Aritmetické a logické operace (posuvy a rotace, sčítání)

Zdeněk Vašíček, Vojtěch Mrázek

2025

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

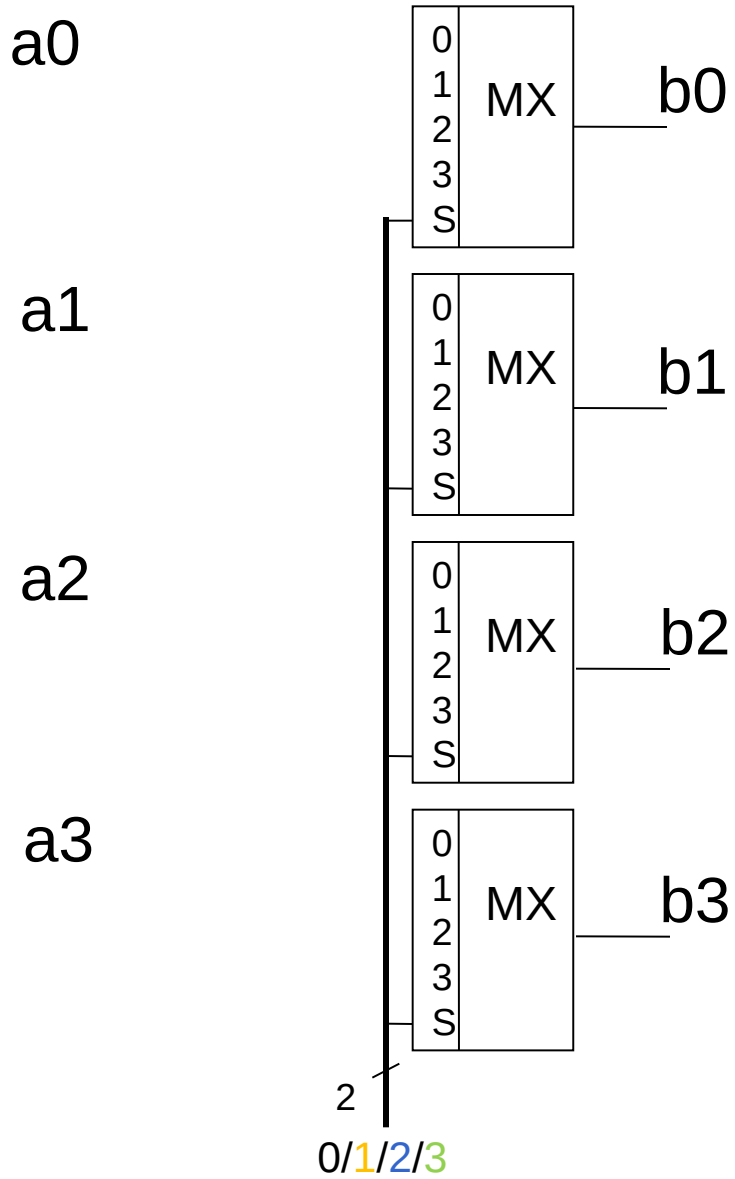
Rotace a posuvy

- Bitové **rotace** a **posuvy** se implementují výhradně pomocí pomocí tzv. Barrel Shifter obvodu, což je kombinační obvod, který je schopen vyčíslit výsledek operace během jediného taktu.
- Počet taktů při použití posuvného registru by závisel na počtu bitů o kolik posuv realizujeme.

- Barrel Shifter je obvod složený z vhodně uspořádaných multiplexorů, používá se označení BS(n_1,n_2,…,n_N), kde N odpovídá počtu stupňů (zpoždění) a n_i šířce multiplexorů v jednotlivých stupních.
- BS(B), kde B je počet bitů operandu odpovídá B multiplexorům s B-bitovým vstupem (velmi drahé).

4-bitový válcový posouvač pro rotace vpravo

nejjednodušší varianta - BS(4)



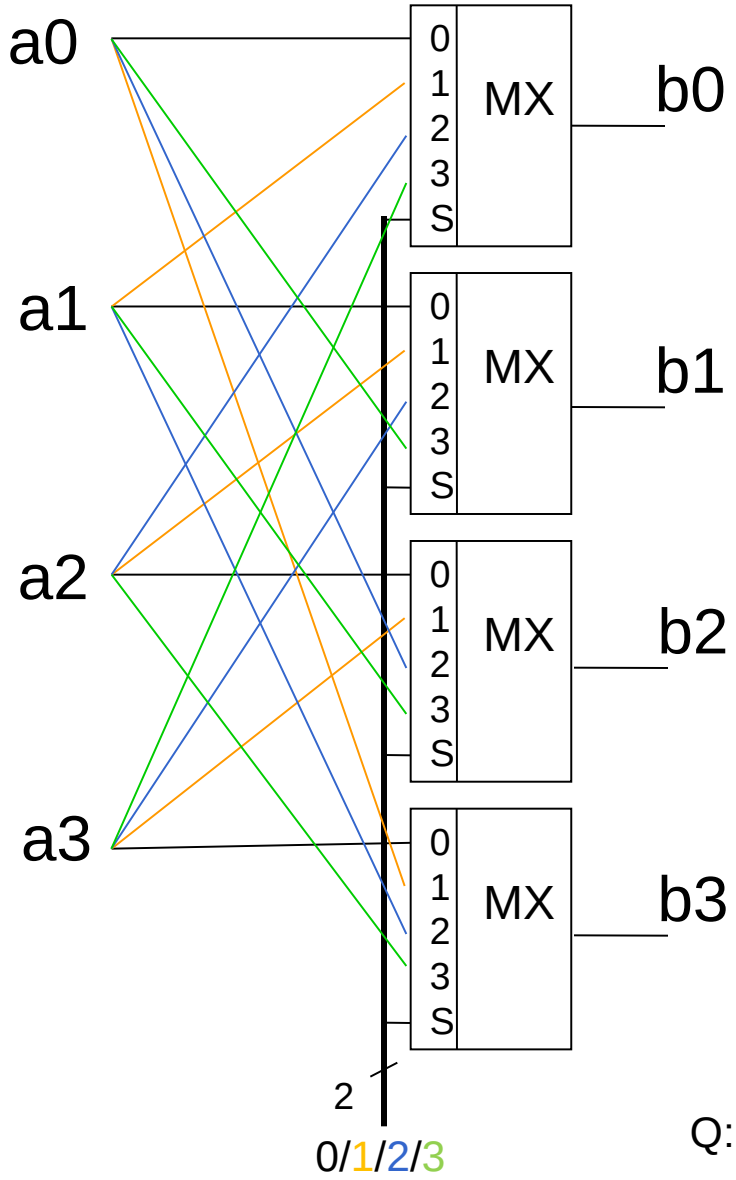
$b_3\ b_2\ b_1\ b_0$

↓ ↓ ↓ ↓

$a_3\ a_2\ a_1\ a_0$ 0 bit

4-bitový válcový posouvač pro rotace vpravo

nejjednodušší varianta - BS(4)

