



Popis systémů pomocí VHDL

Milan Kolář

Ústav mechatroniky a technické informatiky



evropský
sociální
fond v ČR



EVROPSKÁ UNIE



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY



OP Vzdělávání
pro konkurenční
schopnost
EF-ESF

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Projekt ESF CZ.1.07/2.2.00/28.0050
**Modernizace didaktických metod
a inovace výuky technických předmětů.**



Vývoj VHDL

- **HDL - Hardware Description Language**
- **VHDL - Very High Speed Integrated Circuits HDL**
- Vývoj od roku 1983 v rámci projektu **VHSIC**
- **1987 - standard IEEE 1076-1987**
- **1993 - revize IEEE Std 1076-1993**
- 1999 - revize IEEE Std 1076.1-1999
VHDL-AMS (Analogue & Mixed Signals)
- 2002 – revize IEEE Std 1076-2002
- **2008 – revize IEEE Std 1076-2008**





Charakterizace VHDL

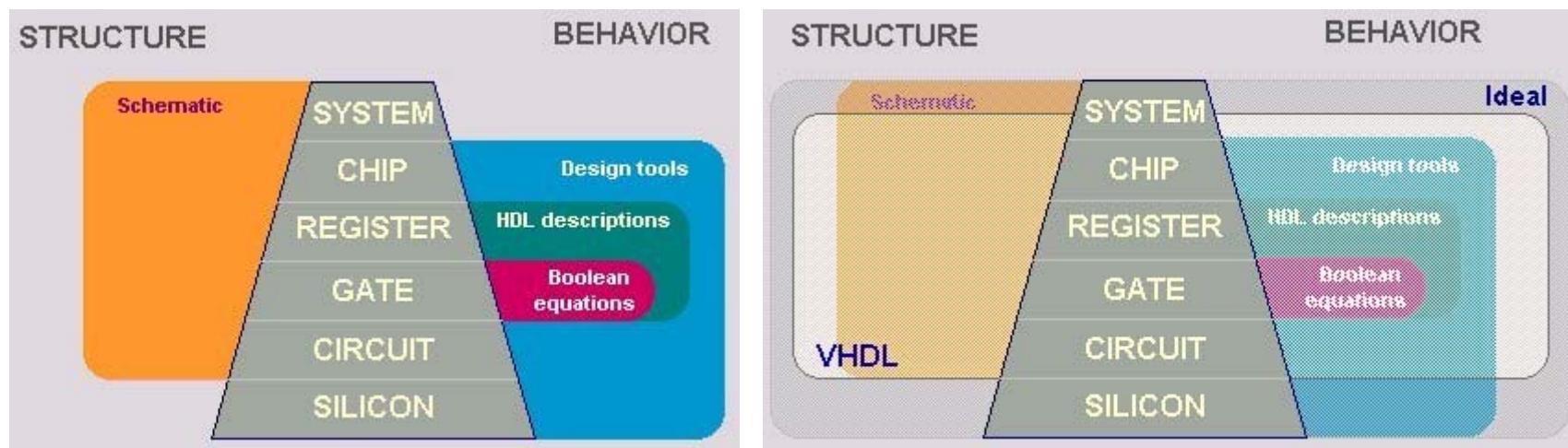
- všeobecně přístupný otevřený standard,
- vhodné pro návrh metodou shora-dolů (top-down),
- nezávislé na budoucí technologii realizace,
- důraz na funkci obvodu (oproštění od detailů),
- umožňuje opakované používání modelů (knihovny),
- využití pro dokumentaci a modelování,
- snadná výměna částí návrhů mezi návrháři (IP core),
- libovolná část návrhu může být osamostatněna,
- model VHDL může být simuloval v různých systémech,
- podpora testovatelnosti (Boundary Scan Architecture),
- „upovídáný“ jazyk (opakování bloků, deklarace),
- ne všechny konstrukce jazyka musí být syntetizovatelné.



VHDL v různých úrovních abstrakce

VHDL lze použít v různých úrovních abstrakce:

- úroveň behaviorální (popis chování obvodu)
- úroveň RTL (Register Transfer Level)
- úroveň hradel (logická úroveň)





Formální vlastnosti VHDL

- při zápisu se nerozlišují malá a velká písmena (není „case sensitive“),
- každý příkaz je ukončen středníkem (;),
- v zápisu jazyka lze pro lepší čitelnost používat libovolný počet mezer („space insensitive“),
- využívá klíčových slov,
- žádná pravidla pro jména souborů (doporuč. jméno souboru shodné se jménem nejvyšší entity).



Pravidla jmen identifikátorů

Syntaxe:

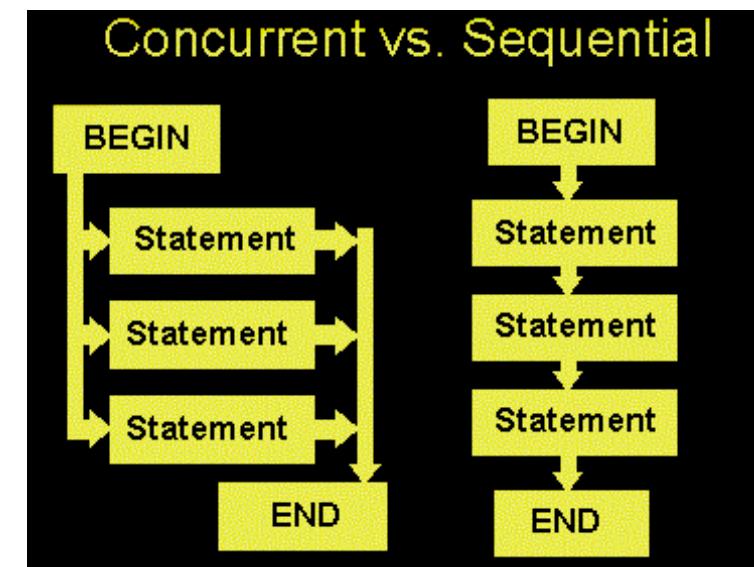
písmeno{ [_]písmeno_nebo_číslice }

- nerozlišují se velká a malá písmena,
- musí začínat písmenem,
- mohou obsahovat písmena, číslice a podtržítka,
- jméno nesmí obsahovat mezeru,
- nelze použít dvě podtržítka za sebou,
- podtržítko nesmí být posledním znakem,
- nesmí být totožná s klíčovými slovy,
- musí být unikátní
 - nelze použít signál A a současně sběrnici A(7 downto 0).



Příkazy

- **Declaration statements**
 - definice konstant, typů, objektů, podprogramů;
- **Concurrent statements** – současně probíhající
 - pro popis kombinační logiky;
 - např. block, signal assignment, procedure call, ...
- **Sequential statements**
 - spouští se v napsaném v pořadí;
 - např. příkazy if, case, loop, next, wait, exit, ...
 - obdoba SW programu.





evropský
sociální
fond v ČR



EVROPSKÁ UNIE



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY



Návrh hardwarových komponent

Modernizace didaktických metod a inovace výuky technických předmětů



Komentář

- začíná dvěma pomlčkami (dash);
- komentář může začínat i za libovolným příkazem;
- komentář končí na konci řádku (nelze jiným způsobem ukončit);
- víceřádkový komentář podporován až od verze VHDL2008 (často ale řešeno již v editorech);
- nedoporučuje se v komentářích používat diakritiku;
- v komentářích se někdy objevují i speciální příkazy návrhového systému (např. pro syntezátor).

-- toto je komentar

c <= a AND b; -- toto je také komentar



Hlavní komponenty VHDL

dvě povinné komponenty: Entity a Architecture

Příklad - hradlo XOR:

ENTITY hr_xor IS

PORT (a, b : IN BIT;
y : OUT BIT);

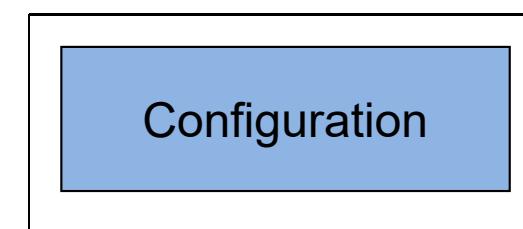
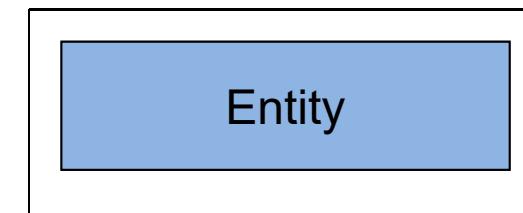
END hr_xor;

ARCHITECTURE ar_hr_xor OF hr_xor IS

BEGIN

y <= a **XOR** b;

END ar_hr_xor;





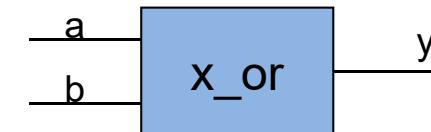
Entita a architektura

- **Entita** - „černá skříňka“ se vstupy a výstupy (obdoba grafického symbolu);
 - Entita nepopisuje chování modulů (nedefinuje funkci).
-
- **Architektura** - určuje chování entit
 - tělo architektury má dvě části:
 - deklarační část (např. definice signálů),
 - příkazová část (uzavřeno do **begin - end**);
 - architektura musí být spojena se specifikovanou entitou.



Příklad: XOR (popis chování)

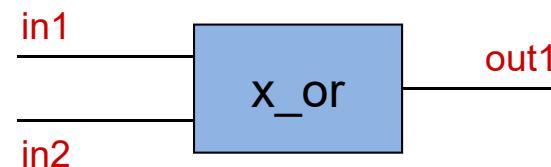
```
ENTITY x_or IS
    PORT ( a, b : IN BIT;
            y : OUT BIT );
END x_or;
```



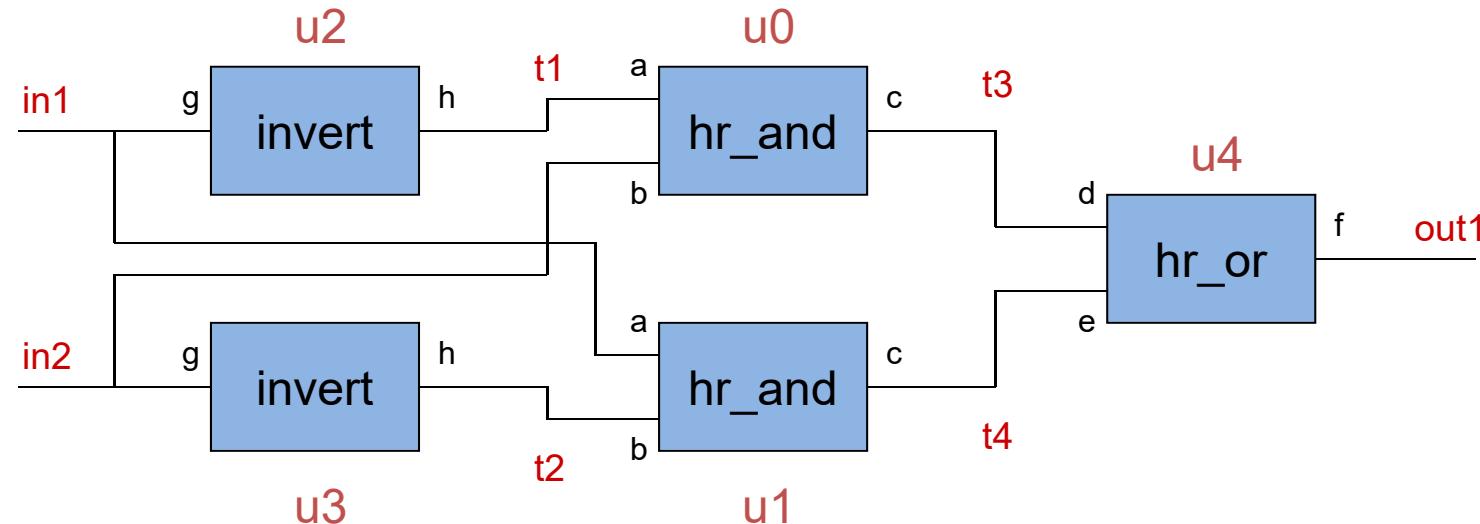
```
ARCHITECTURE behavior OF x_or IS
BEGIN
    PROCESS (a, b)
        BEGIN
            IF ( a = b ) THEN y <= '0';
            ELSE y <= '1';
            END IF;
        END PROCESS;
    END behavior;
```



Příklad: XOR (strukturální popis)



$$(Y = \overline{A}B + A\overline{B})$$





Příklad: XOR (strukturální popis)

```
ENTITY x_or IS
  PORT ( in1, in2 : IN BIT;
         out1 : OUT BIT );
END x_or;
```

```
ENTITY hr_and IS
  PORT ( a, b : IN BIT;
         c : OUT BIT );
END hr_and;
ARCHITECTURE behavior OF
  hr_and IS
BEGIN
  PROCESS (a, b)
  BEGIN
    c <= a AND b;
  END PROCESS;
END behavior;
```

```
ENTITY hr_or IS
  PORT ( d, e : IN BIT;
         f : OUT BIT );
END hr_or;
ARCHITECTURE behavior OF hr_or IS
BEGIN
  PROCESS (d, e)
  BEGIN
    f <= d OR e;
  END PROCESS;
END behavior;
```



Příklad: XOR (strukturální popis)

```
ENTITY invert IS
  PORT ( g : IN BIT;
         h : OUT BIT );
END invert;
ARCHITECTURE behavior OF invert IS
BEGIN
  PROCESS (g)
    BEGIN
      h <= NOT g;
    END PROCESS;
  END behavior;
```

```
ARCHITECTURE structural OF x_or IS
  SIGNAL t1, t2, t3, t4 : BIT;
  COMPONENT hr_and
    PORT (a, b : IN BIT;
          c : OUT BIT );
  END COMPONENT;
```

```
COMPONENT hr_or
  PORT (d, e : IN BIT;
        f : OUT BIT );
END COMPONENT;
COMPONENT invert
  PORT (g : IN BIT;
        h : OUT BIT );
END COMPONENT;

BEGIN
  u0: hr_and PORT MAP (a=>t1, b=>in2, c=>t3);
  u1: hr_and PORT MAP (a=>in1, b=>t2, c=>t4);
  u2: invert PORT MAP (g=>in1, h=>t1);
  u3: invert PORT MAP (g=>in2, h=>t2);
  u4: hr_or PORT MAP (d=>t3, e=>t4, f=>out1);
END structural;
```



evropský
sociální
fond v ČR



EVROPSKÁ UNIE



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY



OP Vzdělávání
pro konkurenčních schopností

Návrh hardwarových komponent

Modernizace didaktických metod a inovace výuky technických předmětů



Porty (brány)

- popisují vnější signály entity
- jsou charakterizovány:
 - jménem (libovolná skupina znaků začínající písmenem)
 - módem (určuje směr toku dat):
IN, OUT, BUFFER, INOUT
 - datovým typem (lze spojovat porty stejného typu)

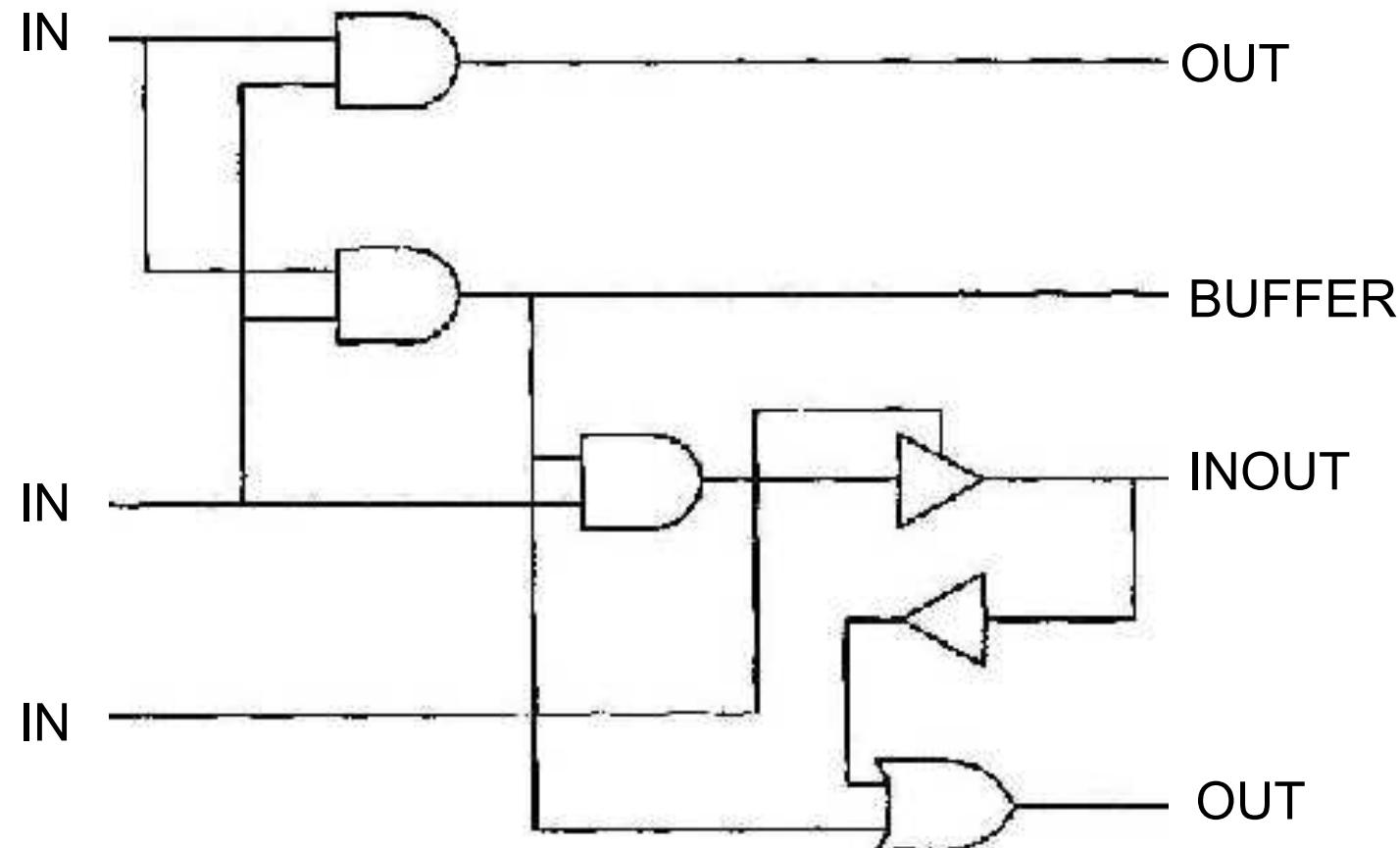


Módy portů

- **IN** – data lze z portu pouze číst;
- **OUT** – data vycházejí z portu (výstupní signál nemůže být použit jako vstup uvnitř entity);
- **INOUT** – obousměrný tok (obousměrné vstupy/výstupy) – slouží pro připojení k třístavové sběrnici.
- **BUFFER** – výstup se zpětnou vazbou (může být buzen pouze z vnitřku entity - data mohou z entity pouze vystupovat, lze zpětně číst) – v nadřazené struktuře lze spojovat pouze s módem IN (ne OUT nebo INOUT);



Příklad portů





Porty (pokračování)

Syntaxe:

```
PORT (jméno_signálu : mód datový_typ) ;
```

Příklad:

```
PORT ( vstup : IN BIT ;
       a1, b1 : IN BIT_VECTOR (3 DOWNTO 0) ;
       vystup : OUT BIT ) ;
```

```
součet : OUT BIT_VECTOR (0 TO 7) ;      -- 0 je MSB, 7 je LSB
operand : IN BIT_VECTOR (4 DOWNTO 0) ; -- 4 je MSB, 0 je LSB
-- nelze: BIT_VECTOR (4 TO 1);
          BIT_VECTOR (0 DOWNTO 5);
```



Položka Generic

- obdoba portů, ale nepředstavuje žádný signál;
- obdoba konstanty (viditelná v entitě a přiřazených architekturách);
- používá se jako parametr (neměnící se v čase);
- použití pro lepší čitelnost, správu a konfiguraci.

Syntaxe:

GENERIC (generic_jméno : datový_typ [:= hodnota] ;

Příklad:

GENERIC (Delay : integer := 5) ; -- časový parametr

GENERIC (BusWidth : integer := 8); -- velikost objektu

GENERIC (Loop : integer := 3); -- proměnný počet smyček



Datové objekty

Ve VHDL jsou 4 třídy datových objektů:

- **Konstanty** (constants) – mají neměnnou hodnotu
(nelze dále měnit ⇒ lze psát pouze na pravé straně přiřazení) – deklarují se v entitách, architekturách, procesech, funkcích, procedurách a slohách (package).
- **Proměnné** (variables) – používají se jako pomocné objekty (nepředstavují skutečné signály, nelze je použít jako porty) – deklarují se v procesech, funkcích a procedurách.
- **Signály** (signals) – většinou jsou fyzicky přítomné ve formě elektrických signálů – deklarují se pouze v deklarační části architektury.
- **Soubory** (files) – používají se převážně pro uložení vstupních a výstupních dat při simulaci.



Datové objekty (pokračování)

Syntaxe:

CONSTANT jméno_konstanty : datový_typ := hodnota ;

VARIABLE jméno_proměnné : datový_typ [:= hodnota] ;

SIGNAL jméno_signálu : datový_typ [:= hodnota] ;

- **Datový_typ** použijeme standardní nebo je nutné jej definovat příkazem **TYPE** ;
- nastavení hodnoty není podporováno syntézou (slouží jen k simulaci)

Příklady:

CONSTANT pi : **REAL** := 3.14 ;

CONSTANT rychlost : **INTEGER** ; -- defaultní hodnota: 0

VARIABLE suma : **BIT_VECTOR** (0 TO 3) := "0010" ;

SIGNAL select : **STD_LOGIC** ;



Viditelnost datových objektů

Datový objekt deklarovaný:

- v procesu je viditelný pouze uvnitř tohoto procesu;
- v architektuře je viditelný ve všech příkazech této architektury;
- v entitě je viditelný ve všech architekturách přidělených této entitě;
- v „package“ je viditelný ve všech návrzích užívajících tohoto „package“.



Jednoduché přiřazení signálu (simple)

- paralelní příkaz;
- vykazuje určité setrvačné zpoždění (neprovede se bezprostředně);
- datové typy na obou stranách musí být stejné.

Syntaxe:

```
jméno_signálu <= výraz ;
```

Příklad:

```
SIGNAL a, b, q : bit ;
```

```
q <= a XOR b ;
```



Přiřazování polí

- velikost polí na levé i pravé straně přiřazení musí být stejná
- jednotlivé elementy jsou přiřazovány podle pozice
(ne podle indexu)

Příklady:

SIGNAL a, b, q : std_logic_vector (0 TO 1);

SIGNAL c : std_logic_vector (1 DOWNTO 0);

a <= b ; -- a(0) <= b(0) ; a(1) <= b(1) ;

c <= b ; -- c(1) <= b(0) ; c(0) <= b(1) ;

q <= a NOR b ; -- q(0) <= a(0) NOR b(0) ; q(1) <= a(1) NOR b(1);

a(0) <= b(1) ;



Polohové a jmenné přiřazování

Příklady:

b, e : bit;

a : bit_vector(3 DOWNTO 2);

d : bit_vector(0 TO 2);

c : bit_vector(8 DOWNTO 1);

c <= ('0', '1', OTHERS => '0'); -- polohové

d <= ('0', '1', '0'); -- polohové

a <= (b, '0'); -- polohové

c <= (8 => '1', 7 => b, 5 DOWNTO 2 => '1', OTHERS => '0');
-- jmenné

d <= (0 => b nand e, 1 to 2 => a); -- jmenné d(1) <= a(3), d(2) <= a(2)

c <= "00000000"; -- nevhodné, raději: c <= (OTHERS => '0');



Výběrové přiřazení signálu (selected)

Syntaxe:

WITH výběrový_signál **SELECT**

jméno_signalu <= hodnota_1 **WHEN** hodnota_1_výběrového_signálu,
hodnota_2 **WHEN** hodnota_2_výběrového_signálu,
...
hodnota_n **WHEN** hodnota_n_výběrového_signálu;

- nemá charakter prioritního přiřazení (obdoba CASE – WHEN)

Příklad:

WITH sel **SELECT**

hd1 <= i0 **WHEN** "0000" **TO** "0100", -- od – do
i1 **WHEN** "0101" **|** "0111", -- nebo
i2 **WHEN** "1010",
i3 **WHEN** OTHERS ; -- ostatní hodnoty



Podmíněné přiřazení signálu (conditional)

Syntaxe:

```
jméno_signalu <= hodnota_1 WHEN podmínka_1 ELSE  
                      hodnota_2 WHEN podmínka_2 ELSE  
                      ...  
                      hodnota_n WHEN podmínka_n ELSE  
                      hodnota_x ;
```

- má charakter prioritního přiřazení => může vést na složitější obvod

Příklad:

```
hd1 <= i0 WHEN w = '0' ELSE  
          i1 WHEN x = '1' ELSE  
          i2 WHEN y = '1' ELSE '0';
```



Datové typy

Datový typ určuje formát dat, který může daný port, signál, proměnná či konstanta přenášet nebo s nimi pracovat. Datové typy mohou být ve VHDL předdefinované (implicitně viditelné ve všech VHDL modelech) nebo se musí deklarovat pomocí příkazu **TYPE**.

Datové typy lze dělit (sdružovat) do skupin, z nichž nejdůležitější jsou:

- **Číselné** – např. integer, real, natural, positive, signed, unsigned;
- **Logické** – bit, boolean, std_logic, std_ulogic;
- **Znakové** – character, string;
- **Složené** – více elementů v jednom objektu (vector, array, record), zápis bitových polí: b"11001001", X"C3F", O"754";
- **Fyzikální** – vyžadují připojení fyzikální jednotky (např. time);



Datové typy - předdefinované

- **charakter, string** – pole 256 znaků – ISO 8859-1 (Latin1)
(lze užívat i některé speciální znaky, např. ! # \$ % & ' / > á Ö),
jsou case sensitive, znaky zapisujeme s apostrofy, stringy
s uvozovkami (např.: "bit" /= "Bit");
- **bit** – '0', '1';
- **bit_vector** – např. "1001";
- **boolean** - false, true;
- **integer** – rozsah $\pm(2^{31}-1)$ + podtypy: **natural** (≥ 0) a **positive** (≥ 1);
- **real** – reálná čísla (nesyntetizovatelné, pouze pro simulace);
- **time** – čas v rozlišení fs, ps, ns, us, ms, s, m, hr (povinná mezera
mezi hodnotou a jednotkou).



Deklarace datových typů

SIGNAL a : integer RANGE 0 TO 255;

u celočíselného typu uvádíme obvykle rozsah, který omezuje počet bitů (jinak 32bitové).

SIGNAL b: bit_vector(7 downto 0);

Datové typy, které nejsou implicitní, musí být deklarovány v package nebo v deklarační části architektury příkazem **TYPE**, případně **SUBTYPE**. Např.:

TYPE word IS ARRAY (31 DOWNTO 0) of BIT;

SIGNAL abc : word;

TYPE byte IS INTEGER RANGE 10 DOWNTO -10 ; -- někdy se slovo „INTEGER“ vynechává (v rozsahu jsou uvedena celá čísla)

SIGNAL c1 : byte;



Výčtový datový typ (enumerated)

- specifikuje se seznam hodnot, kterých může nabývat (jiné hodnoty nelze přiřazovat) – diskrétní typ;
- záleží na pořadí (lze určit předchozí a následující hodnotu);
- nejčastěji se v praxi používá k výčtu stavů stavového automatu.

```
TYPE muj_stav IS (res, id, rw, int) ;  
SIGNAL stav : muj_stav ;  
SIGNAL ekv : STD_LOGIC_VECTOR (0 TO 1) ;  
stav <= res ;      -- nelze psát: stav <= "00" ; stav <= ekv ;
```

```
TYPE prepинac IS ( ON, OFF ); -- výčtový typ  
VARIABLE a: prepинac;  
a := ON;
```



Vícehodnotová logika

Definována v package „std_logic_1164“

std_logic a **std_ulogic** – výčtové typy definující 9 hodnot – všechny hodnoty jsou typu „character“ – ‘U’, ‘X’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘-’ (nutno psát velkými písmeny)

TYPE std_ulogic IS (

- ‘U’, -- neinicializováno (signál nebyl dosud buzen), implicitní
- ‘X’, -- neznámá hodnota (vzniká při konfliktu ‘0’ a ‘1’)
- ‘0’, -- log. 0 z tvrdého zdroje
- ‘1’, -- log. 1 z tvrdého zdroje
- ‘Z’, -- vysoká impedance
- ‘W’, -- neznámá hodnota (vzniká při konfliktu H a L)
- ‘L’, -- log. 0 z měkkého zdroje
- ‘H’, -- log. 1 z měkkého zdroje
- ‘-’) ; -- neurčená hodnota (don’t care), na hodnotě nezáleží.



Vícehodnotová logika (pokračování)

std_ulogic – nedisponuje tzv. vyhodnocovací funkcí (resolution function), a tedy na signál tohoto typu nelze připojit více budičů najednou.

std_logic je podtypem **std_ulogic** s definovanou vyhodnocovací funkcí (tabulkou):

	U	X	0	1	Z	W	L	H	-
U	'U'								
X	'U'	'X'							
0	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
1	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
Z	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
W	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
L	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
H	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
-	'U'	'X'							



Datové typy signed a unsigned

- datové typy obdobné std_logic_vector (logické vektory), které jsou chápány jako čísla ve dvojkové soustavě;
- unsigned – kladná celá čísla bez znaménka;
- signed – čísla ve dvojkovém kódu se znaménkem;
- slouží pro práci s aritmetickými operátory;
- využívá se při volání package **numeric_std**, **numeric_bit**;

Příklad:

A má hodnotu “1010”, B má hodnotu “0100”

A a B jsou typu signed, potom A < B je “pravdivé”

A a B jsou typu unsigned, potom A < B je “nepravdivé”



Příkaz WAIT

- příkaz pro spuštění (pozastavení) procesu nebo procedury
- sekvenční příkaz

Syntaxe:

WAIT [ON seznam_signálů][UNTIL výraz][FOR čas] ;

Příklady:

WAIT ON s1, s2 ; -- čekáme, dokud nenastane změna signálů

WAIT FOR 50 ns ; -- čekáme daný čas (nesyntetizovatelné)

WAIT UNTIL enable = '1' ; -- čekáme na pravdivost podmínky

WAIT ON a, b UNTIL clk = '1' ; -- čekáme, dokud nenastane změna jednoho ze signálů a nebo b, ale současně musí clk = 1.



Příkaz Process

- představuje nezávislý děj, který se provede při aktivaci;
- užívá se zejména pro popis sekvenčních dějů;
- příkazy uvnitř procesu se vykonávají sekvenčně, ale více procesů v architektuře se vykonává paralelně.

Syntaxe:

[jméno] : **PROCESS** [(clock, reset)] --seznam citlivých proměnných
-- deklarace procesu

BEGIN

-- příkazy procesu

END PROCESS [jméno];

- seznam citlivých proměnných je při syntéze ignorován, ale je významný pro správné provádění simulace.



Spuštění procesu

Každý proces je spuštěn automaticky na začátku simulace.
Další spuštění je možné jedním ze dvou možností:

PROCESS (a) **-- spuštění na základě citlivostního seznamu**

BEGIN

 y <= a ;

END PROCESS ;

PROCESS **-- spuštění příkazem WAIT**

BEGIN

 WAIT on a;

 y <= a ;

END PROCESS ;



Process – použití signálů

SIGNAL a, b, c : bit ;

PROCESS (b)

BEGIN

a <= b ;

c <= a ;

END PROCESS ;

Signál	Předešlý stav	Následující stav
b	1	0
a	1	0
c	1	1 !

PROCESS (a, b)

BEGIN

a <= b ;

c <= a ;

END PROCESS ;

Signál	Předešlý stav	Iterace č. 1	Iterace č. 2
b	1	0	0
a	1	0	0
c	1	1	0



Process – použití proměnné

```
SIGNAL b, c : bit ;  
PROCESS (b)  
VARIABLE a : bit ;  
BEGIN  
a := b ;  
c <= a ;  
END PROCESS ;
```

Signál	Předešlý stav	Následující stav
b	1	0
a	1	0
c	1	0

- přiřazování je nahrazeno `:=` (např.: `y := a AND b ;`)
- signály lze přiřazovat do proměnných a naopak
- proměnné deklarované uvnitř procesu si zachovávají svoji hodnotu při dalším průchodu procesem (na rozdíl od funkcí a procedur, kde se vždy inicializují).



Příkaz IF

- lze používat pouze uvnitř procesu (sekvenční příkaz);
- má charakter prioritního přiřazení => může vést na složitější obvodové zapojení (volit raději CASE – WHEN);
- podmínka je výraz vracející hodnotu typu boolean.

Syntaxe:

```
IF podmínka1 THEN {sekvence_příkazů1}
[ { ELSIF podmínka2 THEN {sekvence_příkazů2} } ]
[ ELSE {sekvence_příkazů} ]
END IF ;
```



Příklad multiplexoru

ENTITY mux IS

```
PORT ( In1, In2, Sel : IN BIT;  
       Out1 : OUT BIT );
```

```
END mux ;
```

ARCHITECTURE multiplexer OF mux IS;
BEGIN

```
PROCESS ( Sel, In1, In2 )
```

```
BEGIN
```

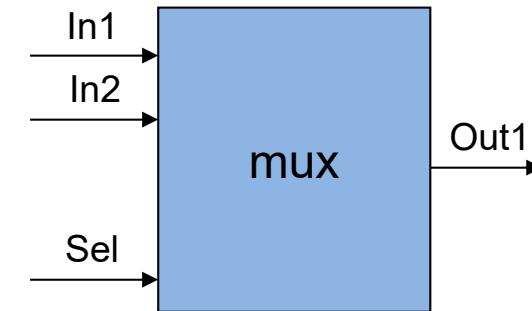
```
IF ( Sel = '1' ) THEN Out1 <= In1;
```

```
ELSE Out1 <= In2;           -- nutné, jinak vznikne latch!
```

```
END IF;
```

```
END PROCESS;
```

```
END multiplexer;
```





Příkaz CASE

- lze používat pouze uvnitř procesu (sekvenční příkaz);
- musí být specifikovány všechny možnosti výrazu;
- hodnoty výrazu se nesmí překrývat;
- nemá charakter prioritního přiřazení
(obdoba příkazu WITH – SELECT – WHEN).

Syntaxe:

CASE výraz **IS**

WHEN hodnota_1 => příkaz;

WHEN hodnota_2 | hodnota_4 => příkaz; -- nebo

WHEN hodnota_m **TO** hodnota_n => příkaz;

WHEN OTHERS => příkaz; -- příp. NULL

END CASE;



Popis KLO procesem

- V seznamu citlivých proměnných uvést všechny vstupy,
- obvod nutno plně specifikovat.

Příklad KLO:

PROCESS (vstupy) **BEGIN**

CASE vstupy **IS**

WHEN "000" => segment <= "0000001";

WHEN "001" => segment <= "1001111";

WHEN "010" => segment <= "0010010";

WHEN "011" => segment <= "0000110";

WHEN "100" => segment <= "1001100";

WHEN "101" => segment <= "0100100";

WHEN OTHERS => segment <= "-----"; -- hardwarově výhodnější

END CASE;

-- než uvést konkrétní logické hodnoty

END PROCESS;



Náběžné a sestupné hrany

clk'event AND clk = '1' -- náběžná hrana

clk'event AND clk = '0' -- sestupná hrana

SIGNAL clock : boolean ;

IF NOT clock AND clock'event -- sestupná hrana

v knihovně IEEE 1164 existují fce **rising_edge ()** a **falling_edge ()**

(knihovnu IEEE 1164 nutno deklarovat – příkazy LIBRARY a USE)

– tyto funkce nelze užívat v logických operacích, např.:

IF NOT falling_edge (clk) ...

– lze psát: enclk <= clk AND clk_en ;

IF rising_edge(enclk) THEN ...

