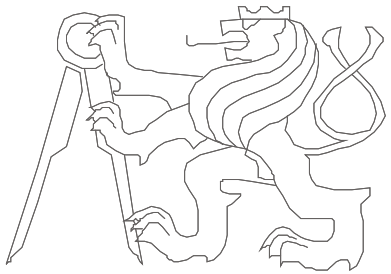
Architektura počítačů

Procesor

V přednášce byly použity (se souhlasem vydavatelství) obrázky z knihy Paterson, D., Henessy, V.: Computer Organization and Design, The HW/SW Interface. Elsevier, ISBN: 978-0-12-370606-5



České vysoké učení technické, Fakulta elektrotechnická

Osnova přednášky

1. Aritmetické operace, základní HW pro realizaci aritmetických a logických operací

2. Návrh jednoduchého procesoru

3. Řadič procesoru, jeho funkce a možnosti realizace

Aritmetické operace – znaménková čísla

Přímý kód

- Zvlášť pracujeme se znaménkem a zvlášť s hodnotou
- Algoritmus pro + a -

-3		1	0	0	1	1
-3	+	1	0	0	1	1
-6		1	0	1	1	0

-4		1	0	1	0	0
+5	+	0	0	1	0	1
?		?	?	?	?	?

Inverzní kód

- Vystačíme si s jedním algoritmem
- nutnost provádět tzv. kruhový přenos = přičtení přenosu z nejvyššího řádu k výsledku

-4		1	1	0	1	1
+5	+	0	0	1	0	1
?		**1**	0	0	0	0
					+	1
+1		0	0	0	0	1

Dvojkový doplněk

- Jeden algoritmus
- Žádný kruhový přenos 

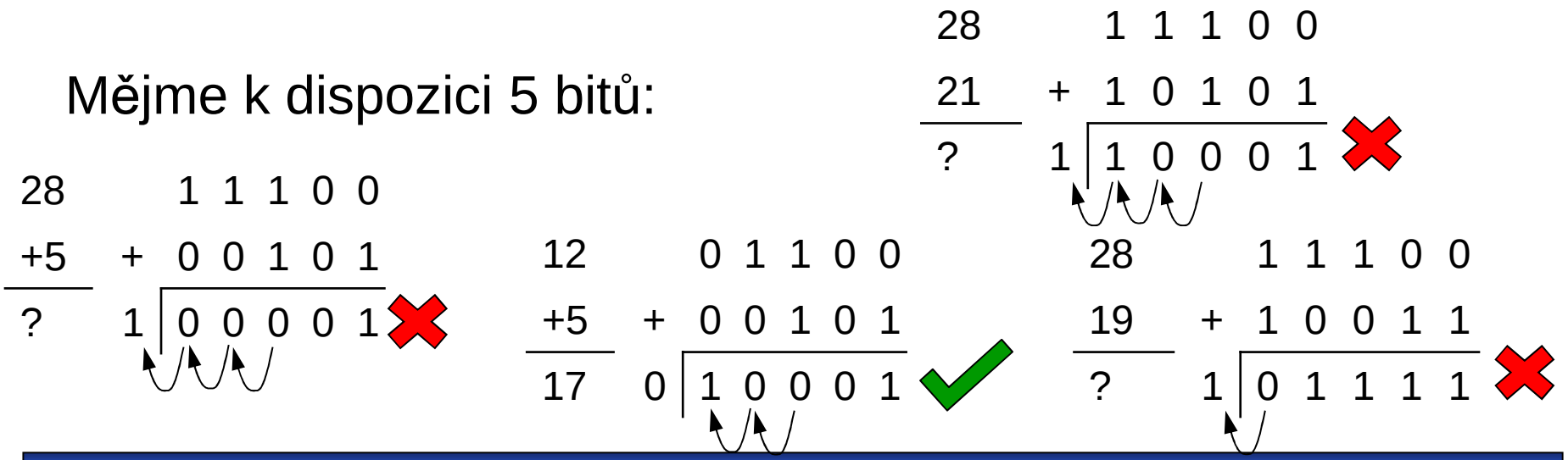
-4		1	1	1	0	0
+5	+	0	0	1	0	1
1	1	0	0	0	0	1

není součástí výsledku

A co čísla **neznaménková**? Na ty nezapomínejme…

- V tomto případě sledujeme **přenos z nejvyššího řádu**
- Opět situace, kdy výsledek operace není správný, protože se nevešel do zobrazitelného rozsahu.

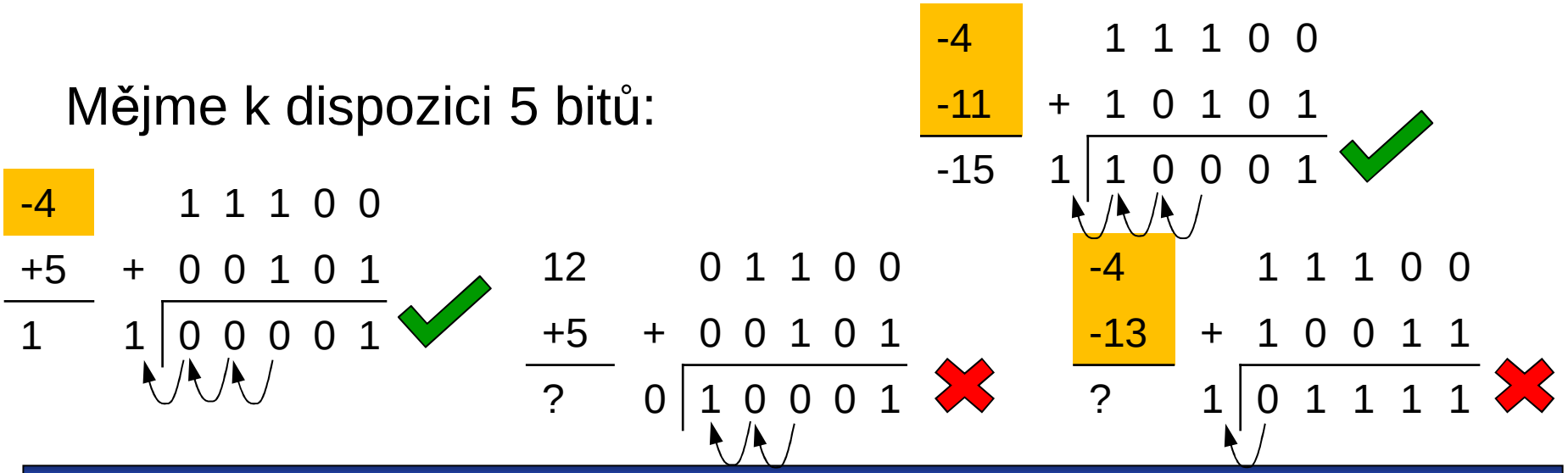
Mějme k dispozici 5 bitů:



Je výsledek správný? Rozumíte pojmu přeplnění?

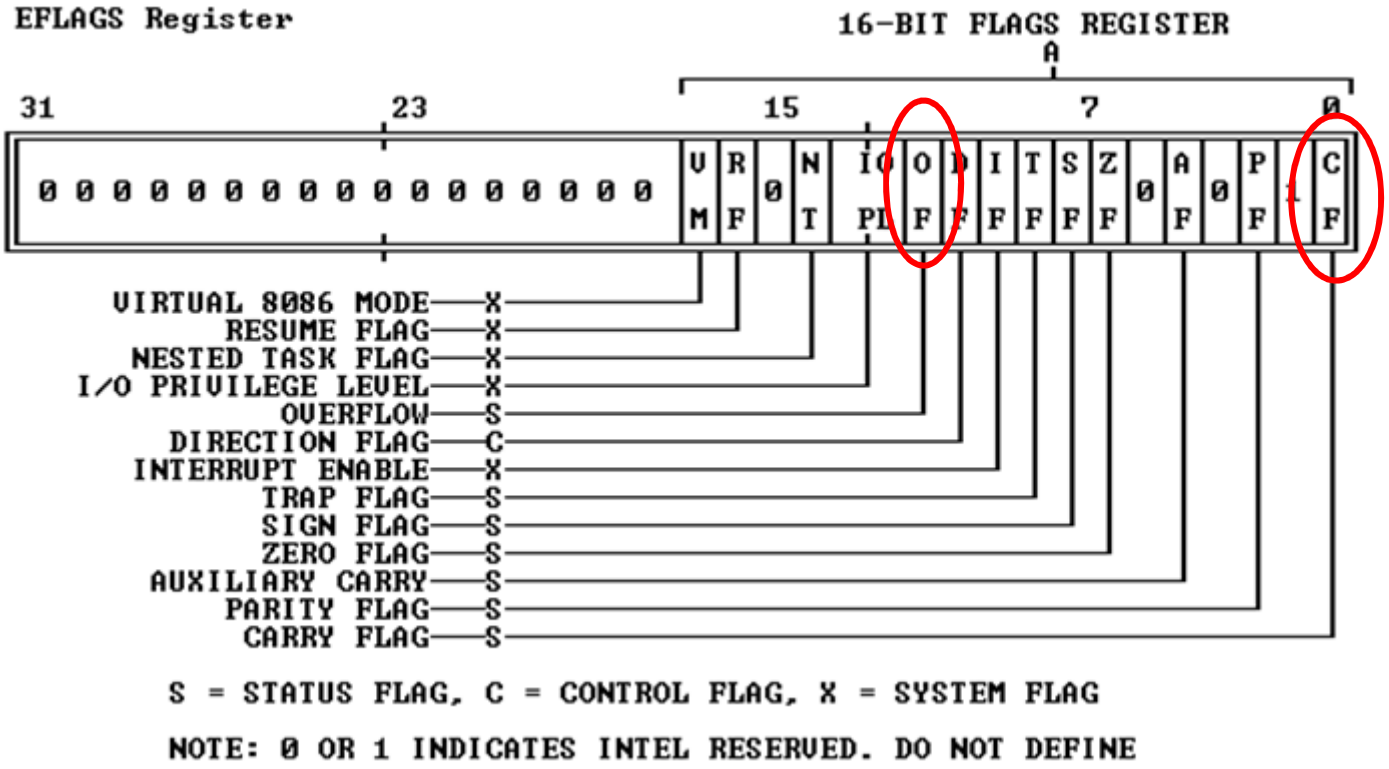
- Přeplnění - říká se tomu také **přetečení (overflow)**.
- Situace, kdy výsledek operace není správný, protože se nevešel do zobrazitelného rozsahu.
- **Nastává v situaci, kdy znaménko výsledku je jiné, než znaménka operandů, byla-li stejná**, nebo
- Nonekvivalencí přenosu **do** a **z** nejvyššího řádu.

Mějme k dispozici 5 bitů:



Jak nás procesor informuje o správnosti výsledku?

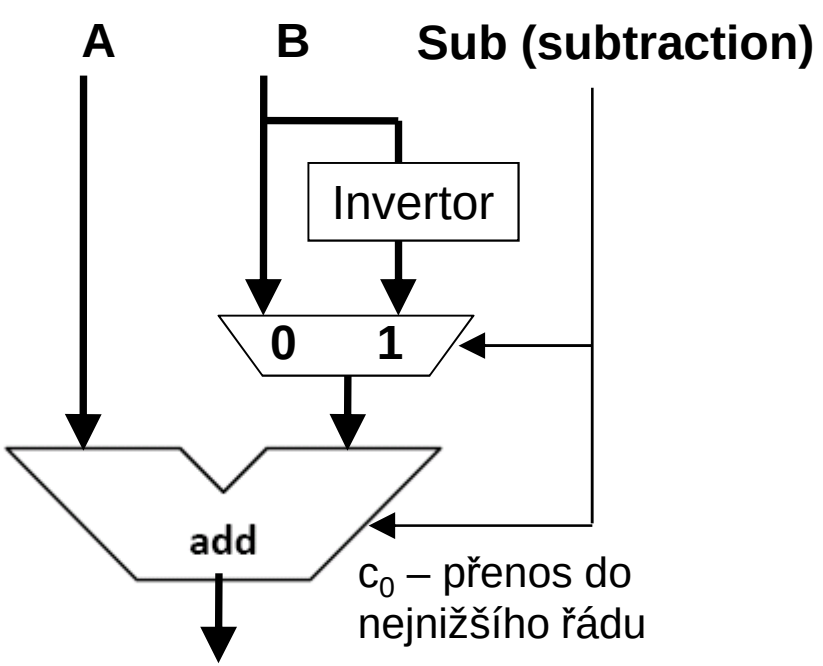
- Musíme se podívat do **příznakového registru** - FLAGS register pro Intel x86 mikroprocesory, CPSR (Current Program Status Register) pro ARM,..
- V něm najdeme příznaky **Carry** a **Overflow** (a další)

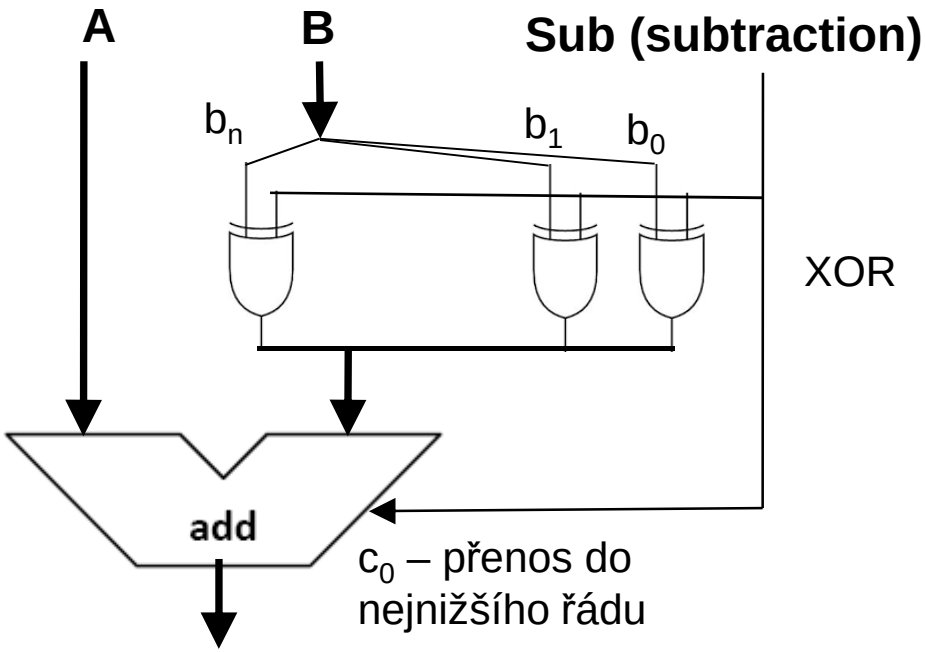


http://css.csail.mit.edu/6.858/2013/readings/i386/s02_03.htm

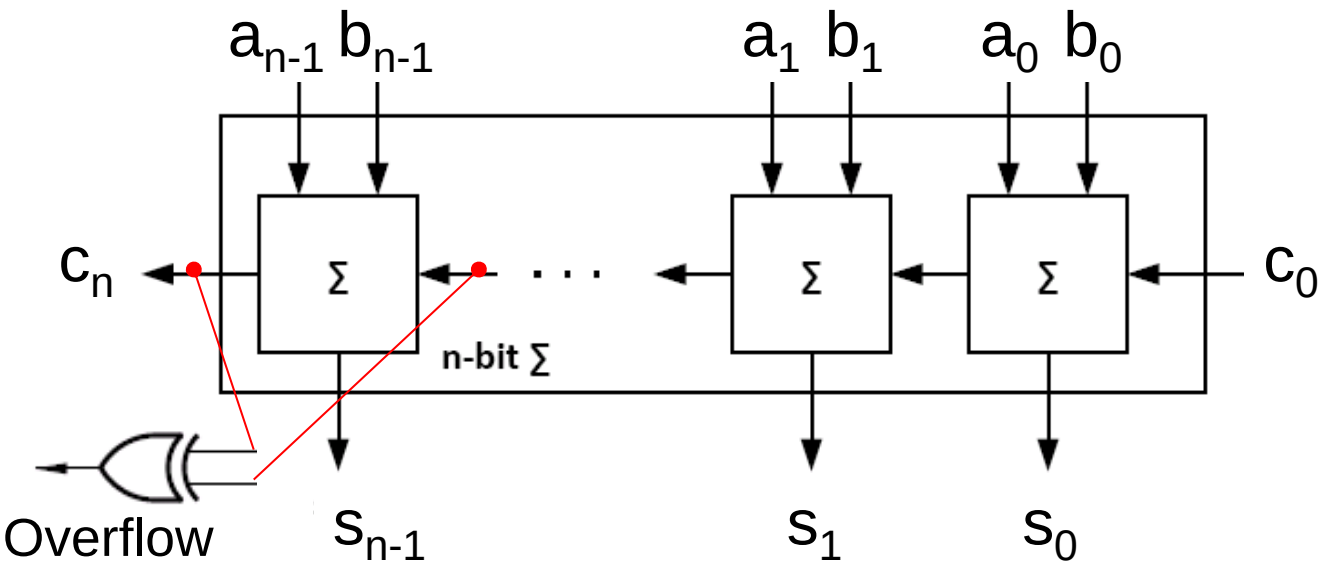
Jak realizujeme rozdíl dvou čísel?

- Na předchozích slidech jsme viděli jak se realizuje operace součtu dvou čísel – ať již znaménkových (kladné+kladné, kladné+záporné, záporné+záporné) nebo neznaménkových (kladné+kladné)
- Otázka k opakování: jak jsme vytvořili k číslu ve dvojkovém doplňku jeho číslo opačné?