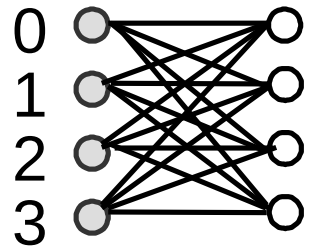


$b_3\ b_2\ b_1\ b_0$

↓ ↓ ↓ ↓

$a_3\ a_2\ a_1\ a_0$	0 bit
$a_0\ a_3\ a_2\ a_1$	1 bit
$a_1\ a_0\ a_3\ a_2$	2 bit
$a_2\ a_1\ a_0\ a_3$	3 bit

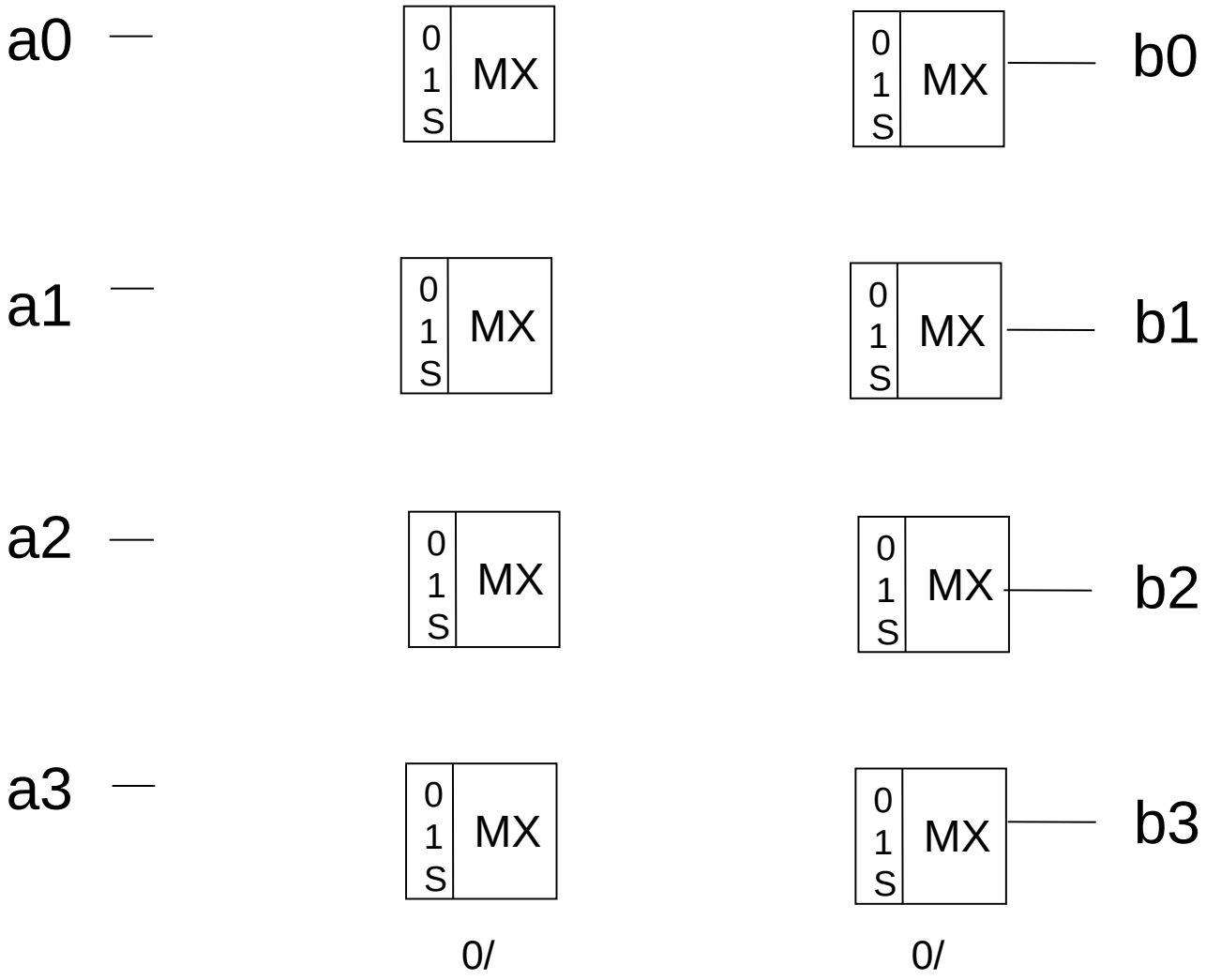
Schematické znázornění:



Q: Jak by se realizoval BS pro posuv nikoliv rotaci?

4-bitový válcový posouvač pro rotace vpravo

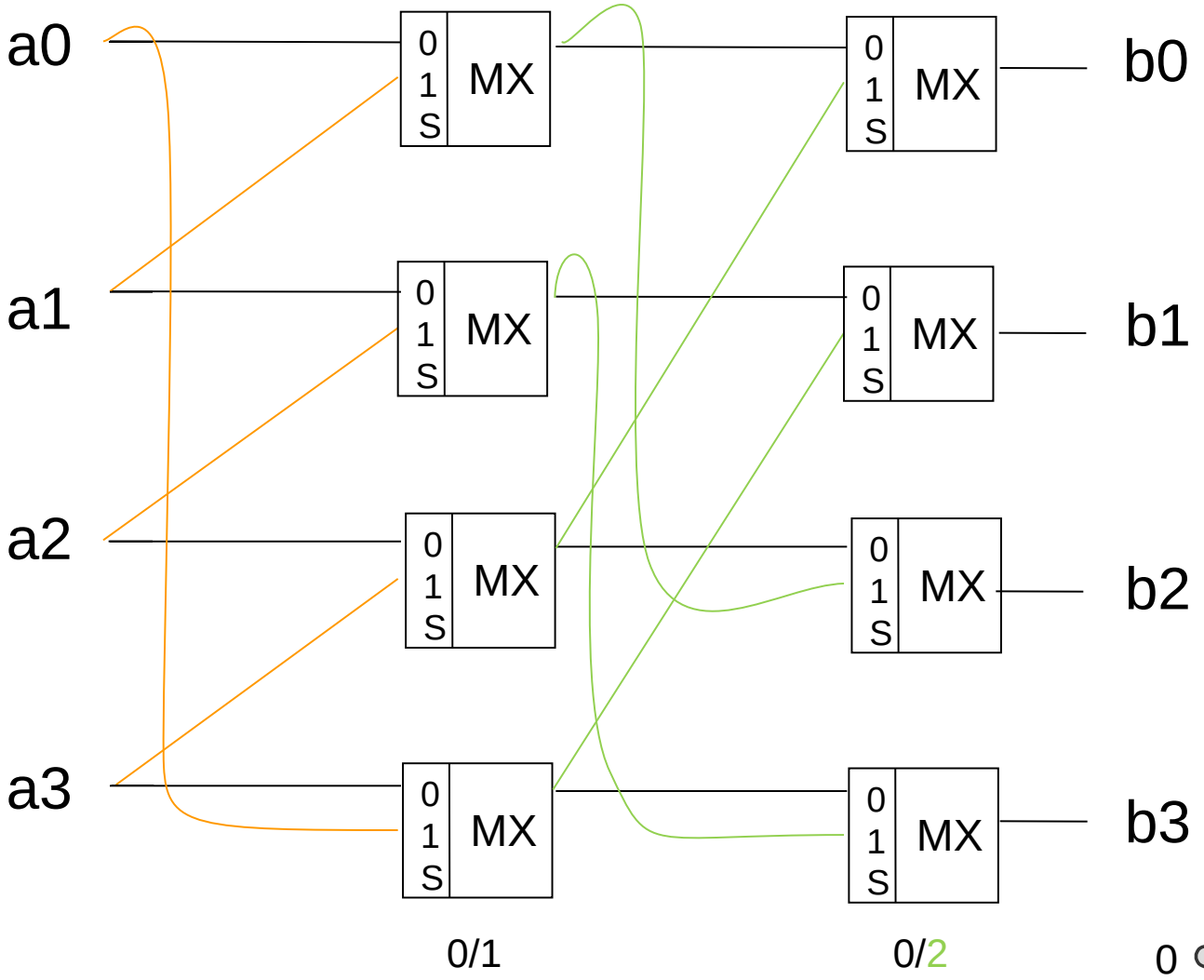
varianta BS(2, 2)