WAIT UNTIL (clk'event AND clk = '1');

IF (num'event AND num = '0') THEN q <= c(5); END IF;



Použití hod. signálu v procesu

PROCESS (hodinový_signál) -- použitím citlivých proměnných

BEGIN

IF (podmínka_hrany_hod_signálu) **THEN**

výstupní_signál <= vstupní signál;

... další sekvenční příkazy ...

END IF;

END PROCESS;

PROCESS

-- použitím příkazu WAIT

BEGIN

WAIT ON (hodinový_signál) **UNTIL** (specifikace_hrany_hod_signálu);

výstupní_signál <= vstupní signál;

... další sekvenční příkazy ...

END PROCESS;



Použití synchronního resetu (1)

PROCESS (hodinový_signál) -- použitím citlivých proměnných

BEGIN

IF (podmínka_hrany_hod_signálu) **THEN**

IF (podmínka_resetu) **THEN**

 výstupní signál <= resetovací_hodnota;

ELSE

 výstupní_signál <= vstupní signál;

 ... další sekvenční příkazy ...

END IF;

END IF;

END PROCESS;

- nepoužívat ELSE ve vnějším IF s podmínkou hrany hod. signálu;
- nepoužívat v procesu více příkazů IF (pouze vložené).



Použití synchronního resetu (2)

PROCESS
BEGIN

-- použitím příkazu WAIT

WAIT ON (hodinový_signál) **UNTIL** (specifikace_hrany_hod_signálu);
IF (podmínka_resetu) **THEN**

výstupní_signál <= resetovací_hodnota;

ELSE

výstupní_signál <= vstupní signál;

... další sekvenční příkazy ...

END IF;

END PROCESS;



Použití asynchronního resetu (1)

Použitím citlivých proměnných:

PROCESS (hodinový_signál, resetovací_signál)

BEGIN

IF (resetovací_podmínka) **THEN**

výstupní signál <= resetovací_hodnota;

ELSIF (podmínka_hrany_hod_signálu) **THEN**

výstupní_signál <= vstupní_signál;

... další sekvenční příkazy ...

END IF;

END PROCESS;

- nepoužívat příkaz ELSE po detekci hrany hodinového signálu;
- nepoužívat v procesu více příkazů IF (pouze vložené).



Použití asynchronního resetu (2)

Použitím příkazu WAIT:

PROCESS

BEGIN

WAIT ON hodinový_signál, resetovací_signál;

IF (resetovací_podmínka) **THEN**

výstupní signál <= resetovací_hodnota;

ELSIF (podmínka_hrany_hod_signálu) **THEN**

výstupní_signál <= vstupní_signál;

... další sekvenční příkazy ...

END IF;

END PROCESS;



Použití asynchronního resetu a setu

Příklad klopného obvodu D s asynchronním resetem a setem:

PROCESS (clk, reset, set)

BEGIN

IF (reset = '0') **THEN**

q <= '0';

ELSIF (set = '0') **THEN**

q <= '1';

ELSIF (clk'event AND clk = '1') **THEN**

q <= data;

END IF;

END PROCESS;

V citlivých proměnných uvedeny pouze hodiny a asynchronní signály!



Klopný obvod řízený hladinou (latch)

- v návrzích raději nepoužívat (vliv střídy hodinového signálu);
- v ASIC obvodech vedou na jednodušší obvodové zapojení;
- někdy se generují v důsledku špatného popisu kombinační logiky (chybějící ELSE, neúplný příkaz CASE);
- v seznamu citlivých proměnných nutno uvést i datové vstupy.

Příklad: (při clk = '0' zachovává předchozí stav)

PROCESS (clk, a, b)

BEGIN

IF (clk = '1') **THEN**

y <= a **AND** b ; -- u KLO by bylo: ELSE y <= '0' ;

END IF ;

END PROCESS ;



Hranový klopný obvod

- nevytvářet hranově řízené klopné obvody z latchů tím, že neuvedeme úplný seznam citlivých proměnných;

PROCESS (clk) -- nevhodné řešení (nereaguje na změnu 'a' nebo 'b')

BEGIN -- a tím se může tvářit jako KO řízený hranou)

IF (clk = '1') **THEN**

 y <= a **AND** b;

END IF;

END PROCESS;

PROCESS (clk) -- vhodné řešení

BEGIN

IF rising_edge(clk) **THEN**

 y <= a **AND** b;

END IF;

END PROCESS;



Více než jeden hodinový signál

- každý hodinový signál musí mít vlastní proces

PROCESS (clk1, clk2)

BEGIN

```
IF rising_edge(clk1) THEN
    q1 <= a;
END IF ;
IF rising_edge(clk2) THEN      -- nelze
    q2 <= b;
END IF;
END PROCESS;
```



Knihovny (Library)

Jazyk VHDL definuje dvě třídy knihoven – pracovní (work) a zdrojové.

Pracovní knihovna může být připojena pouze jedna a jsou do ní automaticky ukládány překládané objekty.

V knihovně se nacházejí tzv. primární a sekundární návrhové jednotky
- mezi primární patří **entity**, **package** a **configuration**, mezi sekundární **architecture** a **package body** (configuration nemá sekundární jednotku).

Primární a jím odpovídající sekundární jednotky se musí nacházet ve stejné knihovně.

V knihovně jsou nejčastěji **package** (knihovní balíky, slohy), příp. **package body**.



Položky LIBRARY a USE

- V systému VHDL je standardně přístupná pouze knihovna „Std“ s package „standard“ - není třeba připojovat (bývá přístupná i knihovna „work“); ostatní knihovny je třeba připojit (zviditelnit) příkazem LIBRARY;
- package se připojují příkazem USE (zviditelňuje specifikované položky v daném package);
- obě položky nutno v návrhu opakovat pro každou entitu a package;
- pokud je třeba zpřístupnit více package z jedné knihovny, užijeme příkaz USE několikrát za sebou.

Syntaxe:

LIBRARY jméno_knihovny ;

USE jméno_knihovny.jméno_package.položka ; -- příp. all



Package (knihovný balík, sloha)

- hierarchicky nad entitou a architekturou;
- položky deklarované v package jsou viditelné v celém návrhu (v package nelze uvést deklaraci entity nebo architektury);
- Package obsahuje dvě části:
 - deklarační část (příkaz **PACKAGE**) – pro deklarace hlaviček funkcí a procedur, typů a podtypů, konstant a signálů, komponent (vlastní popis musí být mimo sekci package), atributů, aj.
 - vlastní tělo (příkaz **PACKAGE BODY**) – pro definice podprogramů, funkcí, aj.).



Package - příklad

PACKAGE system **IS**

CONSTANT pocet : **INTEGER** := 2;

PROCEDURE add (**SIGNAL** a, b : **IN BIT**; suma : **OUT BIT**);

END system;

PACKAGE BODY system **IS**

PROCEDURE add (**SIGNAL** a, b : **IN BIT**; suma : **OUT BIT**);

BEGIN

 suma <= a **XOR** b;

END add;

END system;



Standardizované package IEEE

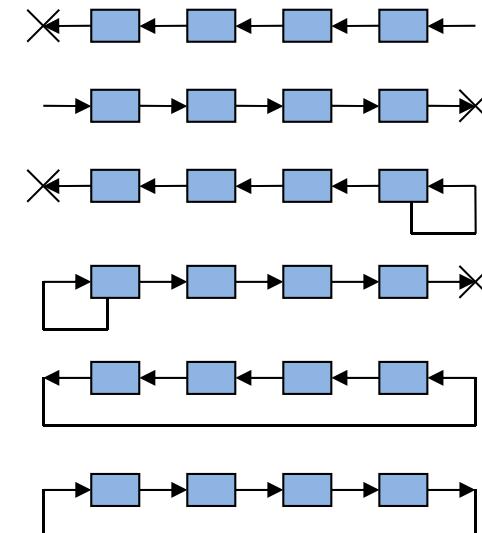
Jsou uloženy v knihovně IEEE;

- **std_logic_1164** – definice vícehodnotovou logiku, vybrané konverzní funkce a logické operátory s std logikou;
- **numeric_bit, numeric_std** – definice aritmetických a logických operací nad typy unsigned a signed (nahrazují starší package **std_logic_arith, std_logic_unsigned a std_logic_signed**);
- **math_real** – definuje operace v plovoucí řádové čárce;
- **fixed_pkg, fphdl_pkg** – připravují se syntetizovatelné balíky pro podporu matematických operací v pevné a plovoucí řádové čárce (Quartus nepodporuje).



Operátory

- logické: **NOT, AND, OR, NAND, NOR, XOR, XNOR**
- relační: **=, !=, >, <, >=, <=**
- aritmetické: **+, -, *, /, mod** (Modulus),
rem (Remainder), **abs** (Absolute Value), ****** (exponent)
- posuvu a rotace
 - **SLL** (Shift Left Logical)
 - **SRL** (Shift Right Logical)
 - **SLA** (Shift Left Arithmetic)
 - **SRA** (Shift Right Arithmetic)
 - **ROL** (Rotate Left Logical)
 - **ROR** (Rotate Right Logical)
- slučující operátor &





Operátory - priorita

- 1) **, ABS, NOT -- nejvyšší priorita
- 2) *, /, MOD, REM
- 3) znaménka +, -
- 4) +, -, &
- 5) SLL, SRL, SLA, SRA, ROL, ROR
- 6) =, /=, <, <=, >, >=
- 7) AND, OR, NAND, NOR, XOR, XNOR -- nejnižší priorita

Prioritu operátorů lze měnit oblými závorkami, jinak se vyhodnocují zleva doprava (pozor na neasociativní operátory NAND a NOR)



Operátory - poznámky

- operandy operátorů musí být stejného typu;
- operandy u logických operátorů musí mít stejnou délku;
- porovnáváme-li u relačních operací operátory různé délky, operátory se porovnávají zleva doprava (zarovnávají se vlevo):
 $\text{bit_vector ("1000")} > \text{bit_vector ("111")}$
-- výsledkem je FALSE, protože $1000 < 1110$
- logické operátory lze používat jen s datovými typy bit, boolean, std_logic, bit_vector, std_logic_vector;
- relační operátory lze používat se skalárními datovými typy a jednorozměrnými poli – vrací hodnotu true a false (u datových typů record pouze = a /=).



Operace posuvu a rotace

Jsou definovány pro jednorozměrná pole typu bit a boolean, pravý operand musí být typu integer (může být i záporný)

Příklad:

```
signal a : bit_vector (7 downto 0) := "10000001";
```

```
signal b, c, d, e, f, g : bit_vector (7 downto 0);
```

```
b <= a sll 1;          -- výsledek "00000010"
```

```
c <= a srl 1;          -- výsledek "01000000"
```

```
d <= a sla 2;          -- výsledek "00000111"
```

```
e <= a sra 2;          -- výsledek "11100000"
```

```
f <= a rol 1;          -- výsledek "00000011"
```

```
g <= a ror 1;          -- výsledek "11000000"
```



Slučující operátor (concatenation) &

Slučuje jednorozměrná pole (včetně řetězců):

Příklad – použití pro sloučení dvou signálů:

a, b : IN bit_vector (1 DOWNTO 0);

c : OUT bit_vector (3 DOWNTO 0);

c <= a & b;

-- c(3) <= a(1); c(2) <= a(0); c(1) <= b(1); c(0) <= b(0);

Příklad – použití při definici posuvného registru:

VARIABLE shifted, shiftin : bit_vector (0 TO 3);

shifted := shiftin (1 TO 3) & '0';

Příklad – použití pro násobení a dělení ($Q = C / 16 + C * 4$):

SIGNAL C, Q : std_logic_vector (7 DOWNTO 0);

Q <= „0000“ & C(7 DOWNTO 4) + C(5 DOWNTO 0) & „00“;



Konverzní funkce

Knihovna ieee.std_logic_1164:

TO_BIT(arg)
TO_BITVECTOR(arg)
TO_STDLOGICVECTOR(arg)

z **std_logic** na **bit**
z **std_logic_vector** na **bit_vector**
z **bit_vector** na **std_logic_vector**

Knihovna ieee.std_logic_arith:

CONV_INTEGER(arg)
CONV_UNSIGNED(arg, b)
CONV_SIGNED(arg, b)
UNSIGNED(arg)
SIGNED(arg)
STD_LOGIC_VECTOR(arg)
CONV_STD_LOGIC_VECTOR(arg, b)

z **(un)signed** na **integer**
z **integer** a **signed** na **unsigned**
z **integer** a **unsigned** na **signed**
z **std_logic_vector** na **unsigned**
z **std_logic_vector** na **signed**
z **(un)signed** na **std_logic_vector**
z **integer** a **(un)signed** na **std_logic_vector**

(druhý parametr udává počet bitů výsledku)



Konverzní funkce (pokračování)

Knihovna ieee.numeric_std (ieee.numeric_bit):

(obsahuje funkce obdobné jako std_logic_arith, některé mají jiný název)

TO_INTEGER(arg)

z **(un)signed** na **integer**

TO_SIGNED(arg, b)

z **integer** a **unsigned** na **signed**

TO_UNSIGNED(arg, b)

z **integer** a **signed** na **unsigned**

Příklad: **library** ieee;

```
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_arith.all;
```

...

```
subtype byte is bit_vector (7 downto 0);
```

```
type mem_16x8 is array (0 to 15) of byte;
```

```
signal RAM : mem_16x8;
```

```
signal data : std_logic_vector (7 downto 0);
```

```
signal addr : bit_vector (0 to 3);
```

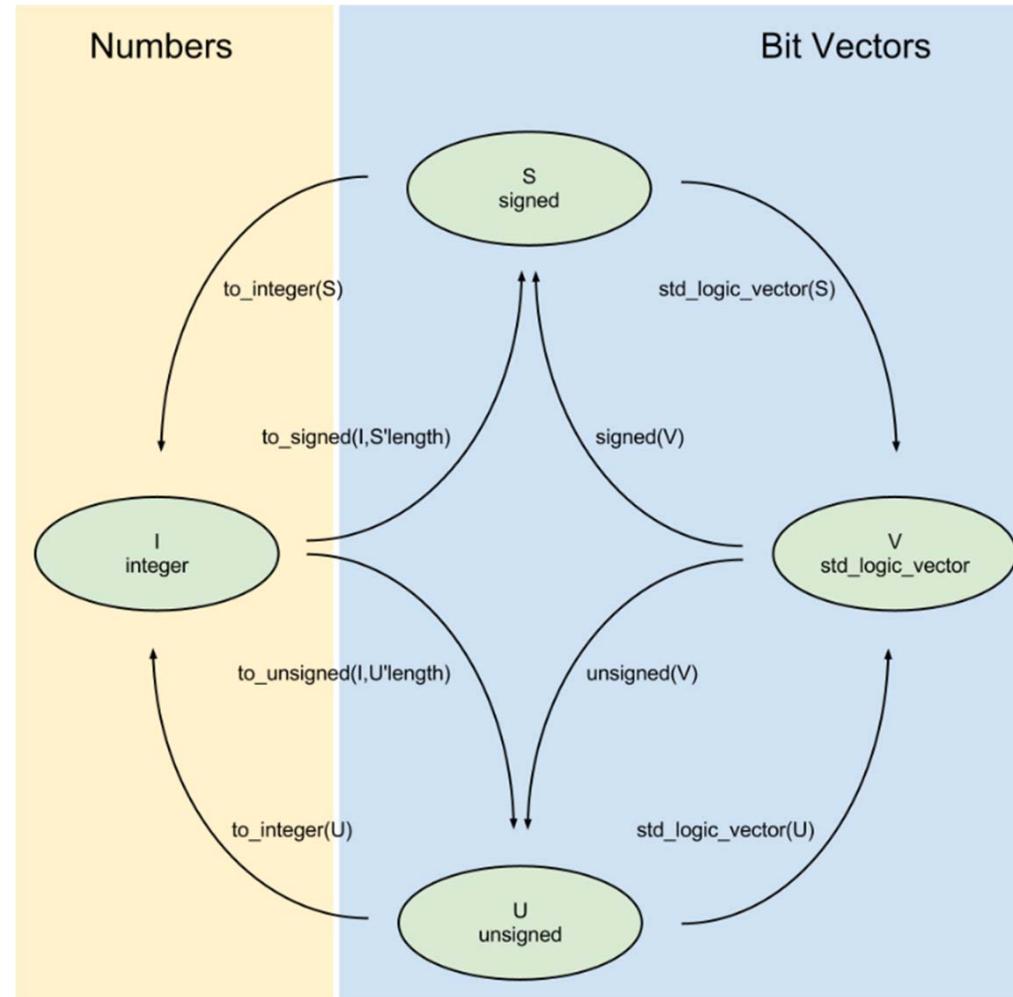
...

```
data <= to_stdlogicvector (RAM (conv_integer (to_stdlogicvector (addr))));
```

```
RAM (conv_integer (to_stdlogicvector (addr))) <= to_bitvector (data);
```



Konverzní funkce (pokračování)





Atributy

- Atributy jsou v podstatě speciální funkce, které poskytují dodatečnou informaci jejich nositelích;
- nositeli mohou být typy, signály, proměnné, vektory, pole, bloky, architektury, jednotky, ...

Atributy: *předdefinované* (typy, pole, signály, objekty);
definované uživatelem (nepříliš standardizované).

Syntaxe:

PREFIX'jméno_atributu[(výraz)]



Předdef. atributy – typy (podtypy)

Celkem 14 funkcí k získávání informací o typech nebo podtypech, nejčastější jsou:

- | | |
|---------------------|---|
| T'low | -- nejnižší mez typu T |
| T'high | -- nejvyšší mez typu T |
| T'left | -- hodnota levé krajní meze typu T |
| T'right | -- hodnota pravé krajní meze typu T |
| T'leftof(X) | -- hodnota na pozici vlevo od X |
| T'rightof(X) | -- hodnota na pozici vpravo od X |
| T'image(X) | -- převede výraz X na textový řetězec |
| T'value(X) | -- převede textový řetězec X na hodnotu |



Předdef. atributy – typy (příklady)

TYPE word_ab IS range 31 downto 0;

-- word_ab'left = 31

word_ab'right = 0

-- word_ab'high = 31

word_ab'low = 0

-- word_ab'leftof(20) = 21

word_ab'rightof(20) = 19

SUBTYPE shorter IS integer range 0 to 100 ;

-- shorter'high = 100

shorter'left = 0



Předdefinované atributy - pole

Celkem 8 funkcí k získávání informací o polích:

- A'low(N)** -- nejnižší hodnota indexu N-té dimenze pole A
- A'high(N)** -- nejvyšší hodnota indexu N-té dimenze pole A
- A'left(N)** -- hodnota indexu levé krajní meze N-té dimenze pole A
- A'right(N)** -- hodnota indexu pravé krajní meze N-té dimenze pole A
- A'length(N)** -- počet prvků N-té dimenze pole A
- A'range(N)** -- rozsah indexů N-té dimenze pole od A'left do A'right
- A'reverse_range(N)** -- obrácený rozsah indexů N-té dimenze pole A
(zamění *to* za *downto* či naopak)
- A'ascending(N)** -- booleovská hodnota *true*, pokud má N-tá dimenze pole vzestupný (*to*) rozsah, jinak *false*

(parametr N je možné u jednorozměrného pole vynechat)



Předdef. atributy – pole (příklady)

SIGNAL ab : bit_vector (7 downto 0);

-- ab'length = 8 ab'left = 7 ab'low = 0

TYPE pole8x4 IS array (8 downto 1, 0 to 3) of boolean;

-- pole8x4'left(1) = 8

$$\text{pole8x4}'\text{left}(2) = 0$$

-- pole8x4'high(1) = 8

pole8x4'right(2) = 3

-- pole8x4'low(1) = 1

pole8x4'length(2) = 4

-- pole8x4'ascending(1) = false

pole8x4'range(2) = 0 to 3

-- pole8x4'reverse_range(1) = 1 to 8



Předdefinované atributy - signály

Celkem 11 funkcí k získání informací o signálech, nejpoužívanější jsou:

- S'stable(T)** -- pravdivý, pokud za čas T nenastala událost na S
- S'event** -- pravdivé, pokud událost na S právě nastala
- S'last_value** -- předcházející hodnota před změnou S
- S'last_event** -- čas mezi současností a minulou událostí na S
- S'quiet(T)** -- pravdivý, pokud za čas T nenastala transakce na S
- S'active** -- pravdivé, pokud transakce na S právě nastala
- S'delayed(T)** -- vytvoří kopii signálu zpožděnou o čas T (není-li čas udán, je signál zpožděn o tzv. delta zpoždění).

Transakce je přepočet (update) konkrétní hodnoty signálu – nemusí znamenat změnu hodnoty (transakce musí nastat vždy, když nastane událost).

Událost je transakce, jejíž výsledkem je změna hodnoty.

Signál je množina událostí (změna hodnoty v konkrétní čas).



Předdef. atributy – signály (příklady)

```
IF (clk'event and clk='1') THEN out1 <= data; END IF;  
-- nastala-li na clk vzestupná hrana
```

```
WAIT FOR 30 ns;  
data <= '1' after 30 ns;  
WAIT FOR 10 ns;  
a := data'stable(20 ns); -- true (na data 20 ns zpátky nenastala událost)  
WAIT FOR 30 ns;  
a := data'stable(20 ns); -- false (před 10 ns nastala na data událost)
```



Atributy definované uživatelem

- slouží např. pro ovládání chodu syntezátoru nebo simulátoru (někde řešeno pomocí komentářů), závislé na konkrétním výrobci

Syntaxe:

ATTRIBUTE attribute_name : string;

ATTRIBUTE attribute_name **OF**
{component_name|label_name
|entity_name|signal_name|variable_name|type_name}:
{component|label|entity|signal|variable|type} IS
attribute_value;



Atributy def. uživatelem (příklady)

ATTRIBUTE loc : string; -- přiřazení pinů

ATTRIBUTE loc OF out_B: SIGNAL IS "P14 P15 P16"; -- piny 14, 15, 16

ATTRIBUTE io_types : string; -- druh V/V pinů

ATTRIBUTE io_types OF port_F: SIGNAL IS "LVCMOS33, 20"; -- 20 mA

ATTRIBUTE pull : string; -- připojení pull-up a pull-down odporů

ATTRIBUTE pull OF enabl_C: SIGNAL IS "DOWN"; -- neaktivní v log. 0

ATTRIBUTE enum_encoding : string;

ATTRIBUTE enum_encoding OF barvy : TYPE IS "000 100 010 001 101";
-- definice kódování hodnot výčtových datových typů



Příkaz LOOP (typy WHILE a FOR)

- příkaz lze používat pouze uvnitř procesu
- spouští opakovaně sekvence příkazů (smyčka)
- sekvenční příkaz

Syntaxe:

[návěstí] : **WHILE** podmínka **LOOP**
 sekvence příkazů
END LOOP [návěstí];

[návěstí] : **FOR** parameter **IN** rozsah **LOOP**
 sekvence příkazů
END LOOP [návěstí];



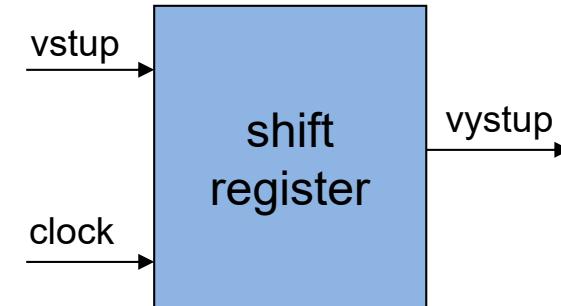
Příklad 4bit. posuvného registru

```

ENTITY shift_register IS
    PORT ( vstup, clock : IN BIT;
              vystup : OUT BIT );
END shift_register;

ARCHITECTURE a_shreg OF shift_register IS
    SIGNAL pom : BIT_VECTOR ( 3 DOWNTO 0 ) := "0000";
BEGIN
    PROCESS ( clock ) BEGIN
        IF ( clock'event) AND ( clock = '1' ) THEN
            vystup <= pom (3);
        FOR i IN 3 DOWNTO 1 LOOP
            pom (i) <= pom (i-1);
        END LOOP;
        pom (0) <= vstup;
    END IF;
    END PROCESS;
END a_shreg;           -- „i“ není nutné deklarovat, inicializovat, inkrementovat

```





Smyčka typu WHILE - příklad

- proměnnou „i“ je nutné deklarovat, inicializovat a inkrementovat

PROCESS (clk, Rst, D_in)

VARIABLE i : integer;

BEGIN

i := 0;

IF Rst = '1' THEN

WHILE i < 7 LOOP

Reg(i) <= (OTHERS => '0'); -- asynchronní nulování

i := i + 1;

END LOOP;

ELSIF (clock'event) AND (clock = '1') THEN

-- popis synchronní funkce

...

END IF;

END PROCESS;



Příkaz GENERATE

- pro popis pravidelných hardwareových struktur
- souběžný příkaz (nepoužívat v procesu)

Syntaxe:

```
label : FOR parametr IN rozsah GENERATE
[ { deklarace }
BEGIN ]
{ souběžné příkazy }
END GENERATE [ label ];
```

```
label : IF podmínka GENERATE           -- nelze použít else
[ { deklarace }
BEGIN ]
{ souběžné příkazy }
END GENERATE [ label ];
```

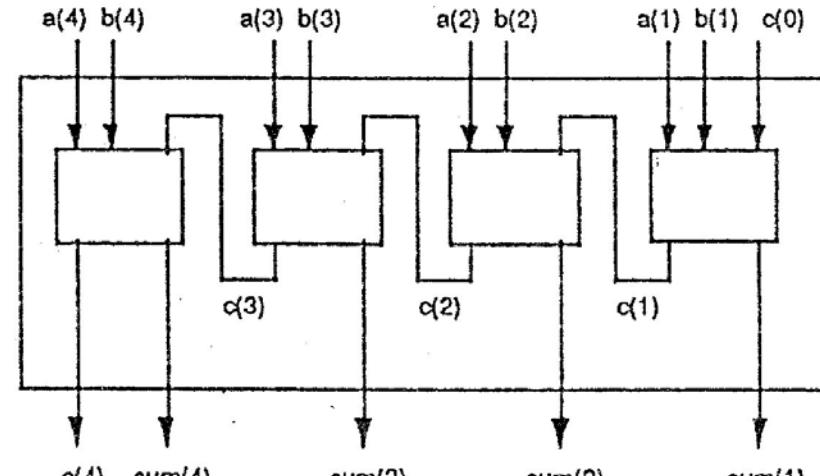


Příkaz GENERATE - příklad

Label : **FOR i IN 1 to 4 GENERATE**

jma PORT (a(i), b(i), c(i-1), c(i), sum(i));

END GENERATE;



Ekvivalentní příkazům:

U1: jma **PORT MAP** (a(1), b(1), c(0), c(1), sum(1));

U2: jma **PORT MAP** (a(2), b(2), c(1), c(2), sum(2));

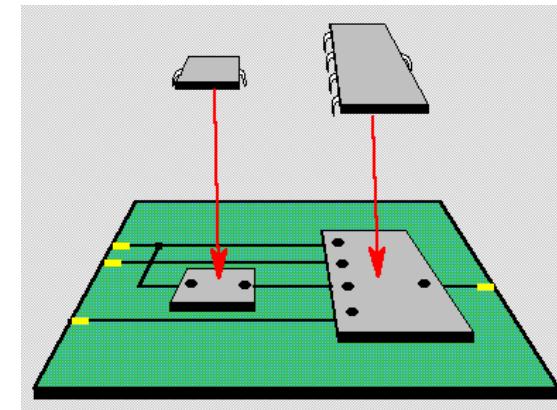
U3: jma **PORT MAP** (a(3), b(3), c(2), c(3), sum(3));

U4: jma **PORT MAP** (a(4), b(4), c(3), c(4), sum(4));



Komponenty (funkční bloky)

- vhodné pro skládání hotových funkčních bloků,
- komponenta musí být definována entitou,
- tato entita musí mít přiřazenu architekturu,
- komponenta musí být deklarována v deklarační části architektury nebo v package,
- lokální signály propojující komponenty musí být deklarovány v deklarační části architektury.





Deklarace komponenty

Syntaxe deklarace komponenty:

```
COMPONENT jméno_komponenty [ IS ]  
[ GENERIC ( jméno_parametru : integer := hodnota {;  
                 jméno_parametru : integer := hodnota } ) ] ;  
PORT (jméno_signálu {, jméno_signálu} : [mód] typ {;  
                 jméno_signálu {, jméno_signálu} : [mód] typ } );  
END COMPONENT [ jméno_komponenty ];
```



Použití komponenty podle jména

Syntaxe použití - asociace podle jména (named association):

jméno_instance : jméno_komponenty

[**GENERIC MAP** (formální_jméno => aktuální_jméno
{, formální_jméno => aktuální_jméno})]

PORT MAP (formální_jméno => aktuální_jméno
{, formální_jméno => aktuální_jméno});

Jméno_signálu při definici komponenty může být následně použito
jako „formální_jméno“ při použití komponenty.

Příklad:

U1 : decode PORT MAP (r => rd, op => in, w =>wr);



Použití komponenty podle polohy

Syntaxe použití - asociace podle polohy (positional association):

jméno_instance : jméno_komponenty

[**GENERIC MAP** (aktuální_jméno {, aktuální_jméno})]

PORT MAP (aktuální_jméno {, aktuální_jméno});

Příklad:

demx23 : demultipl GENERIC MAP (8) PORT MAP (in, rd, wr);



Komponenta - příklad

ENTITY mux2 IS

```
PORT (SEL, A, B : IN std_logic;
      F : OUT std_logic);
```

END mux2;

ARCHITECTURE structure OF mux2 IS

COMPONENT INV

```
PORT (A : IN std_logic ;
      F : OUT std_logic );
```

END COMPONENT ;

COMPONENT AOI

```
PORT (A, B, C, D : IN std_logic ;
      F : OUT std_logic );
```

END COMPONENT ;

SIGNAL SELB : std_logic ;

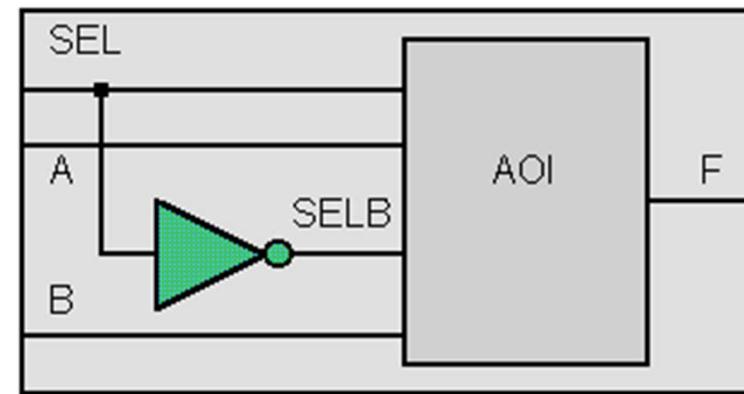
BEGIN

-- tělo architektury (vložení komponent)

G1 : INV PORT MAP (SEL, SELB) ;

G2 : AOI PORT MAP (SEL, A, SELB, B, F) ;

END structure ;





Přímé vkládání entity

- (direct entity instantiation) - dovoluje standard VHDL-93; někdy označováno také jako *přímé vkládání komponenty*;
- není nutné deklarovat komponentu (v deklarační části architektury je možné vynechat části COMPONENT ... END COMPONENT;)

Příklad (ekvivalentní předešlému příkladu):

ARCHITECTURE structure **OF** mux2 **IS**

SIGNAL SELB : std_logic;

BEGIN

G1 : **ENTITY** work.INV **PORT MAP** (SEL, SELB);

G2 : **ENTITY** work.AOI **PORT MAP** (SEL, A, SELB, B, F);

END structure;



Podprogramy

Dva typy: *funkce* a *procedury*

- jsou syntetizovány při každém volání;
- pro viditelnost podprogramů platí stejná pravidla jako pro datové typy;
- příkazy uvnitř funkcí a procedur se vykonávají sekvenčně (obdoba procesu – možnost použití proměnných, příkazů IF, CASE, LOOP, ...);
- proměnné uvnitř funkce nebo procedury si nezachovávají svoji hodnotu - jsou vždy inicializovány při jejich volání (na rozdíl od procesu).



Funkce

- podprogram, který vrací hodnotu (datový typ);
- funkce mají vstupní operandy a jeden výstupní operand;
- typ hodnoty může být skalární nebo složený;
- ve funkci nemohou být deklarovány signály (pouze konstanty a proměnné – jejich platnost je omezena pouze na funkci).

Syntaxe definice funkce:

FUNCTION jméno_funkce (parametry) RETURN typ IS

deklarace

BEGIN

sekvenční příkazy

END jméno_funkce ;



Funkce - příklad

ENTITY decoder **IS**

```
PORT ( input : IN BIT_VECTOR(0 TO 1);  
       output : OUT BIT_VECTOR(0 TO 3));
```

END decoder;

ARCHITECTURE arch_fce **OF** decoder **IS**

```
FUNCTION bit_to_int (bit_in : IN BIT) RETURN INTEGER IS
```

BEGIN

```
    IF bit_in = '0' THEN RETURN 0;  
    ELSIF bit_in = '1' THEN RETURN 1;  
    ELSE RETURN 0; END IF;
```

END bit_to_int;

BEGIN

...

```
    State := bit_to_int (input(0));      -- volání funkce
```

...

END arch_fce;



Posuvný registr – pomocí funkce

PACKAGE ops IS

FUNCTION shift (

shftreg : bit_vector(3 DOWNTO 0);
bitin : bit)

RETURN bit_vector;

END ops;

PACKAGE BODY ops IS

FUNCTION shift (shftreg : bit_vector(3 DOWNTO 0); bitin : bit)

RETURN bit_vector IS

VARIABLE result : bit_vector(3 DOWNTO 0);

BEGIN

result(2 DOWNTO 0) := shftreg(3 DOWNTO 1);
result(3) := bitin;

RETURN result;

END shift;

END ops;



Procedury

- podprogram, který modifikuje vstupní parametry;
- módy parametrů mohou být: IN, OUT, INOUT (implicitně IN);
- proceduře lze předávat signály, konstanty i proměnné;
- z procedury lze volat další proceduru.

Syntaxe definice procedury:

PROCEDURE jméno_procedury (seznam_parametrů) IS

deklarace

BEGIN

sekvenční příkazy

END jméno_procedury;



Volání procedury

- asociace parametrů podle jména nebo podle polohy;

Příklady volání procedury:

```
PROCEDURE examine_data (my_port : IN string;  
                      read_data : OUT bit_vector(0 TO 23);  
                      prop_delay : IN time := 1 ns);
```

- a) `examine_data ("shifter", data_contents, 25 ns);`
- b) `examine_data (my_port => "shifter",
 prop_delay => 25 ns,
 read_data => data_contents);`
- c) `examine_data ("shifter", data_contents);`
-- defaultně je použito `prop_delay = 1 ns`



Volání procedury

v citlivých proměnných uvést všechny IN a INOUT parametry

Dva ekvivalentní příklady:

ARCHITECTURE a OF e IS

BEGIN

 aprocedure (in_sig1, inout_sig2, const1);

END a;

ARCHITECTURE a OF e IS

BEGIN

PROCESS (in_sig1, inout_sig2)

BEGIN

 aprocedure (in_sig1, inout_sig2, const1);

END PROCESS;

END a;



Procedura - příklad

ENTITY add IS

```
PORT ( arg1, arg2 : IN BIT_VECTOR(3 DOWNTO 0);  
       result : OUT BIT_VECTOR(3 DOWNTO 0); carry : OUT BIT );
```

END add ;

ARCHITECTURE arch_proced **OF** add **IS**

```
PROCEDURE add_carry (SIGNAL a1, a2 : IN BIT_VECTOR(3 DOWNTO 0);  
SIGNAL result : OUT BIT_VECTOR(3 DOWNTO 0); SIGNAL carry : OUT BIT) IS  
VARIABLE temp : BIT_VECTOR(4 DOWNTO 0);
```

BEGIN

```
    temp := ("0" & a1) + ("0" & a2);  
    carry <= temp(4);  
    result <= temp(3 DOWNTO 0);
```

END add_carry;

BEGIN

```
    ...  
    add_carry (arg1, arg2, result, carry); -- volání procedury
```

...