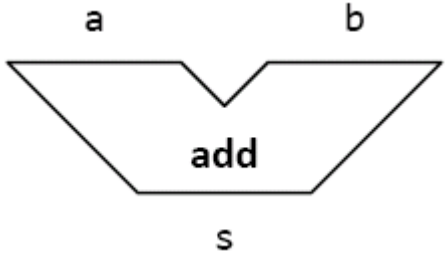




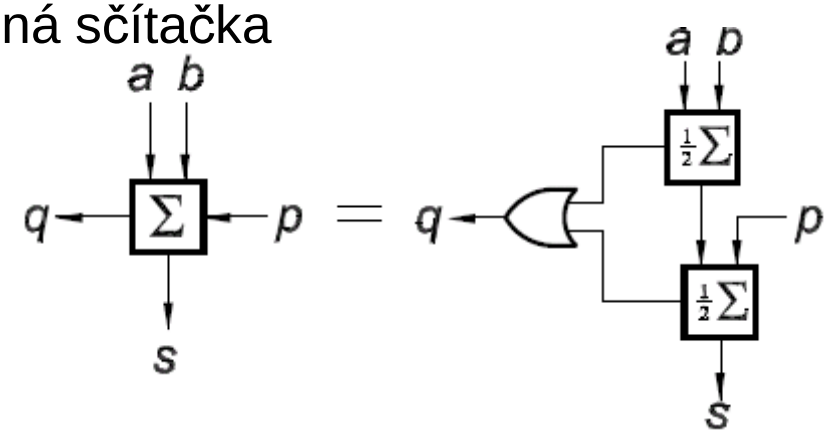
Sčítací HW blokově – Sčítačka s postupným šířením přenosu



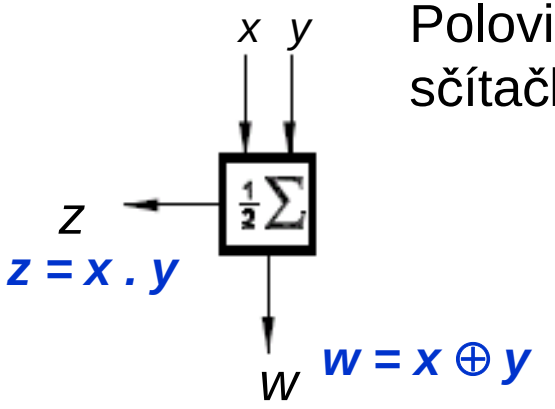
Obvyklý symbol pro funkční blok



Úplná sčítačka



kde



Poloviční sčítačka

Paralelní sčítačka s minimálním zpožděním

- **Sčítačka s postupným šířením přenosu je pro více bitová čísla velmi pomalá…** Pokud označíme Δt zpoždění jednoho hradla, pak generování přenosu v i-tém řádě trvá $2\Delta t$. Pro N bitové číslo generování součtu trvá $2N\Delta t$. **Pro 64-bitovou sčítačku to je 128 Δt.**
- Proto zkusíme navrhnout rychlejší

$$S_n = \overline{A_n} \cdot \overline{B_n} \cdot C_n + \overline{A_n} \cdot B_n \cdot \overline{C_n} + A_n \cdot \overline{B_n} \cdot \overline{C_n} + A_n \cdot B_n \cdot C_n$$

$$C_{n+1} = A_n \cdot B_n + B_n \cdot C_n + A_n \cdot C_n$$

A_n	B_n	C_n	C_{n+1}	S_n
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Paralelní sčítačka s minimálním zpožděním

$$S_0 = \overline{A_0} \cdot \overline{B_0} \cdot C_0 + \overline{A_0} \cdot B_0 \cdot \overline{C_0} + A_0 \cdot \overline{B_0} \cdot \overline{C_0} + A_0 \cdot B_0 \cdot C_0$$

$$C_1 = A_0 \cdot B_0 + B_0 \cdot C_0 + A_0 \cdot C_0$$

$$S_1 = \overline{A_1} \cdot \overline{B_1} \cdot C_1 + \overline{A_1} \cdot B_1 \cdot \overline{C_1} + A_1 \cdot \overline{B_1} \cdot \overline{C_1} + A_1 \cdot B_1 \cdot C_1 =$$

$$= \overline{A_1} \cdot \overline{B_1} \cdot \left(A_0 B_0 + B_0 C_0 + A_0 C_0\right) + \overline{A_1} \cdot B_1 \cdot \overline{\left(A_0 B_0 + B_0 C_0 + A_0 C_0\right)}$$

$$+ A_1 \cdot \overline{B_1} \cdot \overline{\left(A_0 B_0 + B_0 C_0 + A_0 C_0\right)} + A_1 \cdot B_1 \cdot \left(A_0 B_0 + B_0 C_0 + A_0 C_0\right)$$

$$= \ldots$$

$$C_2 = A_1 \cdot B_1 + B_1 \cdot C_1 + A_1 \cdot C_1 = \ldots$$

- Je zjevné, že můžeme pokračovat v zjednodušování výrazu a dosáhnout zpoždění $2\Delta t$ pro každou rovnici. Bude však narůstat počet vstupů do hradla a počet hradel

Poznámka k realizaci rychlé paralelní sčítačky

- Paralelní sčítačka s minimálním zpožděním (kombinační obvod) je prakticky nerealizovatelná.
- Pro 64-bitovou verzi bychom potřebovali 10^{20} hradel.

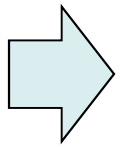
```
  x = 1101 0110 1011
  y = 0011 1011 1101
  c = ???? ???? ????
  ------------------
  s = ???? ???? ????
```

- Jak můžeme urychlit výpočet součtu? = Jak můžeme urychlit výpočet přenosu?
- => **sčítačka s „predikcí“ přenosů** – Carry Look-Ahead

Paralelní sčítačka s předvídáním přenosu

x_j	y_j	c_j	c_{j+1}	s_j
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$C_{n+1} = C_n$
$C_{n+1} = C_n$
$C_{n+1} = C_n$
$C_{n+1} = C_n$



x_j	y_j	c_{j+1}
0	0	0
0	1	C_n
1	0	C_n
1	1	1

Šíření přenosu

Generování přenosu

$$c_{j+1} = x_j y_j \vee (x_j \oplus y_j) c_j$$

$$s_j = \overline{x}_j \overline{y}_j c_j \vee \overline{x}_j y_j \overline{c}_j \vee x_j \overline{y}_j \overline{c}_j \vee x_j y_j c_j$$

$$= c_j (\overline{x}_j \overline{y}_j \vee x_j y_j) \vee \overline{c}_j (x_j \overline{y}_j \vee \overline{x}_j y_j) = c_j \overline{(x_j \oplus y_j)} \vee \overline{c}_j (x_j \oplus y_j)$$

Tyto rovnice jsou klíčové pro pochopení principu…

Označme:

- generování přenosu:

$$g_j = x_j y_j$$

- šíření přenosu (propagation):

$$p_j = x_j \oplus y_j = x_j \bar{y}_j \vee \bar{x}_j y_j$$

Pak:

- součet v j-tem řádu:

$$s_j = c_j \left(\overline{x_j \oplus y_j}\right) \vee \bar{c}_j \left(x_j \oplus y_j\right) = c_j \bar{p}_j \vee \bar{c}_j p_j = p_j \oplus c_j$$

- přenos do vyššího (j+1) řádu:

$$c_{j+1} = x_j y_j \vee \left(x_j \oplus y_j\right) c_j = g_j \vee p_j c_j$$

Paralelní sčítačka s předvídáním přenosu

Takže platí: $c_1 = g_0 \vee p_0 c_0$

$c_2 = g_1 \vee p_1 c_1 = g_1 \vee p_1(g_0 \vee p_0 c_0) = g_1 \vee p_1 g_0 \vee p_1 p_0 c_0$

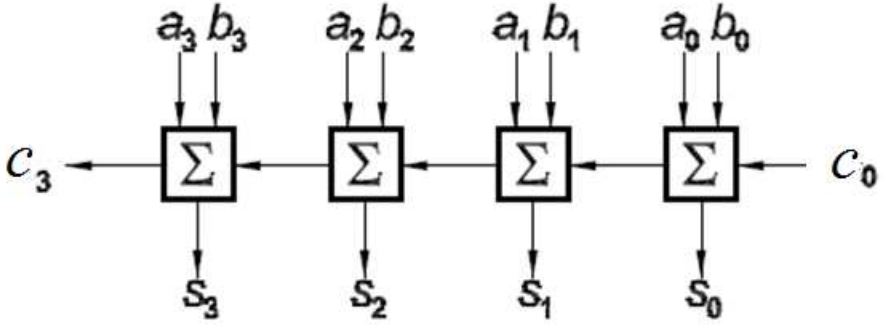
$c_3 = g_2 \vee p_2 c_2 = g_2 \vee p_2(g_1 \vee p_1 g_0 \vee p_1 p_0 c_0) = g_2 \vee p_2 g_1 \vee p_2 p_1 g_0 \vee p_2 p_1 p_0 c_0$

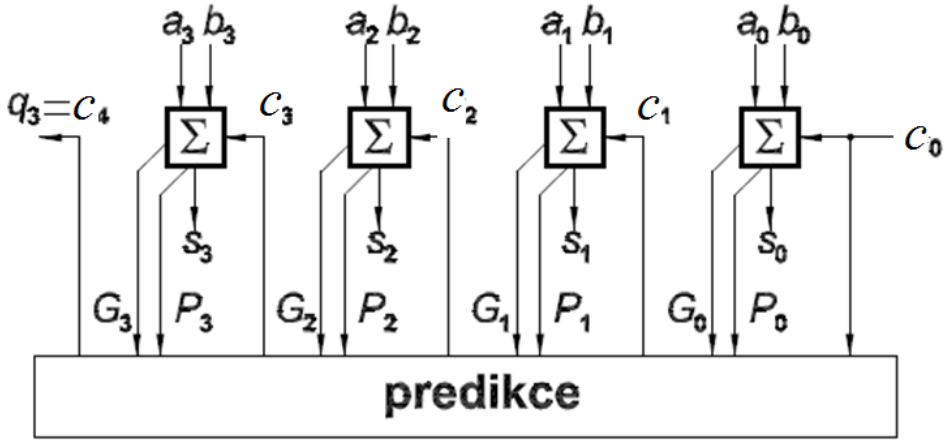
$c_4 = g_3 \vee p_3 c_3 = \ldots = g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0 \vee p_3 p_2 p_1 p_0 c_0$

$c_5 = \ldots$

Například rovnici pro c_3 je možné číst následovně: Přenos do 3. řádu nastane, **pokud** přenos byl generován v 2. řádu, **nebo** se 2. řádem šíří a byl generován v 1. řádu, **nebo** se šíří 2. aj 1. řádem a byl generován 0-tým řádem, **nebo** se šíří druhým, prvním aj 0-tým řádem a byl v c_0 (c_0==1).

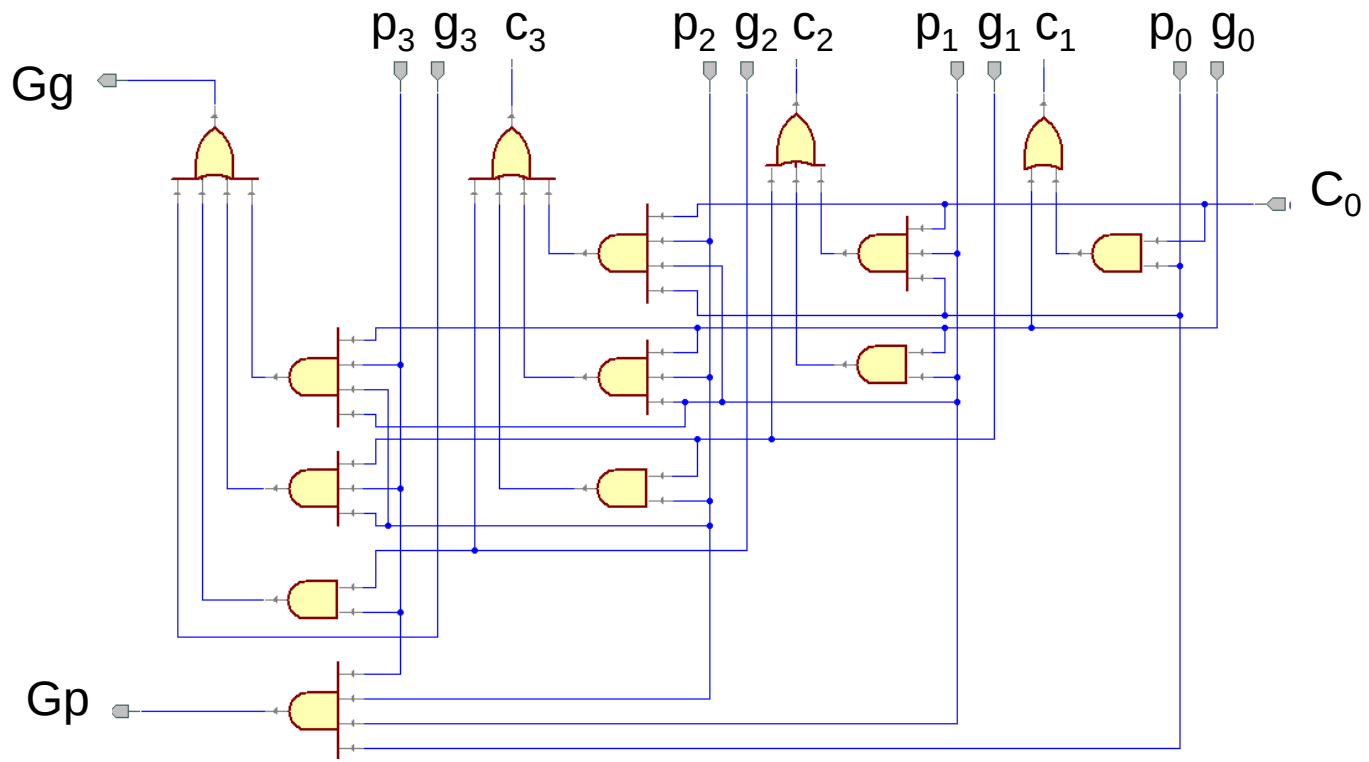
Pro porovnání: Sčítačka s postupným přenosem:





CLA – struktura jednotky „Predikce“

Vycházejíc z rovnic pro c_1 až c_3 sestrojíme jednotku CLA:



$$c_4 = g_3 \vee p_3c_3 = \ldots = \underbrace{g_3 \vee p_3g_2 \vee p_3p_2g_1 \vee p_3p_2p_1g_0}_{Gg_0} \vee \underbrace{p_3p_2p_1p_0}_{Gp_0}c_0$$

$$= Gg_0 \vee Gp_0\,Gc_0$$

Paralelní sčítačka s předvídáním přenosu

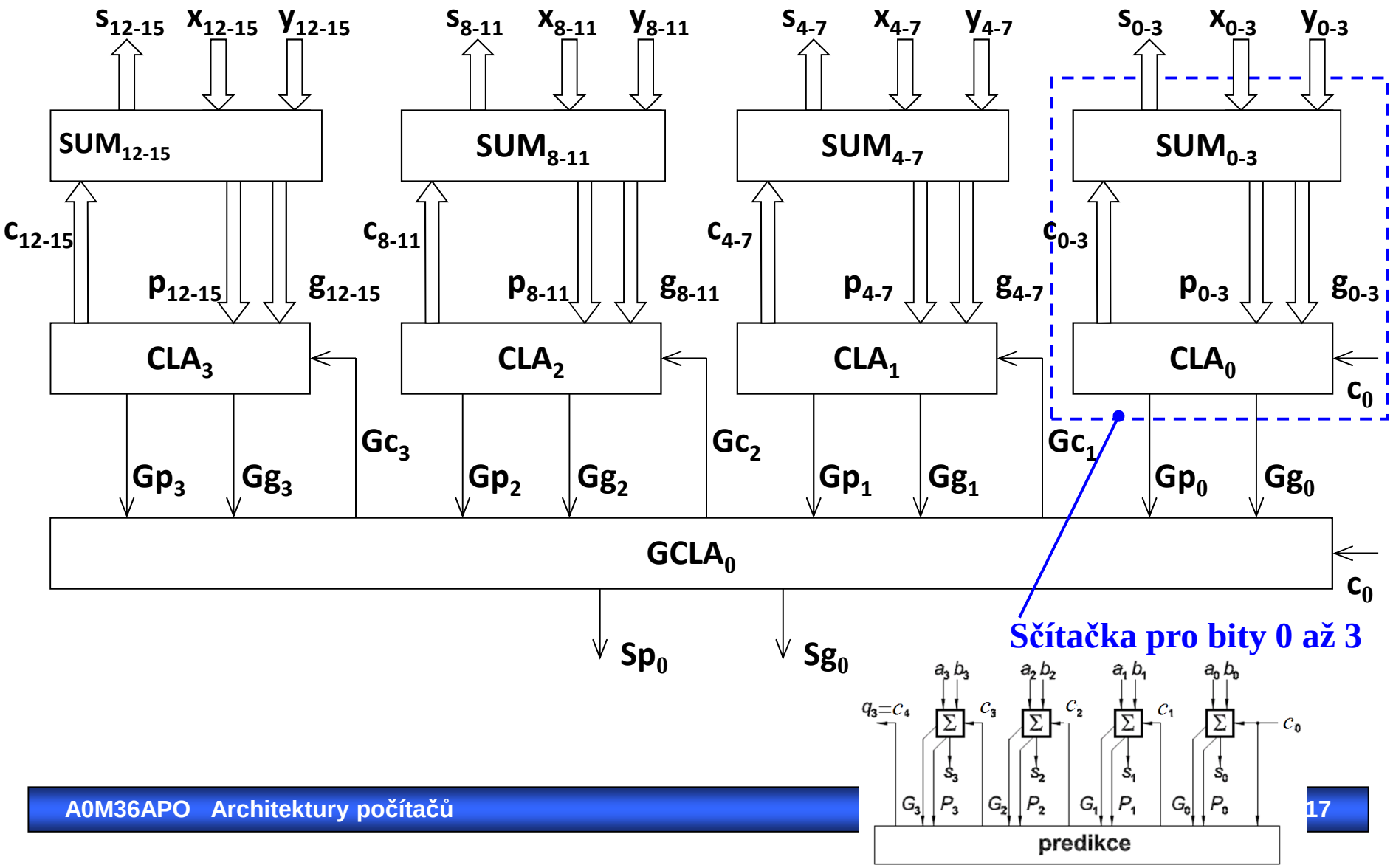
Podobným způsobem jako jsme stanovili c_4 můžeme odvodit rovnice pro grupové přenosy:

$$c_8 = \boldsymbol{Gc_2} = Gg_1 \vee Gp_1 Gc_1 = Gg_1 \vee Gp_1(Gg_0 \vee Gp_0 Gc_0) =$$
$$= Gg_1 \vee Gp_1 Gg_0 \vee Gp_1 Gp_0 Gc_0$$

$$c_{12} = \boldsymbol{Gc_3} = Gg_2 \vee Gp_2 Gc_2 = Gg_2 \vee Gp_2(Gg_1 \vee Gp_1 Gg_0 \vee)$$
$$= Gg_2 \vee Gp_2 Gg_1 \vee Gp_2 Gp_1 Gg_0 \vee Gp_2 Gp_1 Gp_0 Gc_0$$