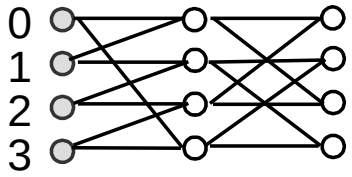


Q: Jak by se realizoval BS pro posuv nikoliv rotaci?

4-bitový válcový posouvač pro rotace vpravo

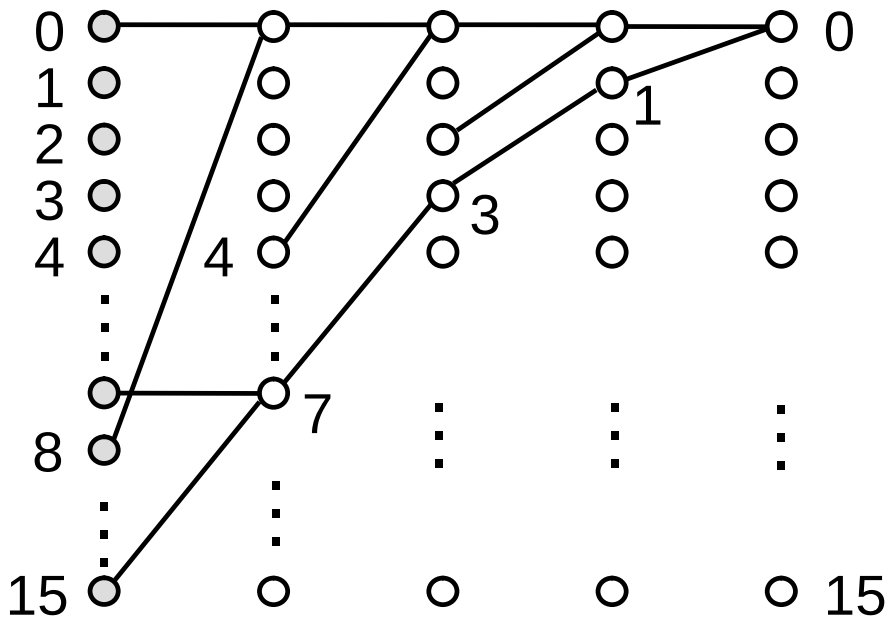
varianta BS(2, 2)





Q: Jak by se realizoval BS pro posuv nikoliv rotaci?

16-bitový válcový posouvač pro rotace vpravo BS(2,2,2,2)



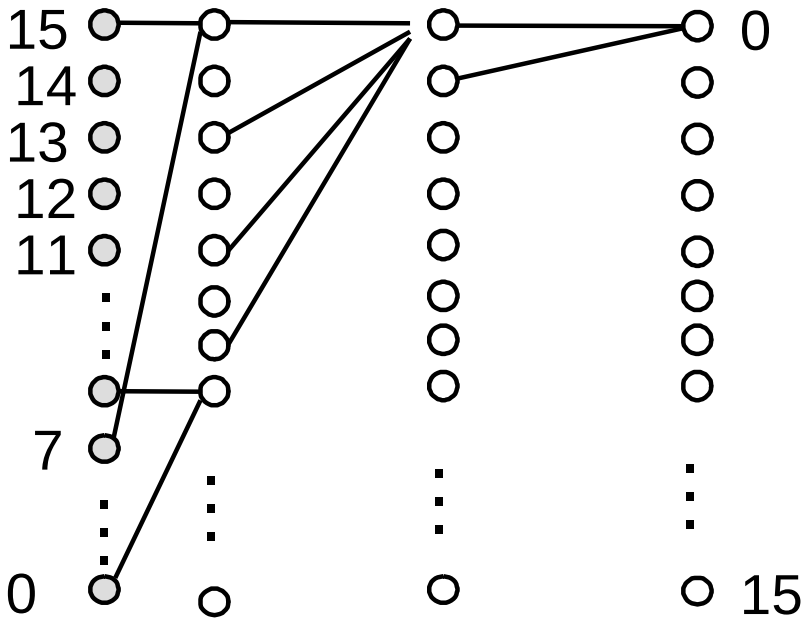
rotace:	0/8	0/4	0/2	0/1	celkem: 0÷15
	Δ	Δ	Δ	Δ	$T = 4\Delta$

Q: Jak bude vypadat BS(2,4,2)?

Q: Jak by se implementoval BS pro rotaci vlevo?

16-bitový válcový posouvač pro rotace vlevo

BS(2,4,2)

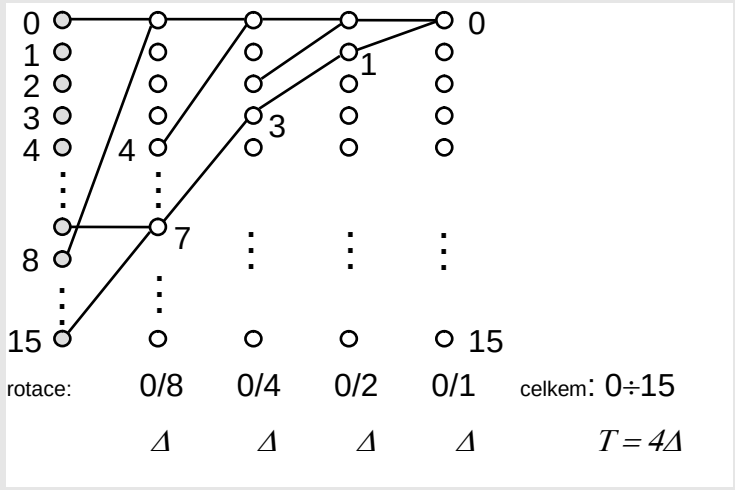


rotace:	0/8	0/2/4/6	0/1	celkem: 0÷15
	Δ	Δ	Δ	$T = 3\Delta$

W-bitový válcový posouvač ve VHDL
BS(2,...,2) rotace doprava

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use work.math_pack.all;