END arch_proced;



Definice stavových automatů

Popis stavového automatu musí obsahovat:

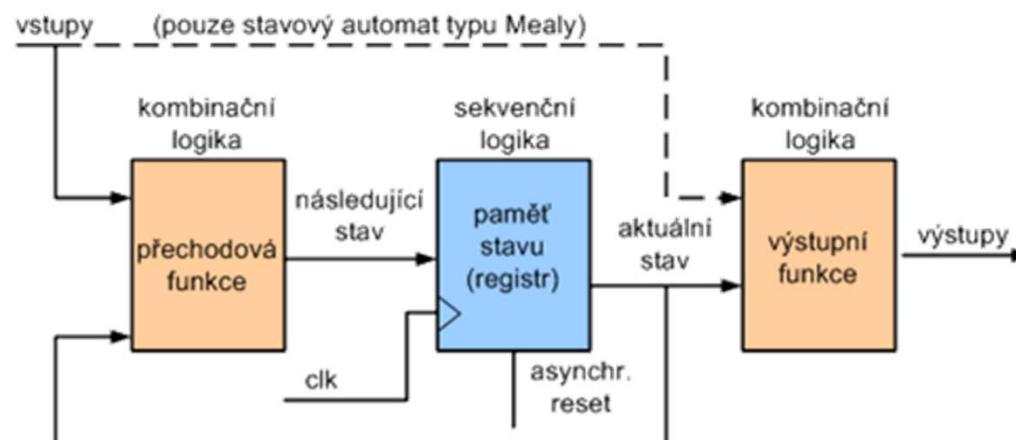
- stavovou proměnnou (současný a následující stav)
- hodinový signál (pouze jediný)
- specifikace změny stavu ($Q_n \Rightarrow Q_{n+1}$)
- specifikace výstupů
- reset (synchronní nebo asynchronní) – není nutné
- každá definice by měla mít svůj proces, příp. procesy
 - a) **jednoprocesorový popis automatu**
(registrová část je popsána procesem)
 - b) **dvouprocesorový popis automatu**
(kombinační i registrová část je popsána jedním procesem)
 - c) **tříprocesorový popis automatu** (proces popisující hodiny, proces pro změnu stavu a proces pro definování výstupů)



Stavové automaty

Automaty typu Mealy a Moore

- oba typy automatů jsou mezi sebou převoditelné,
- výstup Mealyho automatu se mění ihned po změně vstupu,
- Mooreho automat může vést na větší počet vnitřních stavů,
- Mealyho automat může mít složitější přechodovou a výstupní funkci,
- výstupní funkce je u obou automatů kombinační → možnost hazardů → za výstupní funkci se někdy doplňují registry.





Stavové proměnné

- musí být signály nebo proměnné,
- musí být následujícího typu:
enumerated, **integer**, **bit**, **bit_vector**, **boolean**
- nesmí být „port“ signál,
- operace nad stavovou proměnnou je limitována na “=” a “/=”,
- při behaviorálním stylu popisu se obecně ve VHDL nedefinuje chování automatu v nevyužitých stavech,
- pokud automat nemá nevyužité stavы, mluvíme o *spolehlivém* (safe) stavovém automatu,
- ošetření nevyužitých stavů ale způsobuje nárůst logiky.



Kódování vnitřních stavů

- Pro zakódování „ n “ vnitřních stavů je třeba „ k “ bitů ($2^k \geq n \geq k$) (předpokládáme jednoznačné kódování stavů),
- kódování vnitřních stavů vznikne při syntéze (lze ovlivnit nastavením návrhového systému),
- volba kódování má významný vliv na složitost a dynam. parametry.

Kódování	Sekvenční	Gray	Johnson	One-hot	Almost O-h	One-cold
Stav0	0000	0000	00000	0000000001	0000000000	1111111110
Stav1	0001	0001	00001	0000000010	0000000001	1111111101
Stav2	0010	0011	00011	0000000100	0000000010	1111111011
Stav3	0011	0010	00111	0000001000	000000100	1111110111
Stav4	0100	0110	01111	0000010000	000001000	1111101111
Stav5	0101	0111	11111	0000100000	000010000	1111011111
Stav6	0110	0101	11110	0001000000	000100000	1110111111
Stav7	0111	0100	11100	0010000000	001000000	1101111111
Stav8	1000	1100	11000	0100000000	010000000	1011111111
Stav9	1001	1101	10000	1000000000	100000000	0111111111



Moorův stavový automat - příklad

ENTITY moor **IS**

```
PORT (clk, x, rst : IN std_logic;  
      z : OUT std_logic);
```

END moor;

ARCHITECTURE ar_moor **OF** moor **IS**

```
TYPE states IS (s0, s1);
```

```
SIGNAL state : states := s0;
```

```
SIGNAL nxt_state : states := s0;
```

BEGIN

```
clkd: PROCESS (clk, rst) -- Proces pro zachycení hodnoty stavu
```

BEGIN

```
IF (rst = '0') THEN
```

```
state <= s0;
```

```
ELSIF (clk'EVENT AND clk = '1') THEN
```

```
state <= nxt_state;
```

```
END IF;
```

```
END PROCESS cldk;
```



Moorův stavový automat (pokrač.)

state_trans: PROCESS (state, x) -- Proces k určení následujícího stavu

BEGIN

CASE state IS

WHEN s0 => IF (x = '1') THEN nxt_state <= s1;
ELSE nxt_state <= s0; **END IF;**

WHEN s1 => IF (x = '1') THEN nxt_state <= s0;
ELSE nxt_state <= s1; **END IF;**

END CASE;

END PROCESS state_trans;

output: PROCESS (state) -- Proces pro definici výstupu

BEGIN

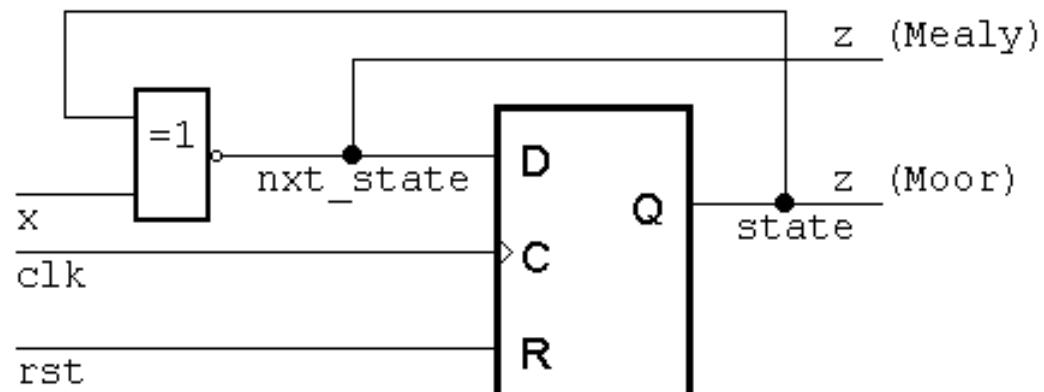
CASE state IS

WHEN s0 => z <= '0';
WHEN s1 => z <= '1';

END CASE;

END PROCESS output;

END ar_moor;





Mealyho stavový automat - příklad

Kromě procesu definujícího výstupní hodnoty je vše stejné s Moorovým automatem

output: **PROCESS** (state, x) -- Proces pro definici výstupu

BEGIN

CASE state **IS**

WHEN s0 => **IF** (x = '1') **THEN** z <= '1';
ELSIF (x = '0') **THEN** z <= '0';
ELSE z <= 'X';
END IF;

WHEN s1 => **IF** (x = '1') **THEN** z <= '0';
ELSIF (x = '0') **THEN** z <= '1';
ELSE z <= 'X';
END IF;

END CASE;

END PROCESS output;



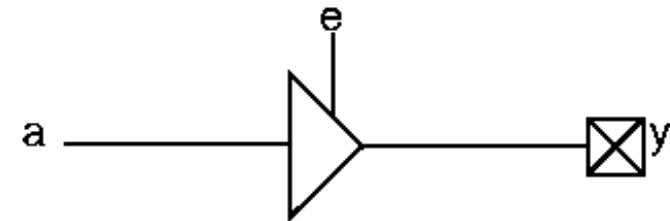
Třístavové výstupy - příklad

```
LIBRARY IEEE;  
USE IEEE.STD_LOGIC_1164.ALL;
```

```
ENTITY tristate IS  
PORT (e, a : IN std_logic;  
      y : OUT std_logic);  
END tristate;
```

```
ARCHITECTURE tri OF tristate IS  
BEGIN  
  PROCESS (e, a)  
  BEGIN  
    IF e = '1' THEN y <= a;  
    ELSE y <= 'Z'; END IF;  
  END PROCESS;  
END tri;
```

Proces je možné nahradit přiřazením: $y \leq a$ WHEN $(e = '1')$ ELSE ' Z ';



Nesyntetizovatelné:

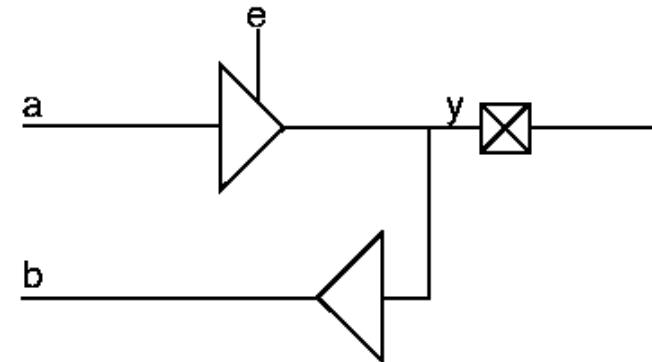
$a \leq b$ and ' Z ';

if $a = 'Z'$ then ...



Obousměrná komunikace - příklad

```
LIBRARY IEEE;  
USE IEEE.STD_LOGIC_1164.ALL;  
  
ENTITY bidir IS  
PORT ( y : INOUT std_logic;  
       e, a : IN std_logic;  
       b : OUT std_logic );  
END bidir;  
  
ARCHITECTURE bi OF bidir IS  
BEGIN  
    PROCESS (e, a) BEGIN  
        CASE e IS  
            WHEN '1' => y <= a;  
            WHEN '0' => y <= 'Z';  
            WHEN OTHERS => y <= 'X';  
        END CASE;  
    END PROCESS;  
    b <= y;  
END bi;
```





Zpoždění

- všechny přiřazovací příkazy ve VHDL představují určité zpoždění, lze ale i přesně specifikovat;
- zpoždění má jednu ze tří forem:
 - *Transport delay*
 - *Inertial delay*
 - *Delta delay* (standardní zpoždění, není-li specifikováno jinak)
 $\text{output} \leq \text{NOT input}$;
- zpoždění se uplatňuje u signálů (ne proměnných);
- zpoždění se nerespektuje při syntéze (pouze při simulaci).

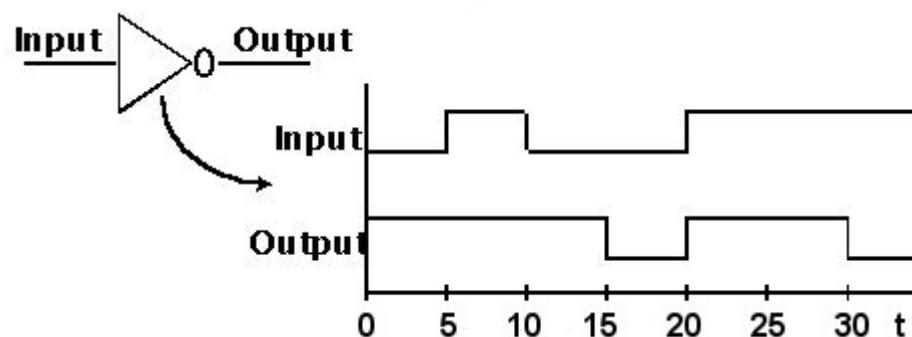


Zpoždění typu Transport

Uvozeno klíčovým slovem TRANSPORT

- signál nabývá nové hodnoty po specifikovaném zpoždění

output <= TRANSPORT NOT input AFTER 10 ns;



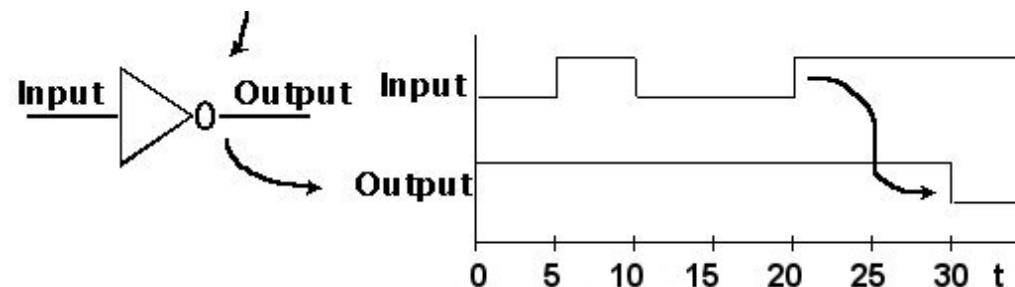


Zpoždění typu Inertial

INERTIAL zpoždění je defaultní, REJECT je volitelný

output <= **REJECT** 5 ns **INERTIAL NOT** input **AFTER** 10 ns;
- neprojde impuls kratší než 5 ns

output <= NOT input **AFTER** 10 ns;
- neprojde impuls kratší než 10 ns





Simulace ve VHDL (Testbench)

- Simulace na nejvyšších úrovních hierarchie (DTL – Design Top Level);
- na DTL aplikujeme budicí signály (stimuly) a sledujeme výstupní signály (response);
- stimuly píšeme ve VHDL
 - deklarace entity je prázdná,
 - deklarace komponenty a signálů je povinná,
 - u signálů definujeme časové průběhy;
- výhodou je přenositelnost mezi návrhovými systémy;
- některé simulátory vybaveny WYSIWYG prostředky pro editaci testbenchů;
- počáteční inicializace – lze specifikovat, implicitně - výčtový typ: první hodnota; integer: 0; real: 0.0; std_logic: 'U'.