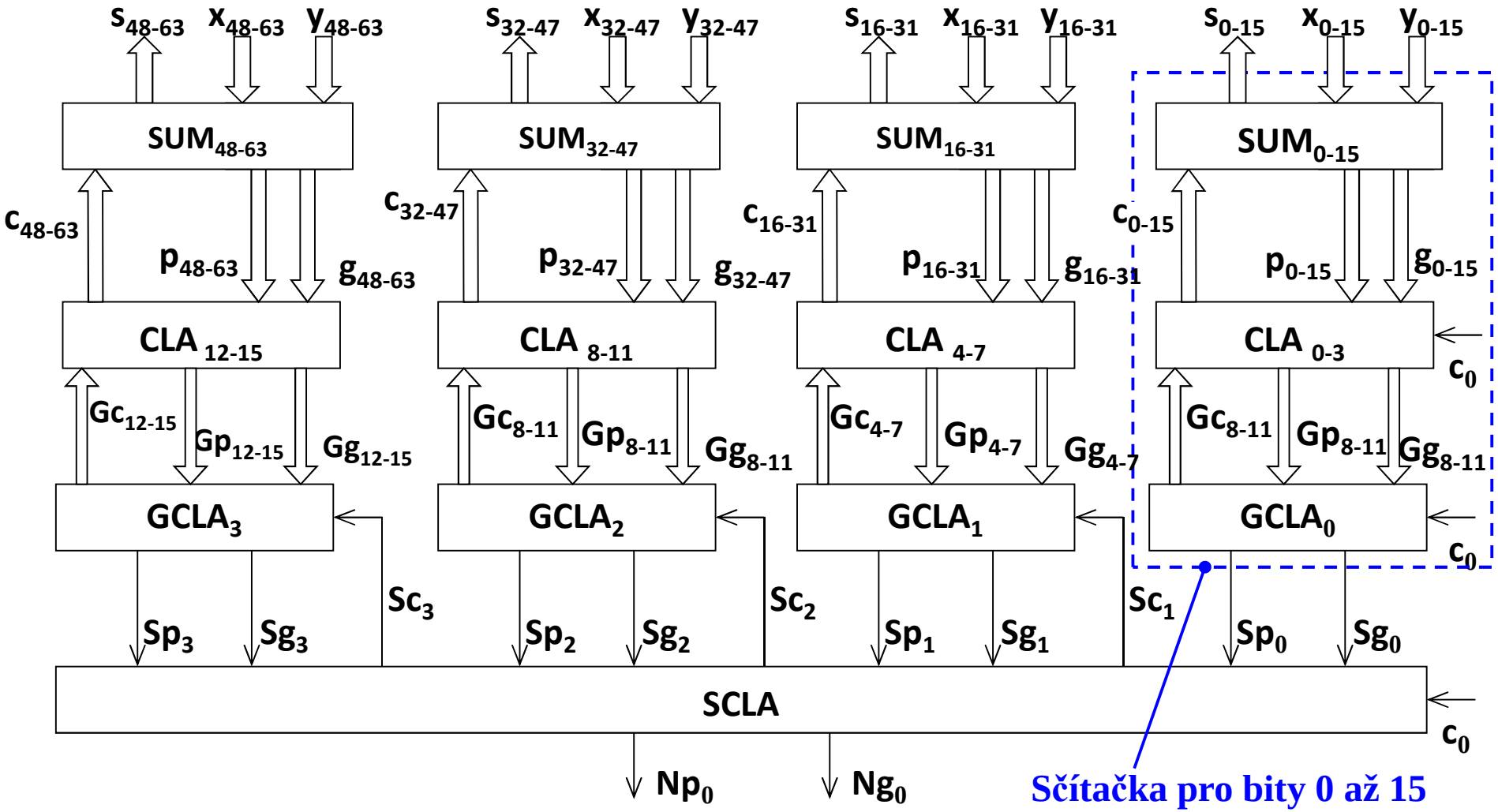
S výjimkou názvů proměnných vidíme, že výše uvedené rovnice pro c_4, c_8 a c_{12} jsou identické s rovnicemi pro c_1 až c_3. Čili, členy grupových přenosů lze vytvářet stejnými obvody CLA, jako pro obyčejné přenosy. Propojení sumátorů a jednotek CLA pro bity 0 až 15 je ukázáno na následujícím obrázku:

16-bitová sčítačka s grupovým předvídáním



64-bitová paralelní sčítačka s předvídáním přenosu

64-bitový sumátor se sekciovým předvídáním:

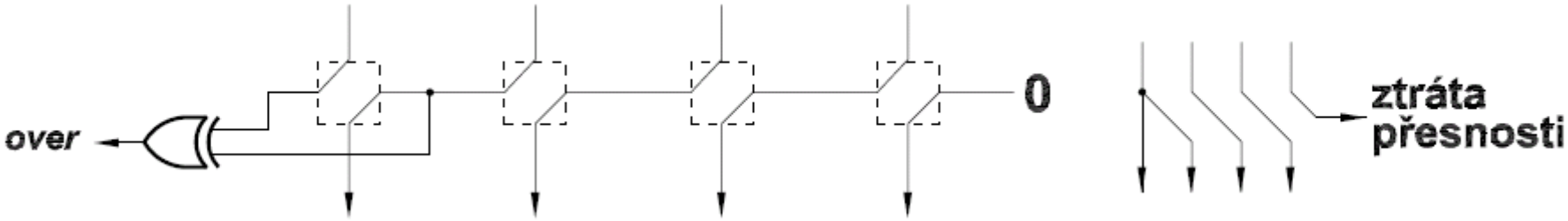


Paralelní sčítačka s předvídáním přenosu

- CLA – Carry Look-Ahead.
- Sčítačka CLA nabízí dostatečné zrychlení v porovnaní se sčítačkou s postupným přenosem při přijatelném nárůstu ceny HW.
- 64-bitová verze zvýší cenu HW o necelých 50% v porovnání se sčítačkou s postupným šířením přenosu, ale rychlost se zvýší 9:1 (128Δt vs. 14Δt).
- To představuje významné zvýšení poměru *rychlost/cena).*

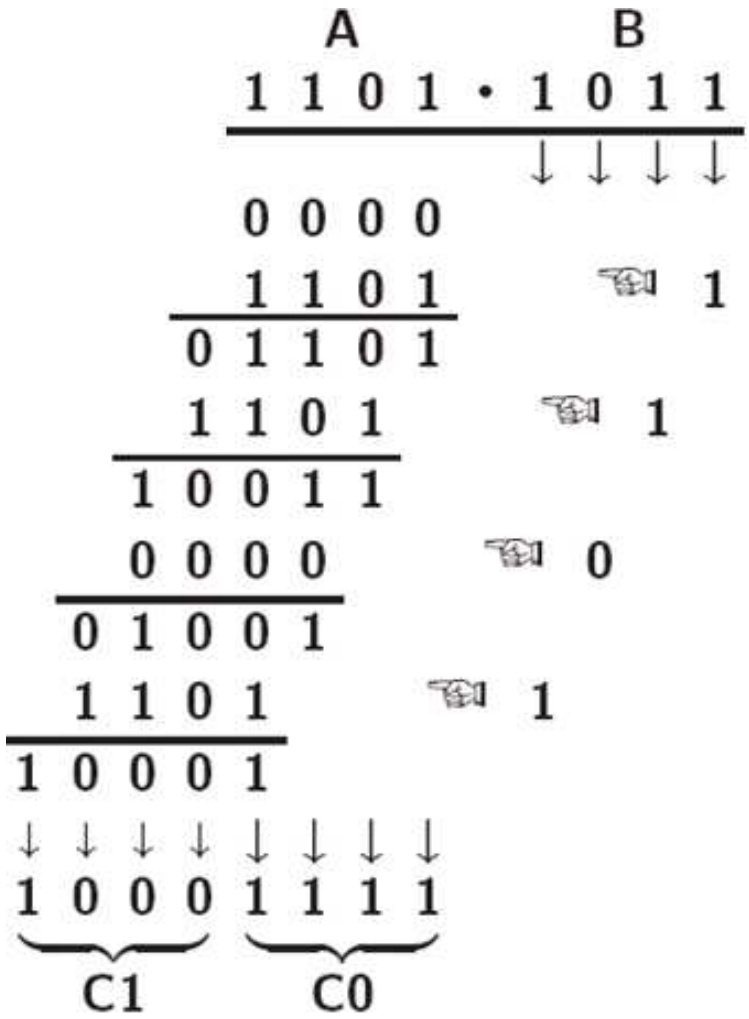
Aritmetické posuvy: vlevo, vpravo

- Aritmetický posuv v doplňkovém kódu o 1 bit (výsledek má odpovídat přísl. násobení nebo dělení)
- vlevo vpravo

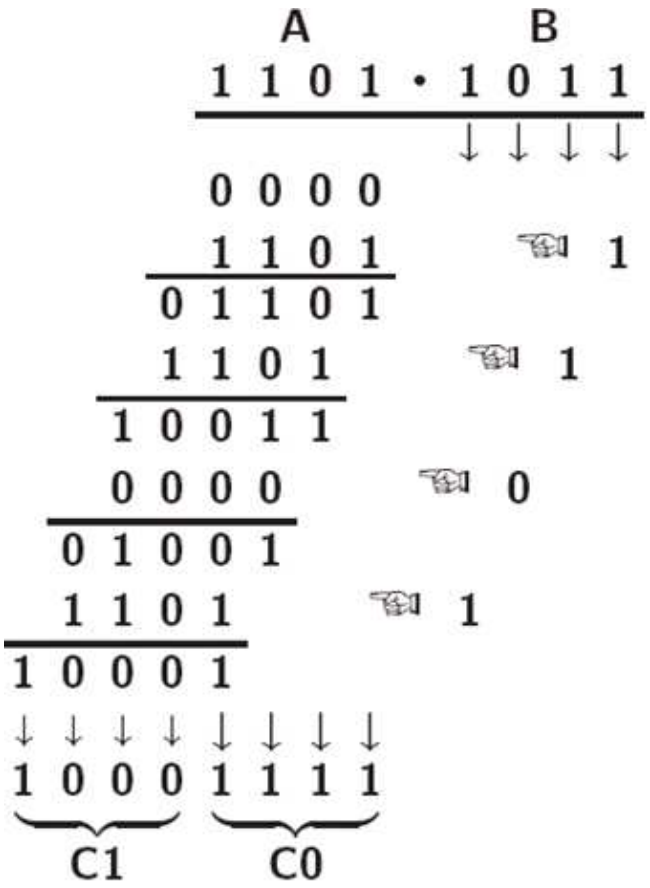


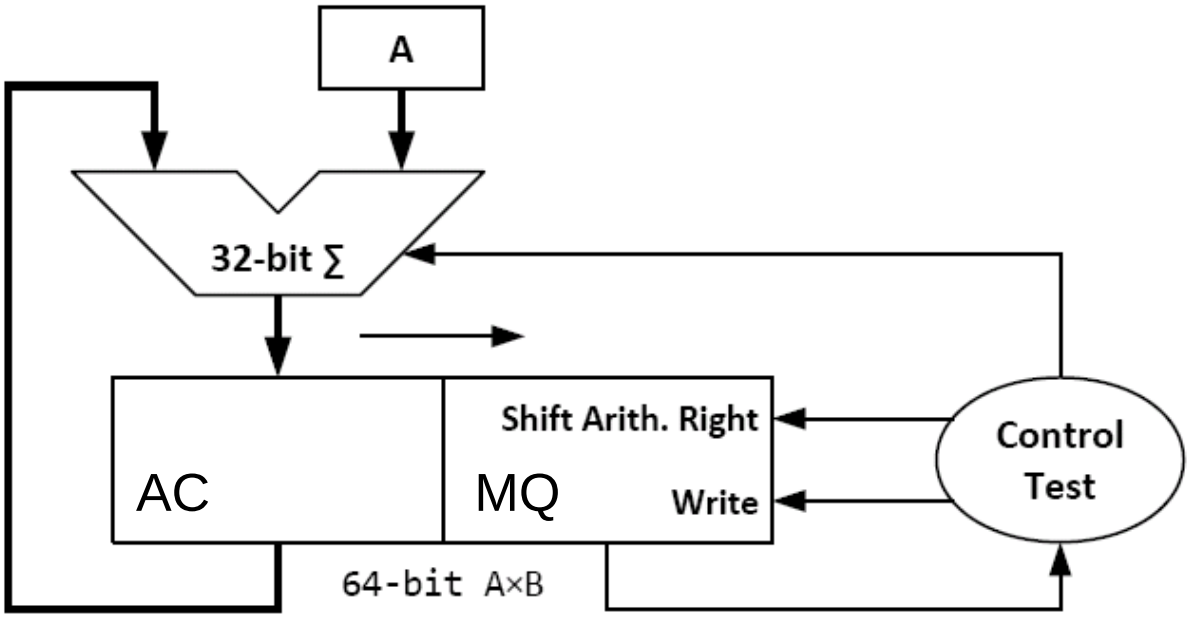
- Aritmetický posuv vs. Logický posuv vs. Cyklický posuv...

Násobení binárních čísel bez znaménka – pro připomenutí



Sekvenční HW násobička (varianta 32b)





Diskuze o rychlosti: ta je ale pomalá, co?

Algoritmus

A = násobenec;
MQ = násobitel;
AC = 0;

for(int i=1; i <= *n*; i++) // *n* je počet bitů
{
  if(MQ_0 = = 1) AC = AC + A; // MQ_0 = nejnižší bit MQ

  SR (posuň registr AC MQ o jedno místo doprava a doplň případný přenos z nejvyššího řádu z předchozího kroku)
}
end.

Nyní je výsledek v AC MQ.

Příklad x.y

Násobenec x=110 a násobitel y=101.

i	operace	AC	MQ	A	komentář
		000	101	110	prvotní nastavení
1	AC = AC+MB	110	101		začátek cyklu
	SR	011	010		
2	nic	011	010		protože MQ_0 = = 0
	SR	001	101		
3	AC = AC+MB	111	101		
	SR	011	110		konec cyklu

Tedy: x×y = 110×101 = 011110, (6×5 = 30)

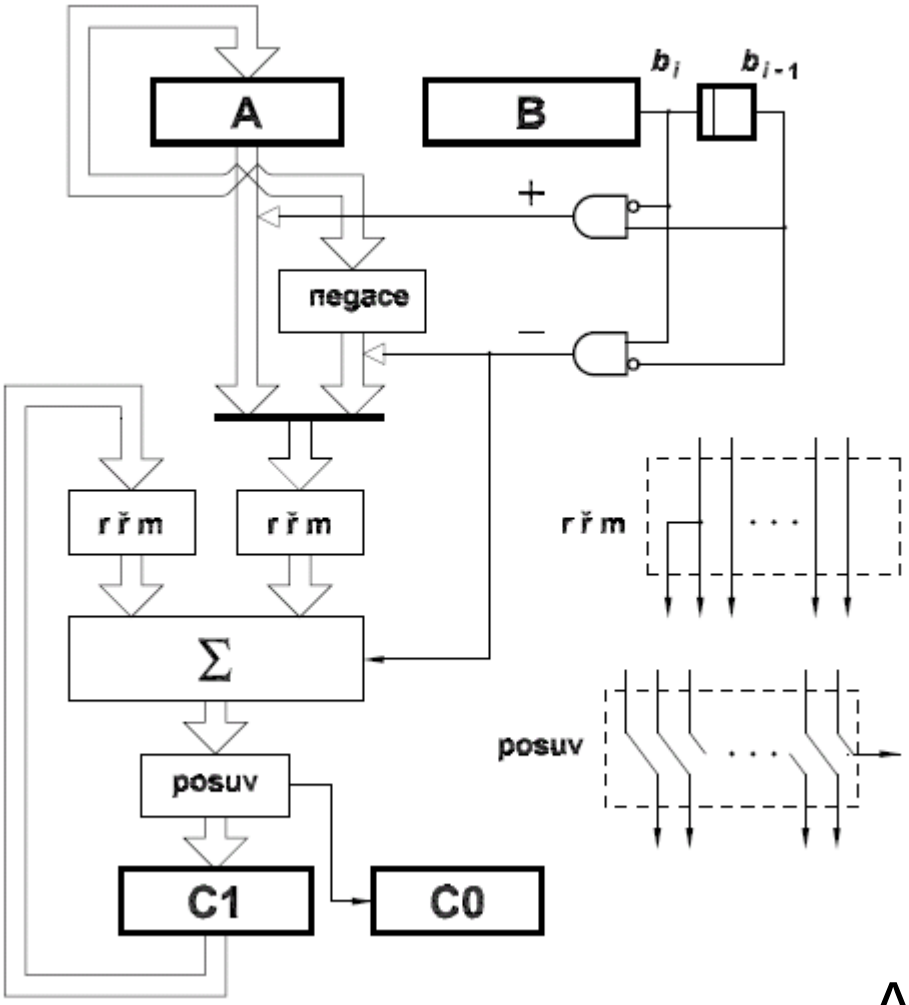
Násobení v doplňkovém kódu

- Lze realizovat, ale je tu problém…
  - **Obraz součinu obecně není roven součinu obrazů!**
- Řešení spočívá v modifikaci dvojkové soustavy na soustavu s relativními číslicemi $0,\ 1,\ \hat{1} = -1$
- Podrobnosti už jsou mimo zamýšlený rozsah APO.