entity BS is
  generic ( W : natural := 16 );
  port (
    DIN   : in  std_logic_vector( W-1 downto 0);
    DOUT  : out std_logic_vector( W-1 downto 0);
    SHIFT : in  std_logic_vector( log2(W)-2 downto 0)
  );
end BS;

architecture rtl of BS is
  constant STEPS : natural := SHIFT'length;
  type tarray is array(STEPS downto 0) of std_logic_vector(W-1 downto 0);
  signal level, unshifted, shifted : tarray;
begin
  level(0) <= DIN;
  DOUT <= level(STEPS);

  levelgen: for i in 1 to STEPS generate
      unshifted(i) <= level(i-1);
      shifted(i) <= level(i-1)(2**(STEPS-i)-1 downto 0) &
                    level(i-1)(W-1 downto 2**(STEPS-i));
      level(i) <= shifted(i) when SHIFT(STEPS-i) = '1' else unshifted(i);
   end generate;
end rtl;
```

i	shifted	shift
1	(7…0) & (15…8)	0/8
2	(3…0) & (15…4)	0/4
3	(1…0) & (15…2)	0/2
4	(0…0) & (15…1)	0/1



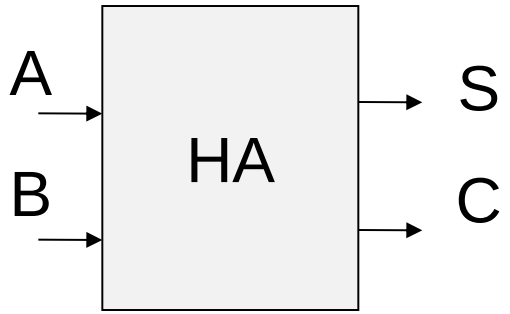
W-bitový válcový posouvač ve VHDL
BS(2,...,2)

```
architecture rtl of BS is
    constant STEPS : natural := SHIFT'length;
    type tarray is array(STEPS downto 0) of std_logic_vector(W-1 downto 0);
    signal level, unshifted, shifted : tarray;
    signal zeros: std_logic_vector(W-1 downto 0);
  begin
    level(0) <= DIN;
    DOUT <= level(STEPS);
    zeros <= (others => '0');

    levelgen: for i in 1 to STEPS generate
        unshifted(i) <= level(i-1);

        --ROTACE DOPRAVA
        shifted(i) <= level(i-1)(2**(STEPS-i)-1 downto 0) &
                      level(i-1)(W-1 downto 2**(STEPS-i));
        --ROTACE DOLEVA
        shifted(i) <= level(i-1)(W - 2**(STEPS-i)-1 downto 0) &
                      level(i-1)(W-1 downto W-2**(STEPS-i));
        --POSUN DOPRAVA
        shifted(i) <= zeros(2**(STEPS-i)-1 downto 0) &
                      level(i-1)(W-1 downto 2**(STEPS-i));
        --POSUN DOLEVA
        shifted(i) <= level(i-1)(W - 2**(STEPS-i)-1 downto 0) &
                      zeros(W-1 downto W-2**(STEPS-i));

        level(i) <= shifted(i) when SHIFT(STEPS-i) = '1' else unshifted(i);
     end generate;
  end rtl;
```

Sčítačky

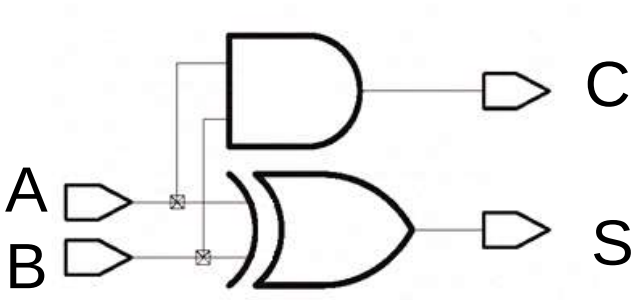
poloviční sčítačka (Half Adder)

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



S – jednobitový součet
C – výstupní přenos