Testbench - příklad

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY test_mux4 IS END;                                -- prázdná entita
ARCHITECTURE bench OF test_mux4 IS
COMPONENT mux4                                         -- deklarace komponenty
    PORT ( sel : IN std_logic_vector(1 DOWNTO 0);
            a, b, c, d : IN std_logic;
            f : OUT std_logic );
END COMPONENT;
SIGNAL sel : std_logic_vector (1 DOWNTO 0);           -- vstupní a výstupní signály
SIGNAL a, b, c, d, f : std_logic;
BEGIN
    M : mux4 PORT MAP ( sel, a, b, c, d, f );          -- vložení instance komponenty
    sel <= "00", "01" AFTER 30 ns, "10" AFTER 60 ns,      -- časové průběhy
        "11" AFTER 90 ns, "XX" AFTER 120 ns, "00" AFTER 130 ns;
    a <= 'X', '0' AFTER 10 ns, '1' AFTER 20 ns;
    b <= 'X', '0' AFTER 40 ns, '1' AFTER 50 ns;
    c <= 'X', '0' AFTER 70 ns, '1' AFTER 80 ns;
    d <= 'X', '0' AFTER 100 ns, '1' AFTER 110 ns;
END bench;
```



TECHNICKÁ UNIVERZITA V LIBERCI
**Fakulta mechatroniky, informatiky
a mezioborových studií**



Zásady návrhu elektronických systémů

Milan Kolář

Ústav mechatroniky a technické informatiky



evropský
sociální
fond v ČR



EVROPSKÁ UNIE



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY



Projekt ESF CZ.1.07/2.2.00/28.0050
**Modernizace didaktických metod
a inovace výuky technických předmětů.**

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ



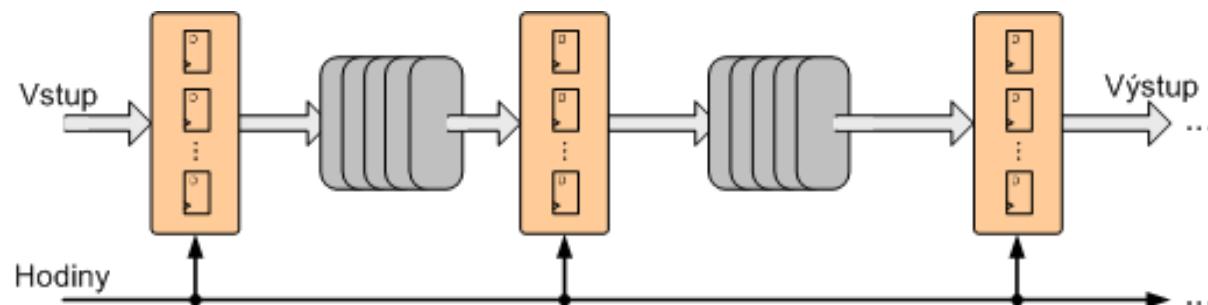
Zápis v HDL jazycích

Nejčastěji používáme úroveň **RTL** (Register Transfer Level).

Datová část - v podstatě se jedná o střídání kombinační logiky (výpočetní jednotky) a registrů pro uchovávání dat.

Řídicí část – stavové automaty - opět se střídá kombinační logika (přechodové a výstupní funkce) s registry (pro vnitřní stav).

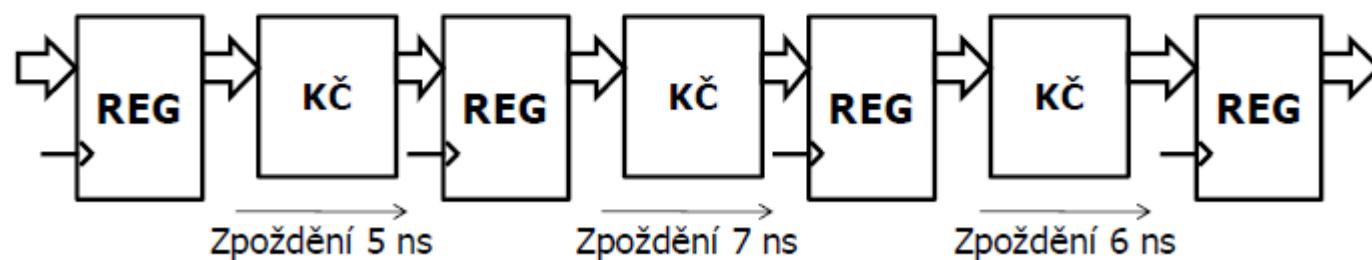
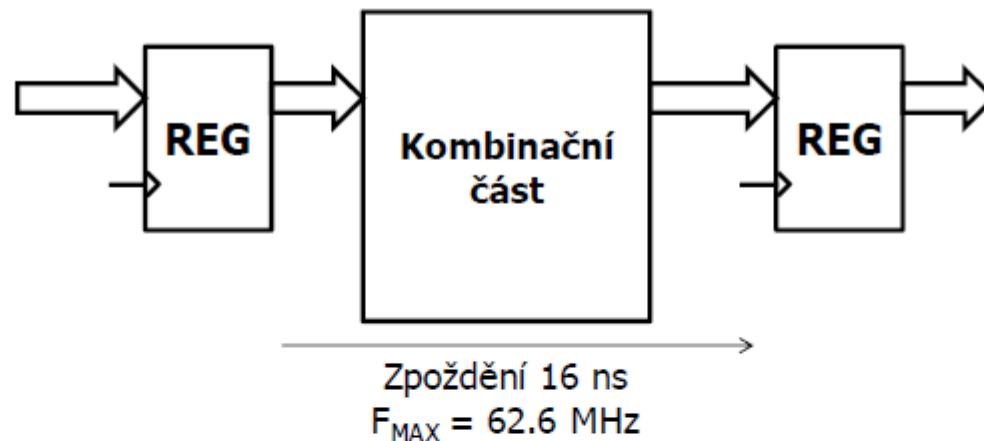
=> vzniká „sendvičová“ struktura registrů a kombinační logiky (přelévání dat mezi registry)
– podpora v hardwaru FPGA (LE = LUT + klopný obvod).





Technika zřetězení (pipelining)

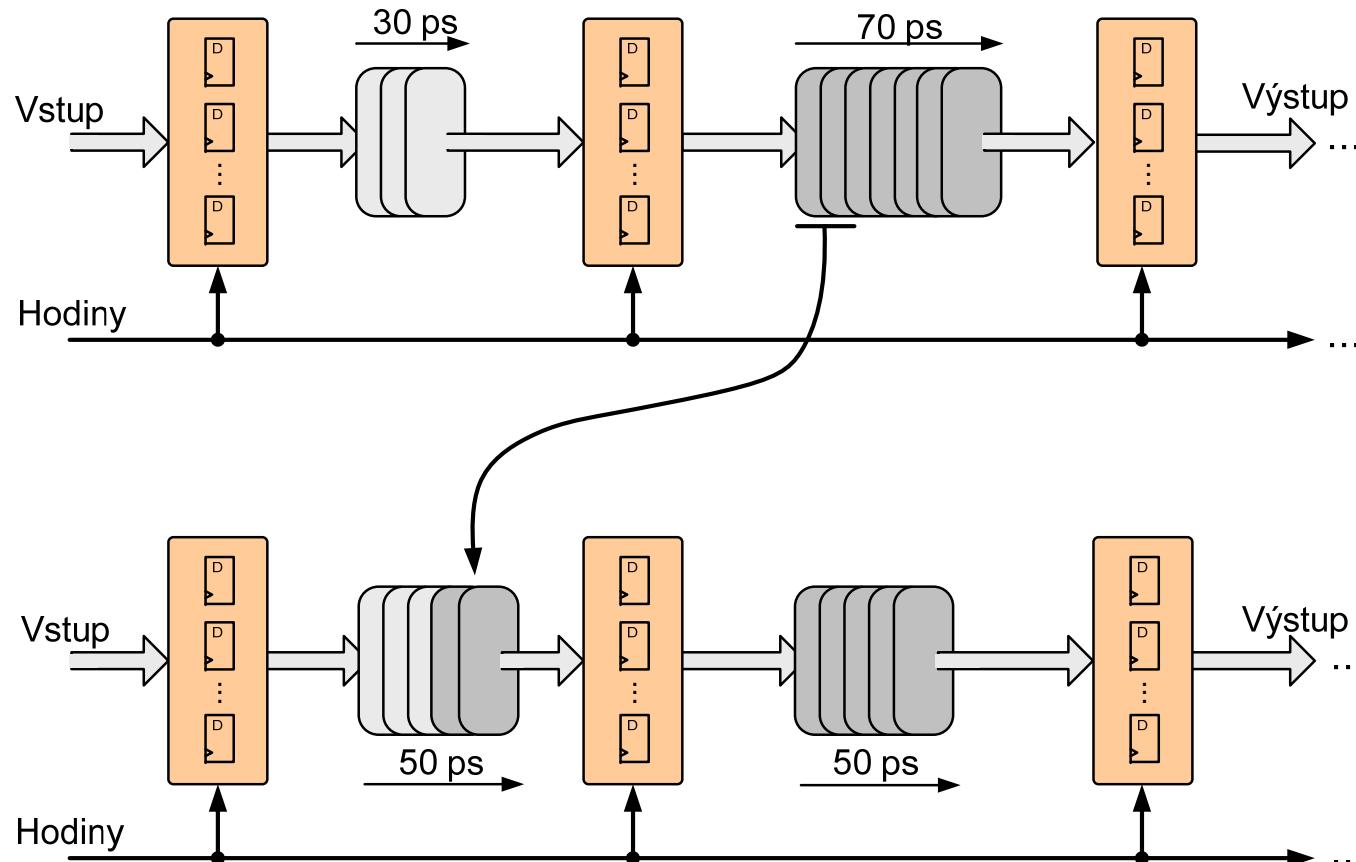
Umožňuje zvyšování pracovního kmitočtu v systému



Největší zpoždění 7 ns => $F_{MAX} = 143 \text{ MHz}$



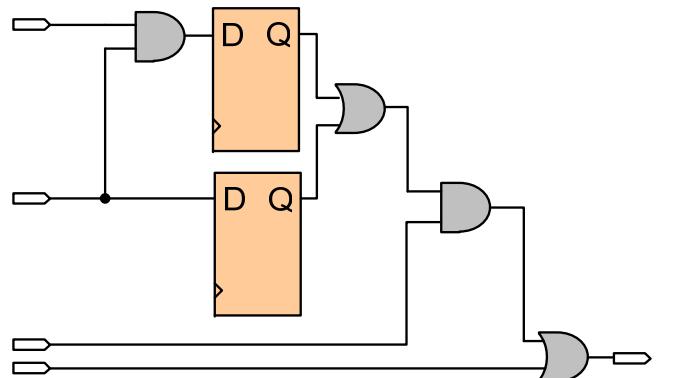
Synchronizace registrů (retiming)



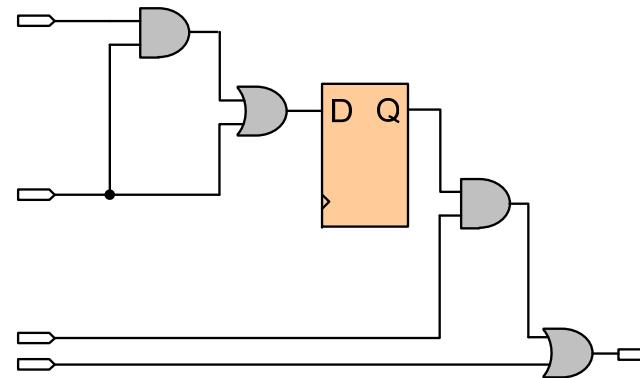


Příklad synchronizace registrů

Vyrovnání kombinačních logických cest mezi registry



Před synchronizací



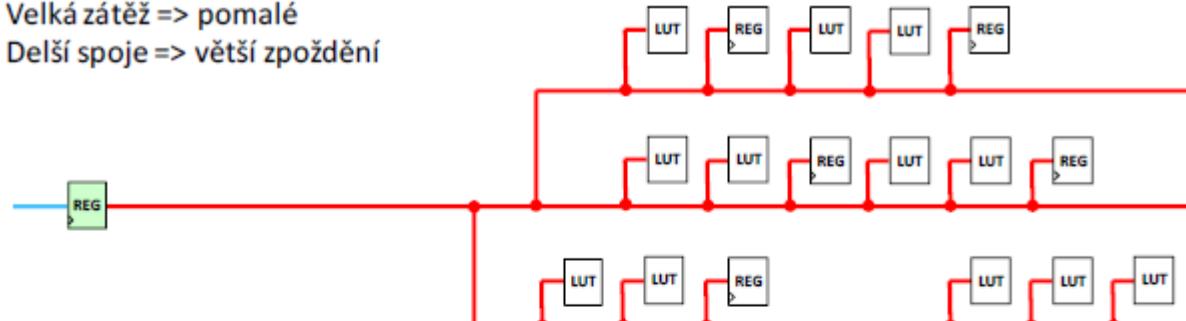
Po synchronizaci



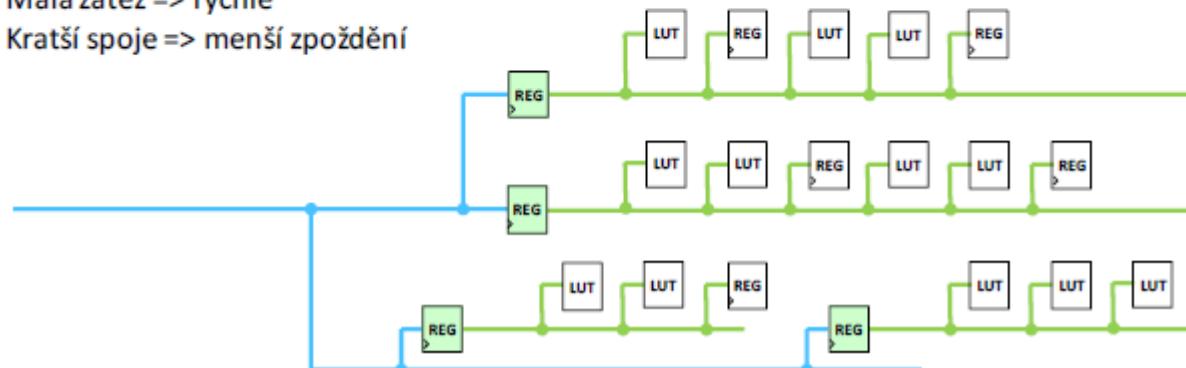
Metoda replikace registrů

Replikace a rozmístění registrů do míst, kde jsou cílové obvody

Velká zátěž => pomalé
Delší spoje => větší zpoždění

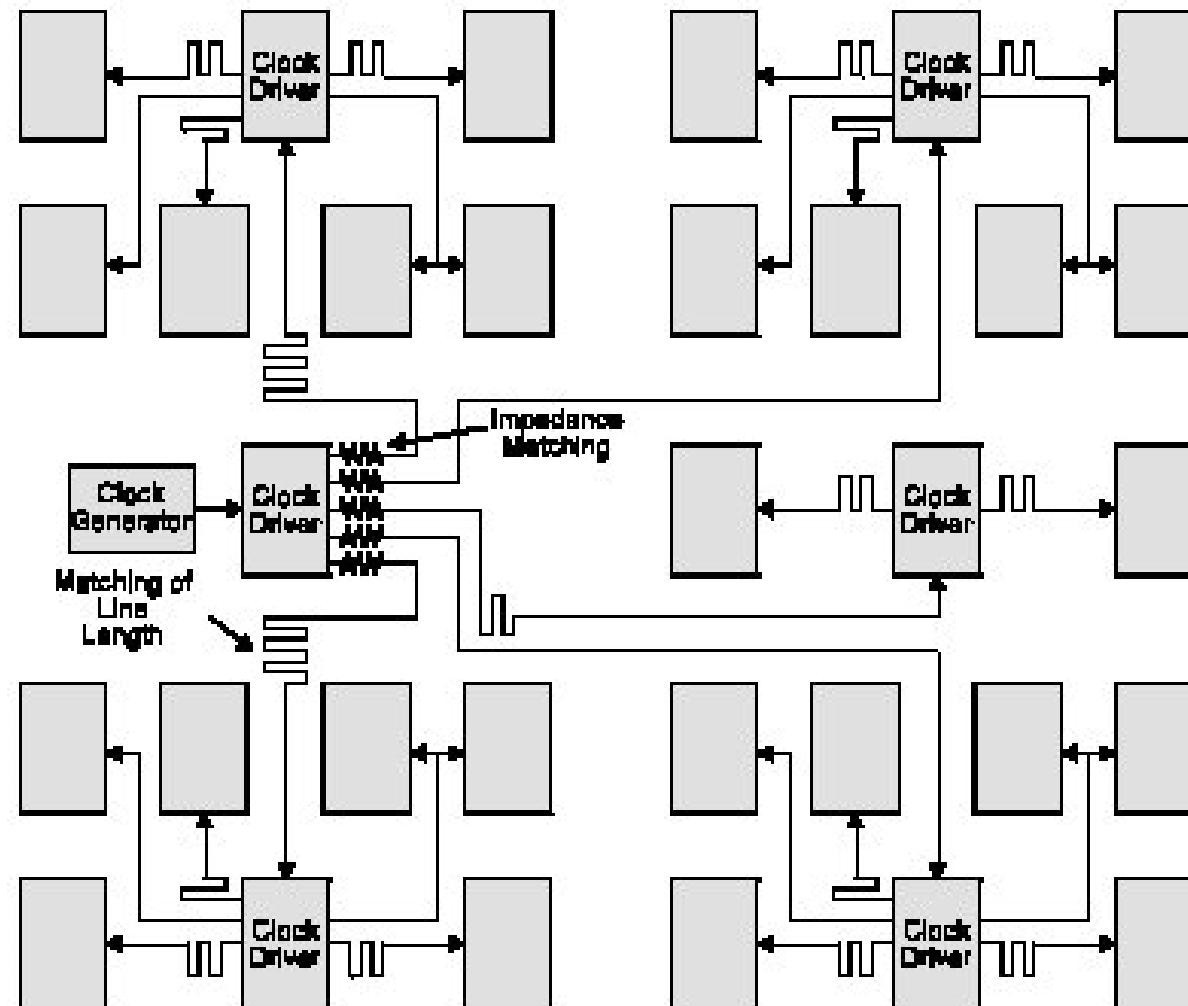


Malá zátěž => rychlé
Kratší spoje => menší zpoždění





Hodinový signál s budiči





Sdílení prostředků

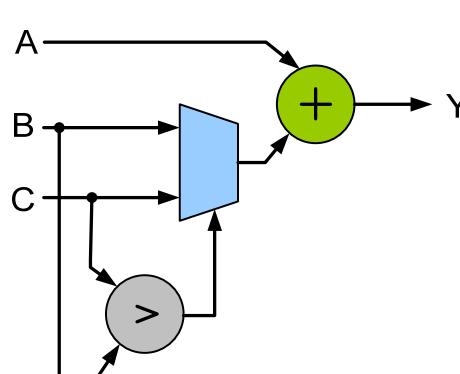
Sdílení prostředků snižuje množství potřebné logiky,
ale snižuje se také maximální pracovní frekvence

Příklad realizace popisu: if (B > C)

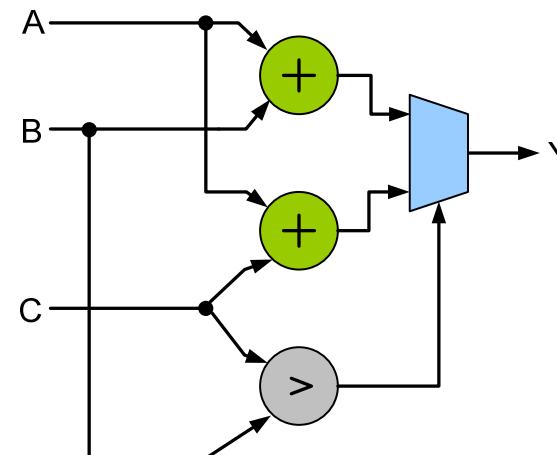
then Y = A + B;

else Y = A + C;

end if;



Se sdílením prostředků



Bez sdílení prostředků



Synchronní návrh

Časová doména – část obvodu, jehož registry jsou buzeny stejným hodinovým signálem.

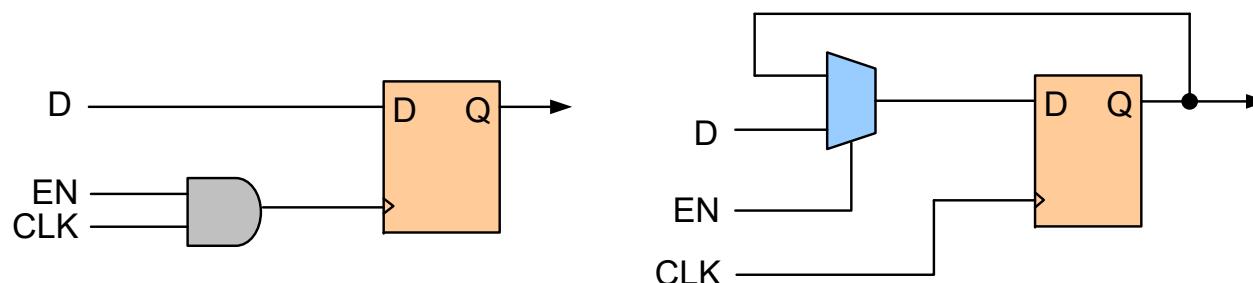
Vše navrhovat synchronně (nejlépe 1 časová doména, příp. důsledně oddělit jednotlivé časové domény)

- podpora v návrhových systémech,
- snadno testovatelné,
- odolnější vůči šumům (přeslechy, odrazy, hazardy, ...),
- funkčnost nezávisí na konkrétním rozmístění a propojení,
- funkční simulace souhlasí se simulací časovou
- při použití více časových domén většinou časové simulace neodhalí problémy (zejména možné metastability).



Pravidla synchronního návrhu

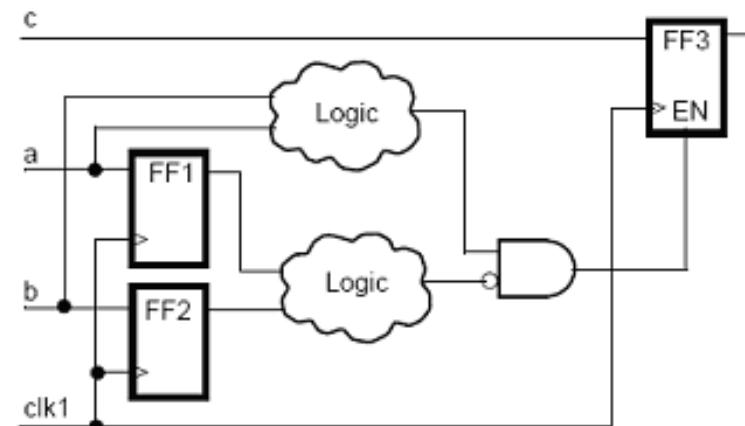
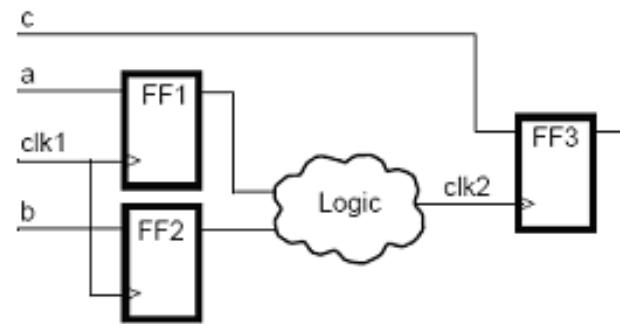
- na hodinové vstupy všech prvků jsou přivedeny pouze hodinové signály (bez přídavné logiky), případné „zastavení“ hodin řešíme klopnými obvody se vstupy „clock enable“;
- všechny klopné obvody jsou hranové řízené (ne hladinově řízené obvody - latches);
- zpětné vazby v kombinačních obvodech se v návrhu nevyskytují (ZV vedou na asynchronní sekvenční logiku);
- asynchronní signály jsou synchronizovány (viz dále).





Jednotné hodinové signály

Výstupy z logiky (přenosy čítačů, výstupy dekodérů a apod.) nejsou použity jako hodinové signály



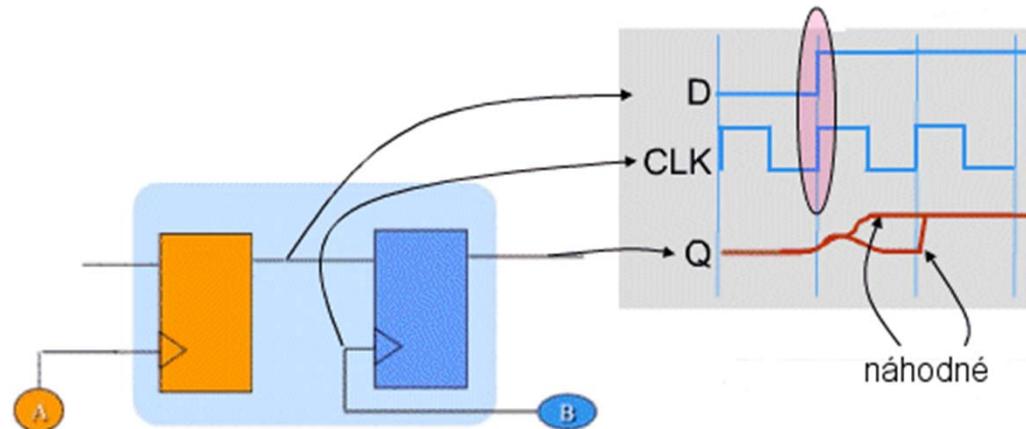


Přenos signálů mezi čas. doménami

Data se na vstupu KO mohou měnit i v době příchodu aktivní hrany hodinového signálu

=> vznikají kritická časová okna, kdy není dodržena doba předstihu t_s (t_{set-up}), doba přesahu t_h (t_{hold}), příp. doba zotavení po resetu t_{rr}

=> na výstupu KO může vzniknout nedefinovaný stav, který mohou následné vstupy vyhodnotit rozdílně.



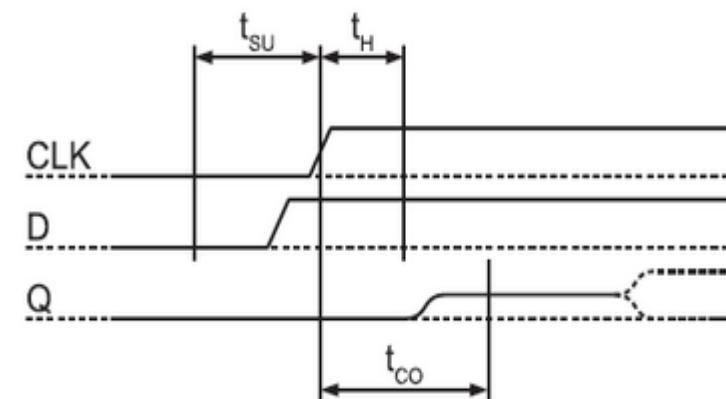
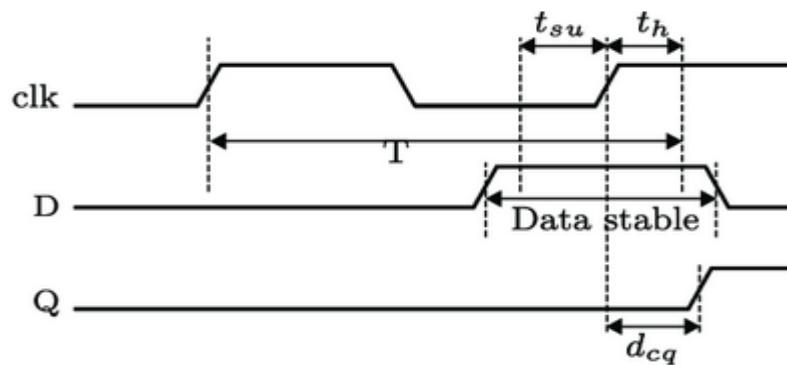


Metastability

Metastabilitou nazýváme neschopnost výstupu registru ustálit se na definované logické úrovni v přesně definovaném čase, obvykle za jednu hodinovou periodu.

Důsledkem metastabilit mohou vznikat:

- proudové špičky na napájení,
- nekorektní přechody mezi stavům stavových automatů,
- nekorektní hodnoty na sběrnicích.

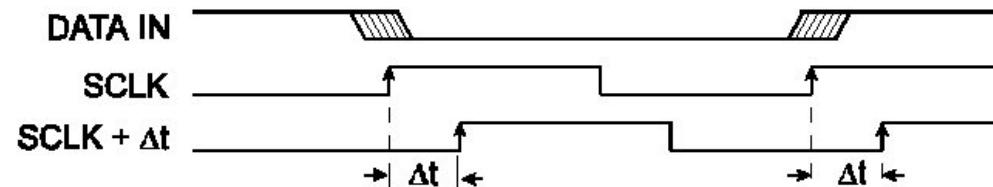




MTBF

MTBF (Mean-time-between-failures) - střední doba bezporuchového provozu (mezi chybami v synchronizaci):

$$MTBF = \frac{e^{T \cdot \Delta t}}{f_{in} \cdot f_{clk} \cdot T_0}$$



T , T_0 ... pravděpodobnosti výskytu chyby charakteristické pro daný typ logického obvodu (souvisí s technologií a architekturou daného obvodu)

f_{in} ... kmitočet vstupního signálu

f_{clk} ... kmitočet synchronizačního signálu

Δt ... doba zpožděněho vzorkování výstupu KO (doba pro stabilizaci)



Poruchovost návrhu

Poruchovost celého návrhu Q_C je dána:

$$Q_C = \frac{1}{MTBF_C} = \sum_{i=1}^n \frac{1}{MTBF_i}$$

⇒ časových domén by mělo být co nejméně a $MTBF_i$ by v návrhu měly být všechny srovnatelné.

Příklad:

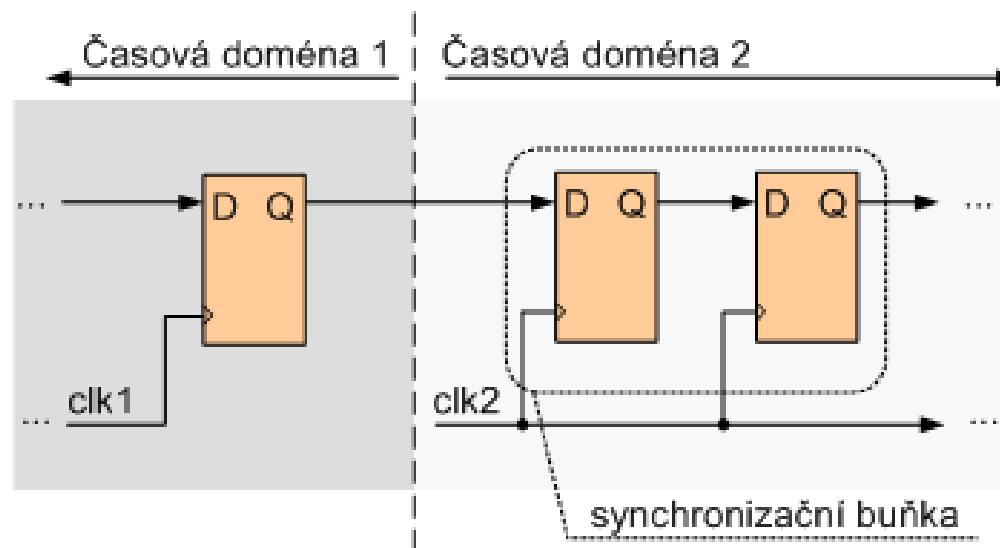
U obvodu Altera Flex10K ($T_0 = 1,01 \cdot 10^{-13}$ s, $T = 1,27 \cdot 10^{10}$ s $^{-1}$) chceme, aby k chybě došlo jednou za 100 let ($MTBF = 3,2 \cdot 10^9$ s) při $f_{in} = 100$ kHz a $f_{clk} = 10$ MHz. Testování signálu musí být prováděno se zpožděním:

$$\Delta t = \frac{\ln(MTBF \cdot f_{in} \cdot f_{clk} \cdot T_0)}{T} = 2 \text{ ns}$$



Synchronizátory

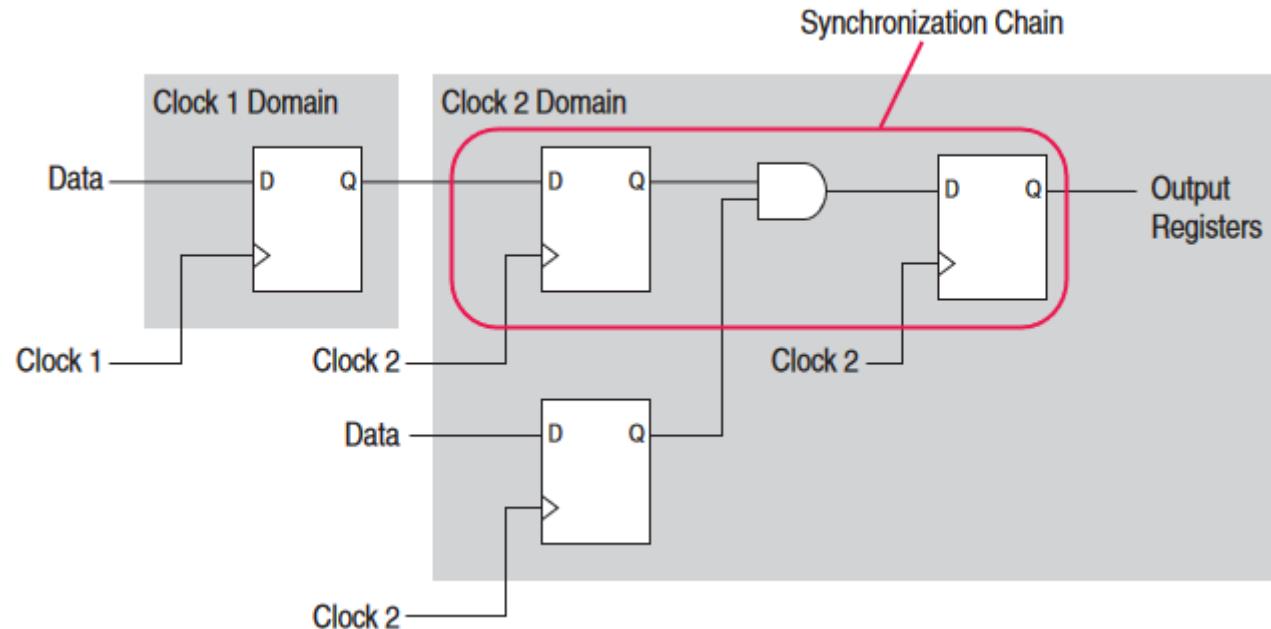
Pro snížení pravděpodobnosti vzniku metastability se používají dvojnásobné (výjimečně i trojnásobné) synchronizátory (synchronizační buňky) – pro jednotlivé signály.