$$
\begin{array}{lrcl}
\text{př.:} & 1\,\hat{1}\,1\,0 & \sim & 1\cdot 8 + (-1)\cdot 4 + 1\cdot 2 + 0\cdot 1 = 6 \\
& \hat{1}\,1\,\hat{1}\,0 & \sim & (-1)\cdot 8 + 1\cdot 4 + (-1)\cdot 2 + 0\cdot 1 = -6 \\
& 1\,1\,0 = 1\,0\,\hat{1}\,0 = 1\,\hat{1}\,1\,0 & \sim & 6
\end{array}
$$

- Nebo v znaménkovém rozšíření na 2N bitů a násobení obvyklým způsobem. Z výsledku bereme pouze 2N bitů. -> „ruční“ násobení

Násobička v doplňkovém kódu Boothova metoda



APO – jen pro informaci

Rychlá násobička podle Walaceova stromu

Q=X .Y, X a Y nechť jsou 8b čísla

(x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0). (y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0) =

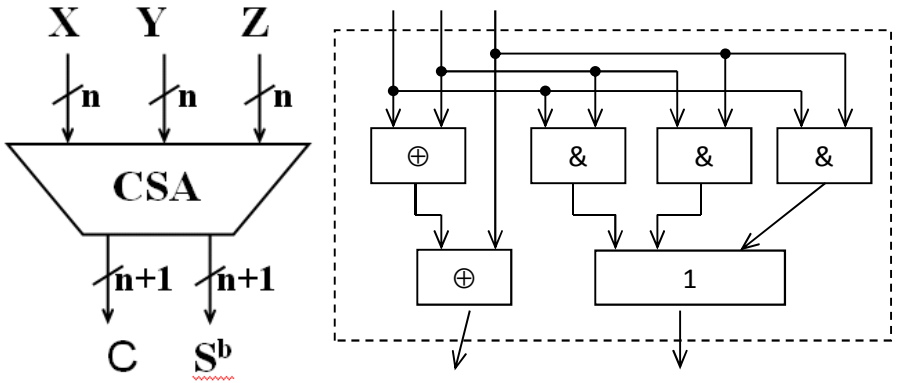
0	0	0	0	0	0	0	0	x_7y_0	x_6y_0	x_5y_0	x_4y_0	x_3y_0	x_2y_0	x_1y_0	x_0y_0	P0
0	0	0	0	0	0	0	x_7y_1	x_6y_1	x_5y_1	x_4y_1	x_3y_1	x_2y_1	x_1y_1	x_0y_1	0	P1
0	0	0	0	0	0	x_7y_2	x_6y_2	x_5y_2	x_4y_2	x_3y_2	x_2y_2	x_1y_2	x_0y_2	0	0	P2
0	0	0	0	0	x_7y_3	x_6y_3	x_5y_3	x_4y_3	x_3y_3	x_2y_3	x_1y_3	x_0y_3	0	0	0	P3
0	0	0	0	x_7y_4	x_6y_4	x_5y_4	x_4y_4	x_3y_4	x_2y_4	x_1y_4	x_0y_4	0	0	0	0	P4
0	0	0	x_7y_5	x_6y_5	x_5y_5	x_4y_5	x_3y_5	x_2y_5	x_1y_5	x_0y_5	0	0	0	0	0	P5
0	0	x_7y_6	x_6y_6	x_5y_6	x_4y_6	x_3y_6	x_2y_6	x_1y_6	x_0y_6	0	0	0	0	0	0	P6
0	x_7y_7	x_6y_7	x_5y_7	x_4y_7	x_3y_7	x_2y_7	x_1y_7	x_0y_7	0	0	0	0	0	0	0	P7
Q_{15}	Q_{14}	Q_{13}	Q_{12}	Q_{11}	Q_{10}	Q_9	Q_8	Q_7	Q_6	Q_5	Q_4	Q_3	Q_2	Q_1	Q_0	

Součtem P0+P1+...+P7 získáme výsledek součinu X a Y.

Q = X .Y = P0 + P1 + ... + P7

Rychlá násobička podle Walaceova stromu

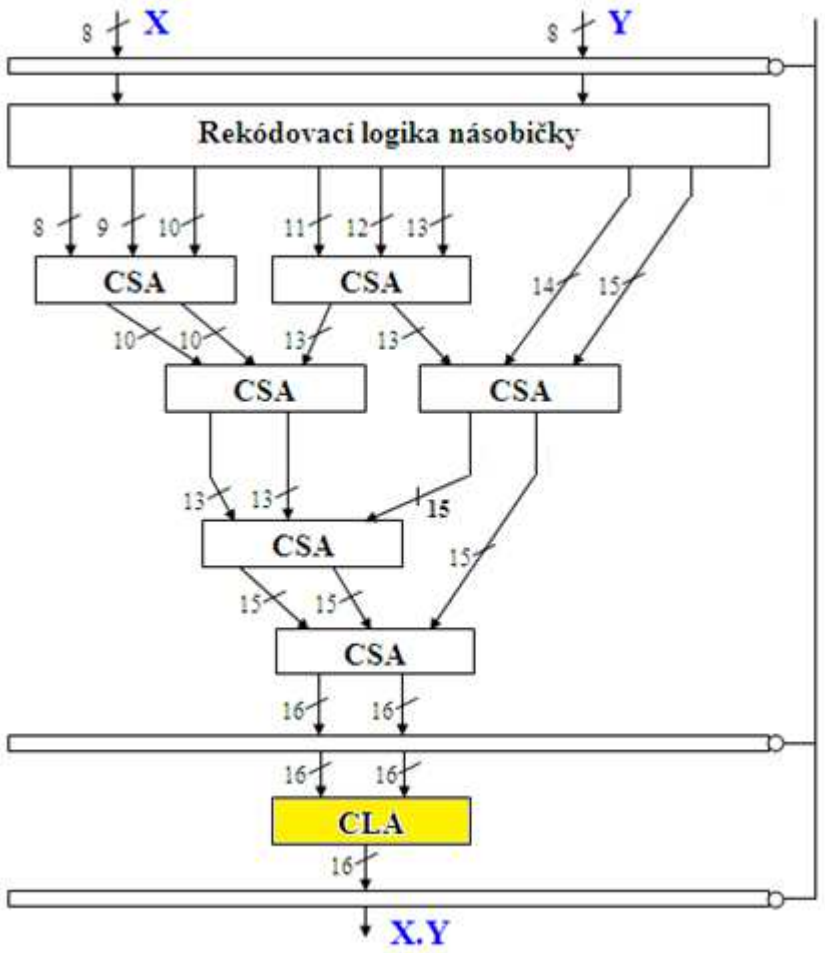
Jejím stavebním prvkem je sčítačka s uchováním přenosu CSA (Carry Save Adder)



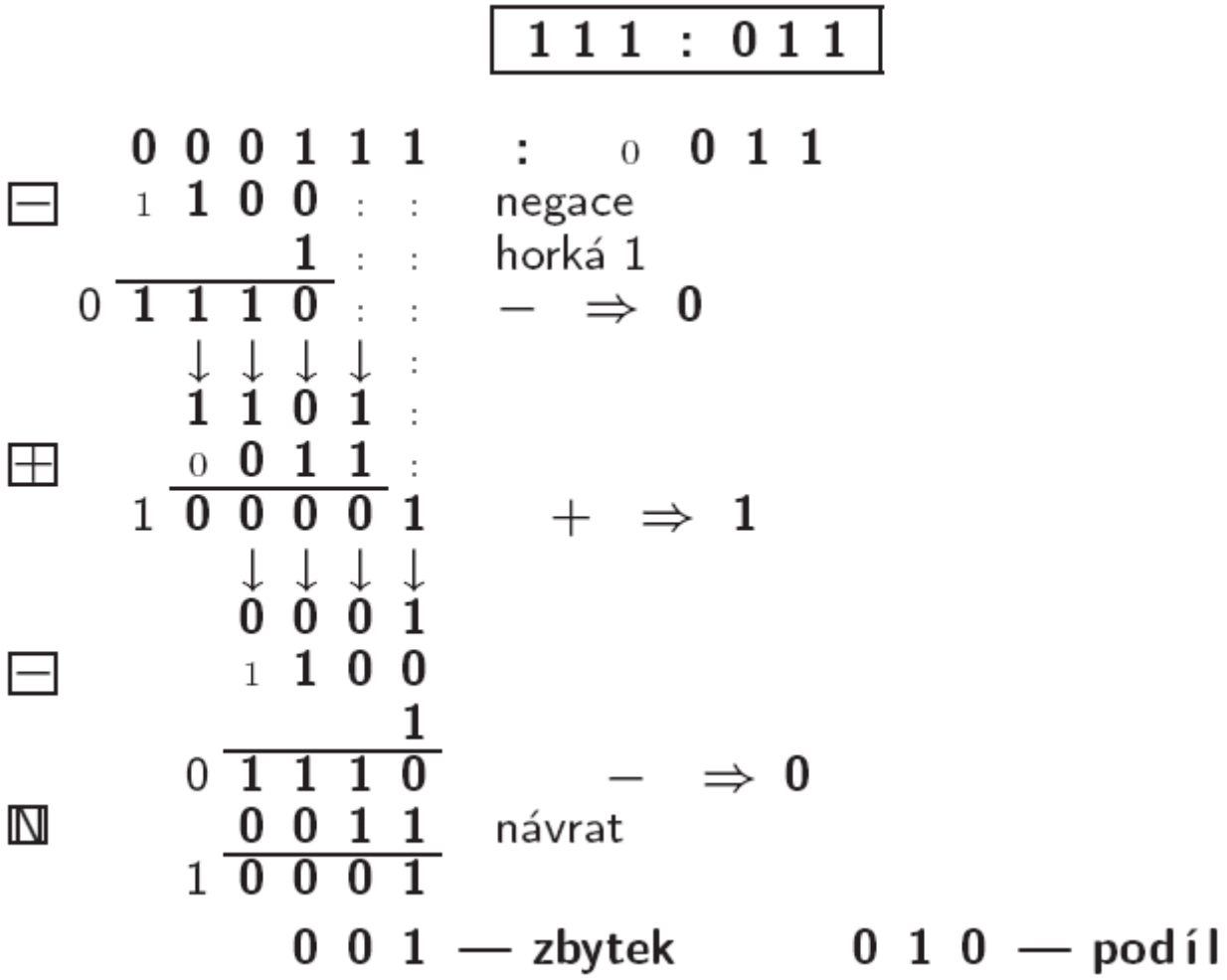
$$S = S^b + C$$

$$S^b_i = x_i \oplus y_i \oplus z_i$$

$$C_{i+1} = x_i y_i + y_i z_i + z_i x_i$$



HW dělička – algoritmus dělení

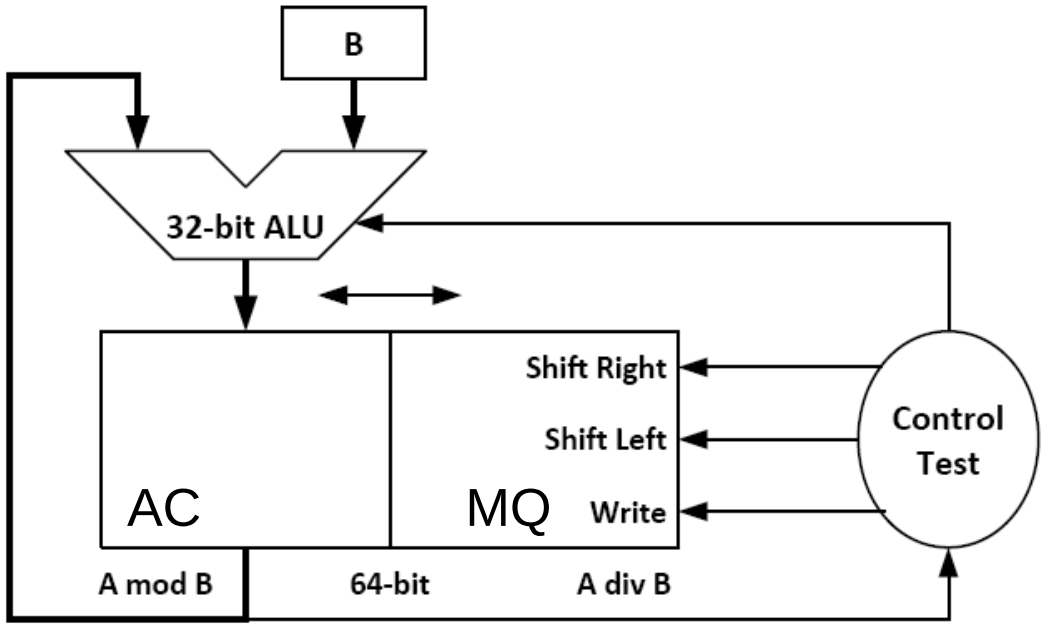


Sekvenční HW dělička (varianta 32b)

1 1 1 : 0 1 1 dělenec = podíl×dělitel + zbytek

```
      0 0 0 1 1 1   :   0 0 1 1
⊟     1 1 0 0   :   :   negace
            1   :   :   horká 1
    0 1 1 1 0   :   :   −  ⇒  0
      ↓ ↓ ↓ ↓   :
      1 1 0 1   :
⊞     0 0 1 1   :
    1 0 0 0 0 1       +  ⇒  1
        ↓ ↓ ↓ ↓
        0 0 0 1
⊟       1 1 0 0
              1
    0 1 1 1 0         −  ⇒  0
N       0 0 1 1   návrat
    1 0 0 0 1
```

0 0 1 — zbytek **0 1 0** — podíl



Algoritmus dělení

MQ = dělenec;
B = dělitel; (Podmínka: dělitel různý od 0!)
AC = 0;

```
for( int i=1; i <= n; i++)          {
   SL (posuň registr AC MQ o jednu pozici vlevo, přičemž vpravo se připíše nula)
   if(AC >= B)   {
         AC = AC – B;
         MQ0 = 1;          // nejnižší bit registru MQ se nastaví na 1
    }
}
```

→ Nyní registr MQ obsahuje podíl a zbytek je v AC

Příklad x/y

Dělenec x=1010 a dělitel y=0011

i	operace	AC	MQ	B	komentář
		0000	1010	0011	prvotní nastavení
1	**SL**	0001	0100		
	nic	0001	0100		podmínka if není splněna
2	**SL**	0010	1000		
		0010	1000		podmínka if není splněna
3	**SL**	0101	0000		r ≥ y
	AC = AC – B; MQ_0 = 1;	0010	0001		
4	**SL**	0100	0010		r ≥ y
	AC = AC – B; MQ_0 = 1;	0001	0011		konec cyklu

x : y = 1010 : 0011 = 0011 zbytek 0001, (10 : 3 = 3 zbytek 1)

Procesor

Počítač podle von Neumanna tvoří

- Řadič
- ALU
- Paměť
- Vstup
- Výstup

Procesor/mikroprocesor

Harvardská architektura je variantou s oddělenou pamětí programu a pamětí dat

V/V podsystém (V/V = I/O)

Řadič - součást (jednotka) počítače/procesoru, která jeho činnost řídí.
Sestává ze dvou částí:

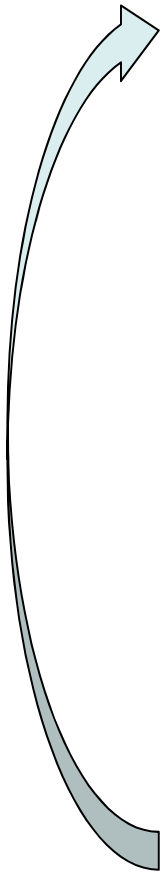
- datové
  - registry,
  - další potřebné obvody,
- vlastní řídicí části, z tzv. jádra řadiče.

Důležité registry řadiče

- PC (Program Counter), programový čítač.
- IR (Instruction Register), registr instrukce
- Další
  - Univerzální nebo pracovní registry,
  - SP (Stack Pointer), ukazatel zásobníku,
  - PSW Program Status Word), stavové slovo programu,
  - IM (Interrupt Mask), maska přerušení.

Základní cyklus počítače – sekvenční postup vykonávání instrukcí

1. Počáteční nastavení, zejména např. PC.
2. Čtení instrukce
   - PC → adresa hlavní paměti,
   - Čtení obsahu,
   - Přečtená data → IR,
   - PC+*l* → PC, kde *l* je délka instrukce.
3. Dekódování operačního znaku (OZ),
4. provedení operace (včetně vyhodnocení efektivních adres, čtení operandů, apod.).
5. Dotaz na možné přerušení. Ano-li, obsluha.
6. Ne-li, opakování od bodu 2.