```
S <= A xor B;
C <= A and B;
```



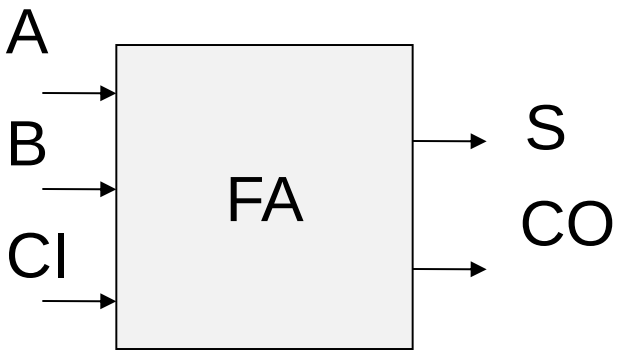
Sčítačky

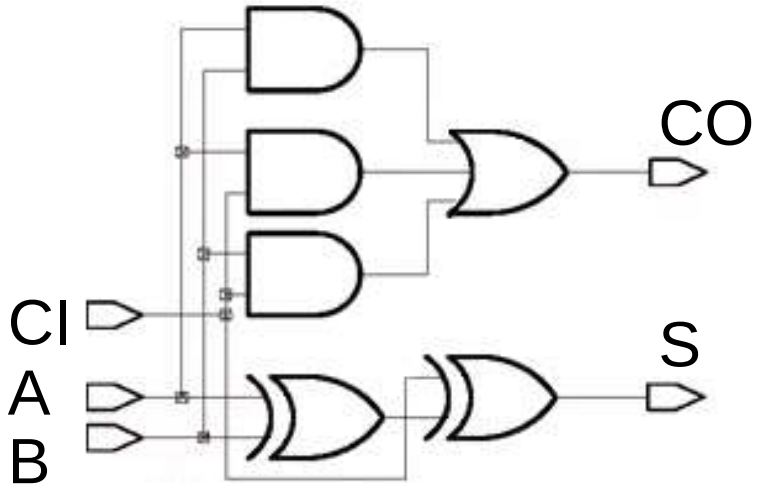
úplná sčítačka (Full Adder)

A	B	CI	S	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

CI – vstupní přenos
CO – výstupní přenos

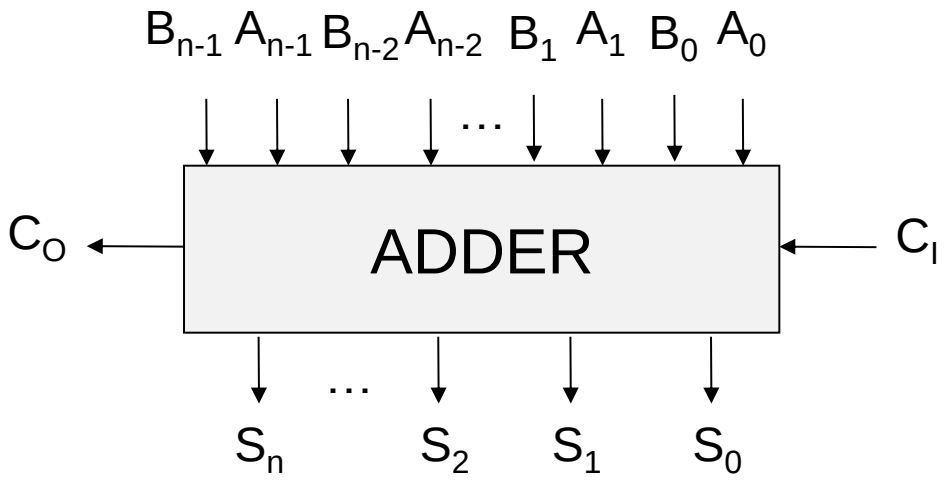
```
S  <= A xor B xor CI;
CO <= (A and B) or
      (A and CI) or
      (B and CI);
```





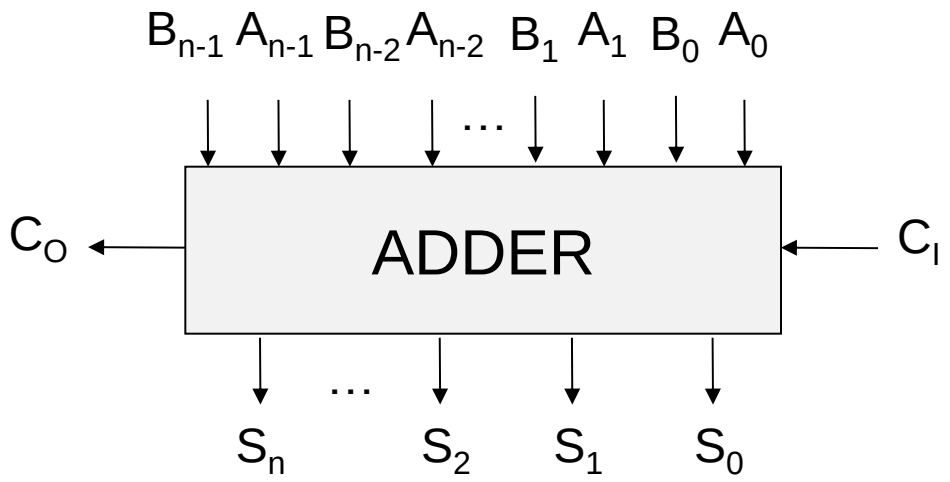
Sčítačky
vícebitové sčítačky

- Existuje několik architektur lišících se v zpoždění a ploše na čipu (od nejpomalejší k nejrychlejší variantě):
  - sčítačka s postupným přenosem (Ripple-Carry Adder)
  - sčítačka s výběrem přenosu (Carry-Select Adder)
  - CLA sčítačka (Carry-lookahead Adder)



Sčítačka s postupným přenosem

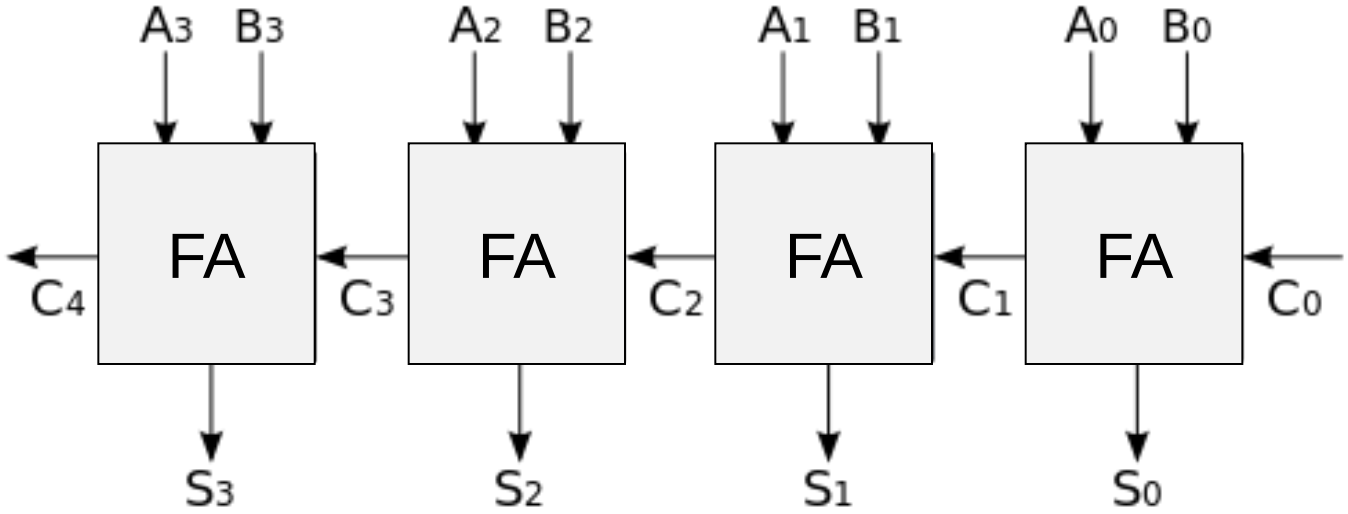
- Lze zkonstruovat pomocí 1-bitových úplných sčítaček
- Zpoždění: $2n\Delta$, kde Δ je zpoždění log. členu, n je počet bitů



Sčítačka s postupným přenosem

- Lze zkonstruovat pomocí 1-bitových úplných sčítaček
- Zpoždění: $2n\Delta$, kde Δ je zpoždění log. členu, n je počet bitů