Synchronizátory

Cesta mezi synchronizačními registry může obsahovat kombinacní logiku, pokud jsou všechny registry řetězce ve stejné časové doméně (nevětvit ale signál mezi synchronizátory).

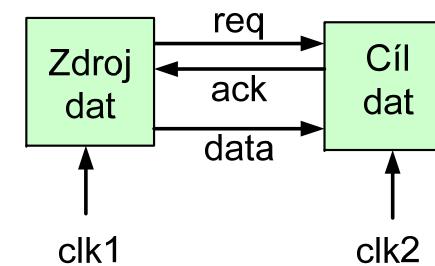
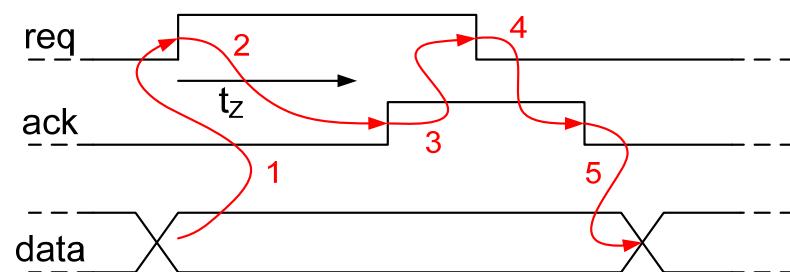
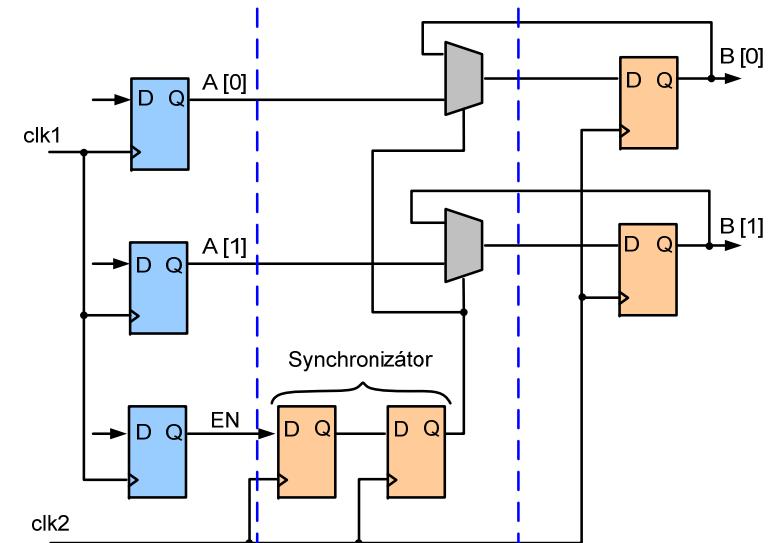




Synchronizace sběrnic

Problém se synchronizací n-bit. sběrnic

- vlivem nestejných zpoždění jednotlivých bitů by mohlo dojít k zápisu nesprávné hodnoty;
- je-li $f_{clk1} \leq f_{clk2}$, lze signálem *EN* potvrzovat platnost dat;
- je-li $f_{clk1} \geq f_{clk2}$, nutno zavést zpětnou vazbu (handshake) – data se nesmí ztratit ani zdublikovat (*req* a *ack* je třeba přesynchronizovat) => pomalejší.

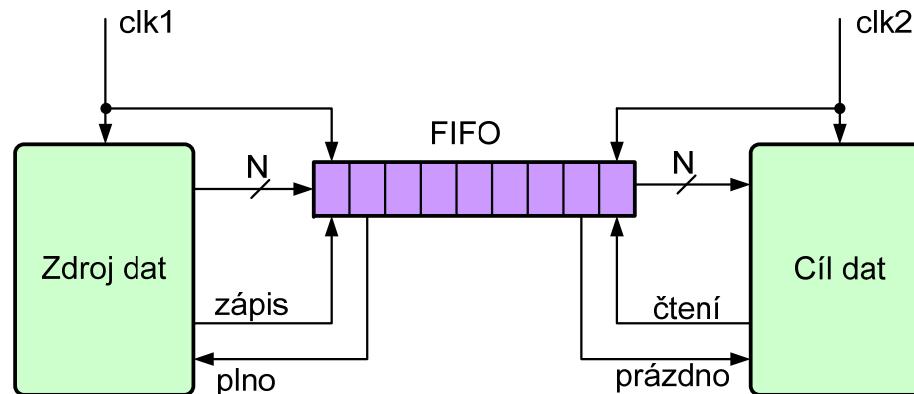




Přenos dat mezi čas. doménamí

Pro přenos dat mezi časovými domény lze použít fronty s využitím paměti FIFO s asynchronními hodinami (dual clock FIFO)

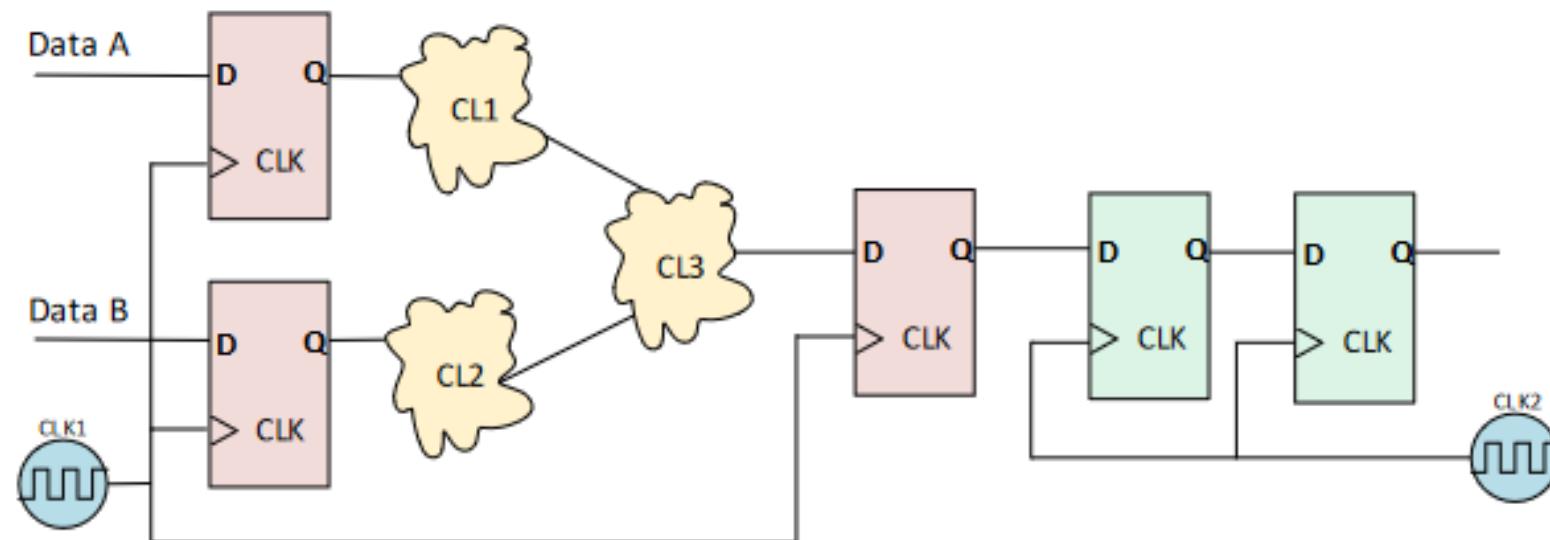
- využíváme zejména v případě, kdy je zpoždění spojené s resynchronizací nepřípustné.





Krátké pulsy v signálu

Velmi krátký puls (glitch) v datovém signálu (v době t_{set-up} nebo t_{hold}) vznikající např. vlivem hazardů, příp. krátký rušivý puls v hodinovém signálu mohou způsobit také metastabilní chování. Proto případnou výstupní kombinační logiku z časové domény doplňujeme KO.

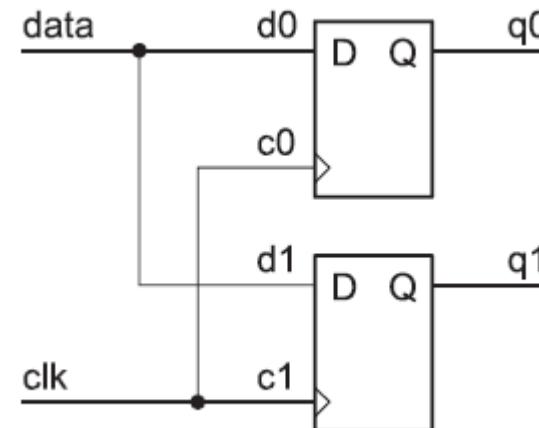




Nekoherence

Nekoherence vzniká pokud přivádíme asynchronní signál do vstupů několika synchronních prvků (klopných obvodů) současně. Zejména v FPGA obvodech může být zpoždění jednotlivých signálů (především datových) dosti rozdílné – to může způsobit rozdílné hodnoty na výstupech (nemusí teoreticky dojít k metastabilitě).

Řešením je opět synchronizace asynchronních signálů (přidání registrů na všechny signály).

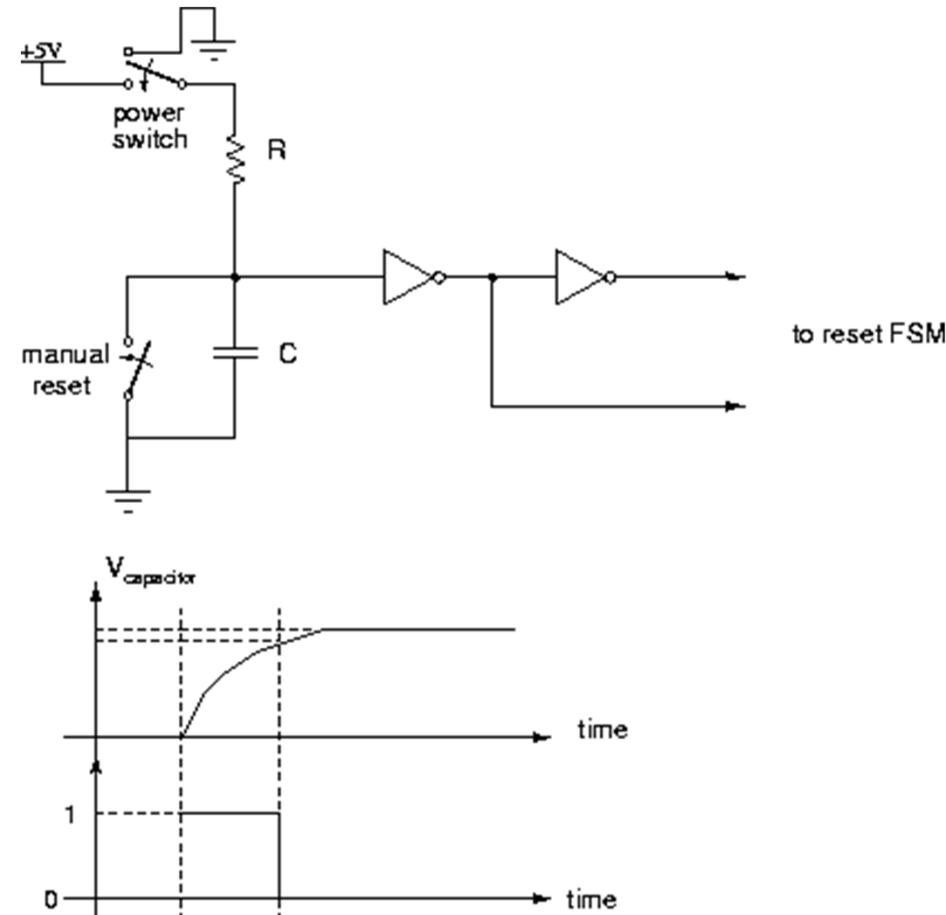




Inicializace obvodů (globální reset)

Důležité pro počáteční nastavení zejména stavových automatů.

Asynchronní nulování a přednastavení může být použito pouze k nastavení počátečního stavu klopných obvodů (ne v průběhu jejich běžné funkce).

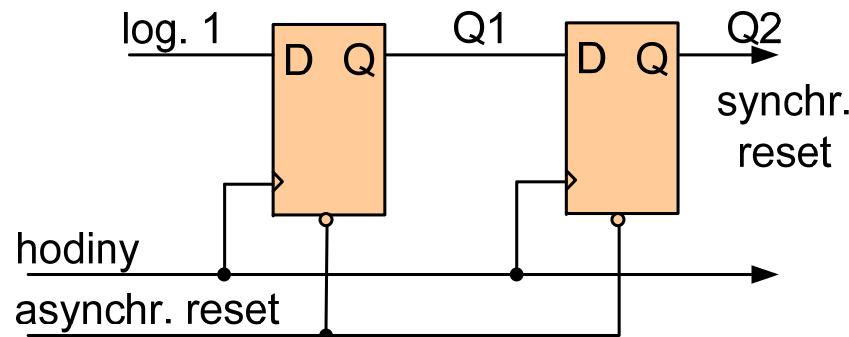




Synchronizace asynchronního resetu

Problém s uvolněním resetu s aktivní hranou hodin (nutno dodržet dobu zotavení po resetu – reset recovery time t_{rr}) – může vést na metastabilní chování;

⇒ reset synchronizer

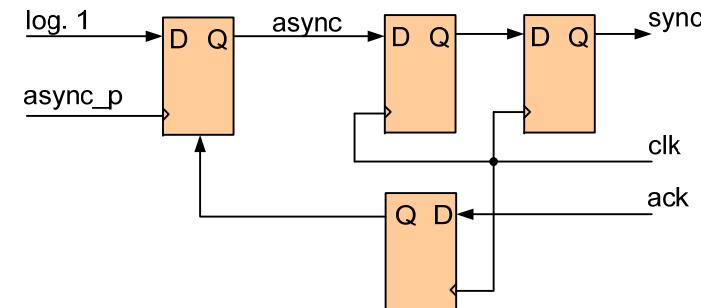
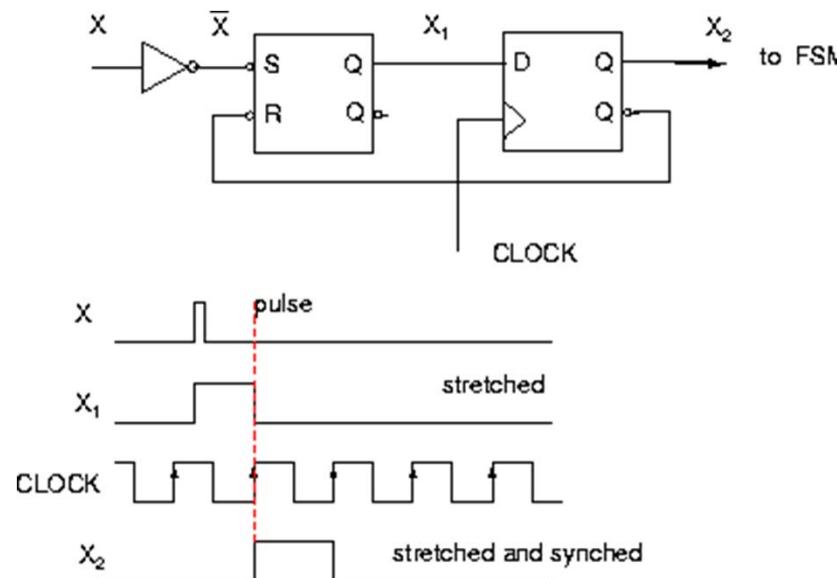


Zastavení hodinového signálu vkládáním logiky není vhodné, lze použít bloky DCM (Digital Clock Manager) či PLL.



Resynchronizace asynchr. vstupů

- synchronizace jednotným hodinovým signálem
- detekce úzkých pulsů



Asynchronní vstupní události nesmí být častější než $4 T_{\text{clk}}$

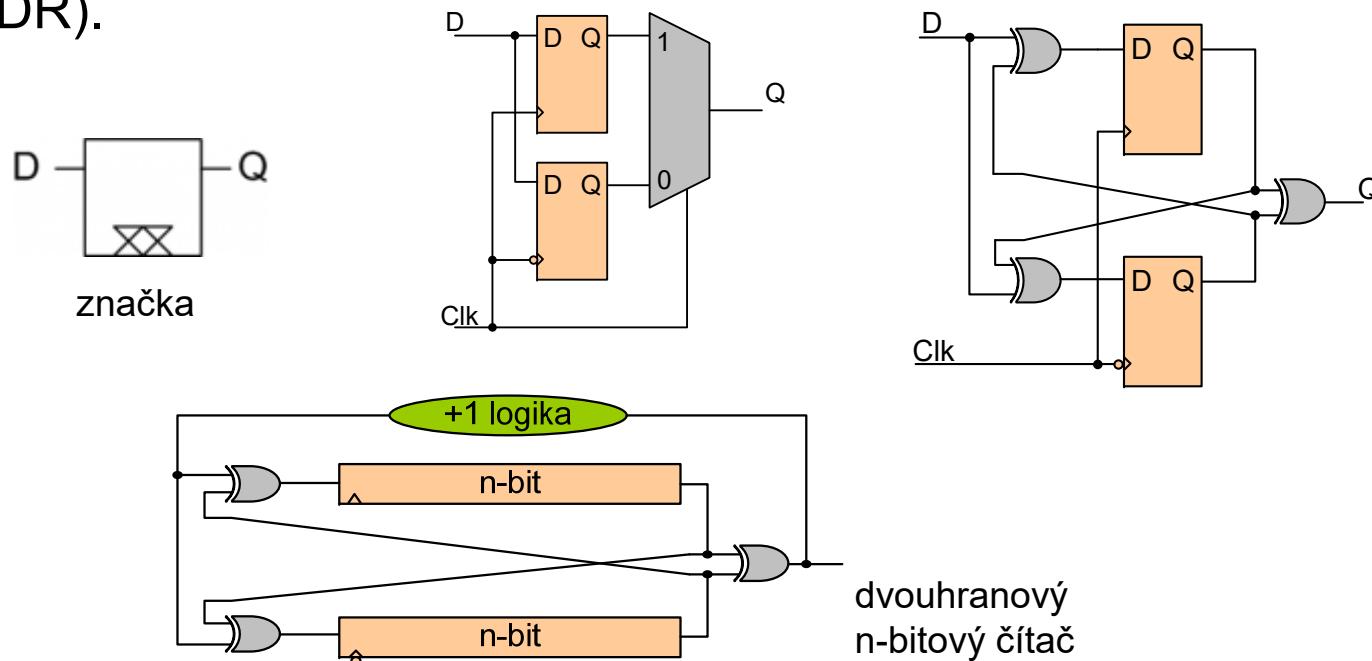


Dvouhranové klopné obvody

V architekturách FPGA jsou většinou jen KO reagující na náběžnou (po negaci clk na sestupnou) hranu.

Ani HDL jazyky většinou nepodporují dvouhranové KO.

U některých zařízení je třeba reagovat na obě hrany (např. paměti DDR).

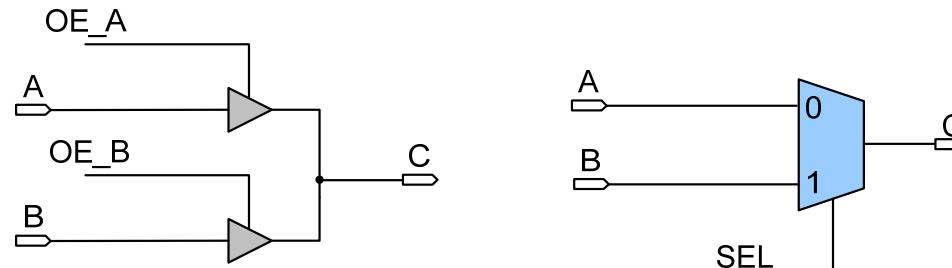




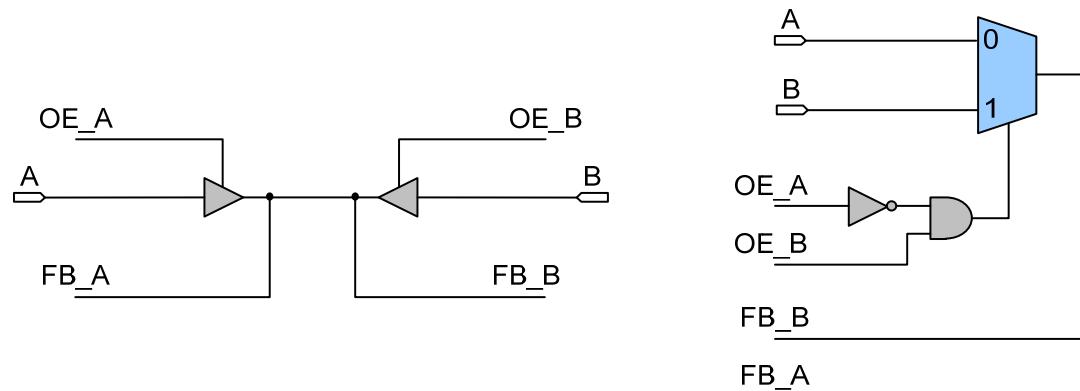
Náhrada sběrnic multiplexorem

Náhrada jednosměrné třístavové sběrnice multiplexorem

- třístavové výstupy jsou pomalejší, někdy uvnitř FPGA nedostupné
- vhodné zejména pro menší počet větví



Náhrada obousměrné třístavové sběrnice multiplexorem:





Makrobloky, jádra (cores)

Makro je definováno jako část navrhovaného systému, která je použitelná jako samostatný stavební blok.

Používání maker výrazně zkracuje a zlevňuje návrh.

Dva typy makrobloků:

- **soft core** – forma syntetizovatelného RTL kódu;
- **hard core** – výstupem je hotový layout (závislé na technologii),
 - úspora místa na čipu (ve srovnání s realizací v log. buňkách),
 - vyšší pracovní kmitočet,
 - snížení spotřeby bloku (malá plocha, optimalizace),
 - možnost implementovat i nestandardní (např. analogové) bloky,
 - funkce pevně dána, nelze upravovat.



Makrobloky (pokračování)

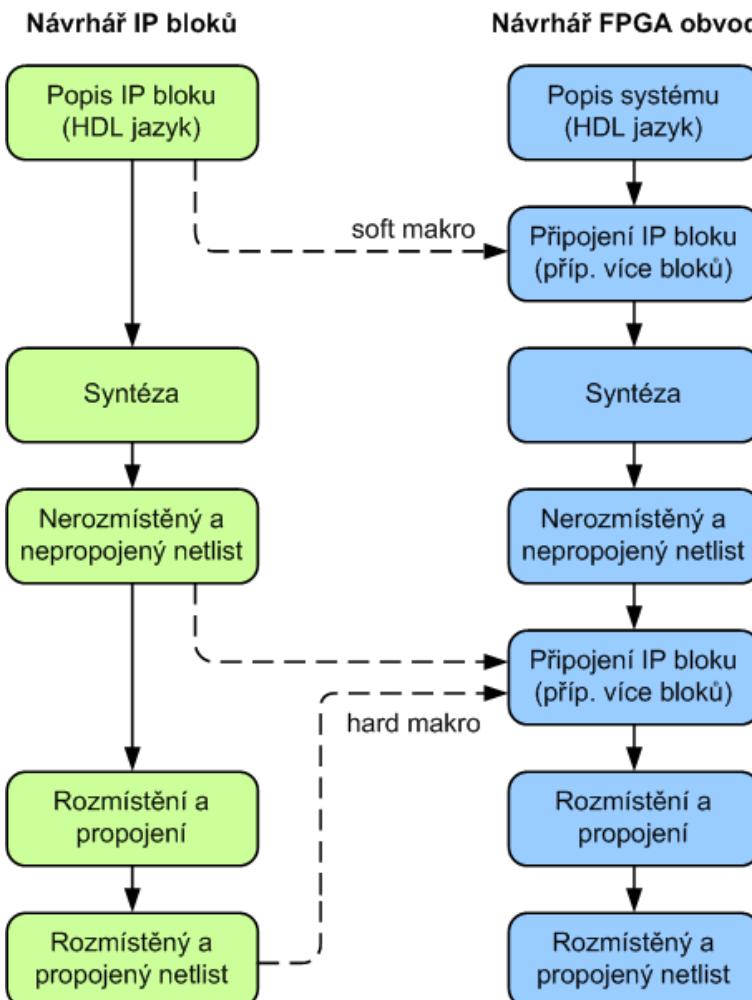
Základní vlastnosti makrobloků:

- souhlasí s příslušnými normami (standard compliant),
- vesměs verifikované pomocí FPGA obvodů, bloky pro testování,
- parametrické (např. v tabulkách uvedená různá velikost makrobloku pro proměnnou šířku sběrnice),
- integrované, spolehlivé a snadno použitelné,
- dobře zdokumentovatelné pomocí kompletní technické specifikace,
- podpora použití (od výrobce), aj.

IP bloky – navržené makrobloky, často nutné zakoupit licenci (Intellectual Property – intelektuální vlastnictví).



Metodika návrhu s IP bloky





TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií



Programovatelné zakázkové integrované obvody

Milan Kolář

Ústav mechatroniky a technické informatiky



evropský
sociální
fond v ČR



EVROPSKÁ UNIE



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY



Projekt ESF CZ.1.07/2.2.00/28.0050
**Modernizace didaktických metod
a inovace výuky technických předmětů.**

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ



Dělení FPLD

FPLD – Field Programmable Logic Device:

- **PLD (Programmable Logic Device)**

SPLD (Simple PLD)

- pevně daná struktura typu: vstup - pole **AND** - pole **OR** - výstup
 - PROM (Programmable Read Only Memory),
 - PAL (Programmable Array Logic),
 - GAL (Generic Array Logic),
 - PLA (Programmable Logic Array),

CPLD (Complex PLD)

- složitější architektury vycházející z SPLD (vrstevnaté, s centrální propojovací maticí).

- **FPGA (Field Programmable Gate Array)**

- pravidelná struktura programovatelných log. bloků s vodorovnými či svislými propojovacími linkami a propojovacími maticemi.

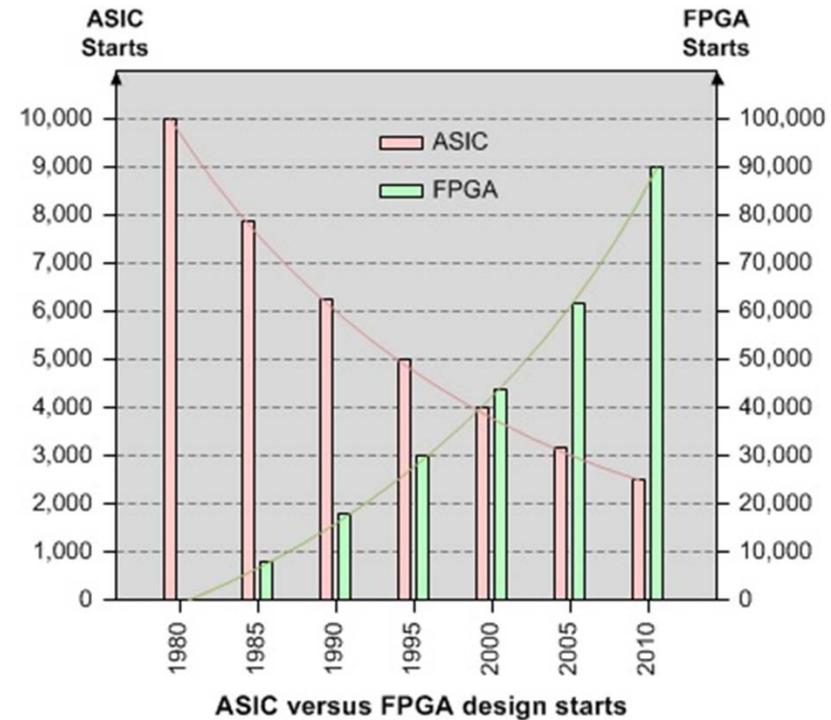


Nejvýznamnější výrobci FPLD

1. AMD, pův. Xilinx (53 %)
2. Intel, pův. Altera (36 %)
3. Microchip, pův. Microsemi (7 %)
4. Lattice Semiconductor (3 %)
5. Achronix Semiconductor
6. QuickLogic
7. Efinix
8. FlexLogix

Návrhové systémy:

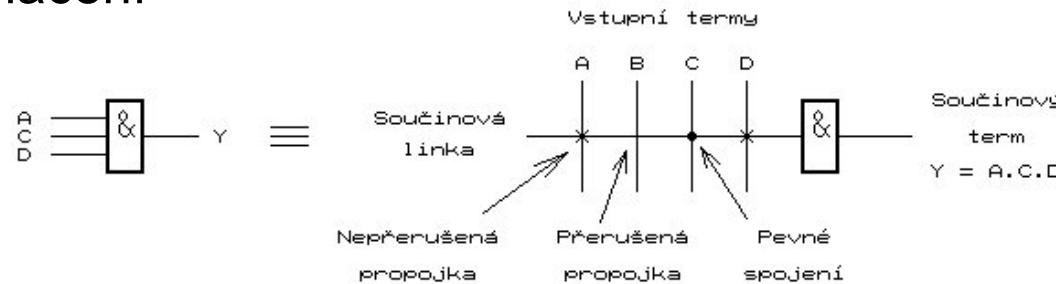
Vivado (AMD),
Quartus (Intel),
Diamond (Lattice),
Libero (Microchip),
HDL Designer (Mentor Graphics).



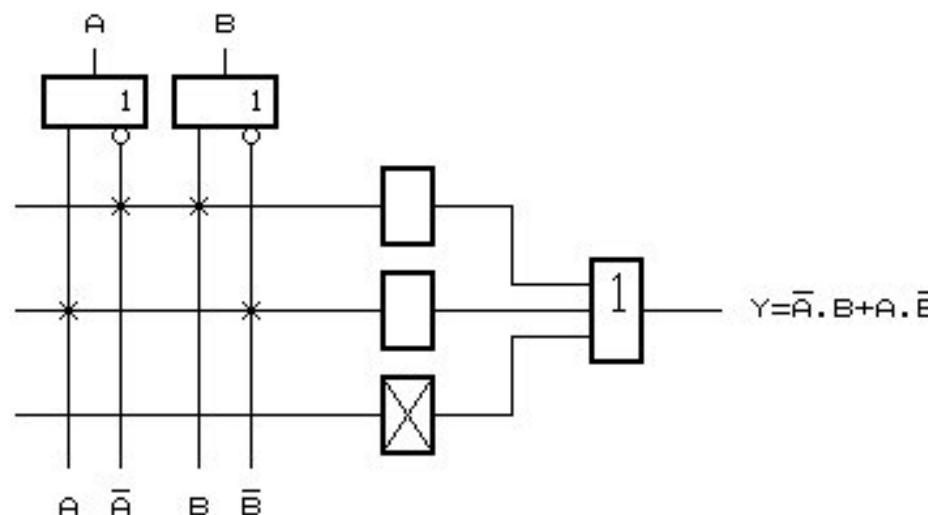


Principy obvodů PLD

Symbolika značení



Příklad realizace funkce XOR





Typy propojek

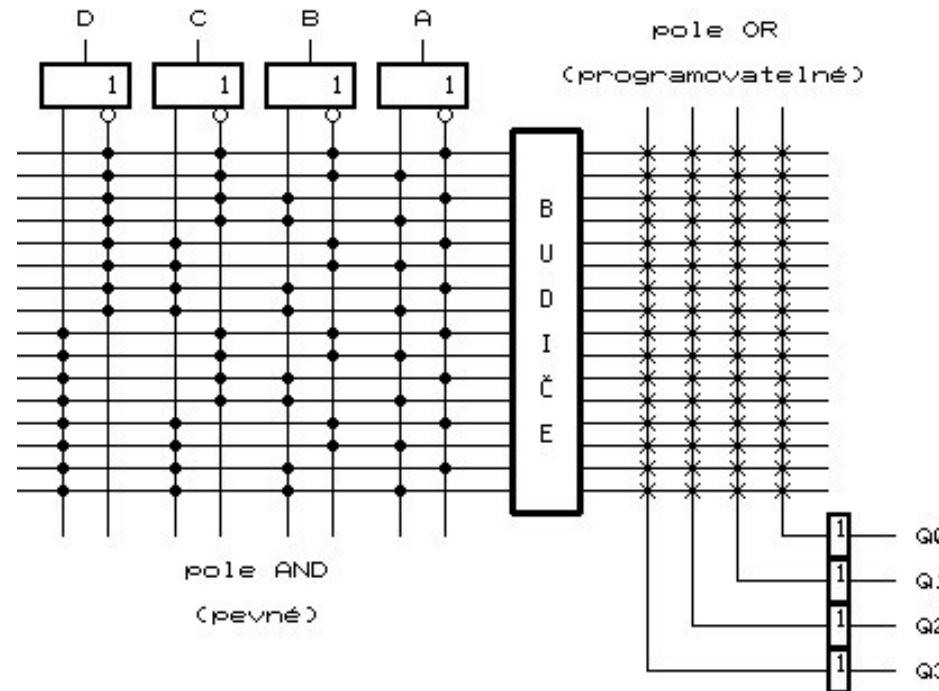
Technology	Symbol	Predominantly associated with ...
Fusible-link		SPLDs
Antifuse		FPGAs
EPROM		SPLDs and CPLDs
E ² PROM/ FLASH		SPLDs and CPLDs (some FPGAs)
SRAM		FPGAs (some CPLDs)



Obvody PROM

- programovatelné pole OR