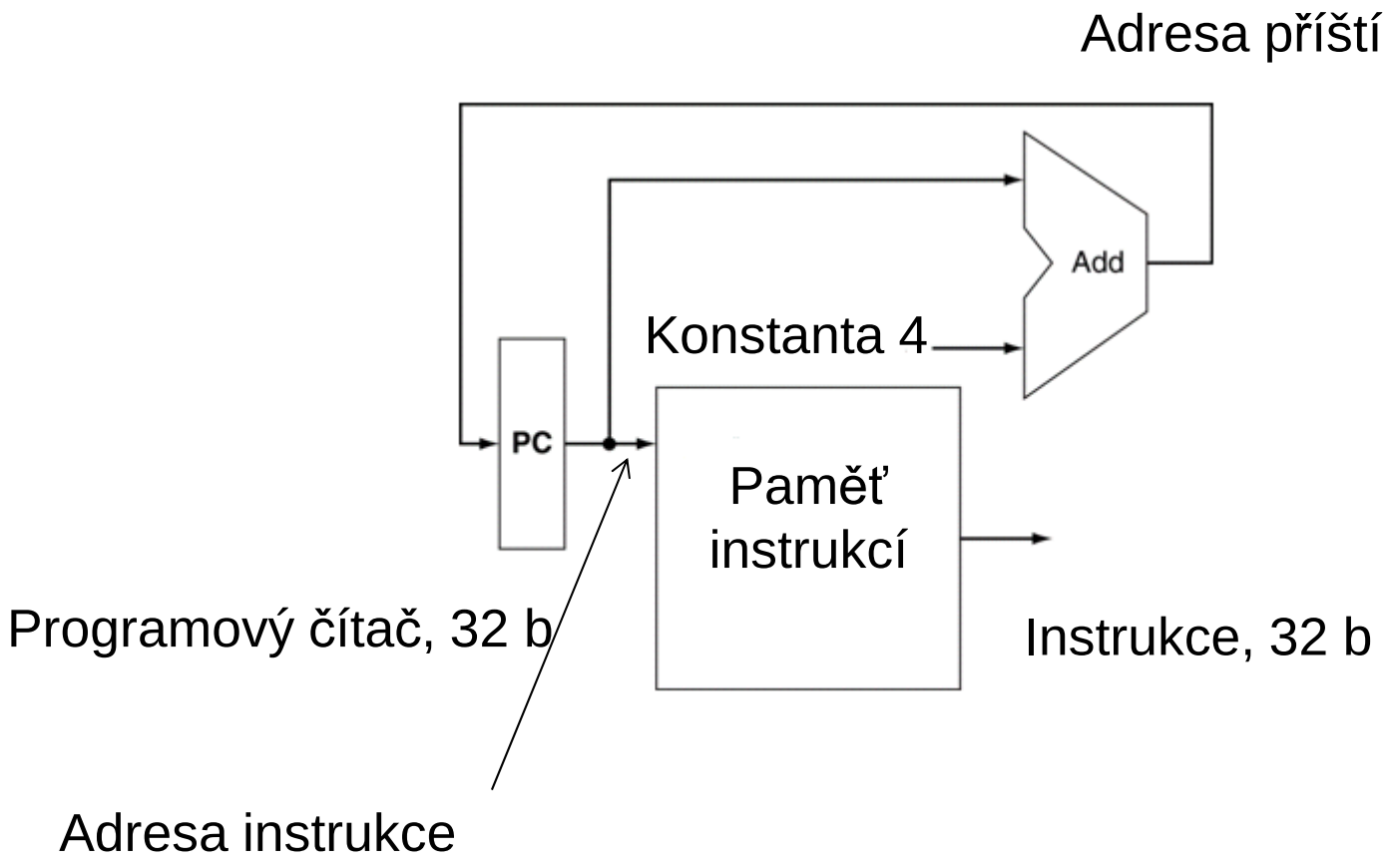
Kompilace a kódování programu

```
int pow = 1;
int x = 0;

while(pow != 128)
{
  pow = pow*2;
  x = x + 1;
}
```

```
addi s0, $0, 1        // pow = 1
addi s1, $0, 0        // x = 0
addi t0, $0, 128      // t0 = 128

while:
  beq  s0, t0, done  // if pow==128, go to done
  sll  s0, s0, 1         // pow = pow*2
  addi s1, s1, 1        // x = x+1
  j    while
done:
```

```
8001FFF4  00 00 00 00          NOP
8001FFF8  00 00 00 00          NOP
8001FFFC  00 00 00 00          NOP
80020000  20 10 00 01  start() ADDI   $16, $00, 0x1
80020004  20 11 00 00          ADDI   $17, $00, 0x0
80020008  20 08 00 80          ADDI   $08, $00, 0x80
8002000C  12 08 00 04  while:  BEQ    $08, $16, 0x4
80020010  00 00 00 00          NOP
80020014  00 10 80 40          SLL    $16, $16, 1
80020018  08 00 80 03          J      0x8003
8002001C  22 31 00 01          ADDI   $17, $17, 0x1
80020020  00 00 00 00  done:   NOP
80020024  00 00 00 00          NOP
80020028  00 00 00 00          NOP
```

Obvodová realizace základního cyklu počítače



Úkol pro tuto přednášku:

- Porozumět implementaci jednoduchého počítače tvořeného procesorem, oddělenými pamětmi instrukcí a dat a ALU, který umí instrukce
  - Čtení a zápis do datové paměti `lw` a `sw`,
  - Aritmetické-logické instrukce `add`, `sub`, `and`, `or` a `slt` a
  - Skokové instrukce `beq`.
- V procesoru bude řídicí jednotka (řadič) i ALU.
- Poznámka:
  - Na této přednášce jej budeme implementovat jednoduše (jako jednocyklový),
  - Na 4. přednášce ukážeme více realistickou, zřetězenou verzi.

Formát instrukcí

- Uvažujme tři typy instrukcí dle tabulky:

Typ	31…					0
R	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**rd**(5), 15:11	**shamt**(5)	**funct**(6), 5:0
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0		
J	**opcode**(6), 31:26	**address**(26), 25:0				

- všechny R instrukce -> opcode=000000, funct – operace
- rs – source, rd – destination, rt – source/destination
- shamt – při operacích posunu, immediate – přímý operand

- K dispozici je 32 pracovních registrů

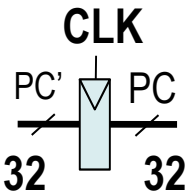
Která je to instrukce?

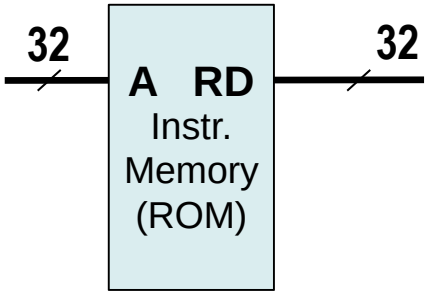
- Na základě **opcode** a **funct** (pokud je to R typ)

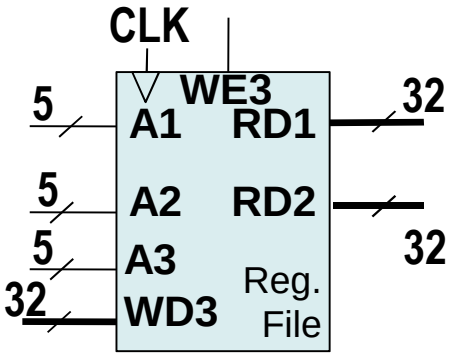
Opcode	
000000	R typ
100011	lw
101011	sw
000100	beq

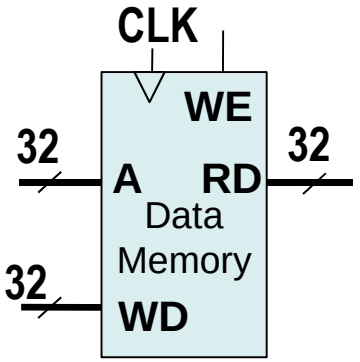
Funct	
100000	add
100010	sub
100100	and
100101	or
101010	slt

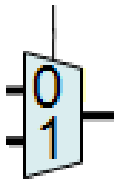
K dispozici máme tyto stavební prvky







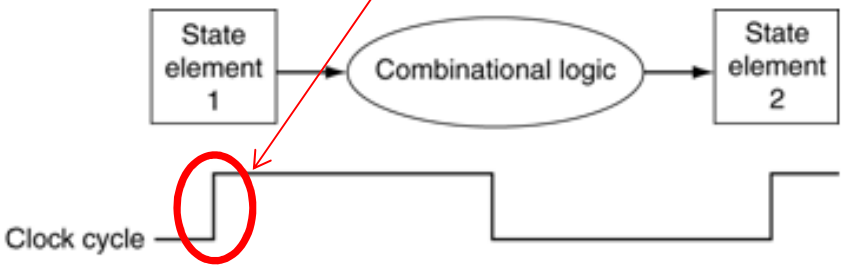




Multiplexor

Zápis náběžnou hranou CLK při WE = 1

Čtení po uplynutí „dostatečně dlouhé“ doby



Výklad syntaxe a sémantiky instrukce: například `lw`

`lw` – load word - čtení slova z datové paměti

Description	A word is loaded into a register from the specified address
Operation:	$t = MEM[$s + offset];
Syntax:	lw $t, offset($s)
Encoding:	`1000 11ss ssst tttt iiii iiii iiii iiii`

Uložme slovo z paměti na adrese 0x4 do registru č.11:

lw $11, 0x4($0)

```
1000 11ss ssst tttt iiii iiii iiii iiii
1000 1100 0000 1011 0000 0000 0000 0100
```

0 11 4

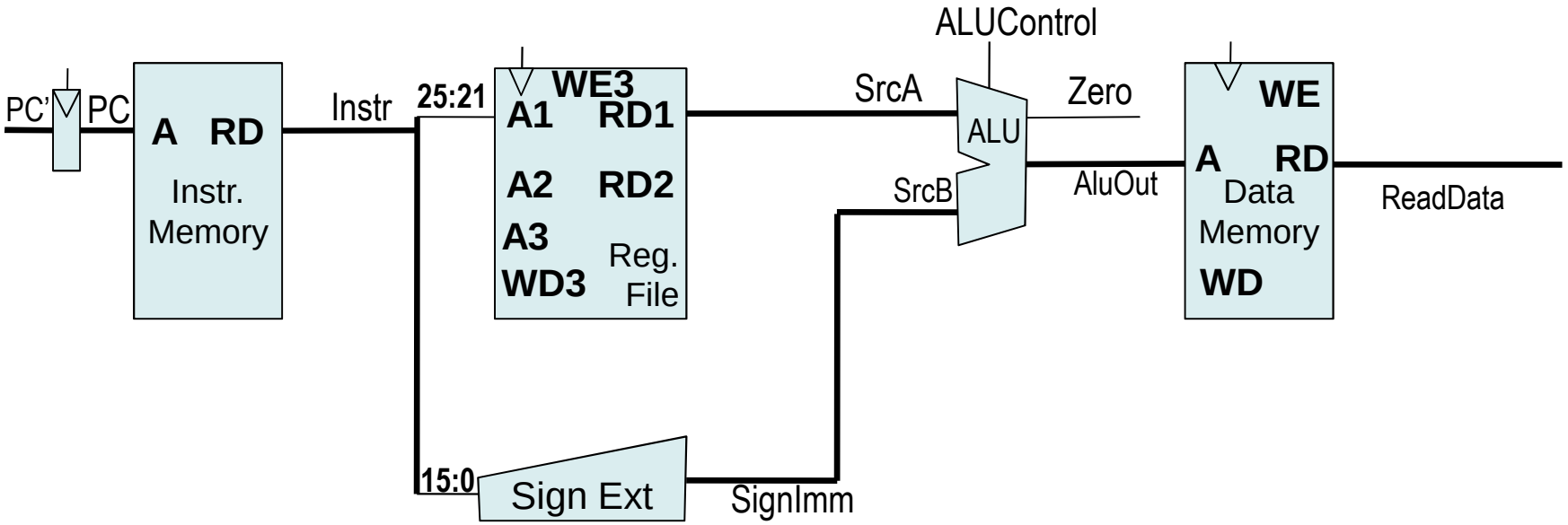
0x 8C 0B 00 04 – strojový kód instrukce lw $11, 0x4($0)

Poznámka: V registu $0 je trvale uložena konstanta 0.

Jedno-cyklový procesor – návrh – podpora čtení z paměti

- **`lw`**: typ I, rs – bázová adresa, imm – offset, rt – kde uložit

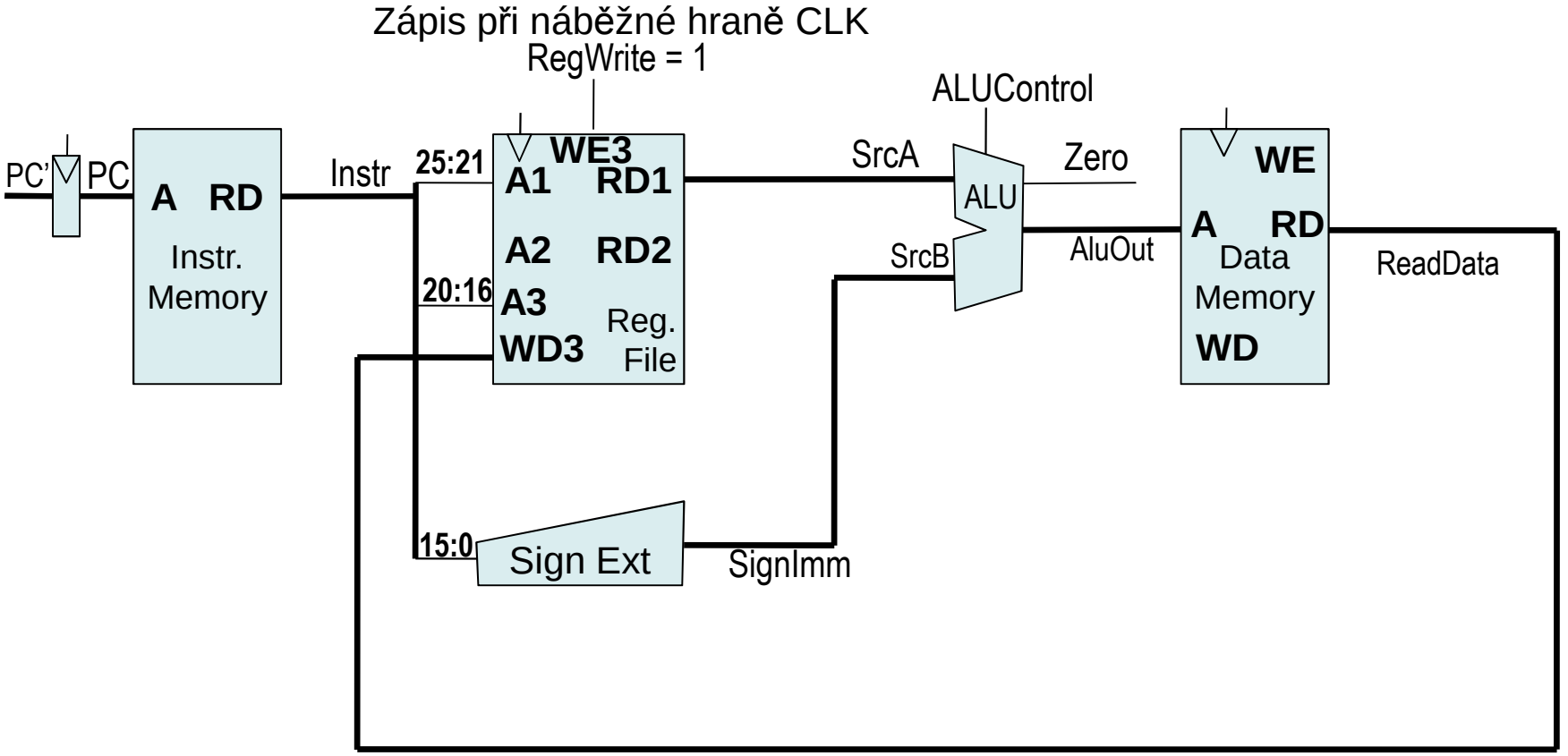
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



Jedno-cyklový procesor – návrh – podpora čtení z paměti

- **`lw`**: typ I, rs – bázová adresa, imm – offset, rt – kde uložit

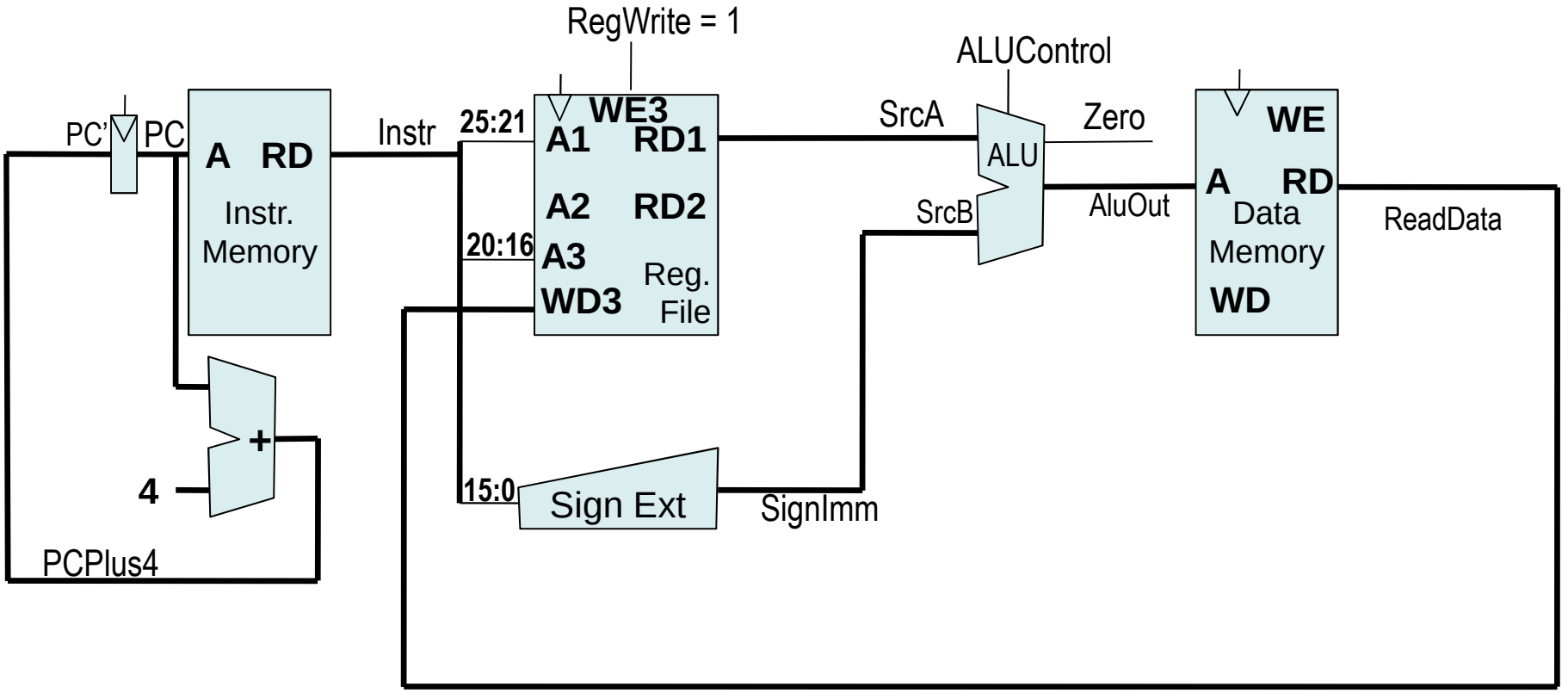
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