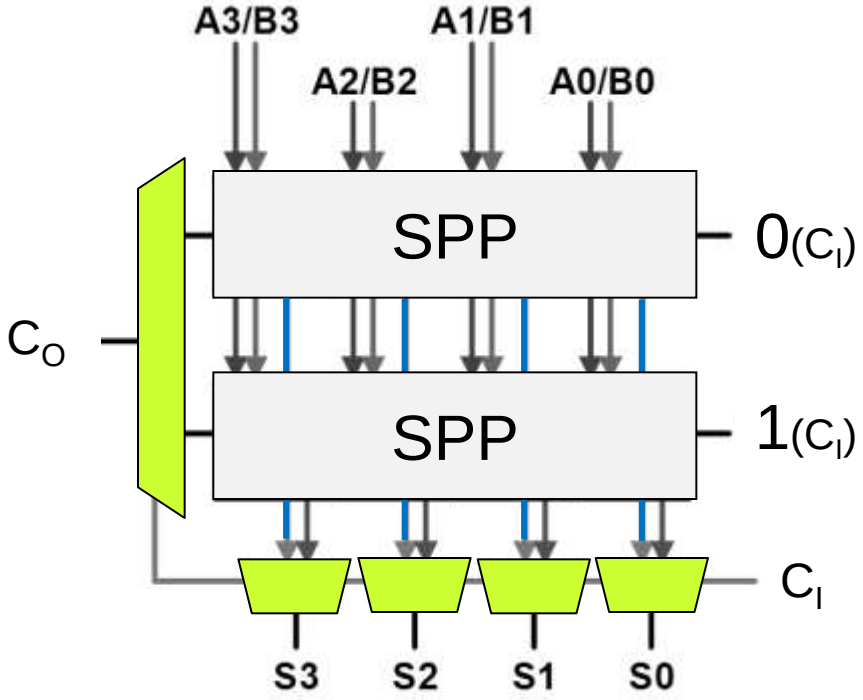
Příklad: 4-bitová sčítačka



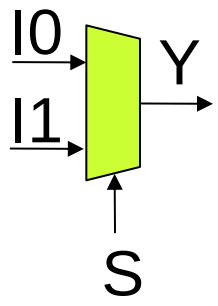
Zpoždění: 8Δ

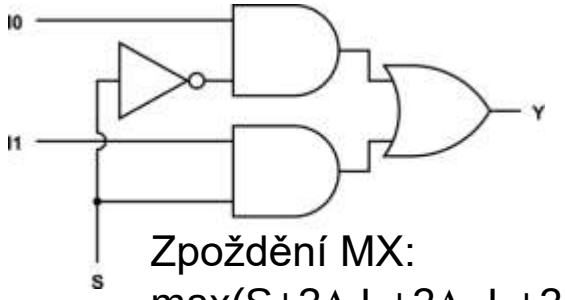
Sčítačka s výběrem přenosu

- Využívá stavební blok skládající se ze dvou sčítaček s postupným přenosem (SPP) a multiplexorů (MX).



S	I0	I1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1





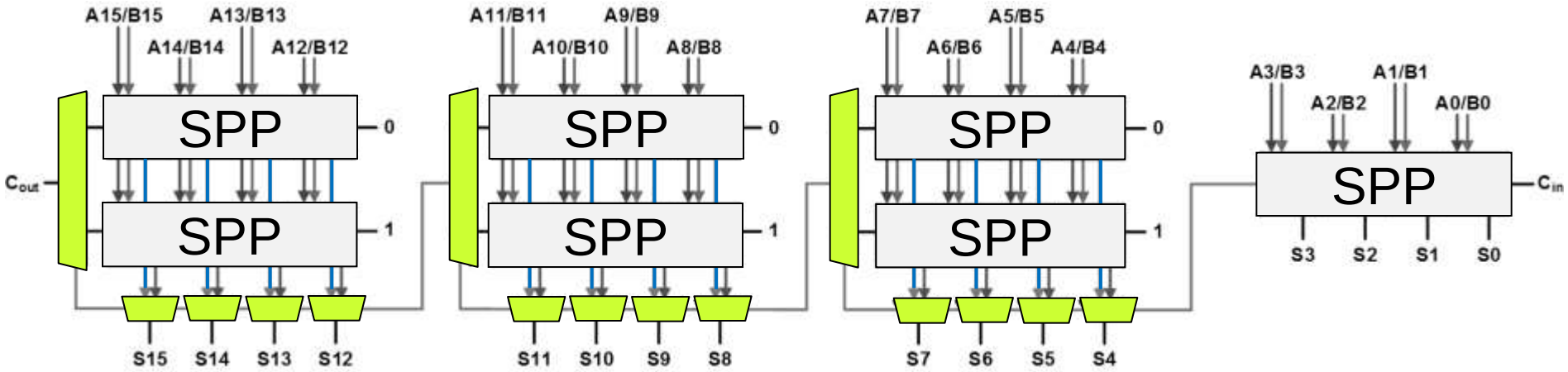
Zpoždění: 10Δ, kde Δ je zpoždění log. členu

Zpoždění MX: $\max(S+3\Delta, I_0+2\Delta, I_1+2\Delta)$

Sčítačka s výběrem přenosu

- Vícebitová varianta sčítačky s výběrem přenosu
- Zpoždění: $O(\sqrt{n})$

Příklad: 16-bitová sčítačka

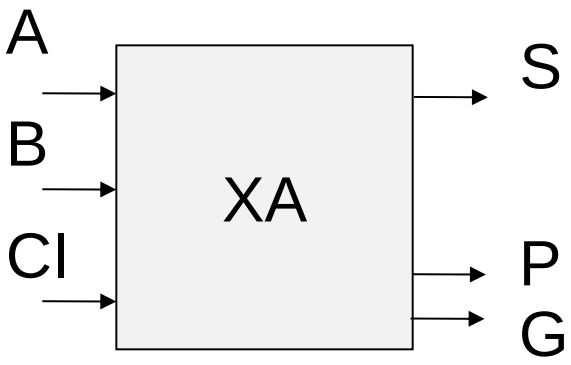


Zpoždění: 17Δ, kde Δ je zpoždění log. členu
16-bit. SPP má zpoždění 32Δ

Sčítačky

rozšířená sčítačka (Extended Adder)

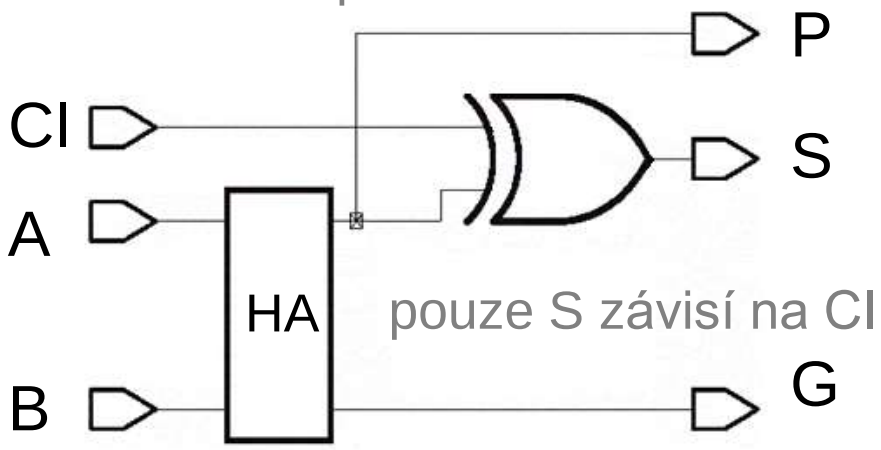
A	B	CI	S	CO	P	G
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	1	0	1	0
0	1	1	0	1	1	0
1	0	0	1	0	1	0
1	0	1	0	1	1	0
1	1	0	0	1	0	1
1	1	1	1	1	0	1



P – šířený (propagate) přenos; tj. přenos sčítačkou prochází

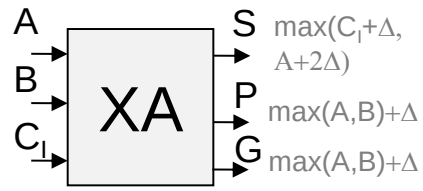
G – generovaný přenos