- počet programovatelných bodů: $N = m \cdot 2^n$
- EEPROM (Electrically Erasable PROM)
- použití jako paměť konstant

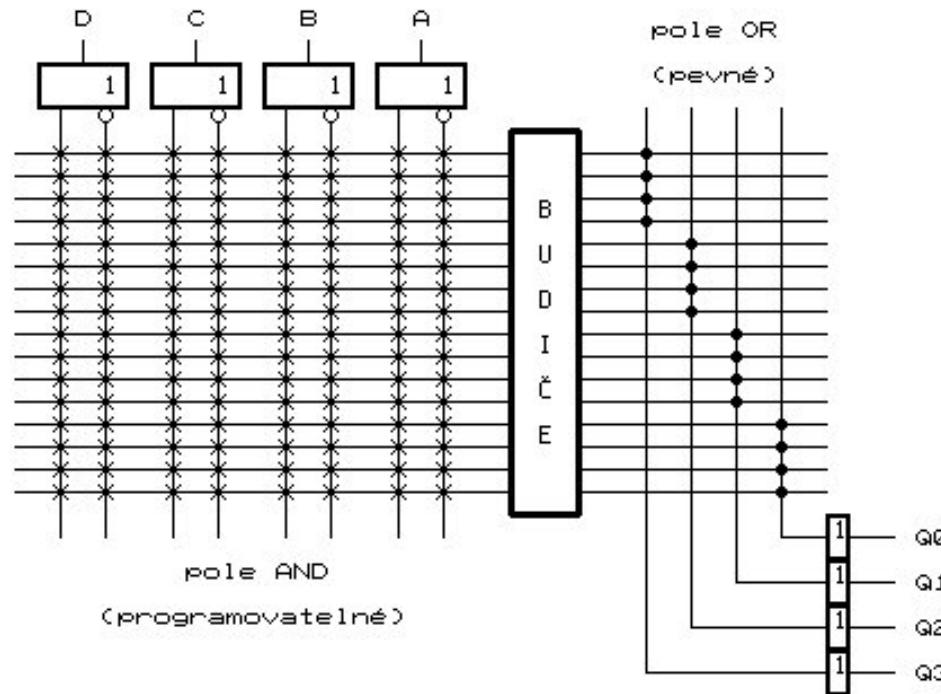
n ... počet vstupů
m ... počet výstupů





Obvody PAL

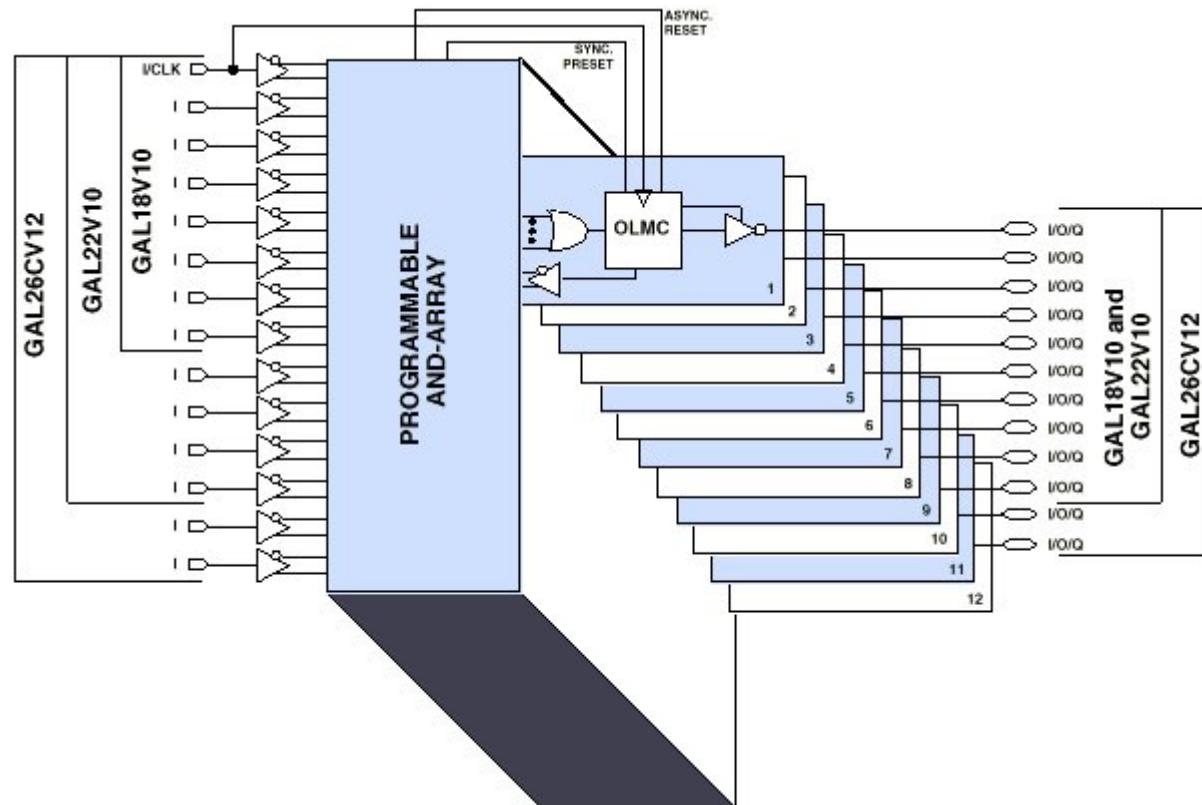
- programovatelné pole AND
- počet programovatelných bodů: $N = 2m.k.n$
- omezený počet součinových termů k
- na výstupu mohou obsahovat klopné obvody





Obvody GAL

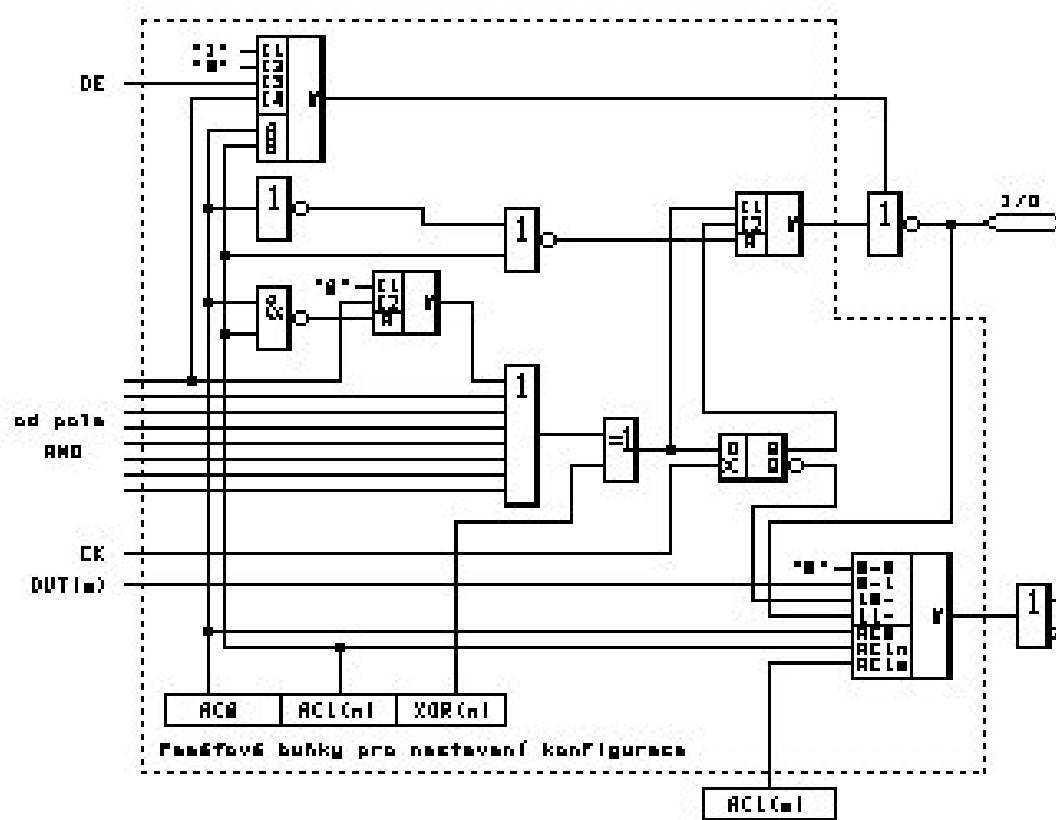
- vychází z obvodů PAL
- na výstupu makrobuňka OLMC (Output Logic Macro Cell)





Makrobuňka obvodů GAL

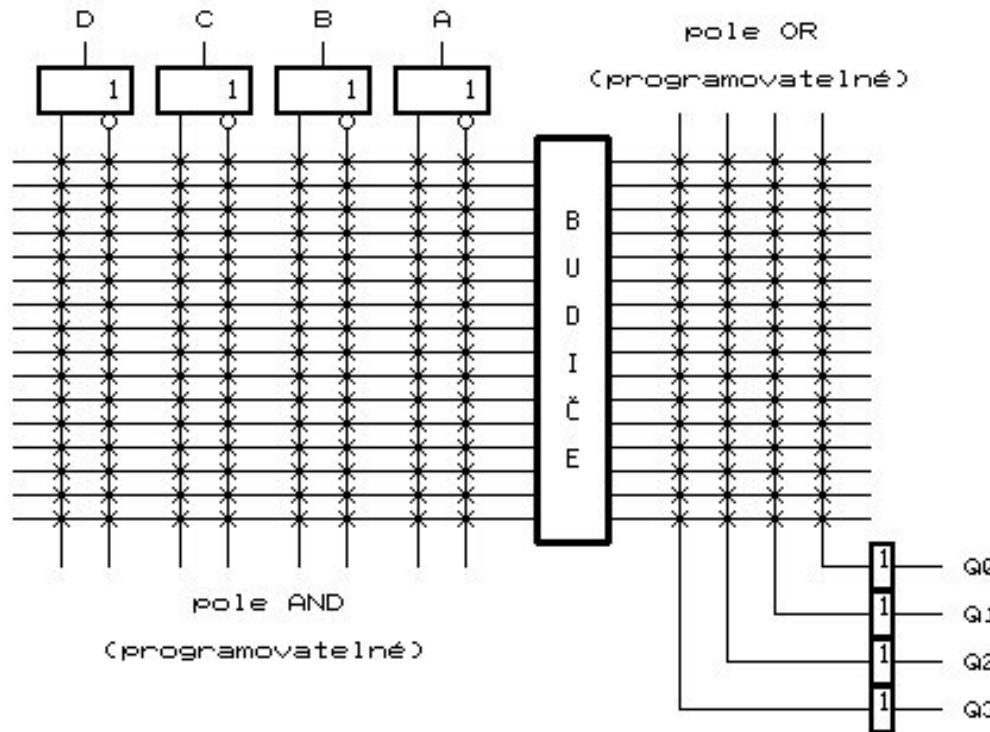
- každý I/O lze konfigurovat jako vstup, výstup nebo třístavový výstup
- některé z konfigurovatelných parametrů pouze globální





Obvody PLA

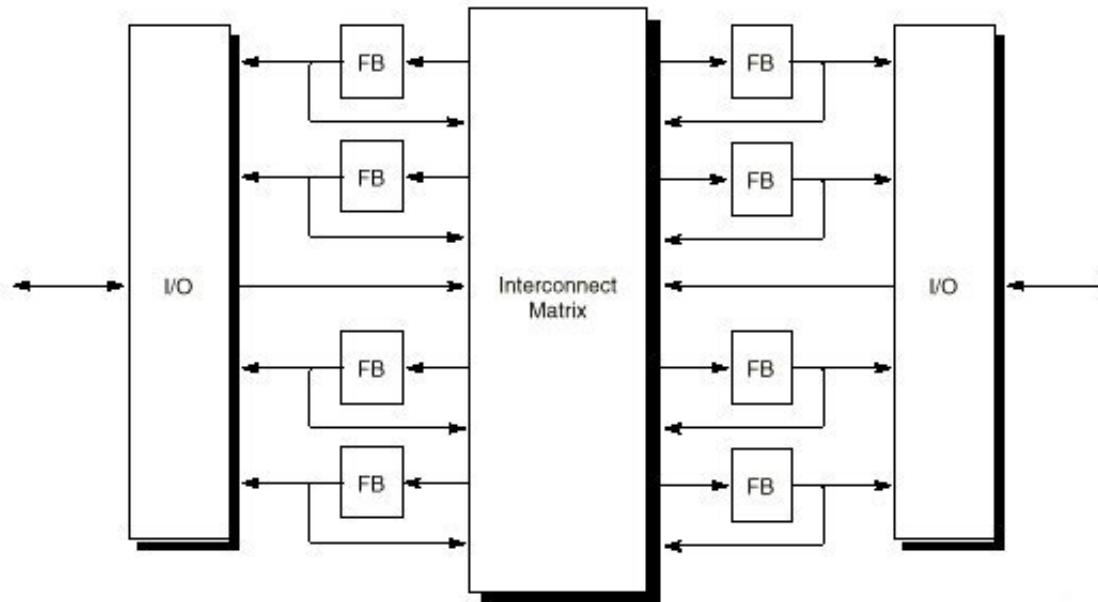
- programovatelné pole AND i OR
- počet programovatelných bodů: $N = m.k + 2k.n$
- odstraňuje omezení v počtu součinových termů





Obvody CPLD

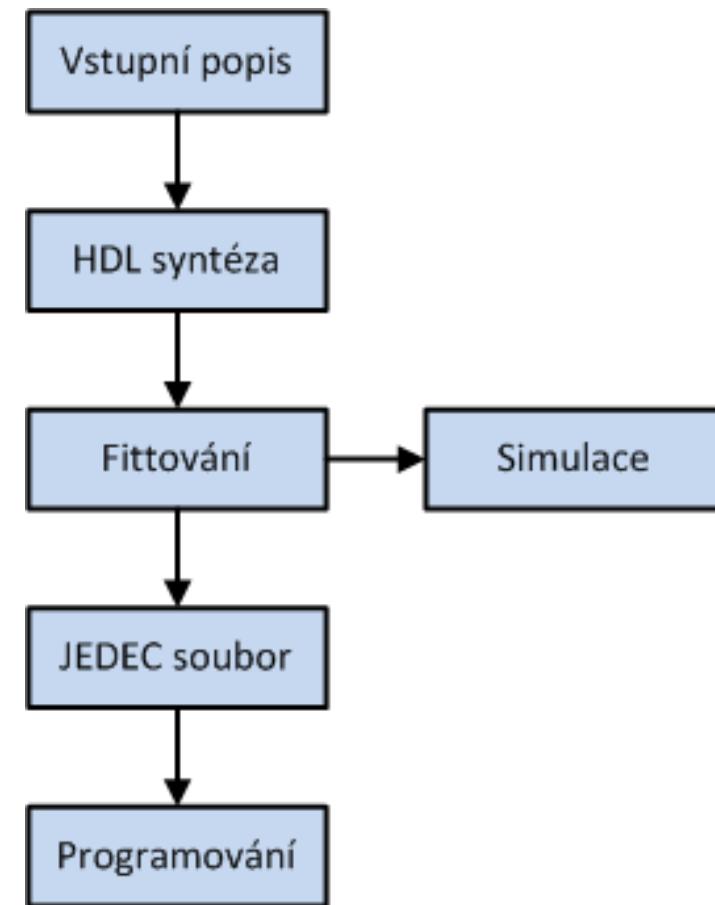
- složitostí mezi PLD a FPGA;
- obsahují centrální propojovací matici, která spojuje jednodušší funkční bloky (obdoba SPLD);
- elektricky reprogramovatelné (EECMOS);
- I/O buňky programovatelné, vesměs obousměrné.





Metodika návrhu obvodů PLD

- definice funkce
 - logické rovnice
 - stavový diagram
 - pravdivostní tabulka
 - indexové rovnice
- generace logických funkcí
- výběr obvodu PLD
- simulace funkce
- generace JEDEC souboru
- programování (v programátoru nebo přímo v systému)
- testování



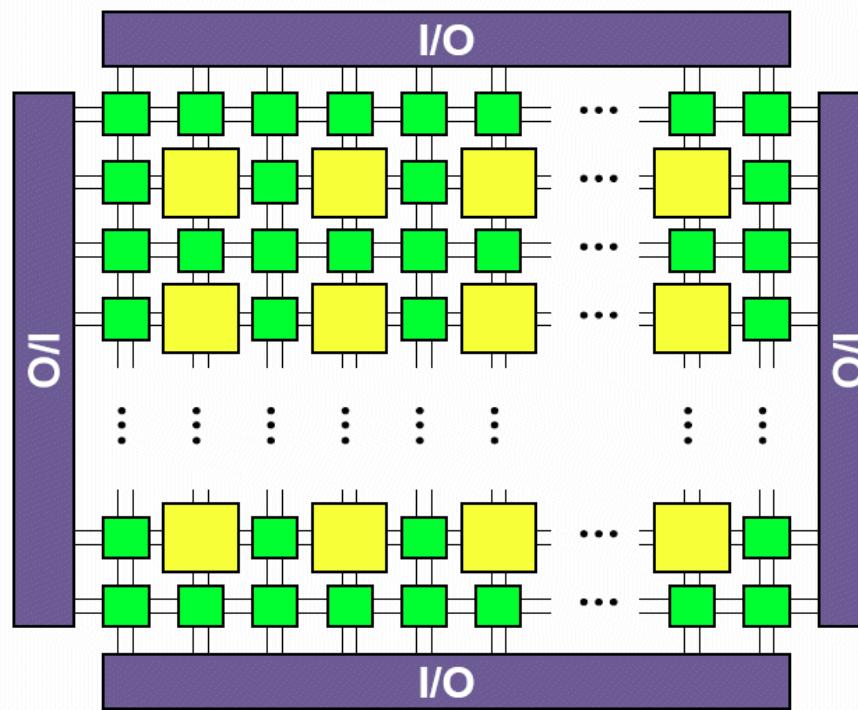


Obvody FPGA

Maticová struktura logických bloků a propojovacích vodičů

Logické bloky

Propojovací přepínače



Programovatelné I/O bloky



Části obvodů FPGA

Obvody FPGA obsahují:

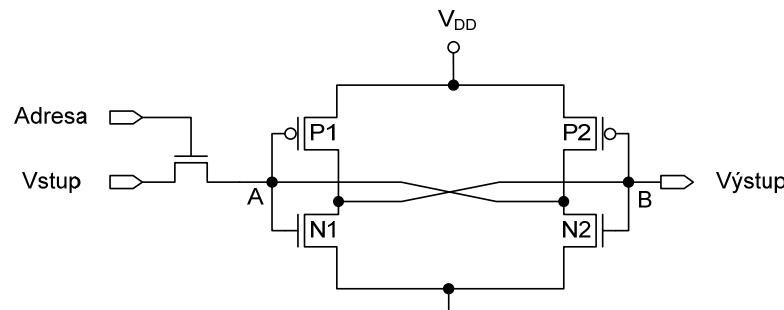
- logické prvky (LE – logic element, LC – logic cell),
- programovatelné propojky (matice),
- I/O prvky s registry,
- globální distribuce a řízení hodinových signálů,
- paměti SRAM (embedded RAM),
- DSP bloky,
- procesorová jádra,
- PLL (fázové závěsy),
- ochrana proti kopírování,
- firmware (výbava pro natažení konfigurace, monitorování stavu)



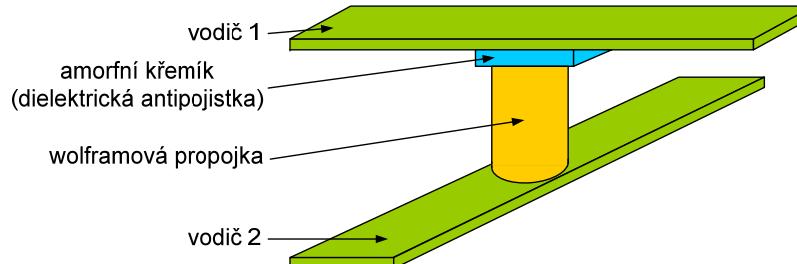
Programovatelné přepínače

V FPGA se nejčastěji používají programovatelné přepínače na principu:

- *SRAM* (nejčastěji z 5-6 tranzistorů);
- *antipojistky* (anti-fuses) - amorfni křemík v místě křížení dvou vodičů – nevodivá dielektrická vrstva se zvýšeným napětím prorazí (odpor $100\text{M}\Omega/50\Omega$);
- *EEPROM/flash* – u FPGA v omezené míře.



Buňka SRAM



Antipojistka



Vlastnosti propojek (technologie)

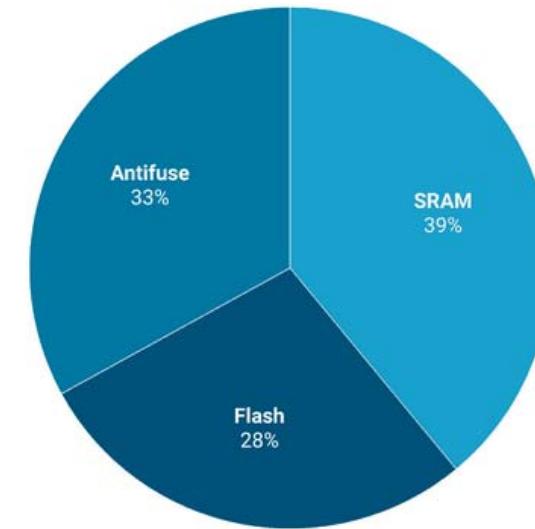
Vlastnosti	SRAM	Antipojistky	EEPROM/flash
Reprogramovatelnost	ano (v systému)	ne	ano (v systému nebo v programátoru)
Volatilní (nutné napájení)	ano	ne	ne
Vyžaduje externí konfigurační soubor	ano	ne	ne
Ochrana proti kopírování	vyhovující (lze zašifrovat)	dobrá	dobrá
Velikost konfig. buňky	velká (5 tranzistorů)	velmi malá	malá (2 tranzistory)
Spotřeba el. energie	vyšší	nízká	střední
Odolnost vůči záření	horší	výborná	střední
Okamžitě použitelné	ne	ano	ano
Programování v systému	ano	ne	ano



Trh FPGA – z hlediska technologií

Všechny tři technologie se v praxi zhruba rovnoměrně používají:

- SRAM - zejména u velkých FPGA, velká rychlosť a výborná rekonfigurovatelnost;
- Antifuse – bezpečné a spolehlivé, vhodné pro sériové použití, vyšší odpor propojek (menší rychlosť);
- Flash – nevolatilní, ale přesto rekonfigurovatelné, bezpečné vhodné pro menší a levnější FPGA.

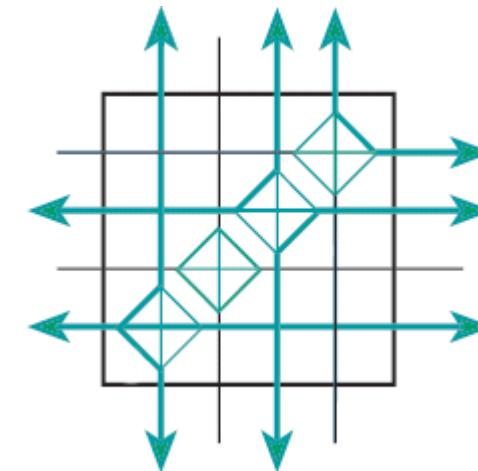
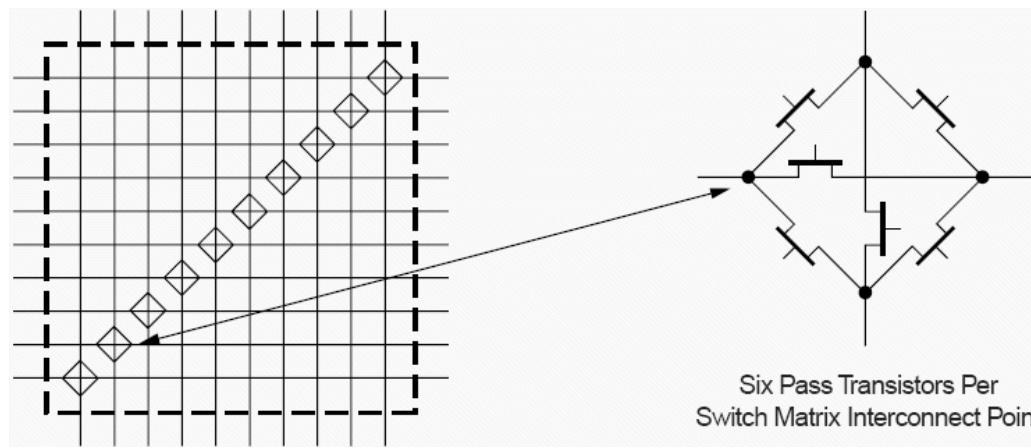




Programovatelné matice

PSM (Programmable Switch Matrix)

Propojuje vertikální a horizontální linie vodičů



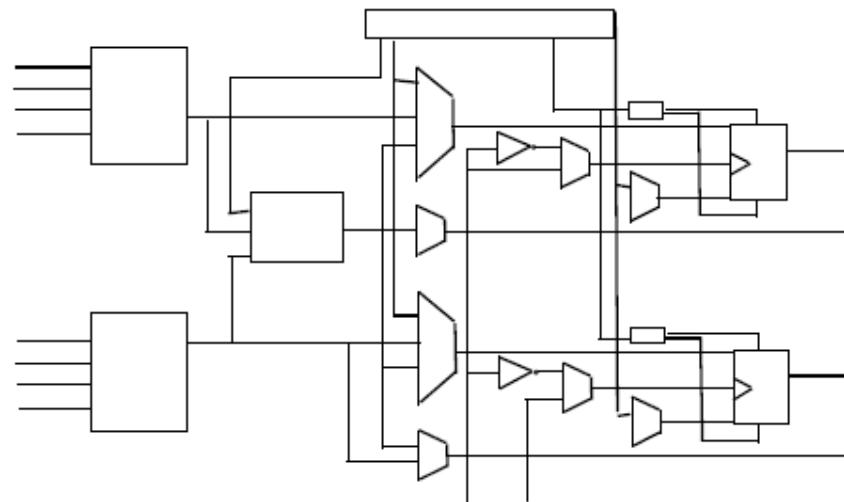


Hrubozrnné struktury

Obvody s "hrubozrnnou" strukturou (Course Grained)

Vlastnosti:

- velké množství logiky v každém modulu
- menší možná využitelnost logiky
- větší a těžko předvídatelné zpoždění
- větší spotřeba energie
- vhodné pro větší funkce
- typické pro PLD, CPLD



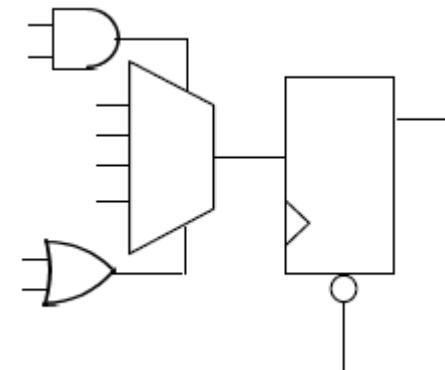


Jemnozrnné struktury

Obvody s "jemnozrnou" strukturou (Fine Grained)

Vlastnosti:

- jednodušší logické moduly
- lepší využitelnost logiky
- snadnější syntéza
- lépe odhadnutelné zpoždění
- obtížnější realizace větších funkcí
- typické pro FPGA

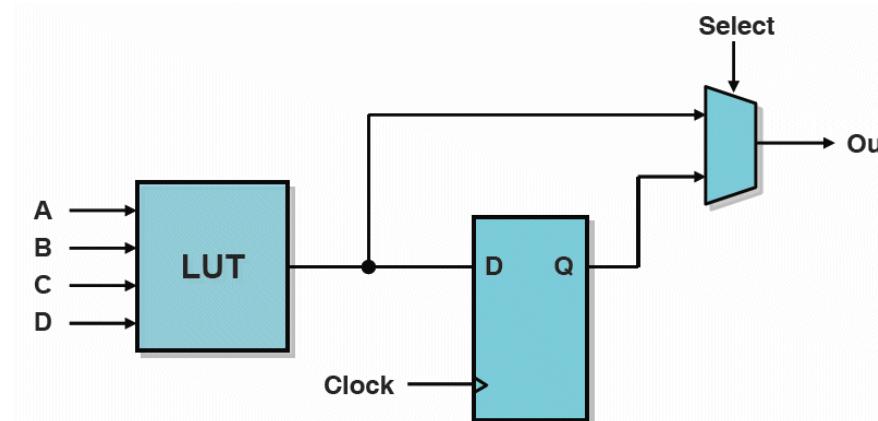




Velikost FPGA

Velikost se uvádí v počtu *logických elementů* (LE)

- LE jsou různě složité,
- nejčastěji LE obsahuje 4vstupový LUT (Look Up Table) a KO (klopný obvod). Rozšiřují se 6vstupové LUTy se dvěma výstupy.



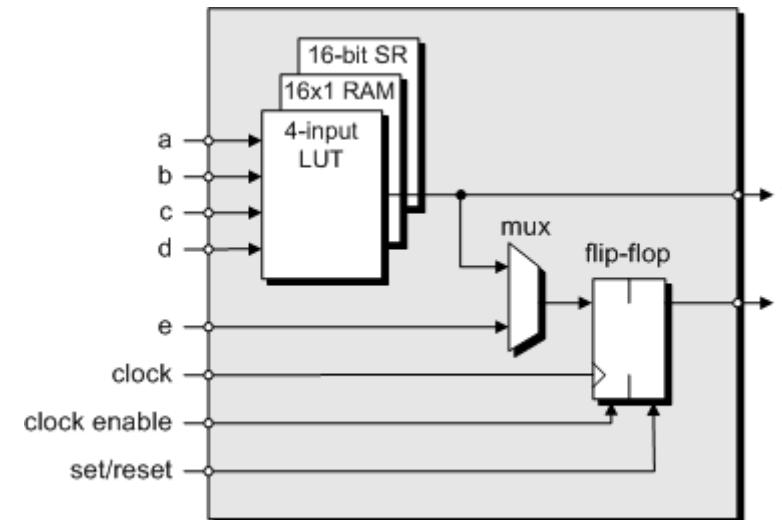


Logická buňka (AMD-Xilinx)

Příklad logické buňky – skládá se z konfigurovatelného LUTu a klopného obvodu.

Mnohofunkční LUT může pracovat jako:

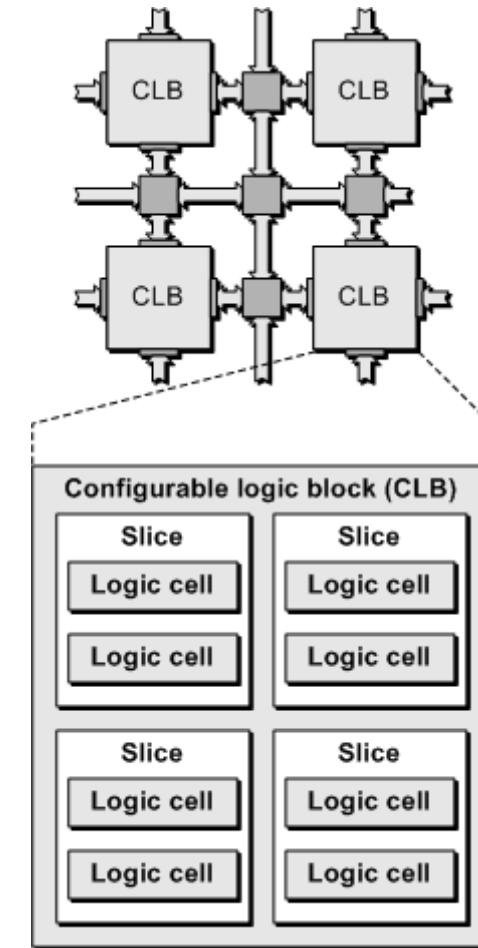
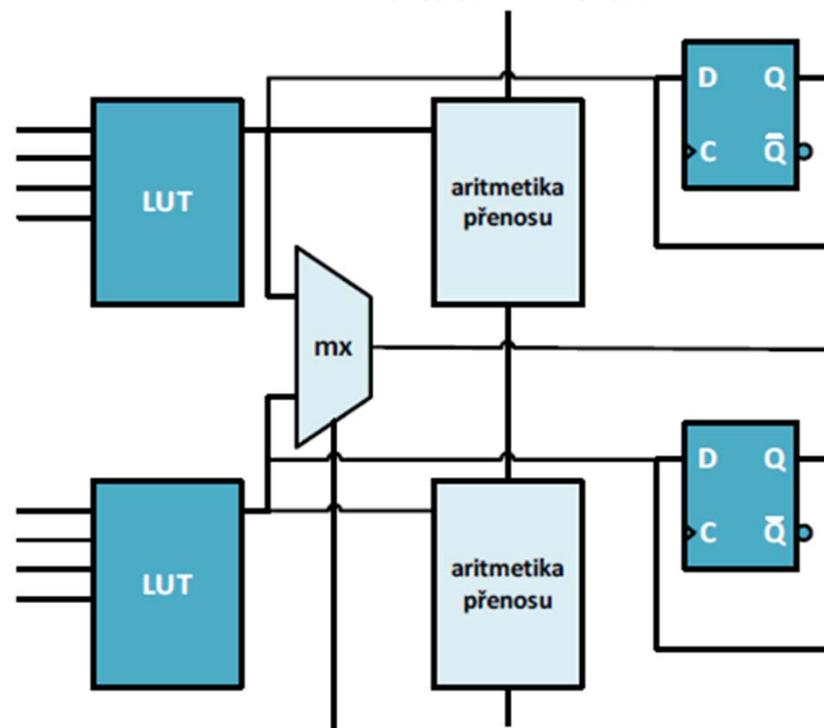
- 16bitový posuvný registr
- 16bitová paměť RAM
- 4vstupový LUT





CLB vs. Slice vs. LC (LE)

CLB – Configurable Logic Block

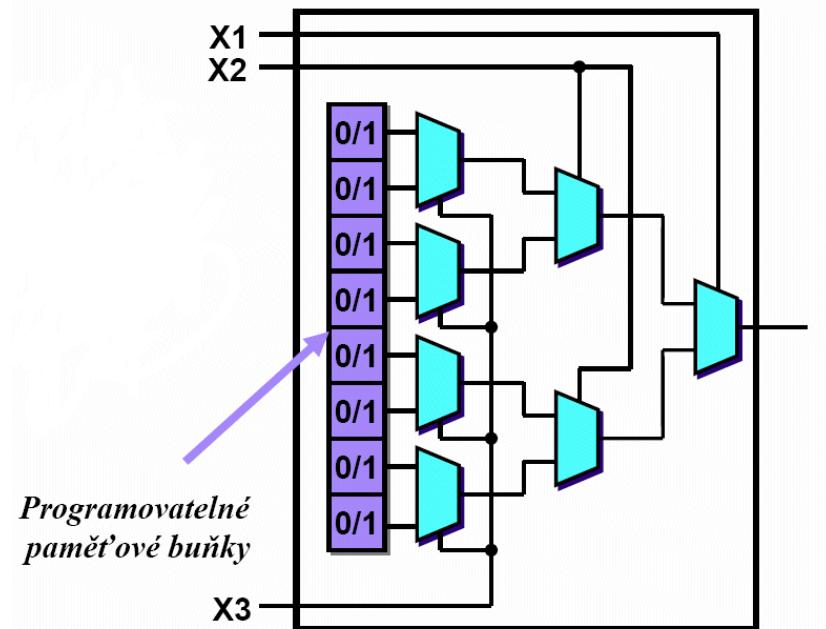




LUT

Look Up Table (LUT) - blok pro implementaci kombinační logiky

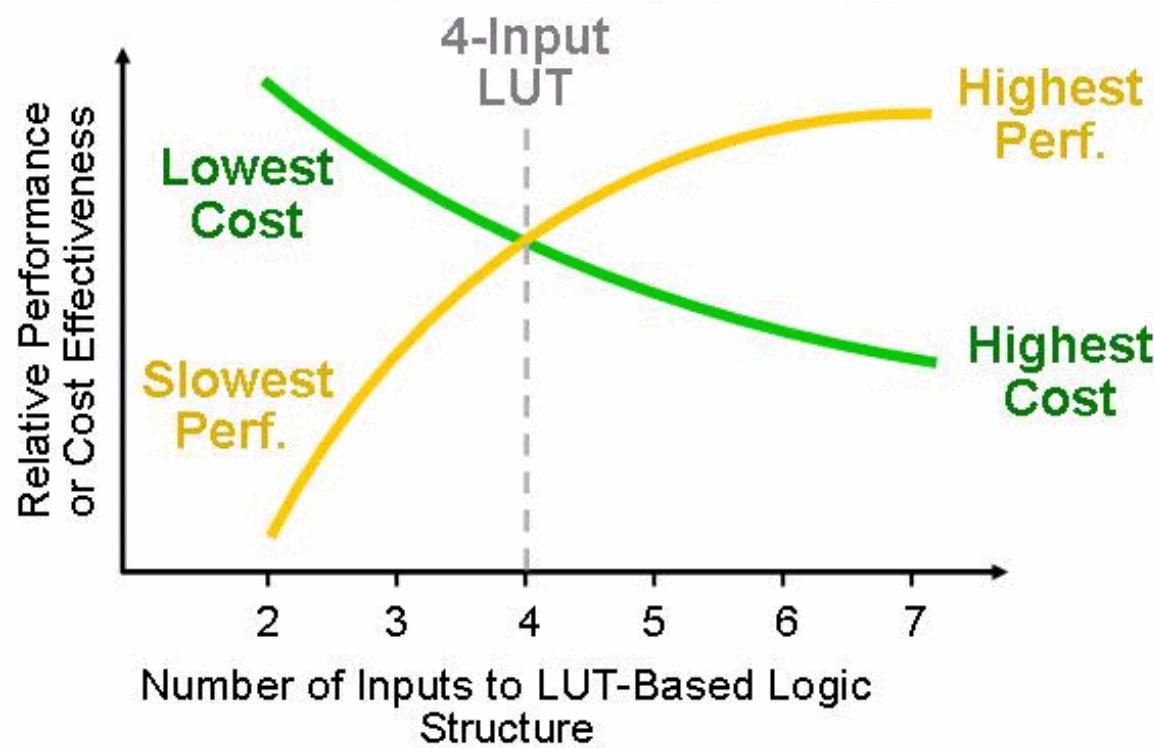
- na principu SRAM
- na principu multiplexorů





Počet vstupů LUT ?

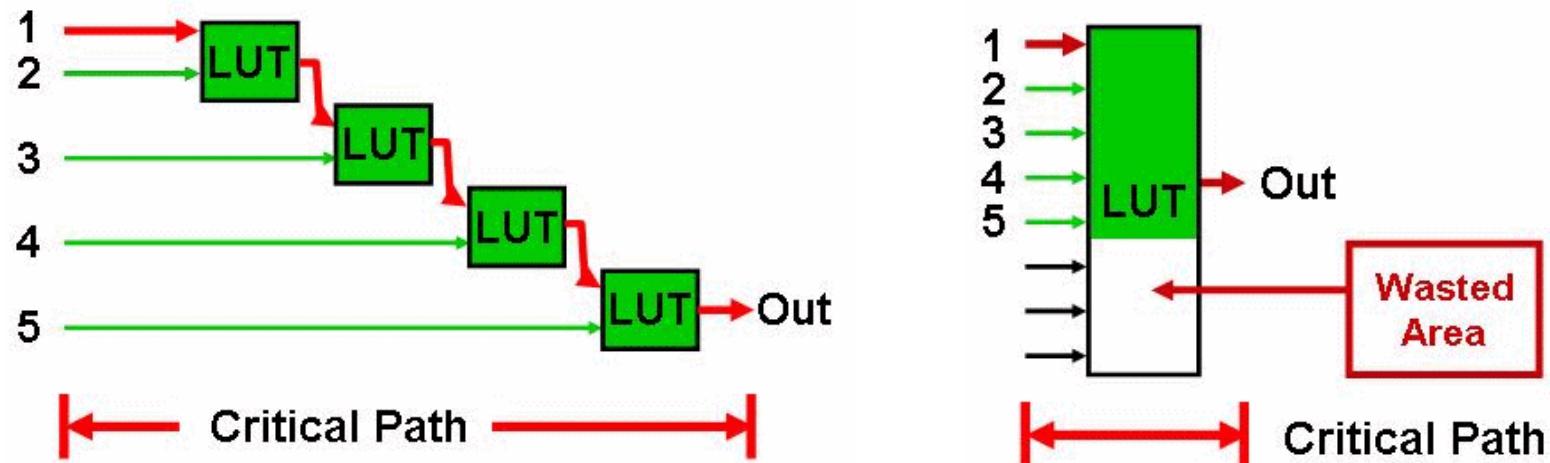
Většina FPGA obvodů má 4-vstupové LUT





Jaký počet je nejvhodnější ?

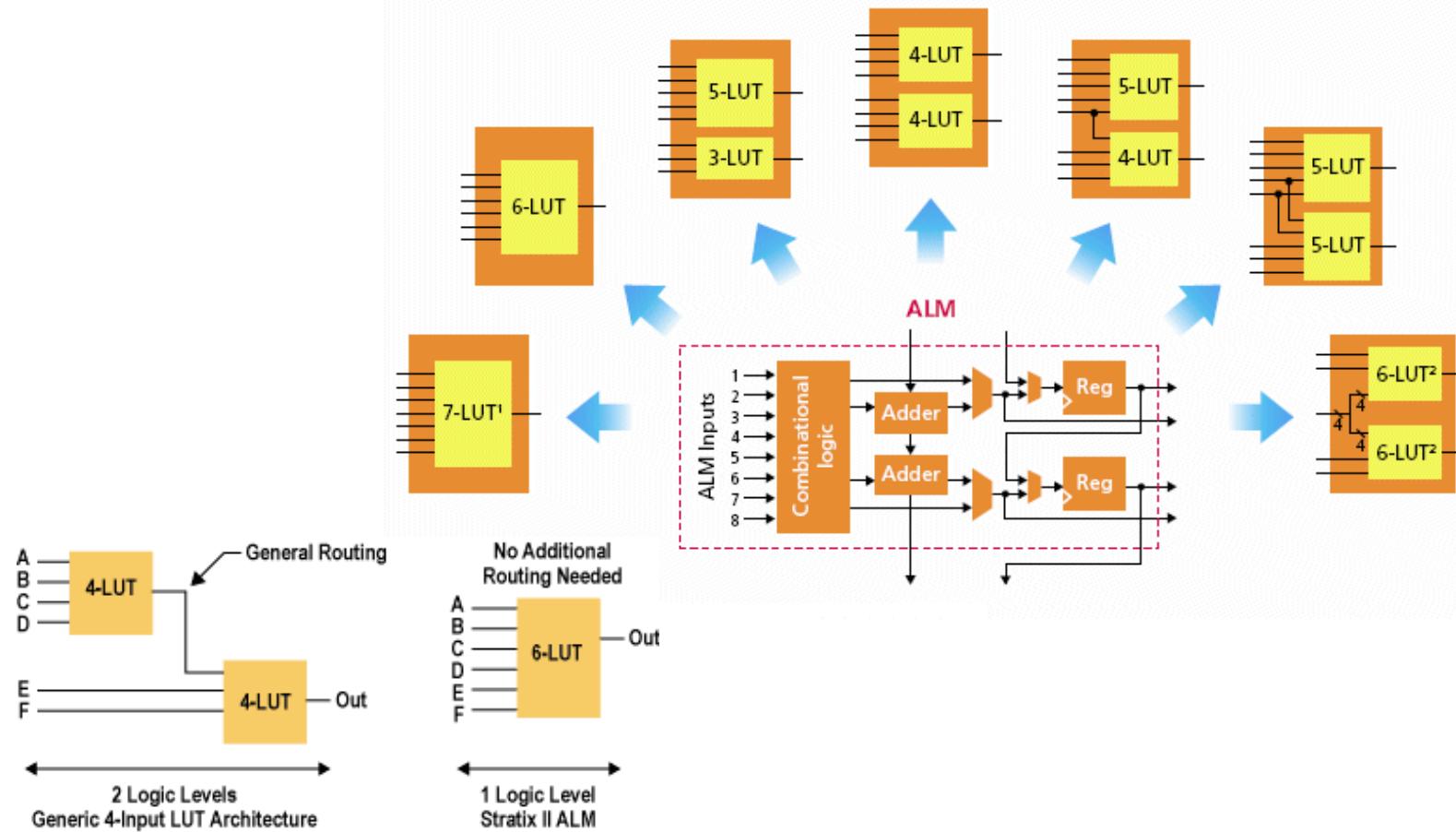
Kompromis mezi délkou kritické cesty a využitím logiky LUTů





Používání adaptivní logiky

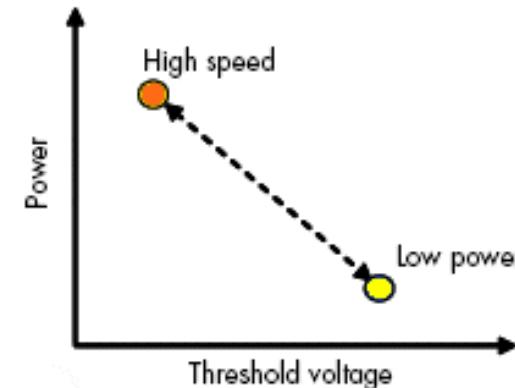
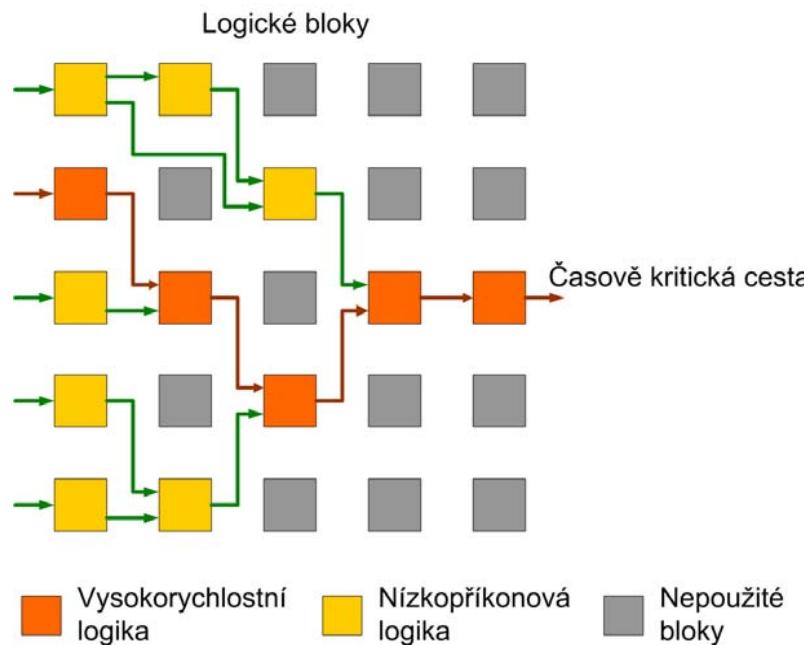
ALM (Adaptive Logic Modules) – v FPGA je pouze část těchto buněk





Programovatelné napájení logiky

Programovatelné řízení napájení s několika úspornými módy (buňky v kritické cestě jsou nejrychlejší, ale mají největší spotřebu; ostatní buňky mohou být v úsporných módech).





Vstupně výstupní buňky

Zajišťují tok dat mezi vnitřní logikou a I/O piny;

- přizpůsobují logické úrovně vně a uvnitř čipu,
- zesilují výstupní signály,
- podporují řadu vstupně-výstupních napěťových standardů.

Buňky jsou rozděleny do *bank* – každou banku lze připojit na jiný napájecí zdroj.

Většina buněk může být konfigurována jako vstupní, výstupní nebo obousměrné.

Struktura buňky obsahuje většinou 3 základní signálové cesty:

- *vstupní cesta* (data z pinu do vnitřní logiky),
- *výstupní cesta* (přenos dat z vnitřní logiky na výstupní pin),
- *cesta ovládající třístavový výstup*.



Vstupně výstupní buňky (pokrač.)

Ve výstupní cestě je zařazen *programovatelný výstupní driver*:

- umožňuje měnit rychlosť přeběhu (2-3 stupně),