Jedno-cyklový procesor – návrh – podpora čtení z paměti

- **`lw`**: typ I, rs – bázová adresa, imm – offset, rt – kde uložit

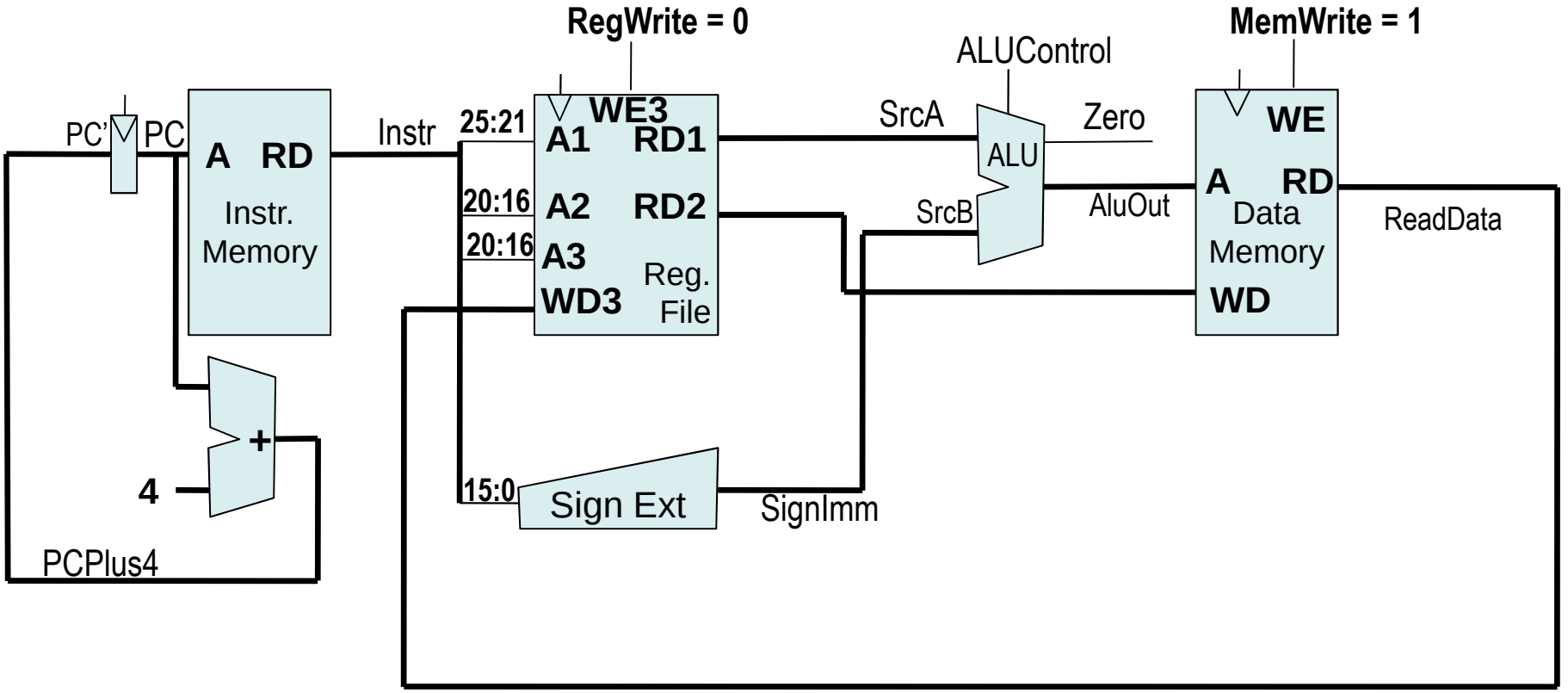
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



Jedno-cyklový procesor – návrh – podpora zápis do paměti

- **sw**: typ I, rs – bázová adresa, imm – offset, rt – co zapsat

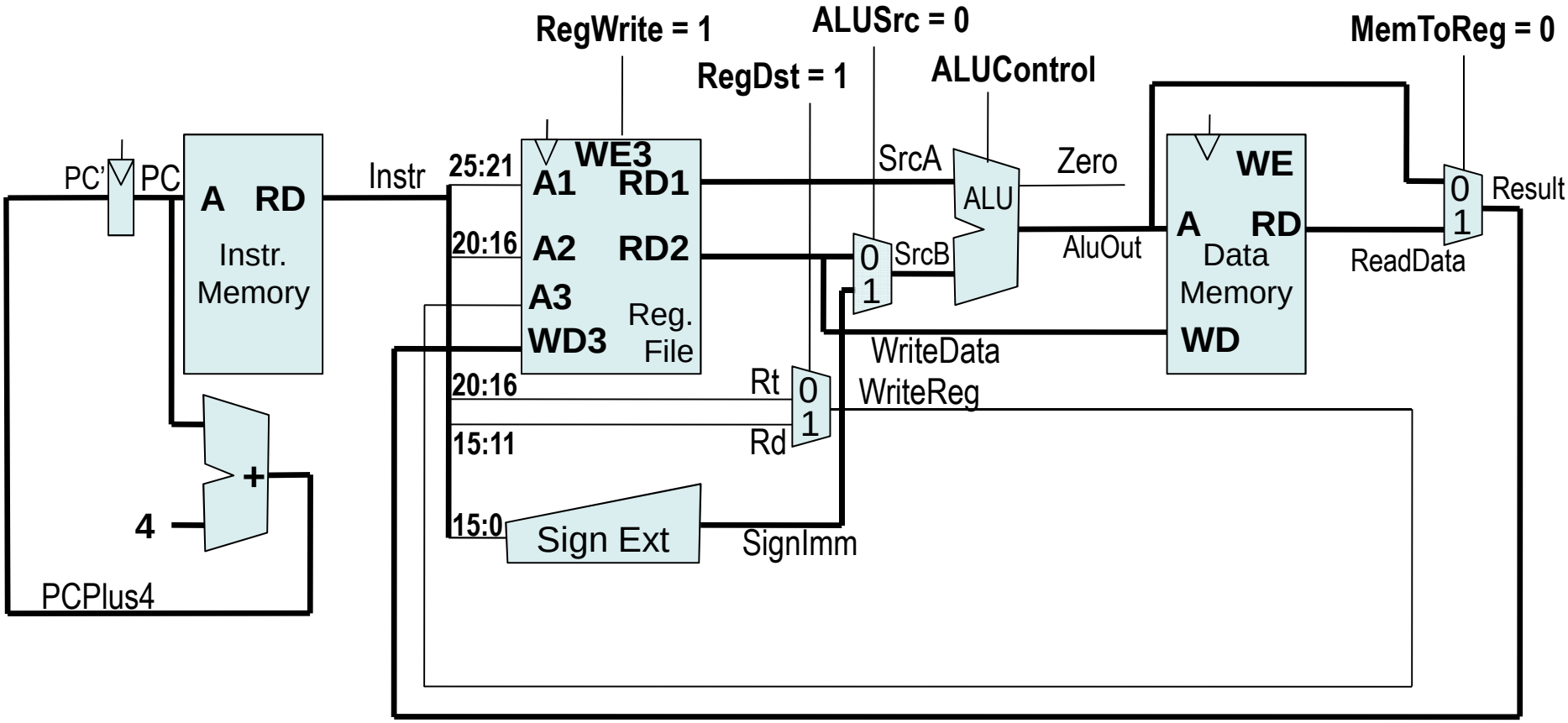
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



Jedno-cyklový procesor – návrh – podpora add

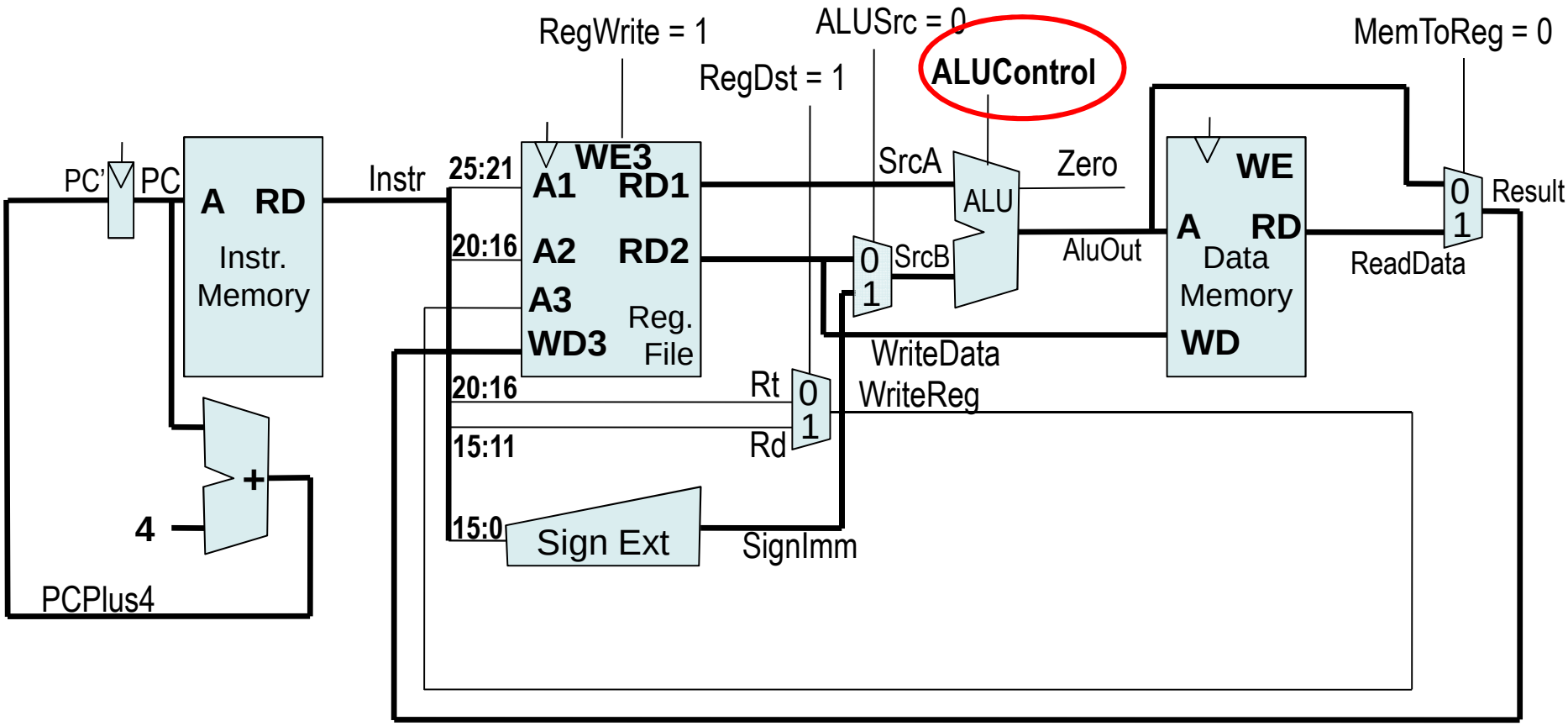
- **add**: typ R; rs, rt – zdroje, rd – cíl, funct – operace součtu

R	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**rd**(5), 15:11	**shamt**(5)	**funct**(6), 5:0



Jedno-cyklový procesor – návrh – podpora `sub`, `and`, `or`, `slt`

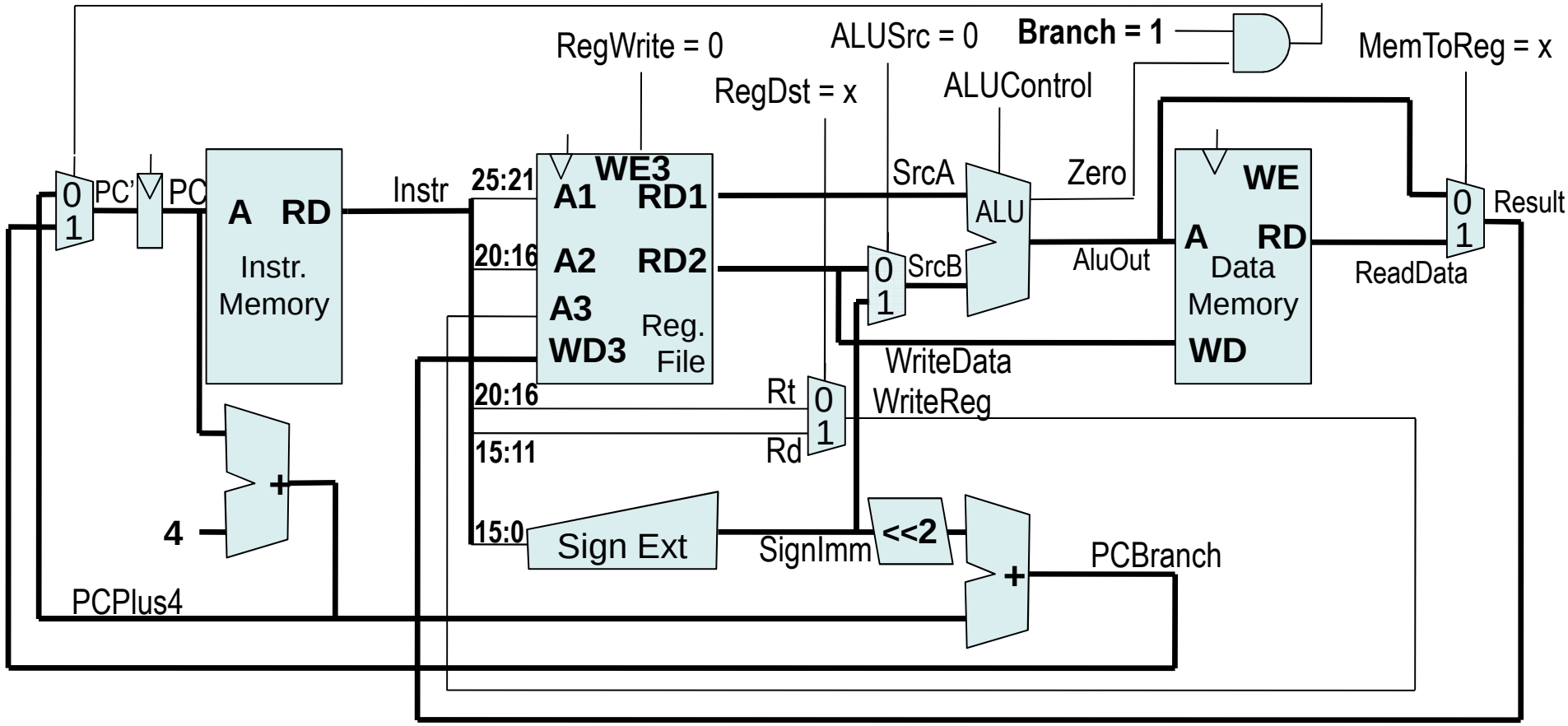
- jediné v čem se liší od `add` je operace ALU -> datapath beze změny; rozdíl v ALUControl



Jedno-cyklový procesor – návrh – podpora beq

- **beq** – branch if equal; imm–offset; PC´ = PC+4 + SignImm*4

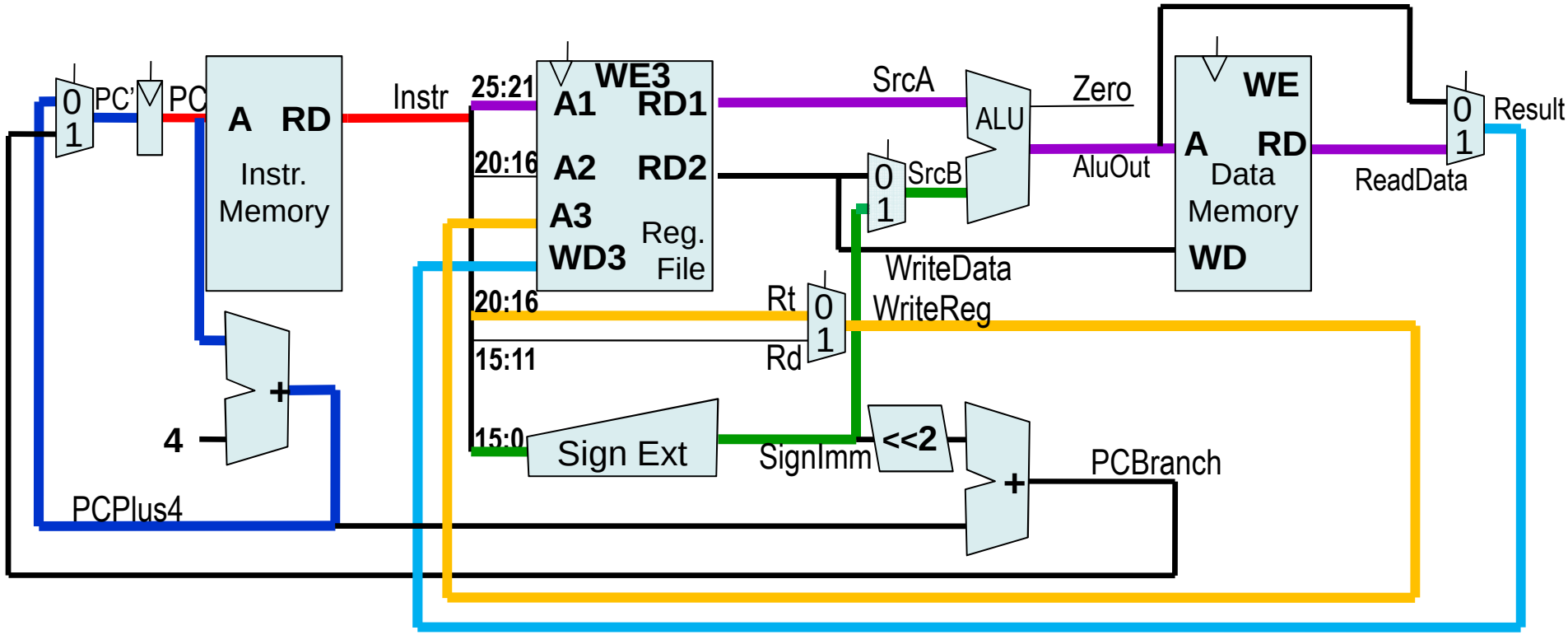
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



Jedno-cyklový procesor – výkon: IPS = IC / T = $IPC_{str}.f_{CLK}$

- Jaká může být maximální frekvence procesoru?
- Zpoždění na kritické cestě – instrukce `lw`:

$$Tc = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}$$



Jedno-cyklový procesor – výkon: IPS = IC / T = $IPC_{str}.f_{CLK}$

- $Tc = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}$
- Předpokládejme:

  t_{PC} = 30 ns
  t_{Mem} = 300 ns
  t_{RFread} = 150 ns
  t_{ALU} = 200 ns
  t_{Mux} = 20 ns
  $t_{RFsetup}$ = 20 ns

  Pak Tc = 1020 ns --> $f_{CLK\ max}$ = 980 kHz,

  IPS = 1. 980e3 = 980 000 instrukcí za sekundu

Důležitá poznámka

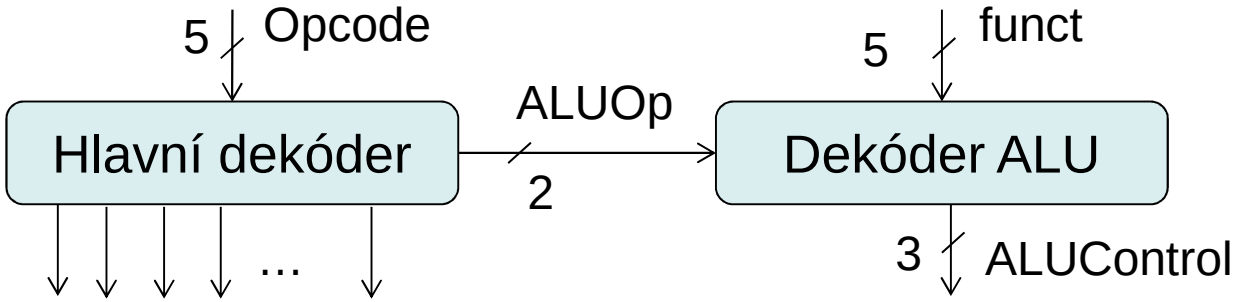
- Tenhle výsledek si, prosím, zapamatujte.
- Budeme s ním pracovat na 5. přednášce.