```
S  <= A xor B xor CI;
P  <= A xor B;
G  <= A and B;
CO <= (P and CI) or G;
```

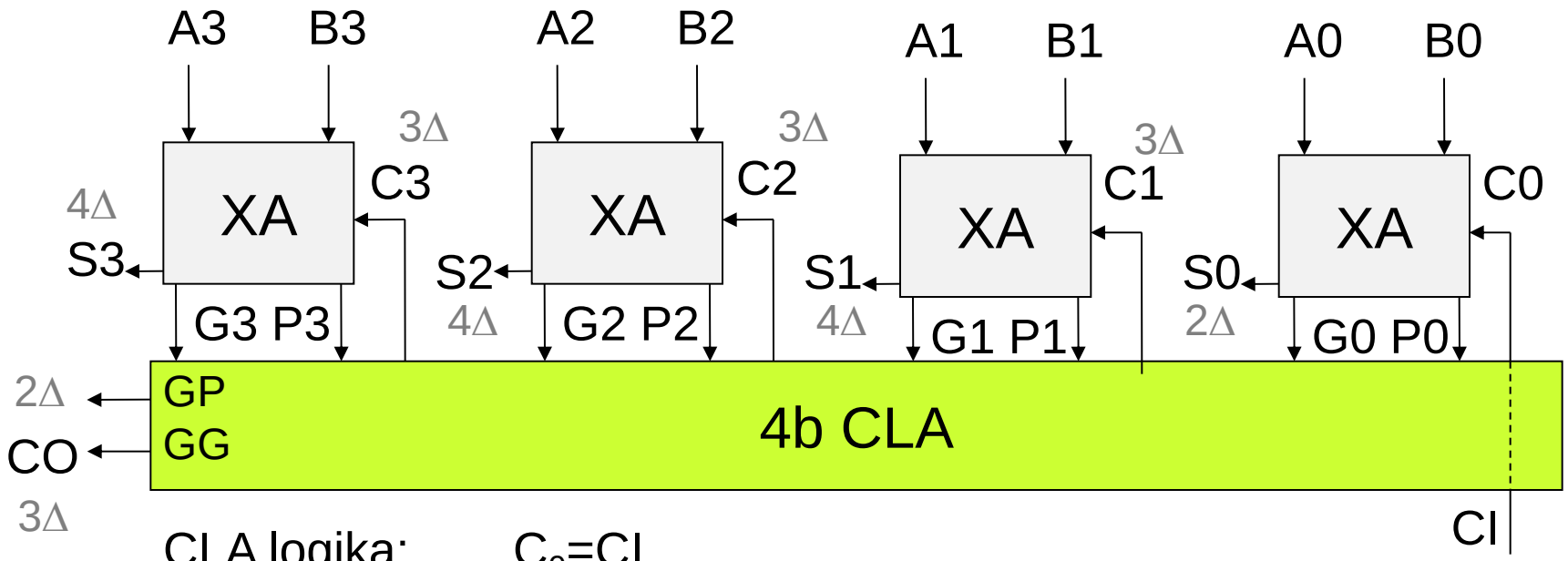


CLA sčítačka

- Lze zkonstruovat pomocí 1-bitových rozšířených sčítaček a CLA logiky (obvod se zpožděním 2Δ)
- Zpoždění: 4Δ, kde Δ je zpoždění log. členu



Příklad: 4-bitová sčítačka



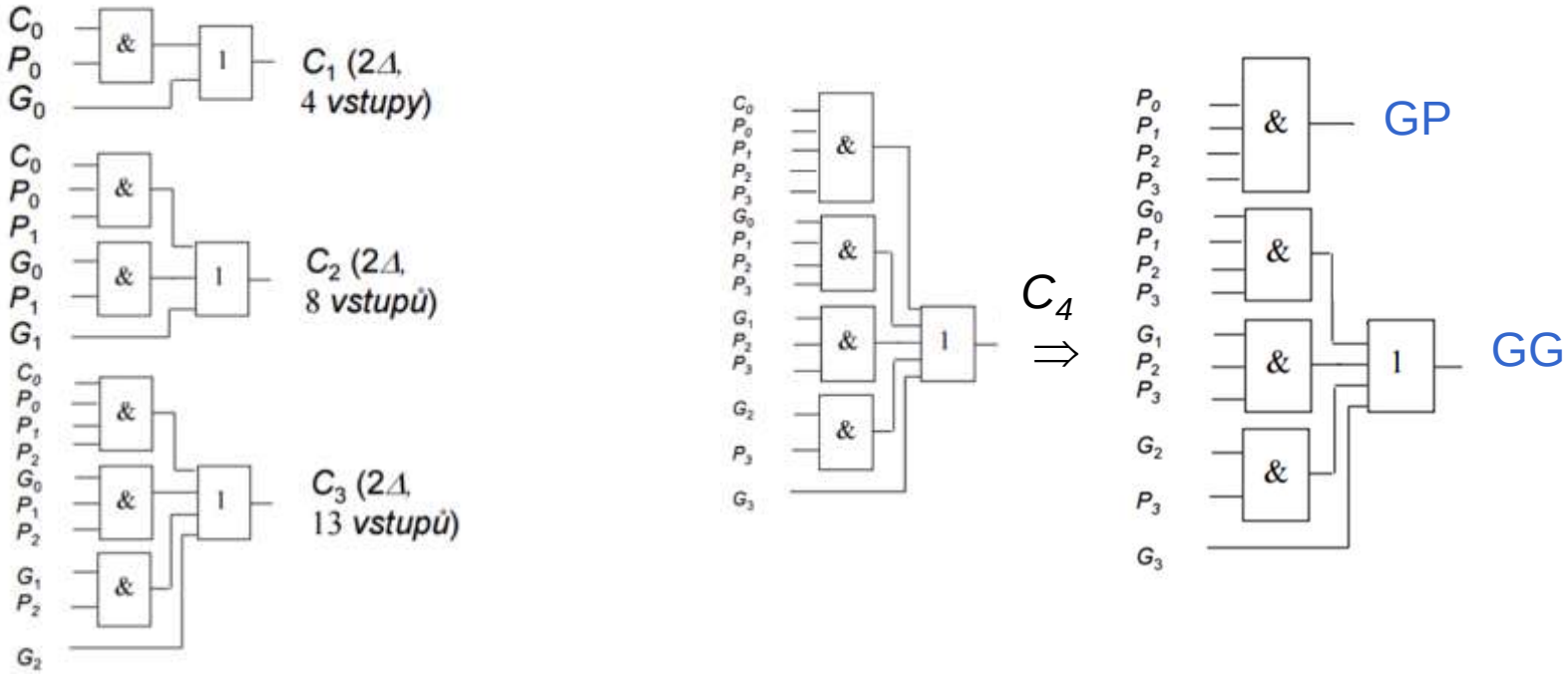
CLA logika: C_0=CI

$C_i=(C_0$ and $P_{i-1})$ or G_{i-1} pro $1<i\leq N$, kde G_{N-1}= CO

CLA sčítačka

složitost CLA logiky

- CLA logika jako obvod se zpožděním 2Δ

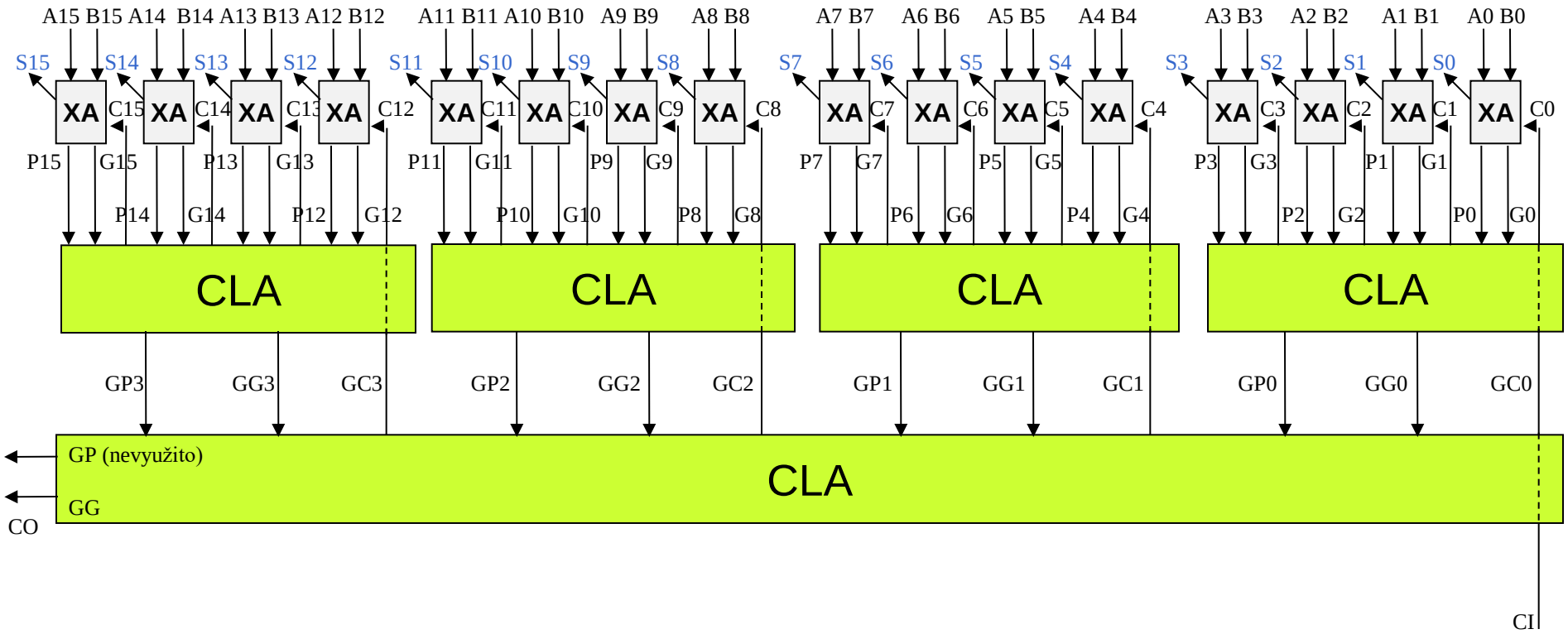


- Plochy CLA jednotky závisí na počtu bitů kvadraticky. Plocha je neúnosně velká již pro 16-bitovou CLA. Řešením je zavést víceúrovňovou CLA logiku (sestavit CLA do stromu).

CLA sčítačka

se stromovou architekturou

Příklad: 16-bitová sčítačka



Implementace stromové CLA sčítačky 1/4