- určuje výstupní proudové zatížení (2-25 mA),
- umožňuje nastavení do stavu vysoké impedance.

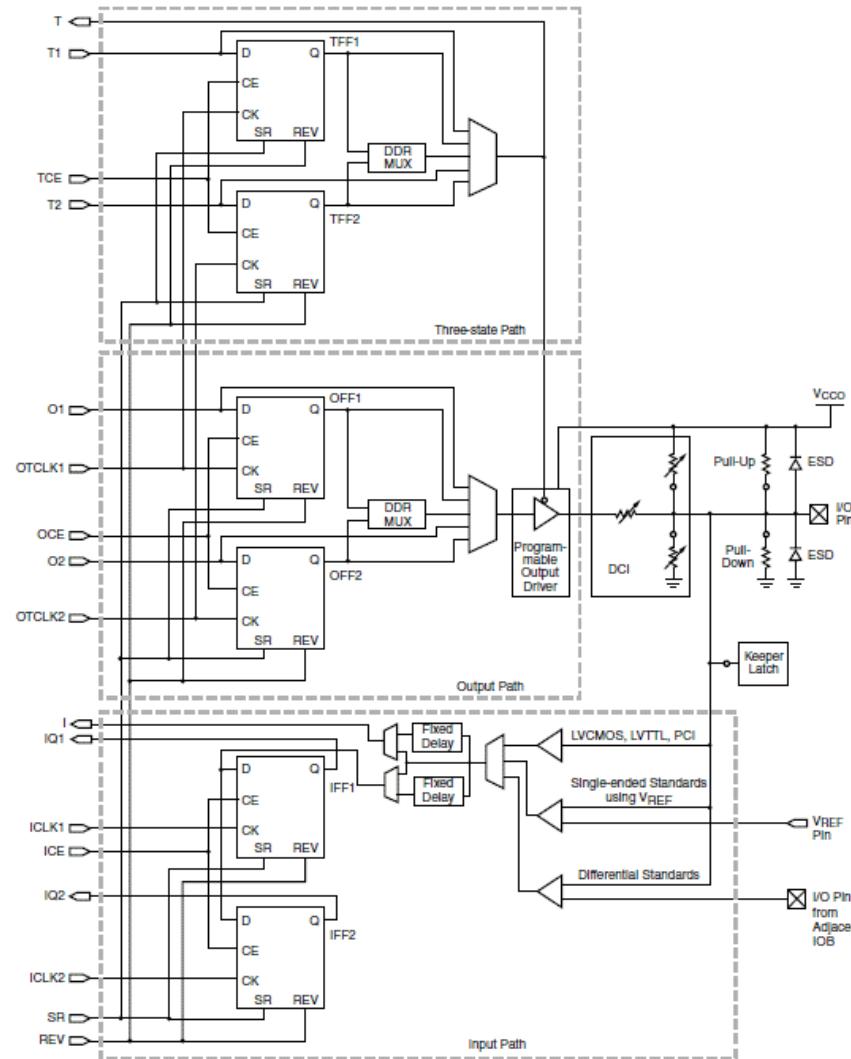
Součástí I/O buňky jsou:

- pull-up a pull-down rezistory,
- ochrany proti kladnému i zápornému přepětí,
- obvody zajišťující impedanční přizpůsobení (OCT – On-Chip Termination, DCI – Digitally Controlled Impedance),
- obvod přidržení úrovně (bus hold, keeper, aktivní terminátor).



Vstupně výstupní buňka

Xilinx Spartan 3





Použití RAM v FPGA

- 1) **Klopné obvody** v logických buňkách (pro menší množství dat, rychlý přístup);
- 2) **Distribuovaná paměť** – využití paměťových buněk v LUTech (omezená velikost, náročné na propojovací síť);
- 3) **Bloková (embedded) paměť** – speciální paměťové bloky vložené do struktury FPGA (snadno konfigurovatelné, plně dvouportové, rychlé, efektivní z hlediska nároků na plochu);
- 4) **Externí paměti** – pro uložení velkých objemů dat (nejčastěji dynamické paměti, podpora DDR řadičů v FPGA).

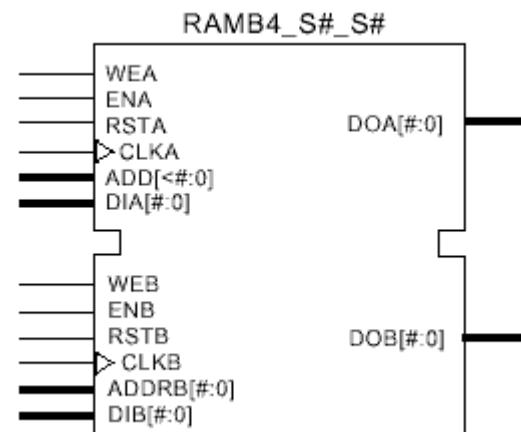


Embedded RAM

Speciální vložené bloky konfigurovatelné RAM v FPGA;

Možnosti:

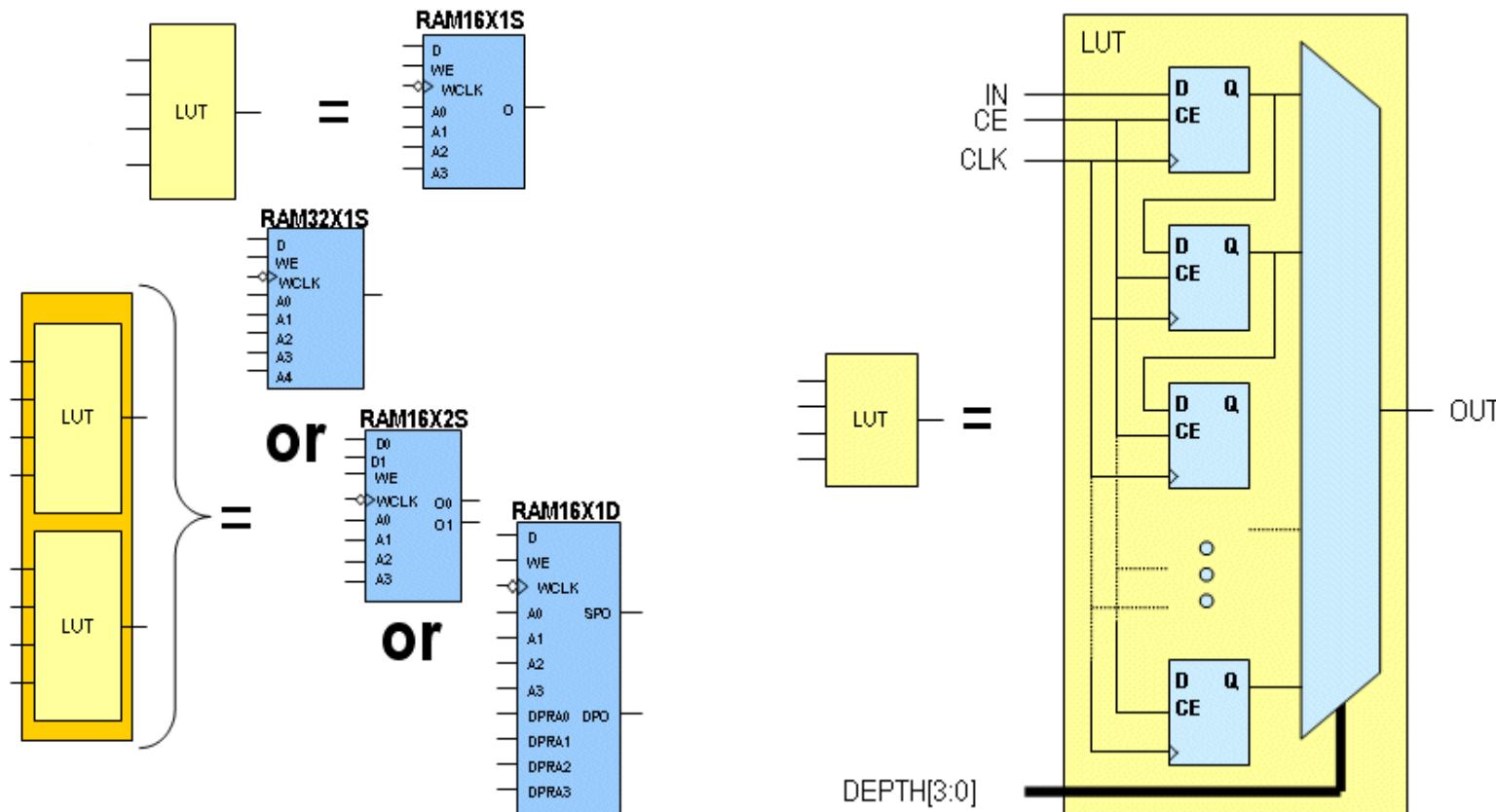
- změny organizace (např. 4Kx1 nebo 256x16),
- krátká vybavovací doba (rychlé dekodéry adres),
- obsah je možno definovat při konfiguraci (ROM),
- možno použít jako FIFO (dual clock),
- jednoportové / dvouportové,
- asynchronní / synchronní.





Distribuovaná RAM

Možno často konfigurovat jako jednoportová nebo dvouportová paměť nebo posuvný registr

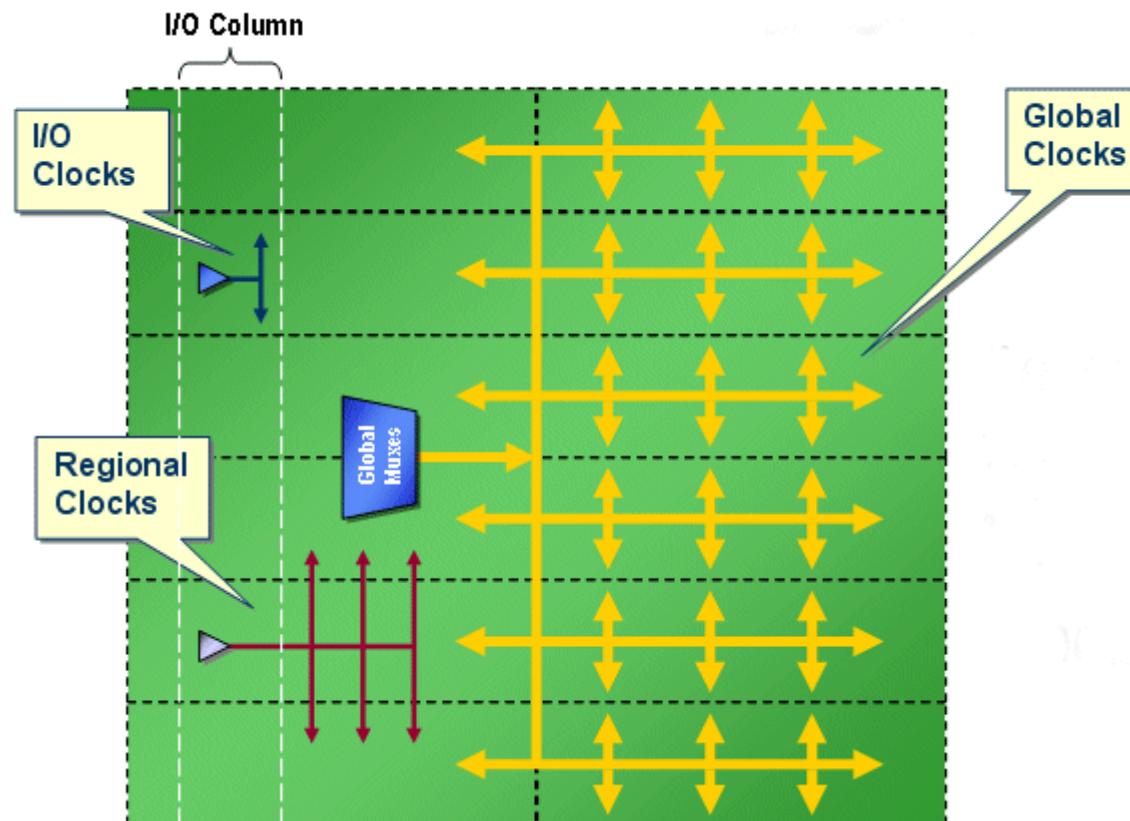




Typy rozvodů hodinového signálu

Snaha o minimální skluz (preferovat speciální vstupní pin pro clock).

Nejčastěji 3 typy rozvodů (I/O rozvody jsou nejrychlejší):





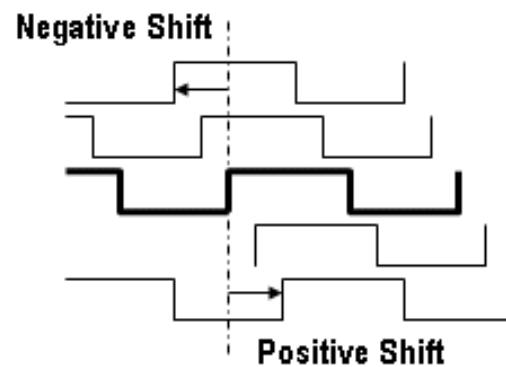
Programovatelné fázové závěsy

dorovnávají zpoždění, dělí nebo násobí frekvenci, mění střídu, zajišťují fázové posuny (rozlišení řádově pod 100 ps);

jitter – kolísání (nestabilita) zdroje hodinového signálu (odchylka od ideální náběžné hrany).

Snaha o nulový skluz hran hodinového signálu v celém FPGA.

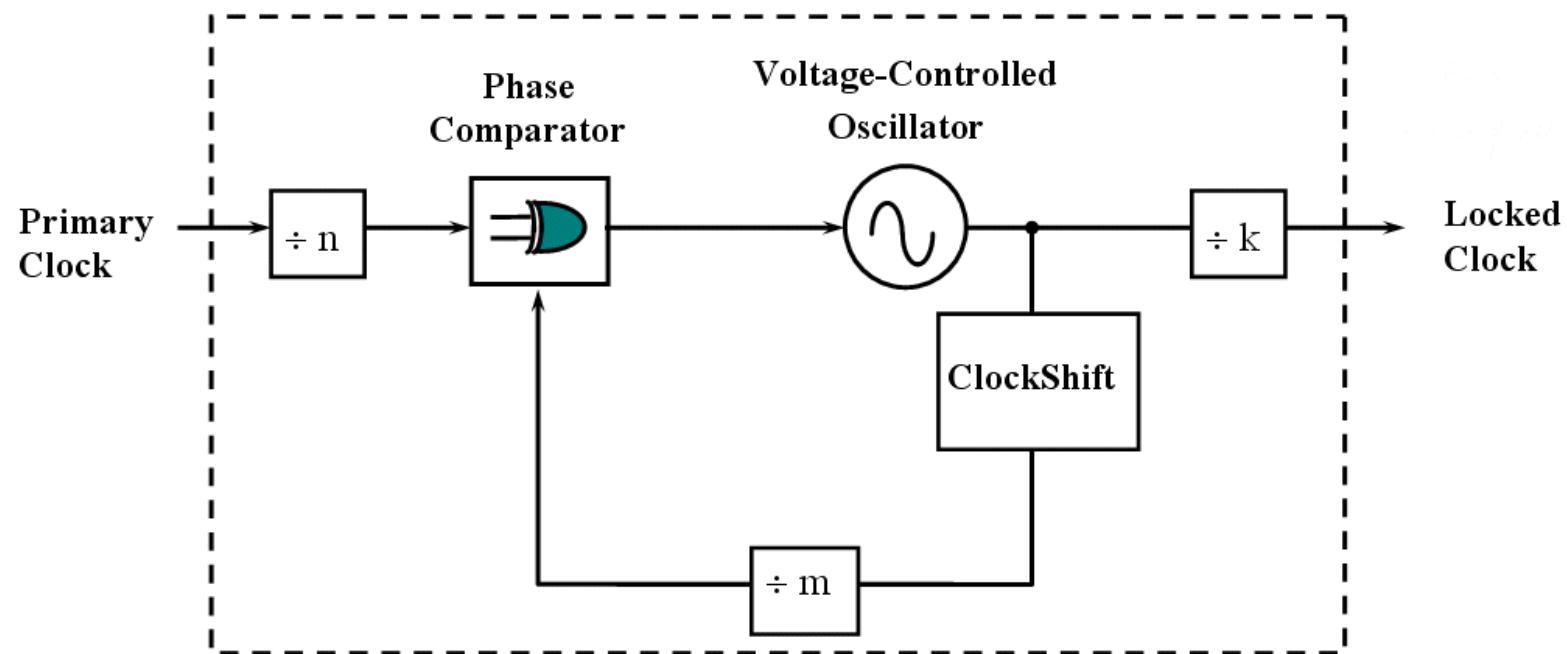
Digital Clock Management (Xilinx), sysCLOCK (Lattice),
PLL (Phase Locked Loop), DLL (Delay Locked Loop).





Princip PLL

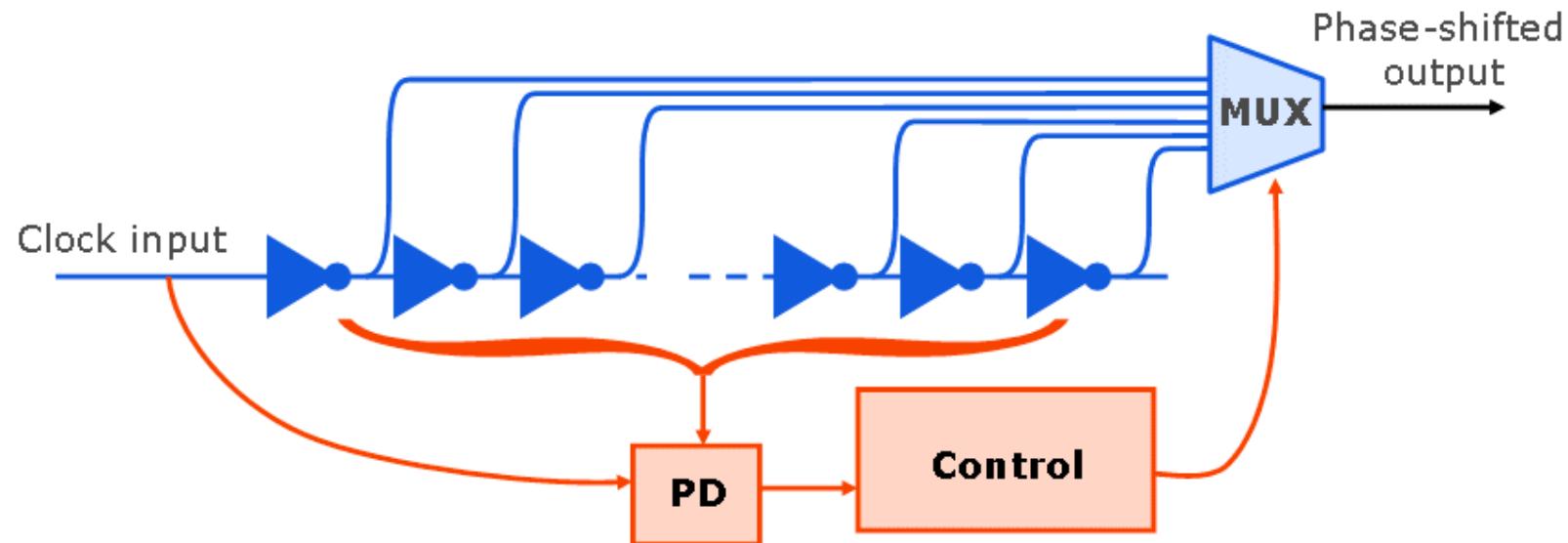
Fázový závěs s fázovým komparátorem a napětím řízeným oscilátorem





Vyrovnávání fázových posunů

PD (Phase Detector) – pozoruje všechna zpoždění a určuje, který výstup nejlépe porovnává zpoždění na 360°

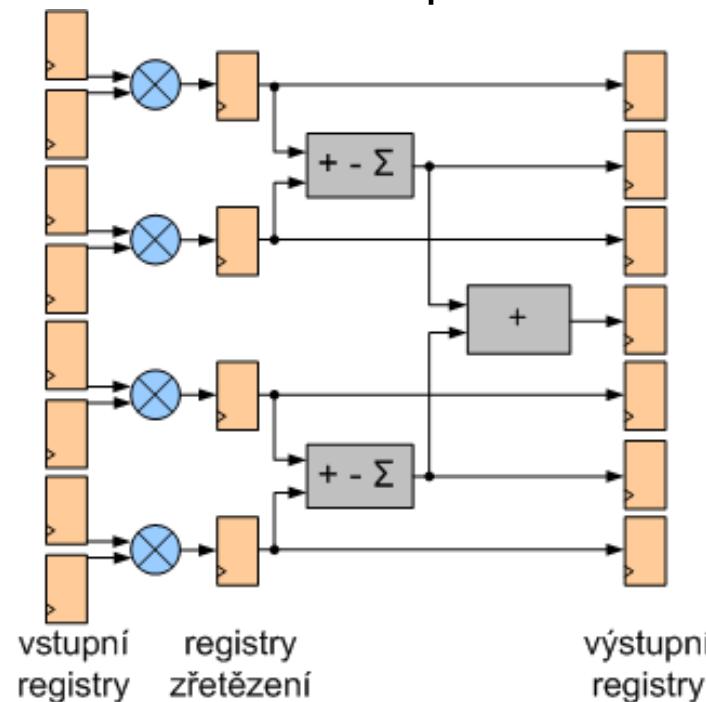




DSP bloky - princip

Speciální bloky optimalizované na DPS aplikace (FIR, FFT,...),

- složeny z násobiček, sčítáček (příp. odečítáček) a registrů;
- jednotlivé bloky jsou mezi sebou vzájemně provázané;
- operace převážně ve formátu pevné řádové čárky.

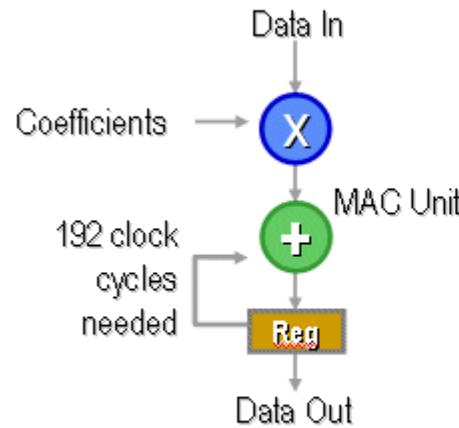




DSP bloky - srovnání

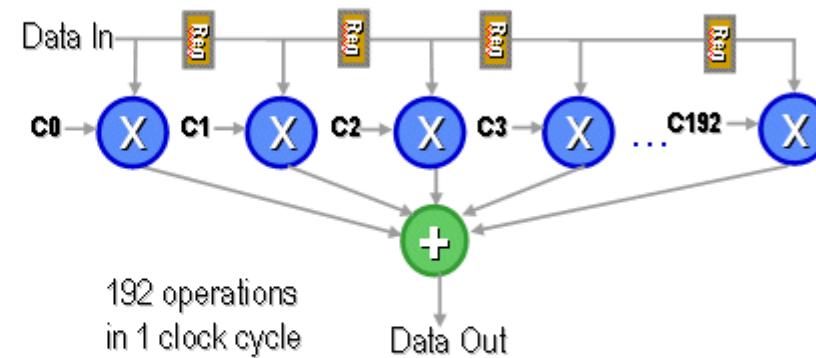
V FPGA lze realizovat např. filtry mnohonásobně rychleji ve srovnání se signálovým procesorem

Programmable DSP - Sequential



$$\frac{1 \text{ GHz}}{192 \text{ clock cycles}} = 5.2 \text{ MSPS}$$

FPGA - Fully Parallel Implementation



$$\frac{550 \text{ MHz}}{1 \text{ clock cycle}} = 550 \text{ MSPS}$$

MSPS (Million Samples per Second)



Výkon FPGA (DSP bloků)

MMAC/s (Million Multiply Accumulates per Second)

pohybuje se v rozmezí řádově od 10^1 do 10^4 (milion násobení a mezisoučtů za sekundu).

často se redukuje na **max. teoretický výkon**

= počet násobiček x max. hod. frekvence násobičky.

Např. LatticeECP-DSP20 má 28 násobiček (18x18)

a max. hod. frekvence je 250 MHz \Rightarrow 7000 MMAC/s



Vysokorychlostní transceivery

Přechod od paralelních sběrnic k vysokorychlostním sériovým rozhraním (lepší časové parametry, integrita signálů, diferenční přenos v párech).

Signály se převádí z/do paralelního rozhraní (8–128 bitů), příp. se ještě linkově kódují (pro potlačení stejnosměrné složky a zajištění dostatečné hustoty hran v signálu).

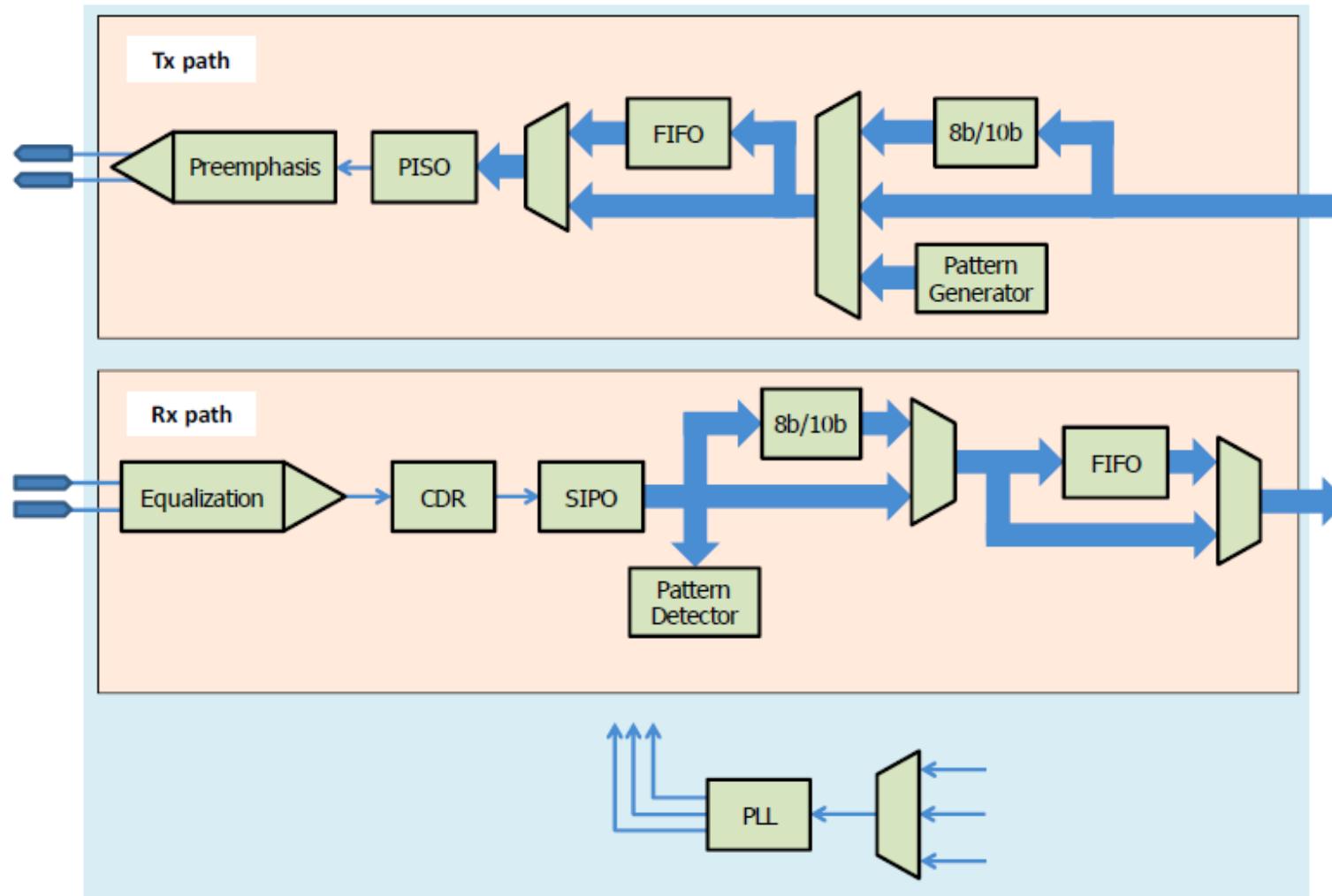
Přenosové rychlosti cca od 0,5 do 58 Gb/s (podle typu FPGA).

Na vstupu bývají bloky pro obnovu synchronizace CDR (Clock and Data Recovery).

Součástí bývají bloky PLL (společné pro více transceiverů) pro generování rychlých hodinových signálů.



Vysokorychlostní transceivery





Spojení FPGA + MCU

Elektronické systémy často používají jak levnější a univerzální procesor (včetně periferií), tak rychlý hardware (FPGA)

- ⇒ realizace obojího uvnitř FPGA
- úspora plochy na DPS, současný společný vývoj systému.

Procesory:

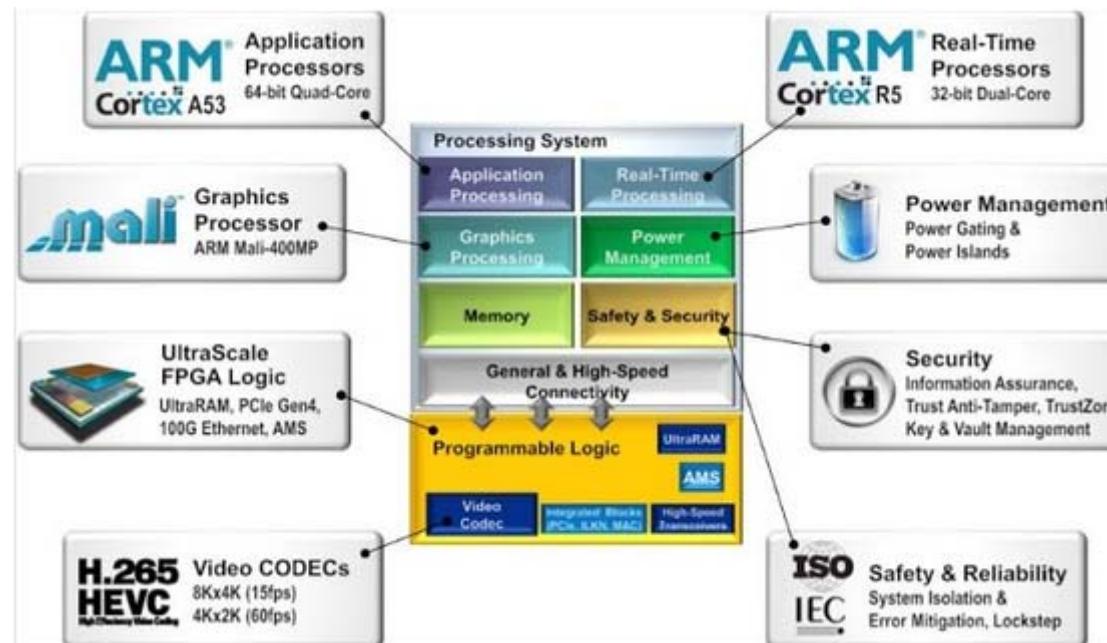
hardware (ARM Cortex, PowerPC, Excalibur, AVR) – na čipu FPGA zaintegrováno CPU jádro (řadiče, ALU), jako paměť slouží embedded RAM, periferie často v log. poli;
omezená přenositelnost návrhů mezi různými typy FPGA;

software (NIOS V (na bázi RISC-V), MicroBlaze, LatticeMico) – sice pomalejší (cca o polovinu vůči HW), ale architekturu lze přizpůsobit potřebám; možno použít i více CPU současně; lze implementovat do různých typů FPGA; energeticky náročnější.



Hardwarevý procesor ARM

V dnešních FPGA jsou nejrozšířenější různé varianty ARM procesorů např. dvoujádrový Cortex-A9 (32-bit) nebo čtyřjádrový Cortex-A53 (64-bit), RISC, superskalární, frekvence přes 1 GHz.



Xilinx Zynq UltraScale+



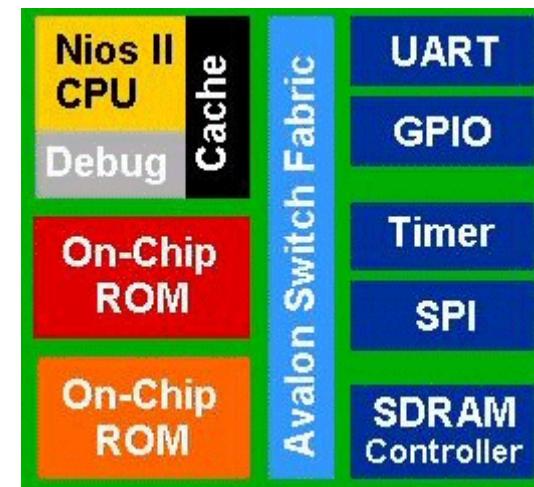
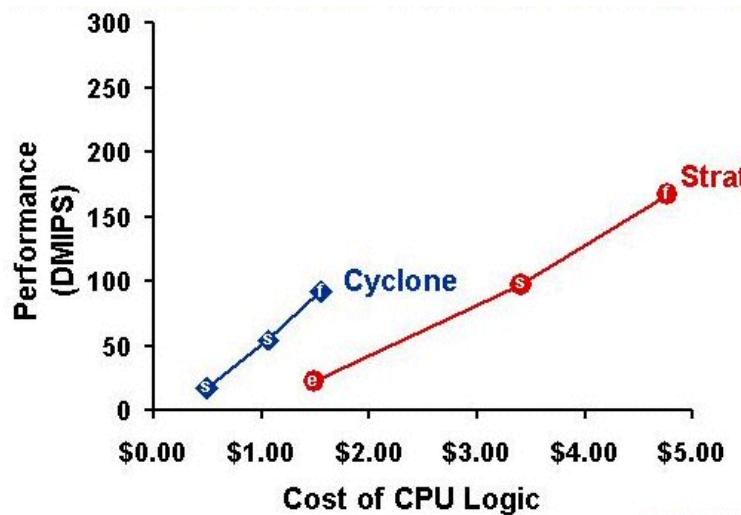
Procesor NIOS II (Intel)

Softwarový 32-bitový procesor, RISC

32 všeobecných registrů, 32 externích přerušení

Varianta Fast – Standard – Economy, výkonnost až 250 DMIPS;

Operace: aritmetické (+, -, *, /), relační (=, ≠, ≥, <) s typy signed a unsigned, logické (AND, OR, NOR, XOR), posuny a rotace.



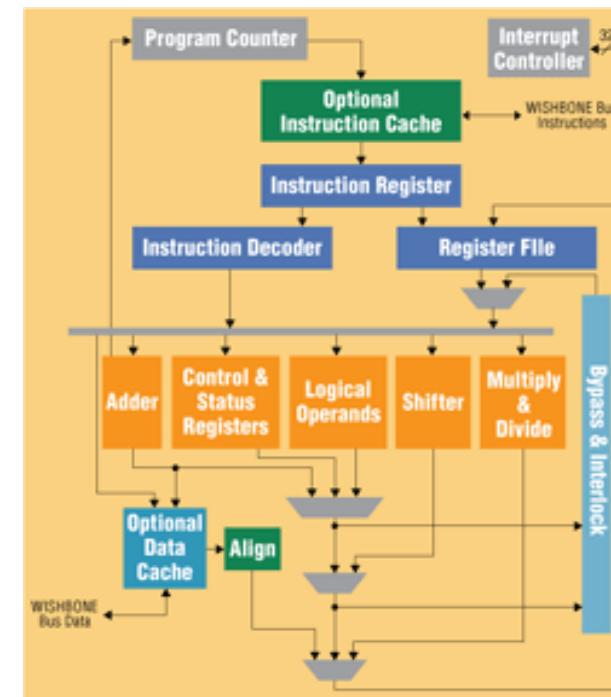


Procesor LatticeMico32

- 32bitový softwarový procesor (data i instrukce), RISC,
- 32 všeobecných registrů, až 32 externích přerušení,
- řadiče paměti (pro asynchronní SRAM, pro paměť DDR1, pro blokovou paměť na čipu)
- 32bitový časovač, DMA řadič, řadič I2C, SPI, Ethernet řadič, UART)

Performance and Resource Utilization for LatticeEC/ECP Devices ¹		
Configuration	LUTs	fMAX (MHz)
Basic	1,830	81
Standard	2,040	89
Full	2,230	92

Performance and Resource Utilization for LatticeECP2/M Devices ¹		
Configuration	LUTs	fMAX (MHz)
Basic	1,571	98
Standard	1,816	116
Full	2,158	116





Napájecí napětí

Napájení se rozděluje do 3 skupin:

- napájení vlastního jádra s log. bloky;
- napájení vstupně-výstupních buněk (1,2 – 3,3 V);
- napájení speciálních bloků (např. fázové závěsy).

Napětí jádra souvisí s výrobní technologií: 3,3 V (350 nm),