- Dnes se dále budeme zabývat porozuměním funkci **řadiče (řídicí jednotky)**.

Jedno-cyklový procesor – návrh – řídicí část

R	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**rd**(5), 15:11	**shamt**(5)	**funct**(6), 5:0
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0		
J	**opcode**(6), 31:26	**address**(26), 25:0				

- řídicí signály na základě **opcode** a **funct**



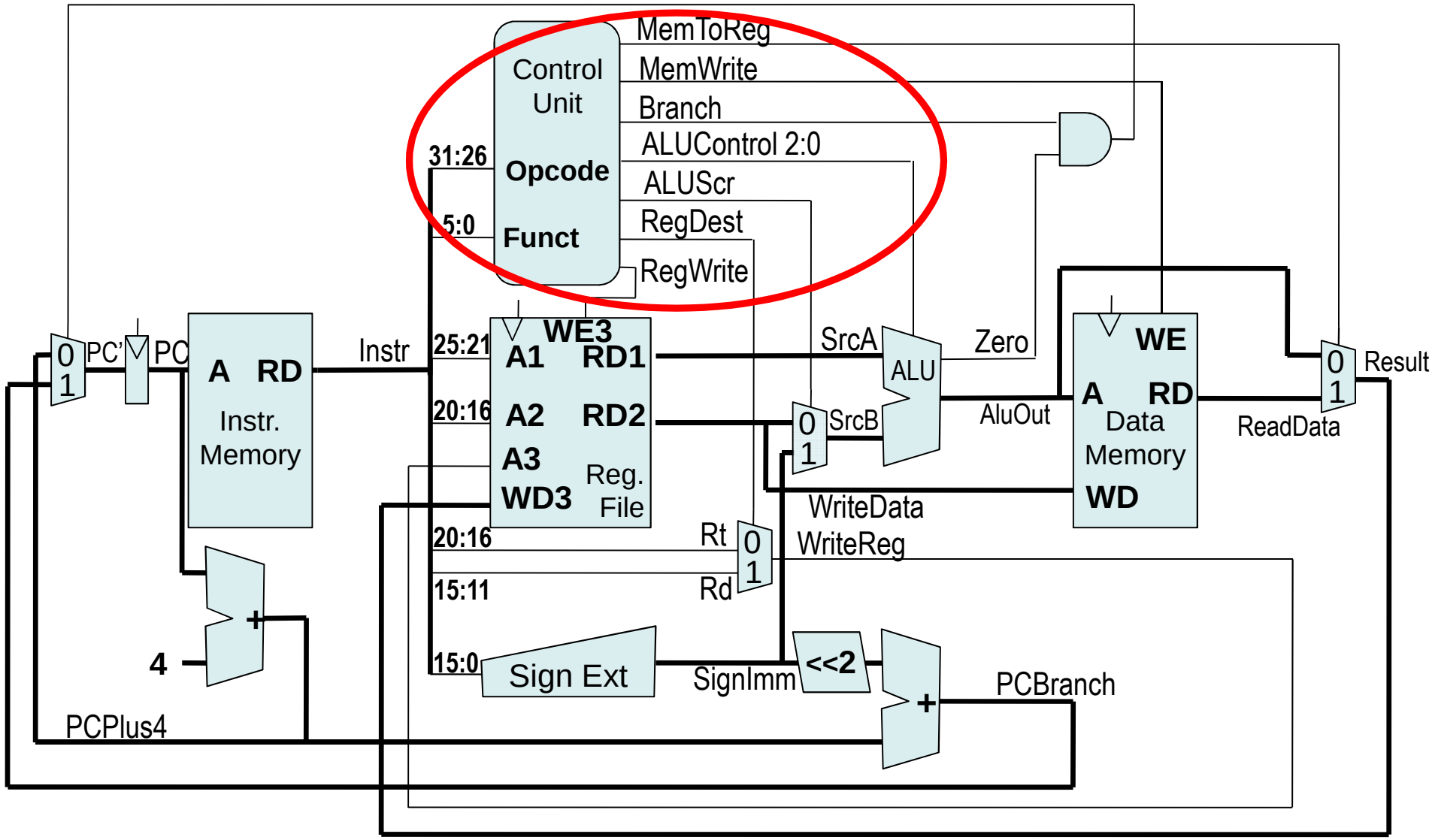
ALUOp	
00	součet
01	rozdíl
10	podle funct
11	-nepoužito-

	Opcode	RegWrite	RegDst	ALUSrc	ALUOp	Branch	Mem Write	MemTo Reg
R typ	000000	1	1	0	10	0	0	0
lw	100011	1	0	1	00	0	0	1
sw	101011	0	X	1	00	0	1	X
beq	000100	0	X	0	01	1	0	X

Řízení ALU (Funkce dekodéru ALU)

ALUOp	Funct	ALUControl
00	X	010 (add)
01	X	110 (sub)
1X	add (100000)	010 (add)
1X	sub (100010)	110 (sub)
1X	and (100100)	000 (and)
1X	or (100101)	001 (or)
1X	slt (101010)	111 (set less than)

Řadič jedno-cyklového procesoru



Co je řadič (řídicí jednotka) procesoru?

- Funkce řadiče: V příslušný časový okamžik generovat řídicí signály a přijímat signály stavové.
- Řadič — jednotka/sekvenční obvod,
  - výstupy: řídicí signály,
  - vstupy: stavové signály.
- Poznámka pro náš specifický případ: náš řadič reaguje např. na stavový signál Zero.

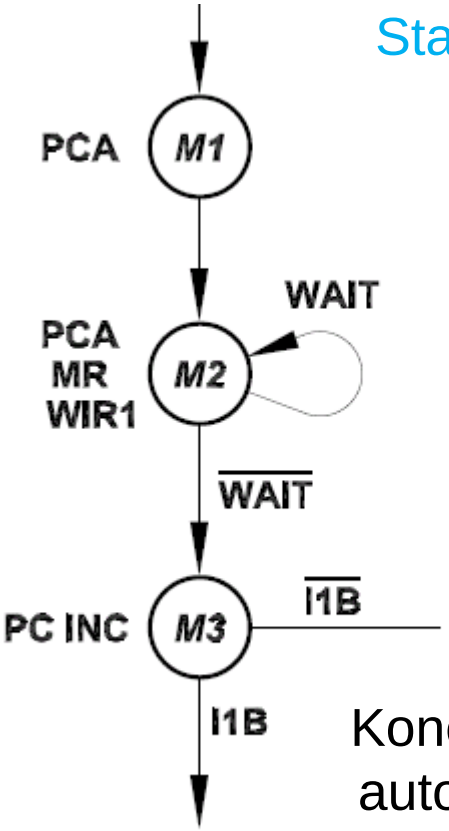
Co je řadič procesoru?

Obecněji: Řadič řídí činnost jednotlivých jednotek, koordinuje jejich aktivity a zajišťuje tok informace mezi nimi. Z hlavní paměti získává instrukce (sekvenci instrukcí), které mají být vykonány. Dekóduje je, a na základě typu instrukce nastaví příslušné hradla a datové cesty aby mohla být instrukce vykonána. Obecně, **funkcí řadiče je generovat sekvenci řídicích signálu různým subsystémům počítače ve správném pořadí tak, aby byly vykonány požadované operace (aritmetické, změny toku programu aj.).** Každý krok v sekvenci kroků vykonávaných řídicí jednotkou v průběhu vykonávání dané instrukce může být definován jako mikro-operace. Mikro-operace je tedy elementární operace (obvykle) vykonaná nad jedním nebo několika registry. Výsledkem mikro-operace je typicky změna obsahu registru (registrů).

Možné realizace řadiče

- Řadič klasický, též obvodově realizovaný, tedy tzv. „obvodový“:
  - řadič s řídicími řetězci,
  - řadič na bázi čítače,
  - jinak navržený.
- Řadič mikroprogramovaný (řízený mikroprogramem):
  - horizontální,
  - vertikální,
  - diagonální.

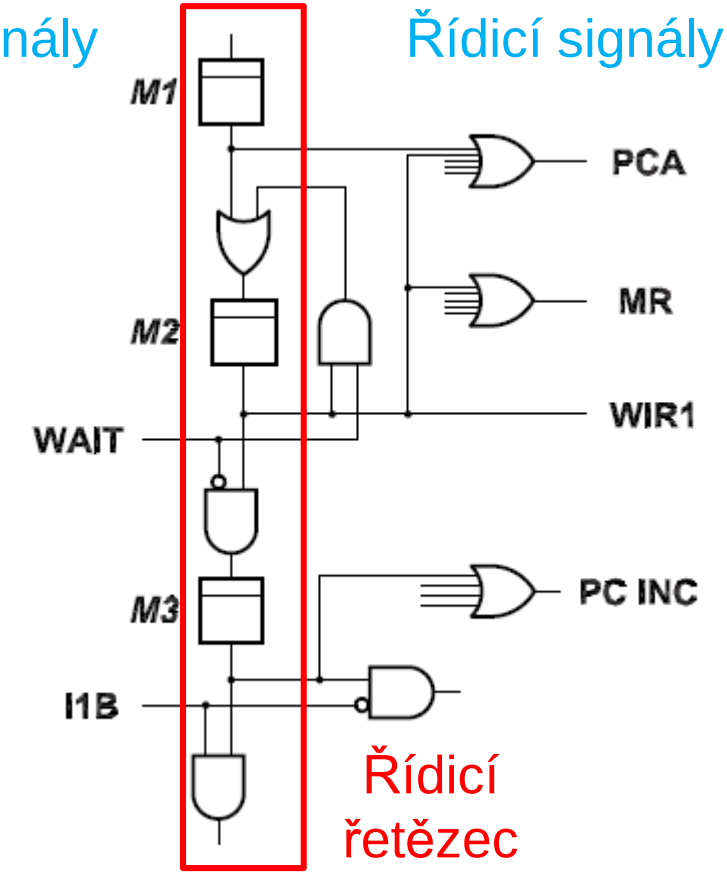
Realizace: řadič s řídicími řetězci



Konečný automat

Stavové signály

Řídicí signály

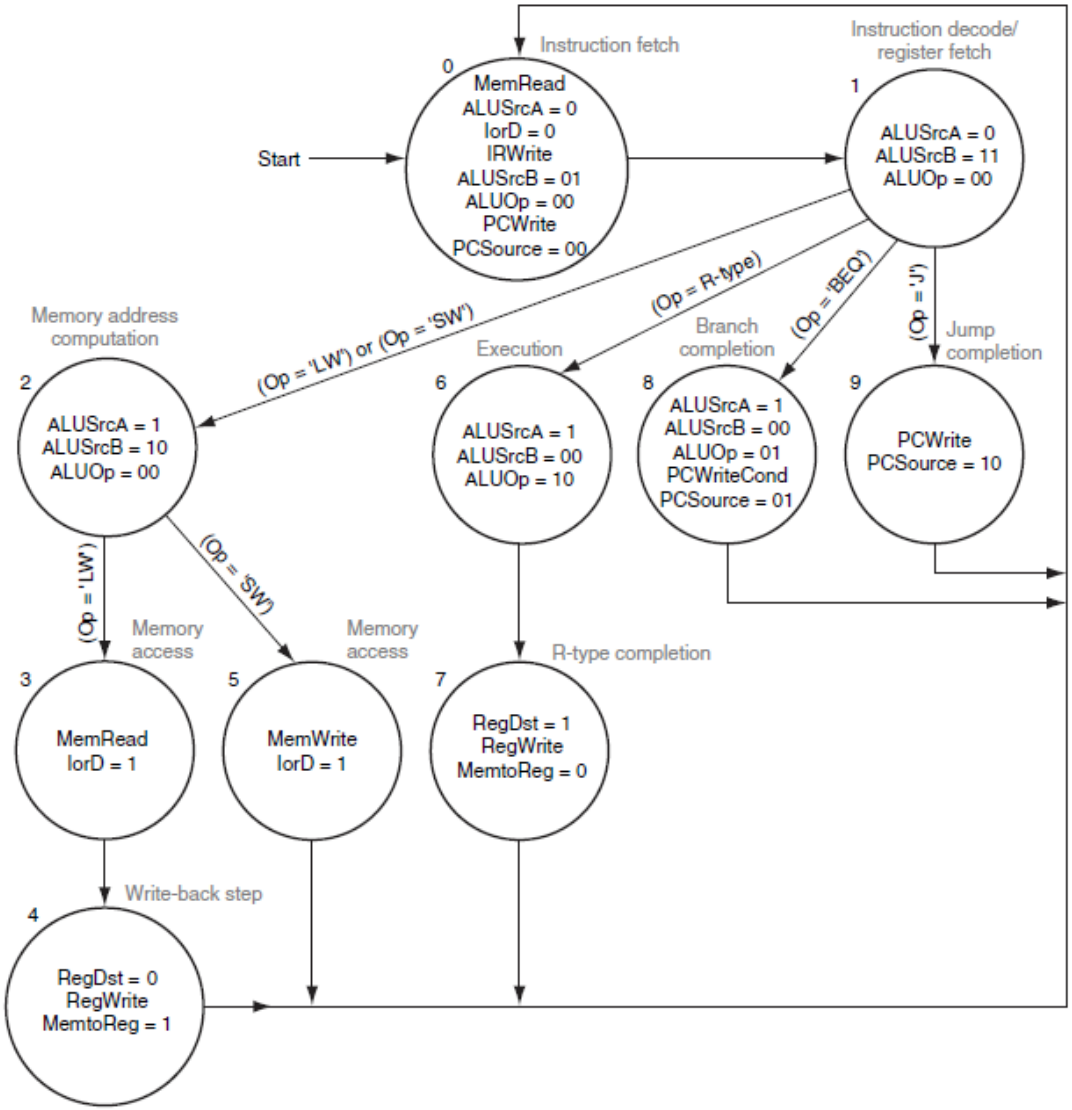


Řídicí řetězec

Důležitá poznámka: označení stavů a názvy řídicích a stavových signálů na obrázku **neodpovídají** našemu specifickému případu!

Činnost řadiče může být popsána konečným automatem (lze relativně snadno popsat jazykem na popis HW – Verilog/VHDL), zejména při multicyklovém vykonávání instrukcí. Jednotlivým stavům automatu přináleží konkrétní nastavení řídicích signálů, přechodům pak podmínky (stavová hlášení, typ instrukce,...), při kterých se mezi těmito stavy přechází.