```
entity CLA16 is
port (
    A, B : in  std_logic_vector(15 downto 0);
    CI   : in  std_logic;
    S    : out std_logic_vector(15 downto 0);
    CO   : out std_logic);
end CLA16;

architecture RTL of CLA16 is

component XA
port (
    A, B, CI : in std_logic;
    S, P, G  : out std_logic);
end component;

component CLAU
port (
    P, G   : in  std_logic_vector(3 downto 0);
    CI     : in  std_logic;
    CO     : out std_logic_vector(3 downto 0);
    GP, GG : out std_logic);
end component;
```

Implementace stromové CLA sčítačky 2/4

```
constant N : integer := 4; -- number of 4 bits
signal P, G, C : std_logic_vector(N*4-1 downto 0);
signal GP,GG,GC : std_logic_vector(N-1 downto 0);

begin

 fagen : for i in 0 to N*4-1 generate
   xau : XA port map (A => A(i), B => B(i), CI => C(i),
       S => S(i), P => P(i), G  => G(i)
   );
 end generate;

 claugen : for i in 0 to N-1 generate
   clau : CLAU port map (
       P => P(i*4+3 downto i*4),
       G  => G(i*4+3 downto i*4), CI => GC(i),
       CO => C(i*4+3 downto i*4),
       GP => GP(i), GG => GG(i));
 end generate;

 clau1 : CLAU port map (P => GP, G => GG, CI => CI, CO => GC,
                        GP => open, GG => CO);
end RTL;
```

Implementace stromové CLA sčítačky 3/4

```
library IEEE;
use IEEE.std_logic_1164.all;
entity CLAU is
  port (
    P, G   : in  std_logic_vector(3 downto 0);
    CI     : in  std_logic;
    CO     : out std_logic_vector(3 downto 0);
    GP, GG : out std_logic);
end CLAU;

architecture RTL of CLAU is
begin
  CO(0) <= CI;
  CO(1) <= (CI and P(0)) or G(0);
  CO(2) <= (CI and P(0) and P(1)) or
           (G(0) and P(1)) or
           G(1);
  CO(3) <= (CI and P(0) and P(1) and P(2)) or
           (G(0) and P(1) and P(2)) or
           (G(1) and P(2)) or
           G(2);
  GG    <= (G(0) and P(1) and P(2) and P(3)) or
           (G(1) and P(2) and P(3)) or
           (G(2) and P(3)) or
           G(3);
  GP    <= P(3) and P(2) and P(1) and P(0);
end RTL;
```

Implementace stromové CLA sčítačky 4/4

```
library IEEE;
use IEEE.std_logic_1164.all;

entity XA is
  port (
    A, B, CI : in  std_logic;
    S, P, G  : out std_logic);
end XA;

architecture RTL of XA is
  signal Gsig, Psig : std_logic;
begin
  Gsig  <= A and B;
  Psig  <= A xor B;
  S     <= Psig xor CI;
  G     <= Gsig;
  P     <= Psig;
end RTL;
```

Python implementace

- https://github.com/ehw-fit/ariths-gen/tree/main/ariths_gen/multi_bit_circuits/adders