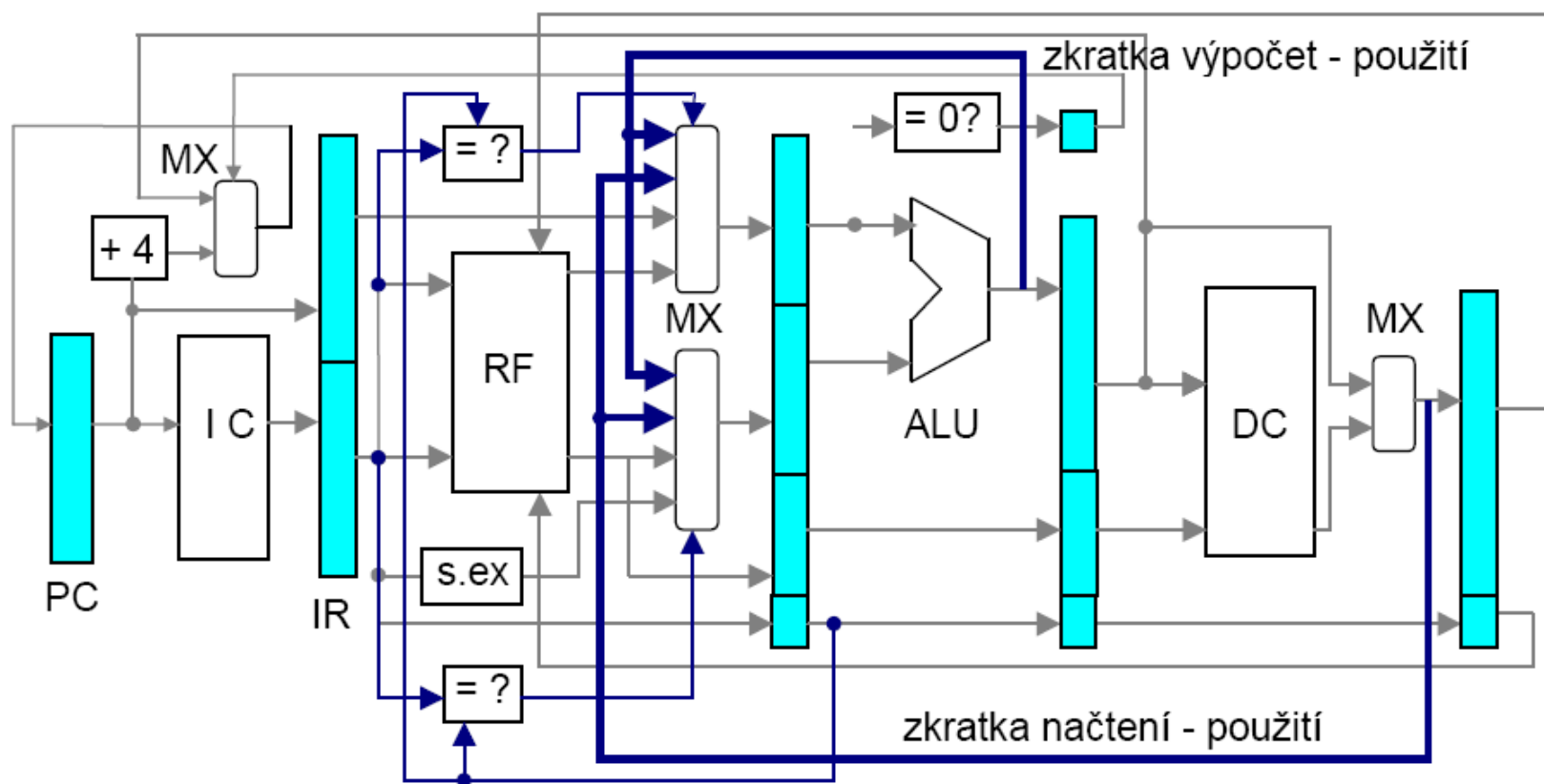
Datové konflikty

	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	<i>stall</i>	EX _{sub}	MEM	WB		
AND	R6, R1 R7			IF	<i>stall</i>	ID	EX	MEM	WB	
OR	R8, R1, R9				<i>stall</i>	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 řádce 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 12
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 12
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.

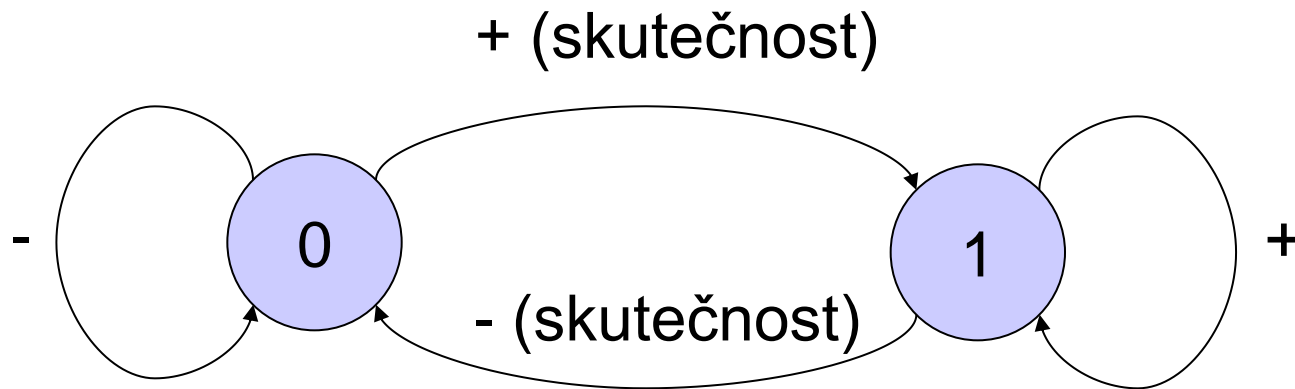
U všech skoků je třeba zrychlit zjištění **cílové adresy** – k tomu se používá malá paměť cache pro uložení cílových adres (**BTB** – Branch Target Buffer), která se postupně naplňuje a aktualizuje. Pro nepodmíněné skoky a pro správně predikované podmíněné skoky se zřetězená linka nepozastaví.