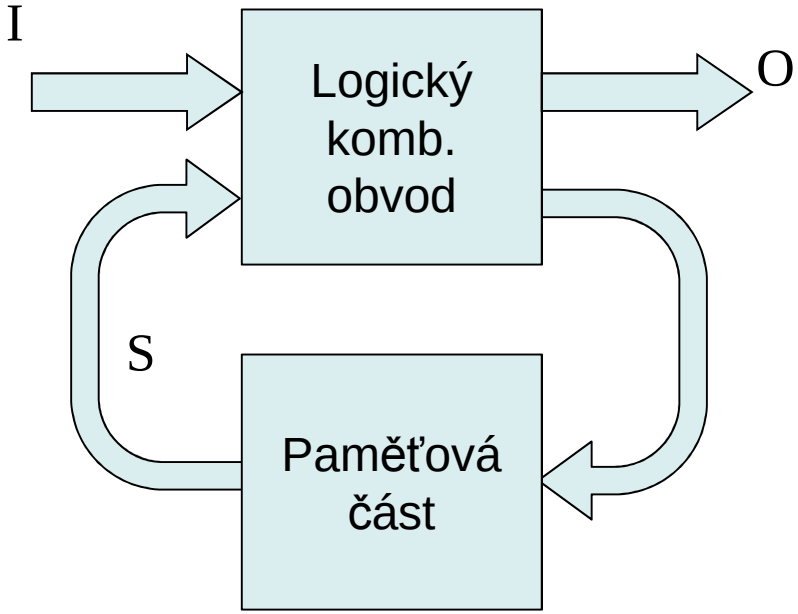
Stavový automat řadiče - příklad



Stavový automat řadiče - příklad

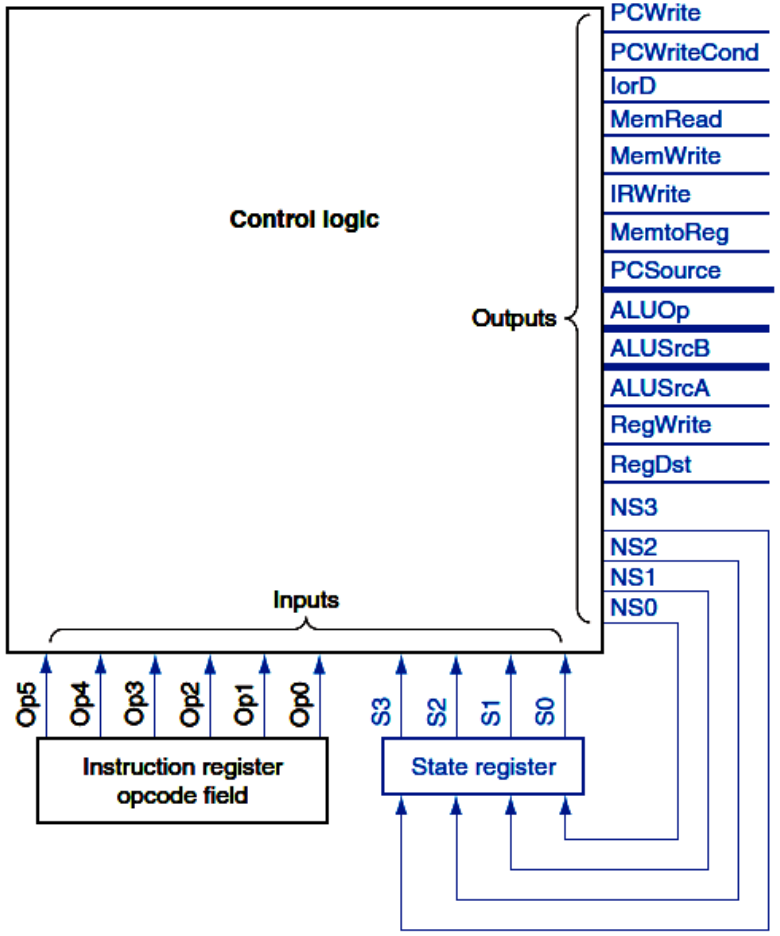
Obecný model logického sekvenčního obvodu (Huffmann)

$O^{t+1}=g(S^t, I^t)$



$S^{t+1}=f(S^t, I^t)$

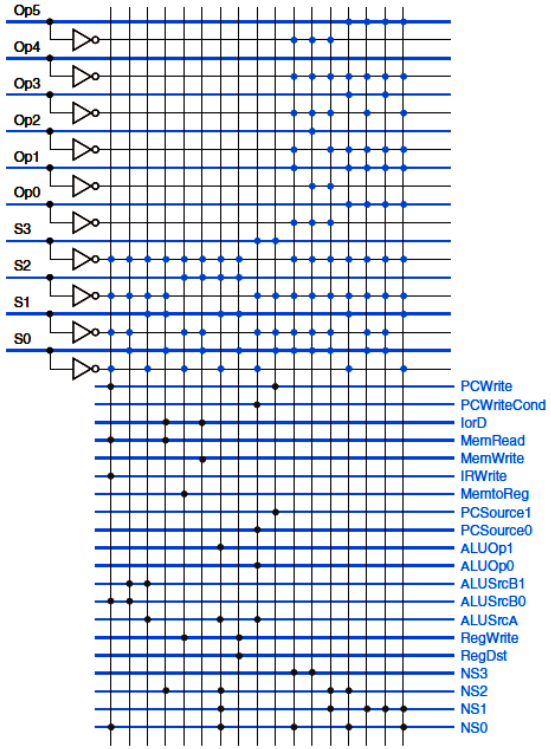
řadič



Stavový automat řadiče - příklad

Implementovat „Control logic“ z předchozího slide prakticky můžeme třema způsoby:

- Jako kombinační logický obvod
- Pomocí paměti ROM
  (vstupy do řadiče budou představovat adresní vodiče pro paměť ROM)
- Pomocí PLA (programmable logic array)



Mikroprogramovaný řadič

Nedílnou součástí *mikroprogramovaného řadiče* je *řídicí paměť* obsahující *mikroinstrukce*, přičemž každá ze strojových instrukcí je provedena pomocí jedné nebo několika jednodušších mikroinstrukcí.

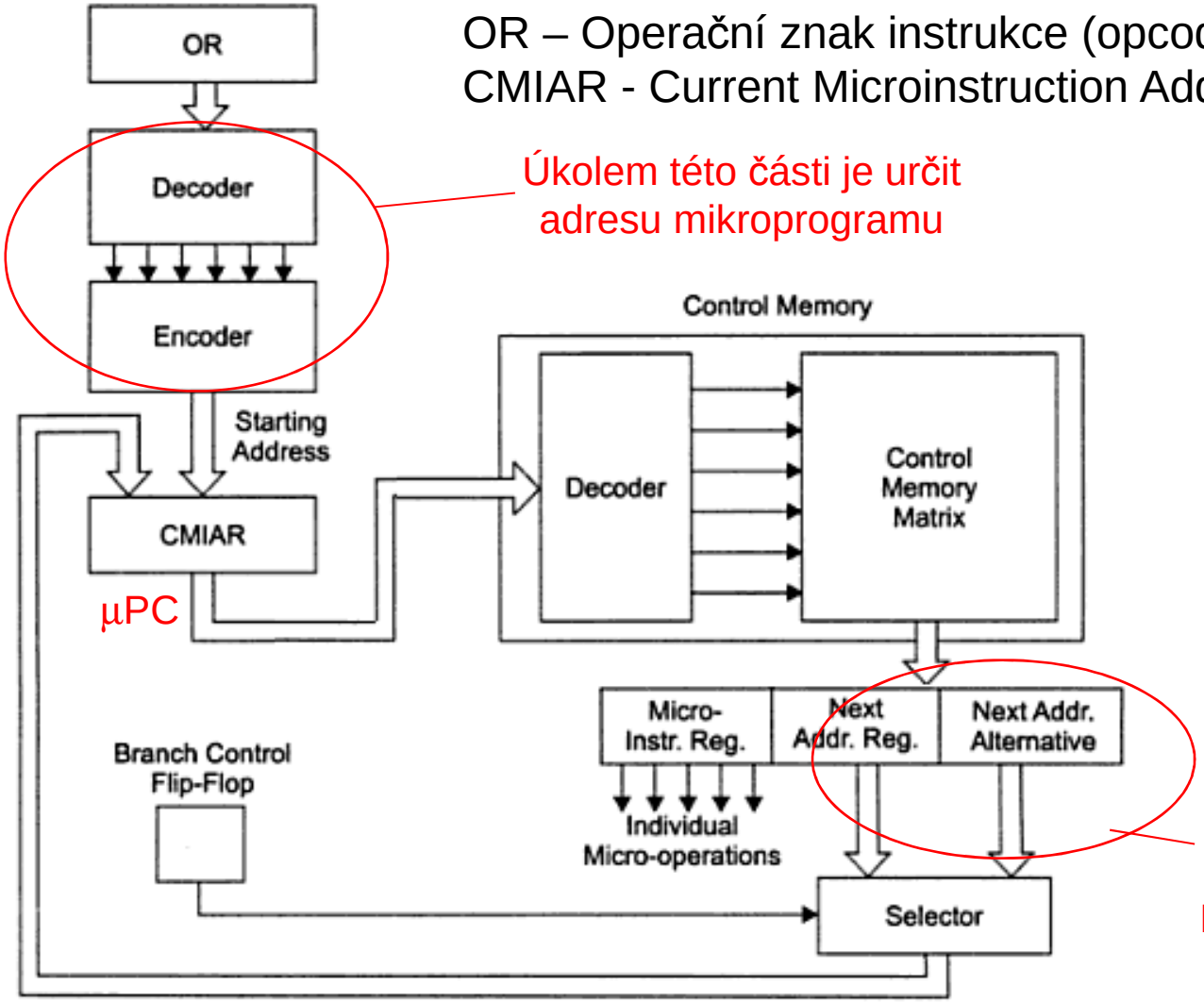
Vstup - kódy instrukcí načtené z operační paměti
Výstup - řídicí signály uvnitř procesorou (ALU, interní sběrnice…), externí signály (paměť,…)

Operační znak instrukce udává adresu první mikroinstrukce z řídicí paměti, která implementuje danou operaci.

Mikroprogram implementuje programátorovi viditelné strojové instrukce (add, sub, lw, xor, jmp…) - ISA

Mikroprogramovaný řadič

OR – Operační znak instrukce (opcode)
CMIAR - Current Microinstruction Address Register (µPC)



Úkolem této části je určit adresu mikroprogramu

Zde může alternativně být tzv. sequencer = Next Address Generator

Mikroprogramovaný řadič

- Příklady mikro-operací:
  - R(MAR) ← R(CIAC)
    Obsah Current Instruction Address Counter do Memory Address Register
  - R(CIAC) ← R(CIAC)+1
    Inkrementace registru CIAC
  - M(MBR) ← M(MAR)
    Výběr z paměti
  - IF F(S) THEN R(A) ← R(MDR)

K realizaci mikro-operace může být zapotřebí jednoho nebo několika řídicích signálů.

Jak může vypadat mikroprogram? Realizace instrukce add

Addr.	Mux Ctl	BrUn	BrCON	Brn≠0	Brn=0	End	PC_{out}	MA_{in}	Other Control Signals	Br Addr.	Actions
100	00	0	0	0	0	0	1	1	• • •	XXX	MA ← PC: C ← PC+4;
101	00	0	0	0	0	0	0	0	• • •	XXX	MD ← M[MA]: PC ← C;
102	01	1	0	0	0	0	0	0	• • •	XXX	IR ← MD; μPC ← PLA;
200	00	0	0	0	0	0	0	0	• • •	XXX	A ← R[rb];
201	00	0	0	0	0	0	0	0	• • •	XXX	C ← A + R[rc];
202	11	1	0	0	0	1	0	0	• • •	100	R[ra] ← C; μPC ← 100;

μinstrukce

μoperace

0. PC_{out}, MAR_{in}, Read, Zero A, Set Carry-In, Add, Z_{in}
1. Z_{out}, PC_{in}, Wait MFC
2. MDR_{out}, IR_{in}
3. $AddressFieldOfIR_{out}$, MAR_{in}, Read
4. $R1_{out}$, Y_{in}, Wait MFC
5. MDR_{out}, Add, Z_{in}, Set CC
6. Z_{out}, $R1_{in}$, End

John Franco: What is Microprogramming and Why Should We Know About it?
http://gauss.ececs.uc.edu/Courses/c4029/exams/Spring2013/Review/microcode.pdf

Mikroprogramovaný řadič

Organizace mikroinstrukcí – horizontální, vertikální a diagonální

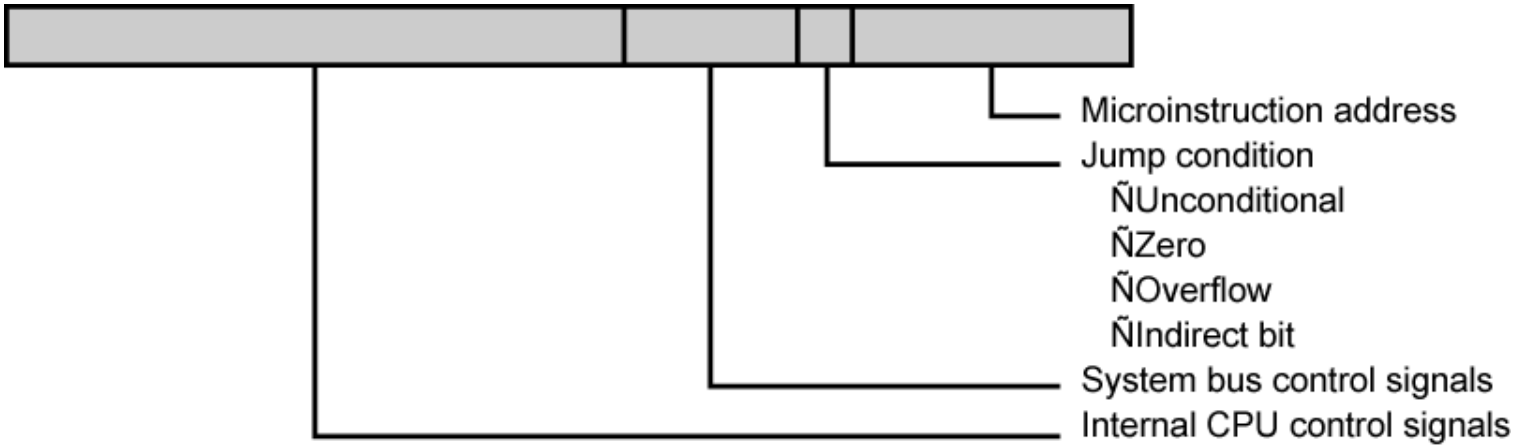
Vertikální:

- Pokud budeme uvažovat N typů mikrooperací, postačuje $\log_2 N$ bitů pro specifikaci mikrooperace. Nicméně potřebujeme dekodéry pro generování řídicích signálů. Jedna mikroinstrukce pak vykoná pouze jednu mikrooperaci. Kódování po skupinách -> Jedna mikroinstrukce pak může vykonat několik mikrooperací.

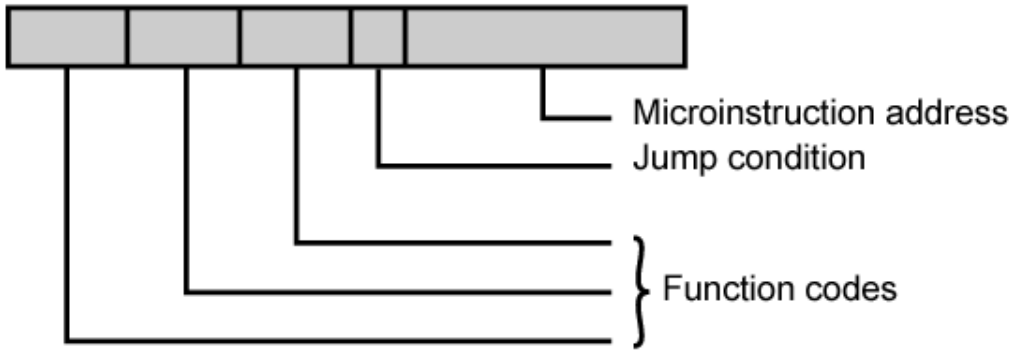
Horizontální:

- Každý bit mikroinstrukce může reprezentovat specifickou mikrooperaci. Takže pro N typů mikroinstrukcí potřebujeme N bitů. Můžeme vykonat libovolnou množinu mikrooperací paralelně v jediné mikroinstrukci.

Mikroprogramovaný řadič



(a) Horizontal microinstruction



(b) Vertical microinstruction

Diagonální:

Kompromis. Některé bloky zakódovány „horizontálně“ a jiné „vertikálně“. Horizontální formát má bity pro jednotlivé signály přímo uloženy v mikroinstrukci (bez nutnosti dalšího dekódování). Vertikální formát kóduje větší skupinu navzájem se vylučujících signálů (aktivní je pouze jeden) společně do jednoho bloku.

Poznámka k terminologii: **Horizontální** vs. **vertikální** primárně slouží k rozlišení toho, zda mikroinstrukce přímo řídí součásti CPU (horizontální), nebo potřebuje další dekódovací stupně (vertikální).

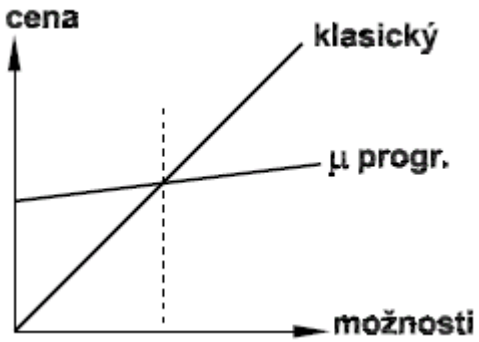
Mikroprogramovaný řadič je vlastně počítačem v počítači.

Závěr k mikroprogramovaným řadičům:

Mikroprogram je vlastně vrstvou mezi strojovými instrukcemi a samotným HW. Slouží k implementaci strojových instrukcí v řídicí jednotce CPU, GPU, řadičů disků, řadičů síťových rozhraní, atd. Pomáhá oddělit strojové instrukce od elektronických obvodů, což rovněž pomáhá implementovat komplexní instrukce bez nárůstu složitosti HW. Programování pomocí mikroinstrukcí se označuje mikroprogramování. Napsaný kód, který je uložen (ROM, PLA, flash) uvnitř řadiče je pak mikroprogram. **Při návrhu řadičů moderních procesorů využívajících zřetězení použití mikroprogramovaného řadiče není populární**, mj. i z důvodu samotné podstaty sekvenčního vykonání mikroprogramu.

Mikroprogramovaný vs. klasický řadič - srovnání

- Rychlost - klasický je rychlejší.
- Cena – levnější je
  - Klasický, ale jen v případě velmi jednoduché variantě.
  - Ve složitější je jím řadič mikroprogramovaný.



- Flexibilita – mikroprogramovaný.
- Změna mikroprogramu – změna chování procesoru.
- Řídicí paměť
  - ROM – pevné mikroinstrukce
  - RWM - µprogramovatelný procesor, možná emulace jiné instrukční sady.
- Mikroprogramovaný řadič – neefektivní pro zřetězené procesory (alespoň jako centralizovaný) – každý stupeň vykonává jinou instrukci. Bylo by potřeba zajistit správné provedení všech instrukcí napříč stupni spolu s řešením hazardů.