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: Jednabitový prediktor

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



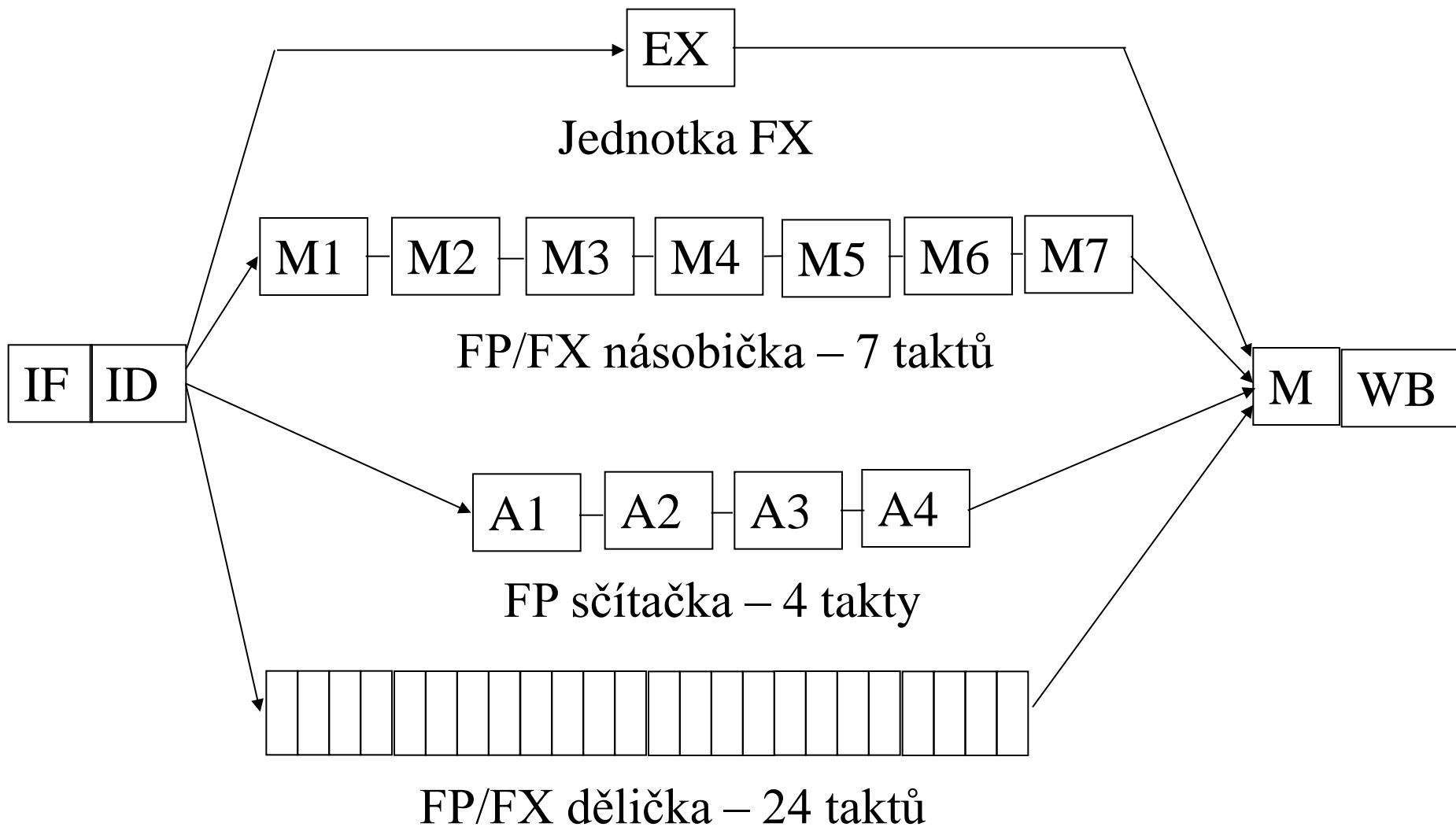
stav 0 = skok minule neproveden, předpověď je neskoč (-)

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 zřetěženému zpracování

- Zřetěžené zpracování přináší urychlení výpočtu nejen v procesorech, ale i v jiných číslicových obvodech (např. pro zpracování obrazu, bioinformatických dat apod.).
- Pokud použijeme zřetěžené zpracování v procesoru, musíme dodat řadu podpůrných obvodů a řešit řadu nových problémů.
- Podrobnější analýza problémů zřetěženého 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>