2,5 V (220 nm), 1,8 V (150 nm), 1,5 V (130 nm), 1,2 V (90 nm),
1,0 V (65 nm), 0,9 V (40 nm), 0,85 V (28 nm).

Spotřeba: statická $P_s = \sum U_i \cdot I_i$

dynamická $P_d = \sum C_i \cdot U_i^2 \cdot f_i$



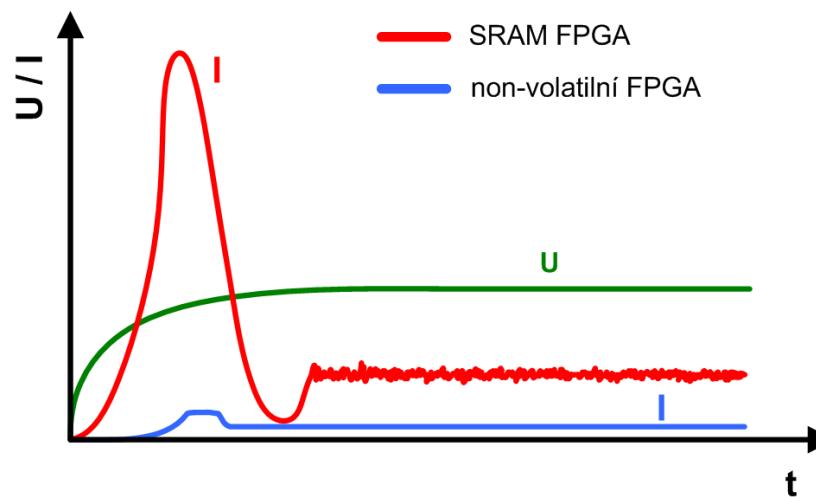
Náběh napájecího napětí

Náběh napájecích zdrojů by měl být monotónní.

Na pořadí náběhu zdrojů většinou nezáleží.

Obvody POR (Power On Reset).

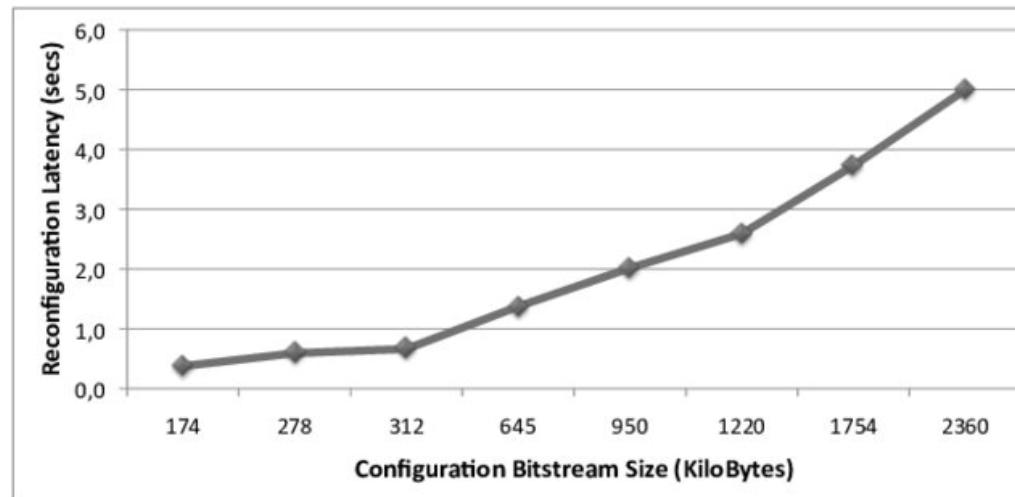
Zdroje řešeny bud' spínanými nebo *lineárními* regulátory.





Doba konfigurace

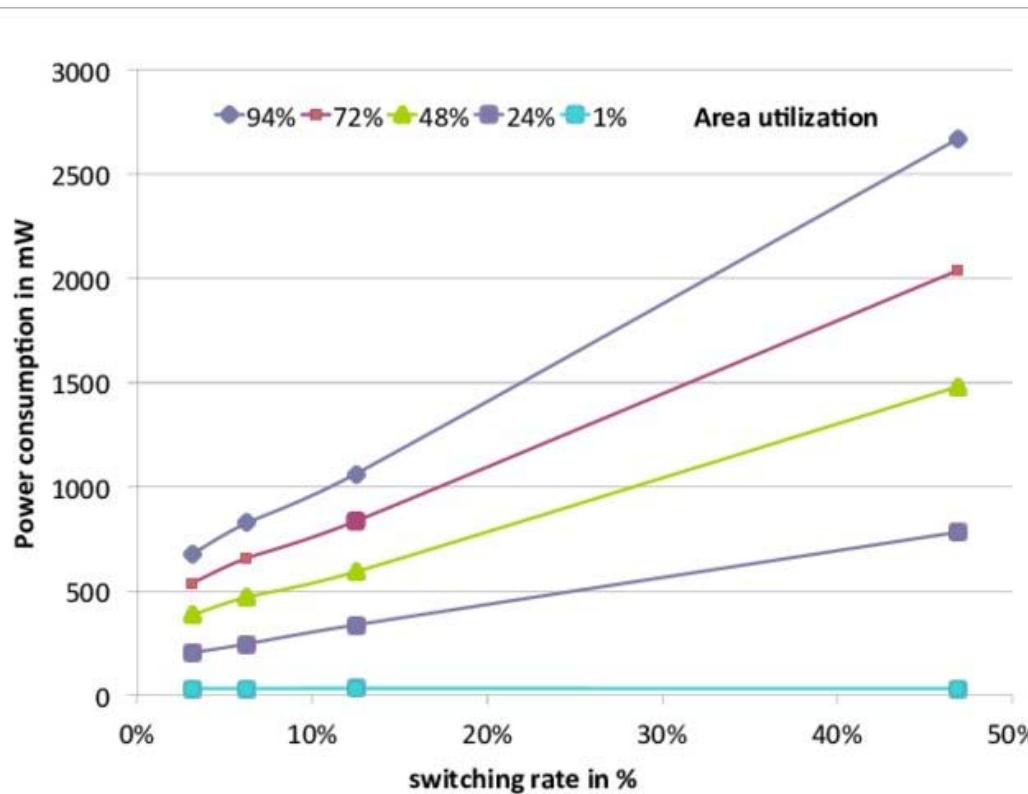
Záleží na velikosti obvodu a na paralelnosti načítání





Souvislost P, f a využití obvodu

Vzájemný vztah příkonu, rychlosti a využití obvodu





Virtex UltraScale+ (AMD)

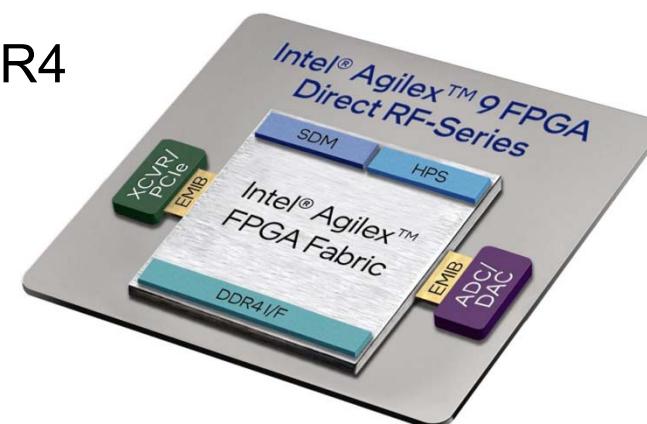
- 14/16 nm technologie, až 832 I/O pinů, napěť. úrovně 1,0-1,8 V;
- až 4,4 mil. log. buněk;
- až 95 Mb blokRAM + 432 Mb UltraRAM;
- až 11904 DSP bloků;
- až 128 transceiverů (rychlosť až 32,75 Gb/s);
- Integrované řadiče PCI Express, Ethernet 100G, 150G interlaken;





Agilex 9 (Intel)

- 10 nm technologie, 768 I/O pinů, až 2,693 mil. LE;
- až 17056 DSP bloků;
- až 287 Mb embedded paměti;
- 24 PLL bloků;
- Quad-core 64bit ARM Cortex-A53;
- 384 LVDS kanálů;
- 40 transceiverů 58 Gb/s;
- Hard IP: PCIe, Ethernet, podpora DDR4

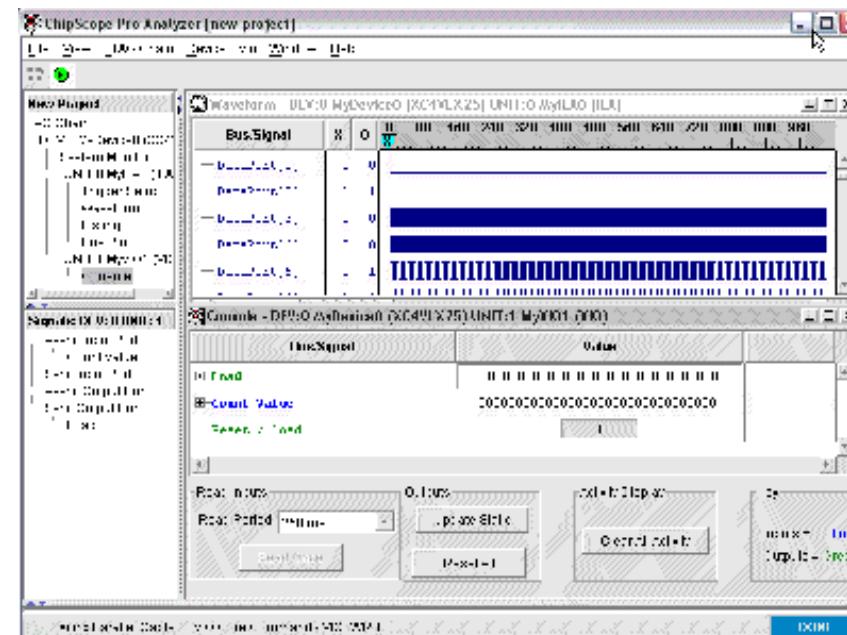




ChipScope (AMD)

Programový blok využívající volné buňky obvodu FPGA k vytvoření logického analyzátoru (přes JTAG rozhraní)

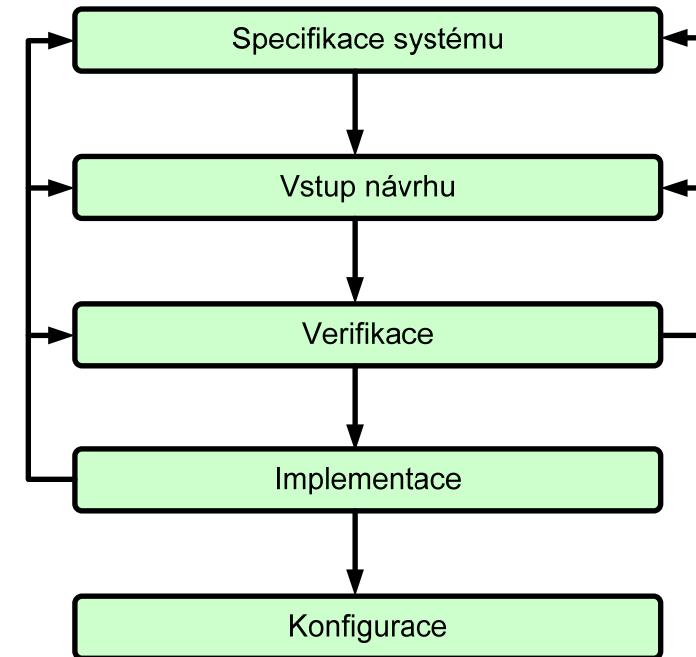
- dovoluje na monitoru sledovat vnitřní signály, sběrnice, uzly, vestavěné hw i sw procesory, ...
- zkracuje se doba vývoje (rychlejší než simulace);
- určeno pro vybraná FPGA.





Metodika návrhu obvodů FPGA

- specifikace systému,
- vstup návrhu (popis funkce)
 - VHDL, Verilog, HDL, ...
- simulace (nejčastěji časová),
- syntéza logického obvodu,
- mapování na technologii,
- rozmístění a propojení,
- generace výstupního souboru,
- konfigurace (programování).

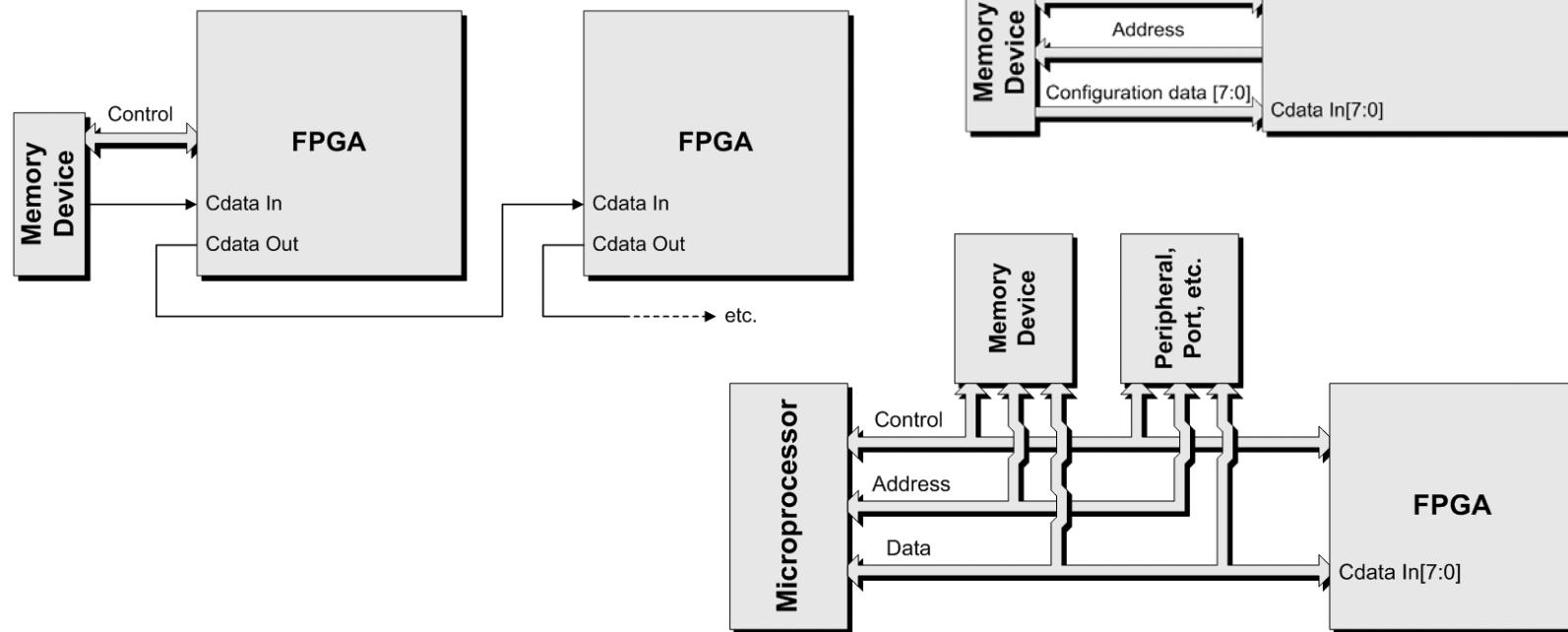




Inicializace FPGA

Možnosti inicializace FPGA obvodů na bázi SRAM:

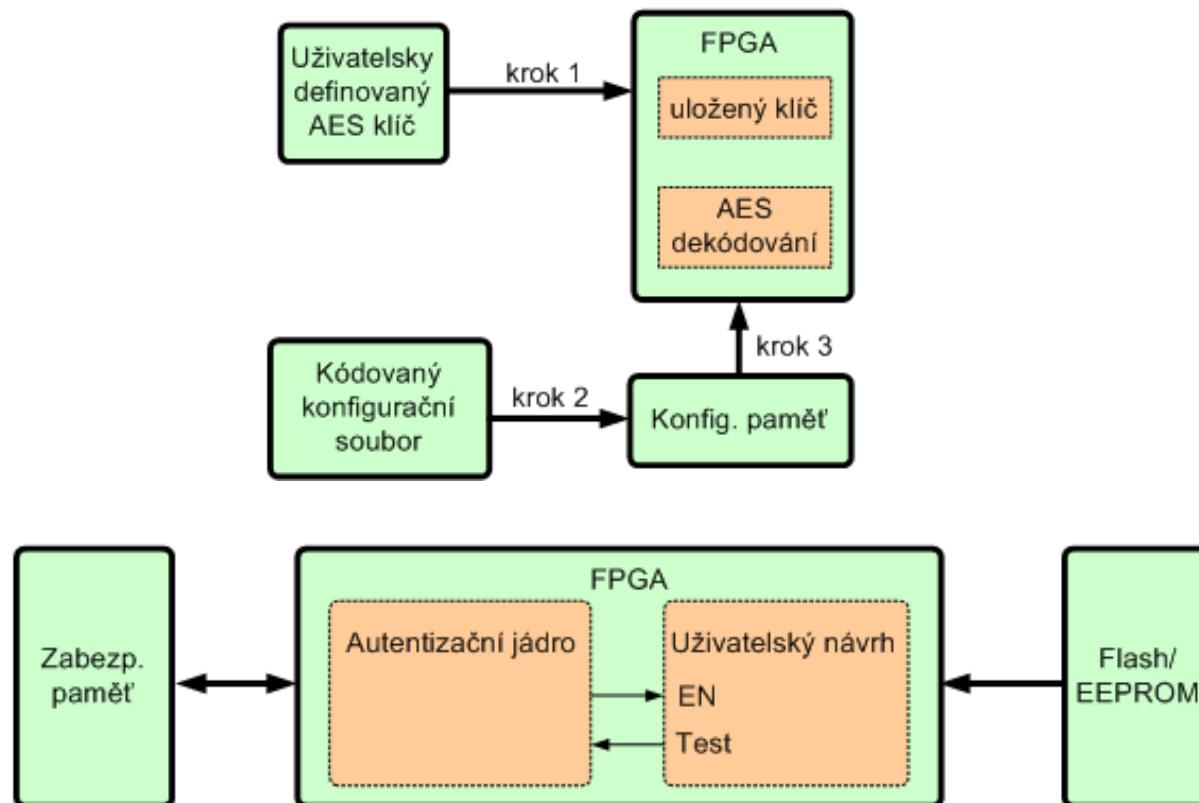
- sériově z vnější EEPROM
- paralelně z vnější EEPROM
- z mikroprocesoru





Zabezpečení dat

Zejména u konfiguračních dat uložených v externích pamětech



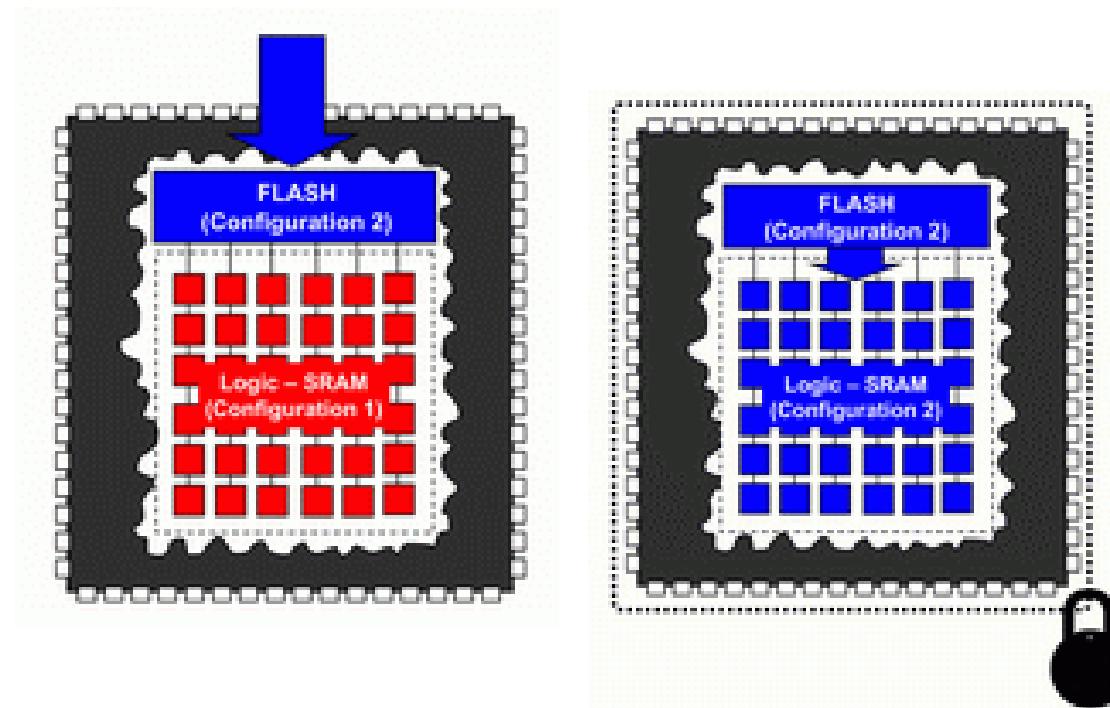


Transparentní rekonfigurace

Za běhu aplikace je možno přehrát obsah flash paměti, pak se

uzamknou piny a přehraje se nový program do RAM

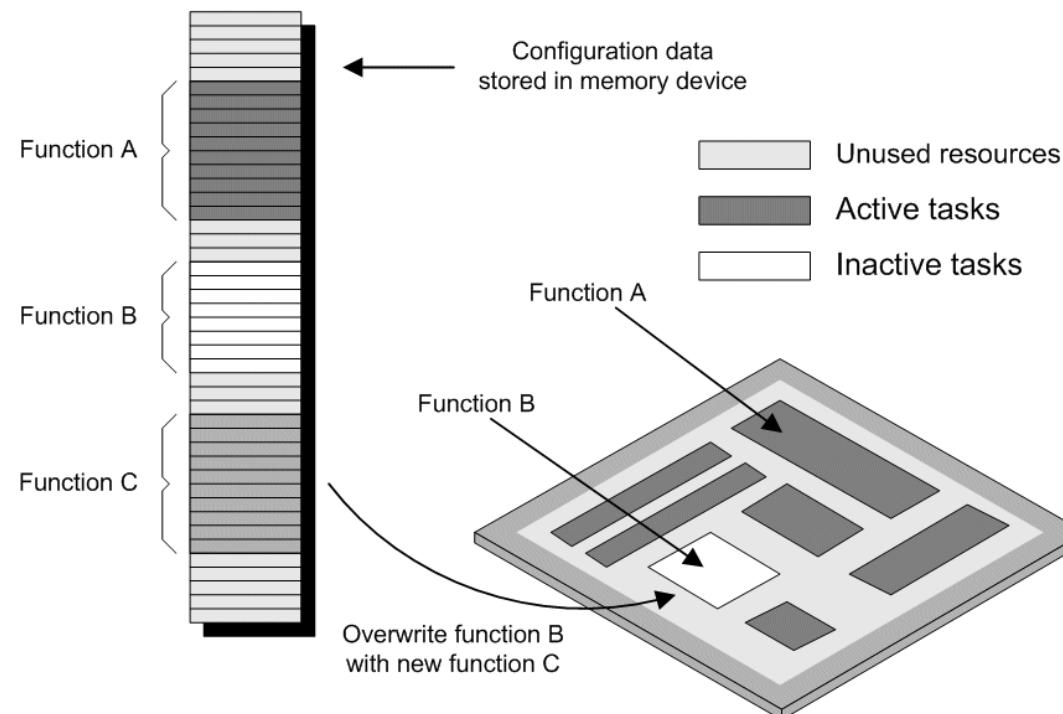
- minimální doba přerušení (cca za 1 ms) – Lattice Semiconductor





Dynamická rekonfigurace

Za provozu lze měnit část obsahu FPGA (oblast se uzamkne, přehraje se část konfigurace, která je již připravena v paměti)
- Atmel (AT94k), Xilinx





Trendy vývoje FPGA obvodů

- růst velikosti FPGA obvodů (miliony logických buněk),
- stále rozsáhlejší vkládané bloky (embedded core) – RAM, DSP,
- hybridní čipy – kombinace procesorových jader (HW i SW) a programovatelné logiky,
- rekonfigurovatelná logika (on-the-fly),
- možná vzdálená změna konfigurace - šifrováno
- snižování napěťových úrovní $3.3V \rightarrow 2.5V \rightarrow 1.8V \rightarrow 1.5V \rightarrow 1.0V$ (jádro $1.2V \rightarrow 0.9V$),
- technologie $65\text{ nm} \rightarrow 40\text{ nm} \rightarrow 28\text{ nm} \rightarrow 16\text{ nm} \rightarrow 10\text{ nm}$,
- frekvence řádově stovky MHz.



TECHNICKÁ UNIVERZITA V LIBERCI
**Fakulta mechatroniky, informatiky
a mezioborových studií**



Technologie výroby integrovaných obvodů

Milan Kolář

Ústav mechatroniky a technické informatiky



evropský
sociální
fond v ČR



EVROPSKÁ UNIE



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY



Projekt ESF CZ.1.07/2.2.00/28.0050
**Modernizace didaktických metod
a inovace výuky technických předmětů.**

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ



Technologie výroby

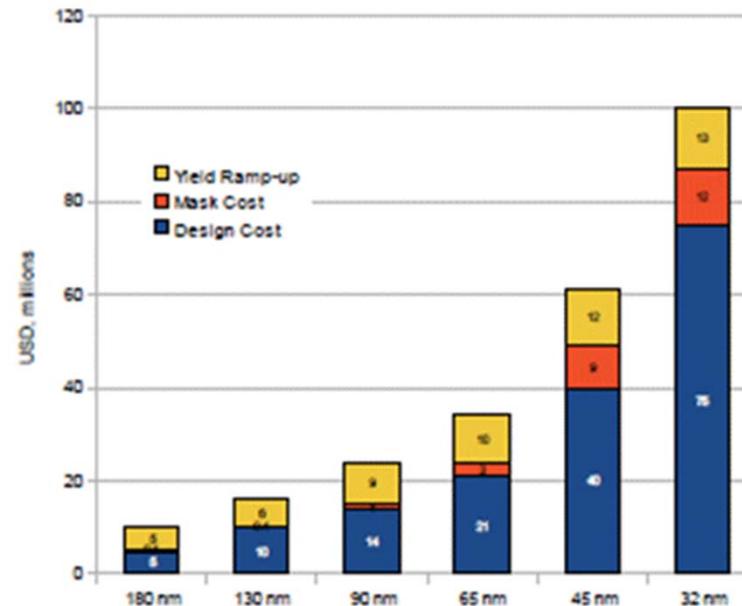
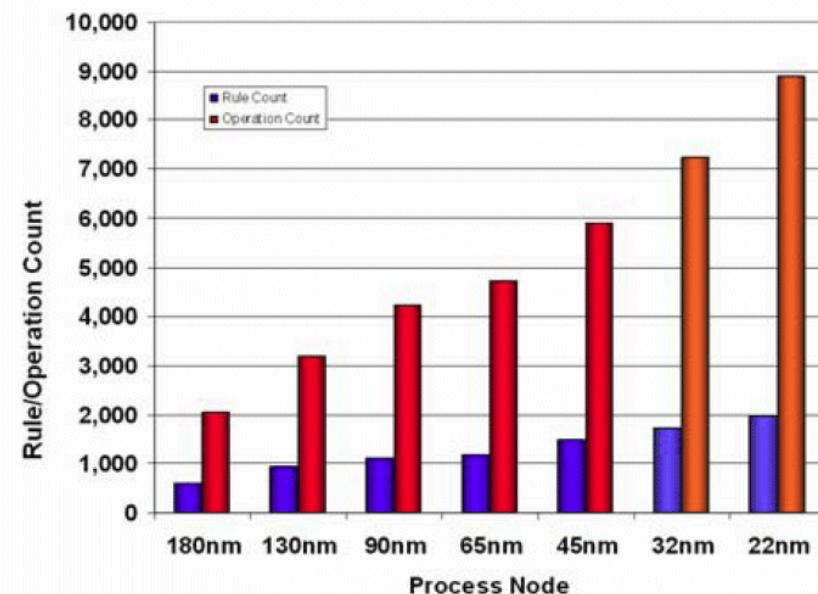
Technologie založeny na **fotolitografickém procesu** - technologické operace na oblasti vymezené maskami (difuse, iontová implantace, leptání), rentgenová litografie, nově se používá bezmasková elektronová litografie.

Je velice problematické a drahé dosahovat litografickými metodami rozlišovací schopnosti pod 10 nm. V budoucnu se očekává přechod od litografického opracovávání homogenního materiálu (někdy označováno jako metody „top-down“) k metodám využívajících schopnosti samouspořádání zejména u organických molekul (označováno jako metody „bottom-up“).

Přecházíme od mikroelektroniky a nanoelektroniky k **molekulární elektronice** (na pomezí elektroniky, chemie a biologie).



Růst nákladů na vývoj

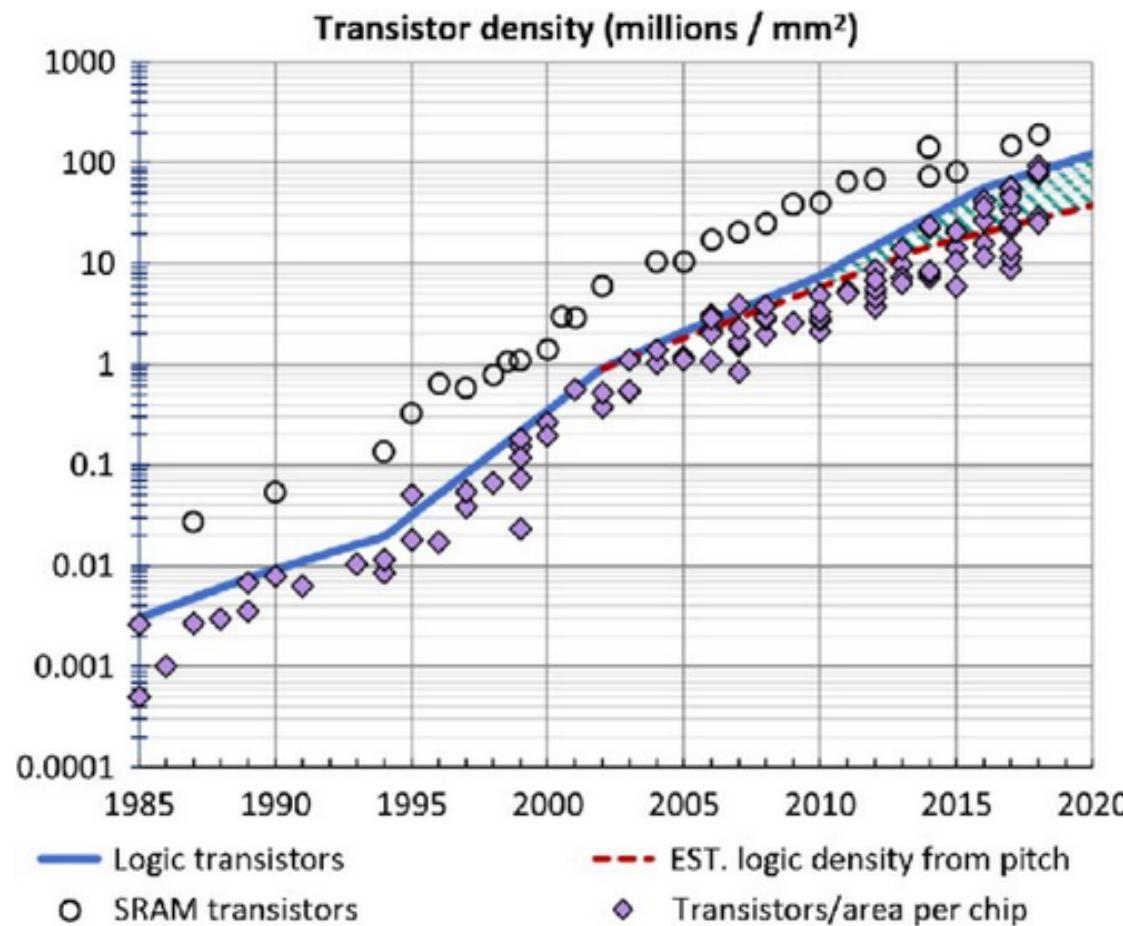


Rockův zákon (někdy označován jako druhý Moorův zákon): Investice do nových zařízení na výrobu čipů se zdvojnásobuje každé čtyři roky.



Efektivnost nových technologií

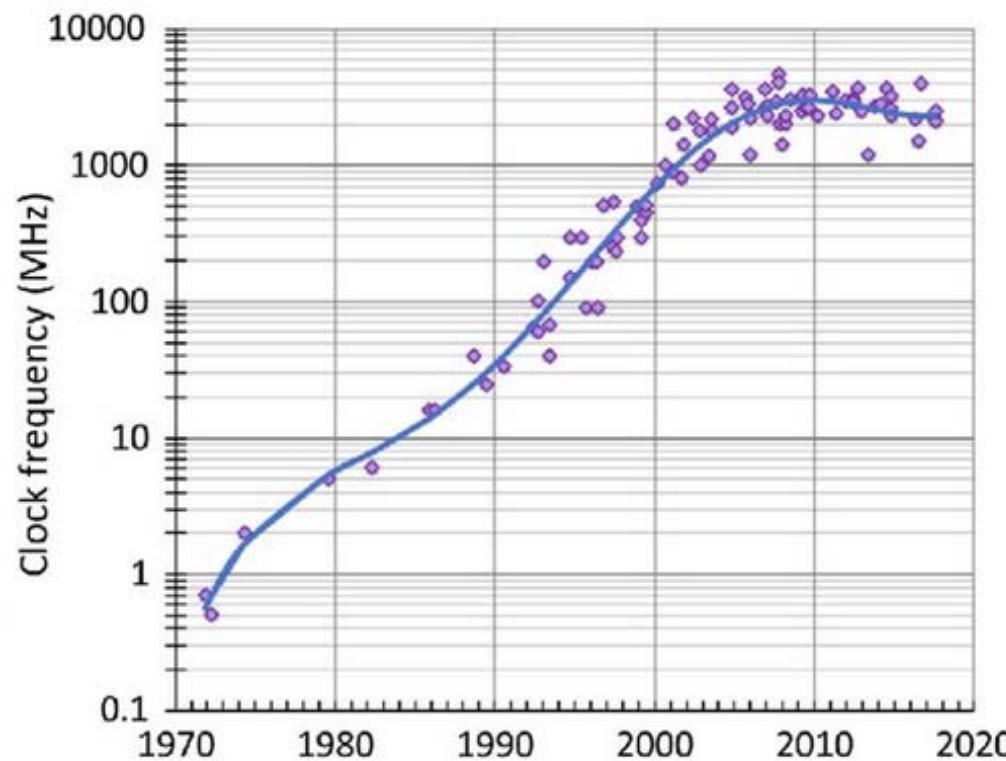
Výrazně se zvyšují počty tranzistorů, ale klesá cena za tranzistor





Efektivnost nových technologií

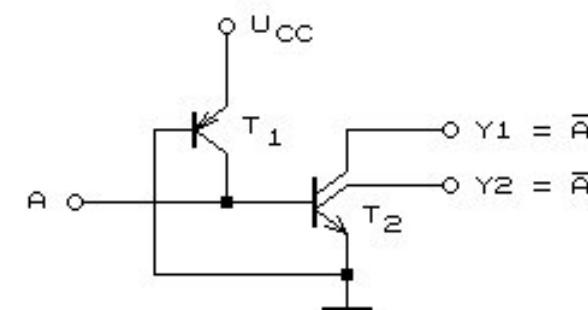
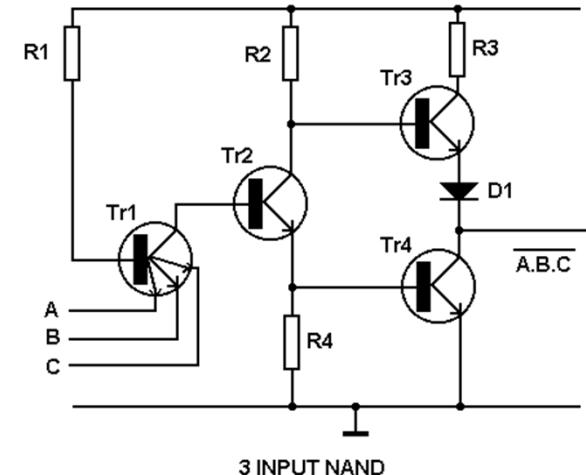
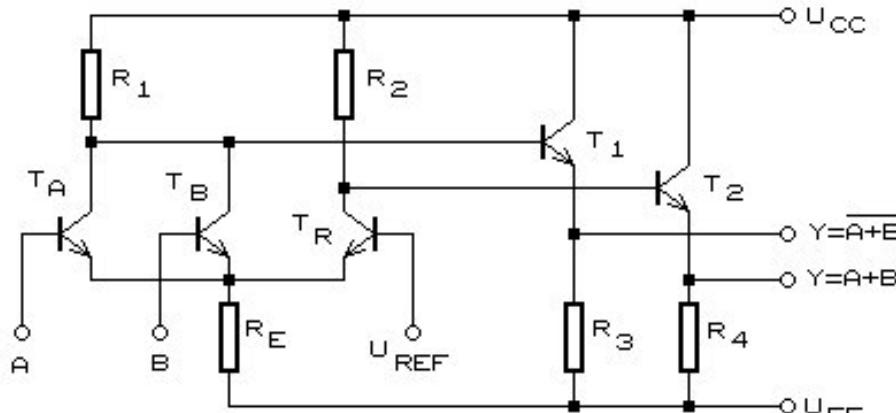
Nezvyšují se hodinové frekvence systémů, přesto roste výkonnost
(vlivem paralelnosti architektur)





Bipolární technologie výroby

- dělení podle realizace základních logických členů:
 - TTL, ECL, I²L





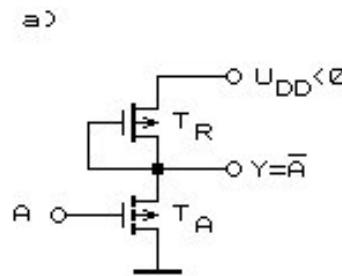
Unipolární technologie výroby

- Charakteristické vlastnosti:
 - jedna součástka (MOSFET) ve funkci spínače, odporu, kapacitoru
 - velká hustota integrace
 - velmi malý příkon (závislé na frekvenci)
 - široký rozsah napájecího napětí
 - nízké nároky na stabilitu napájecího napětí
 - velké výstupní větvení
 - malý výstupní výkon (výstupní proud)
 - citlivá na rušivá napětí
 - zpoždění závislé na zatížení
- Druhy unipolárních obvodů:
 - PMOS, NMOS, CMOS, BiCMOS

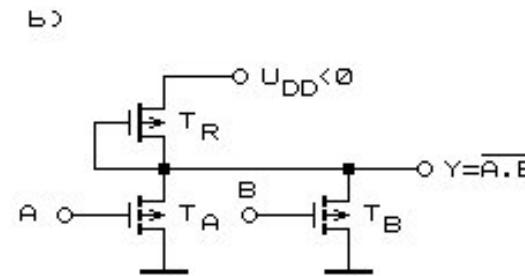


Technologie PMOS a NMOS

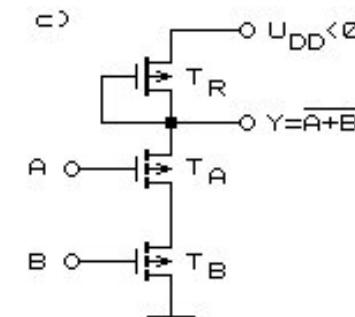
- PMOS a) invertor



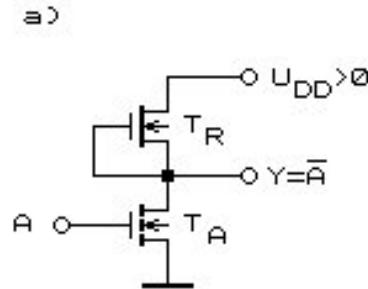
- b) NAND



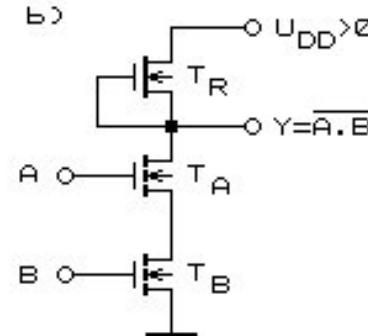
- c) NOR



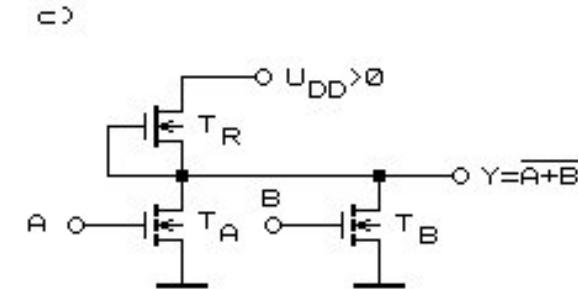
- NMOS a) invertor



- b) NAND



- c) NOR

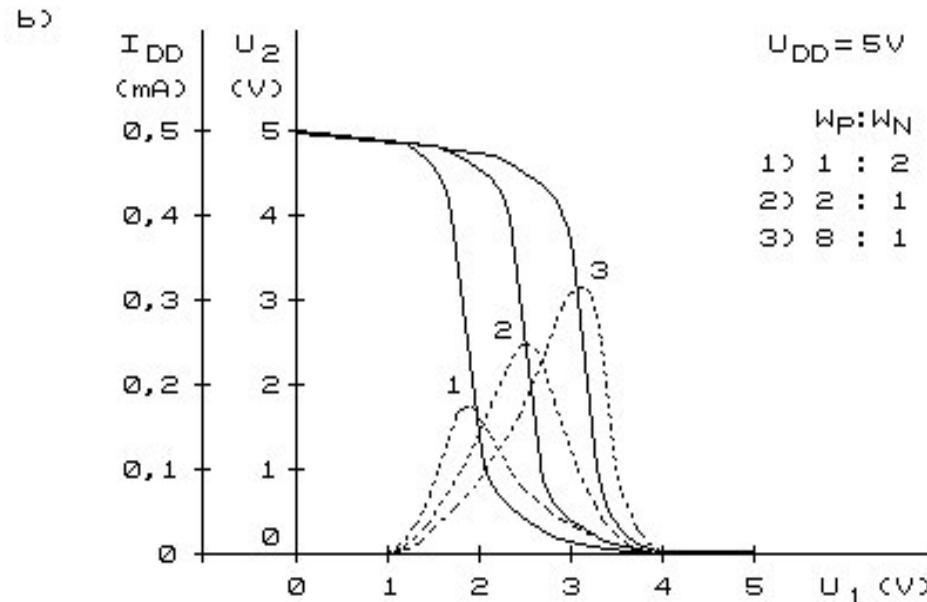
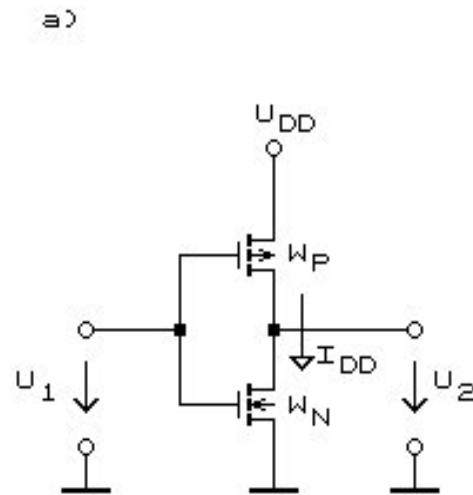




Technologie CMOS

Invertor

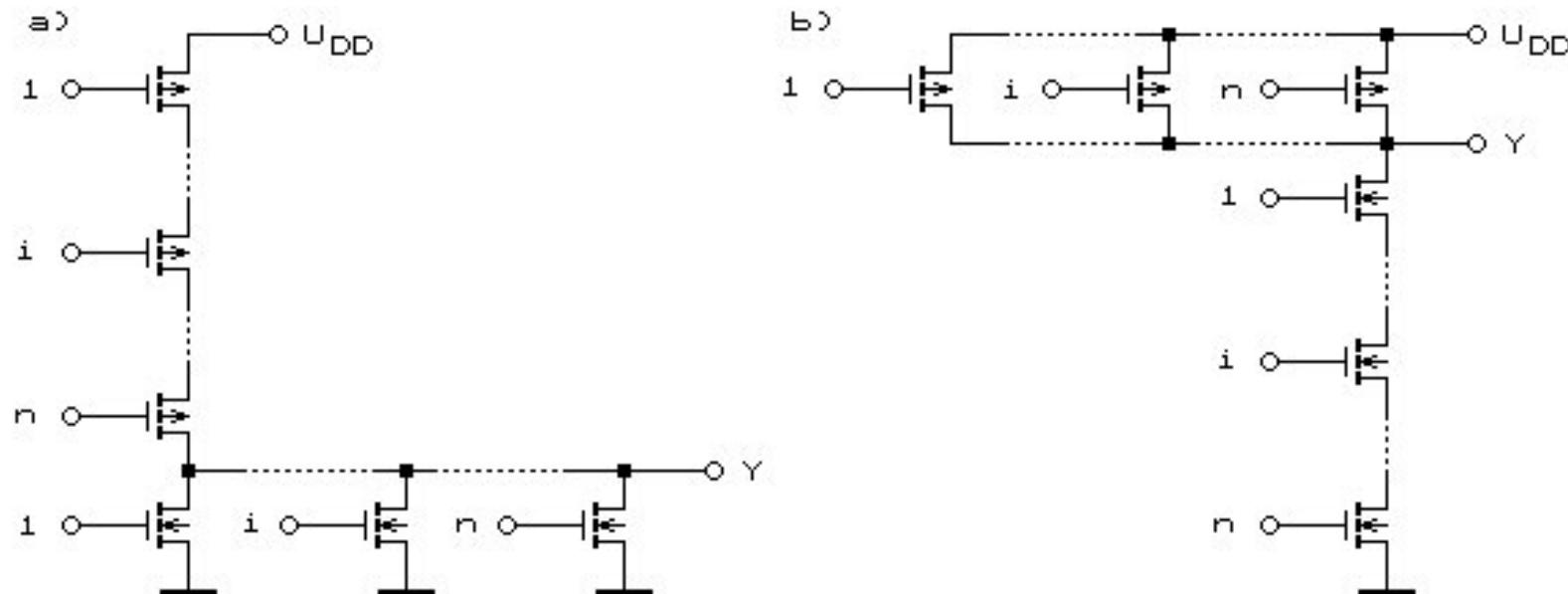
kanál tranzistoru N dvakrát vodivější vůči tranzistoru s kanálem P





Technologie CMOS (pokračování)

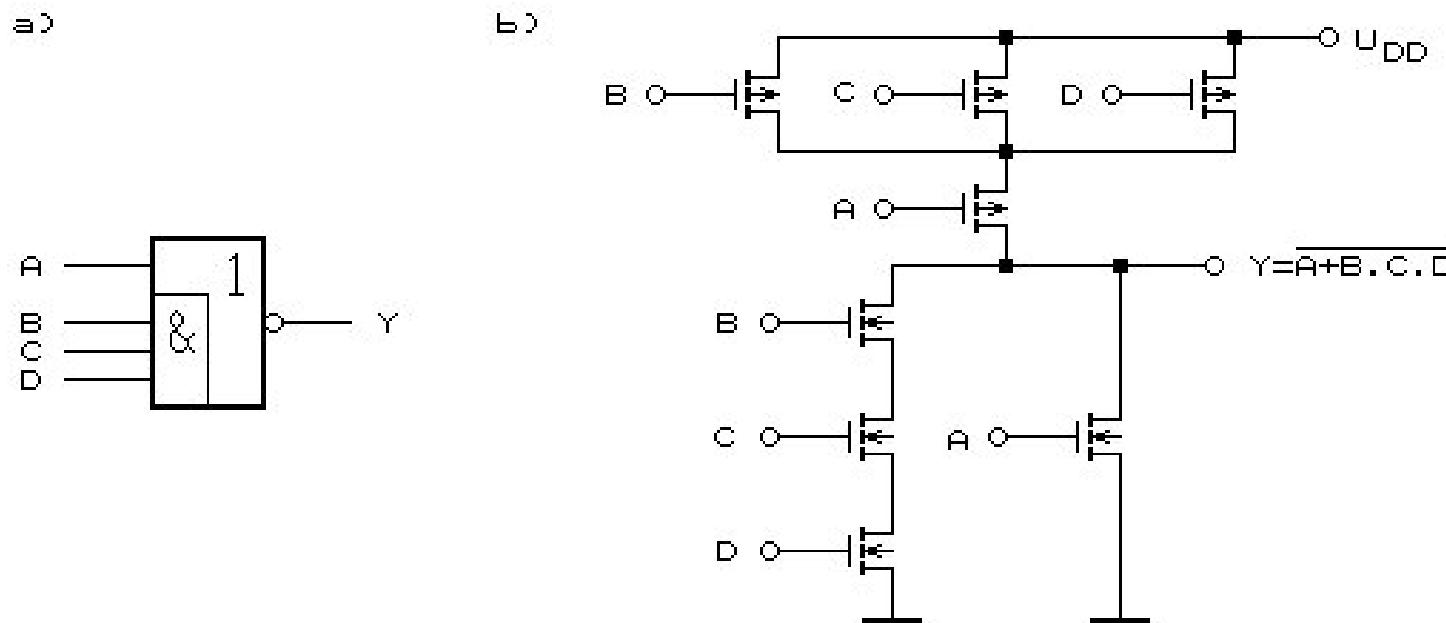
Logické členy NOR a NAND





Technologie CMOS (pokračování)

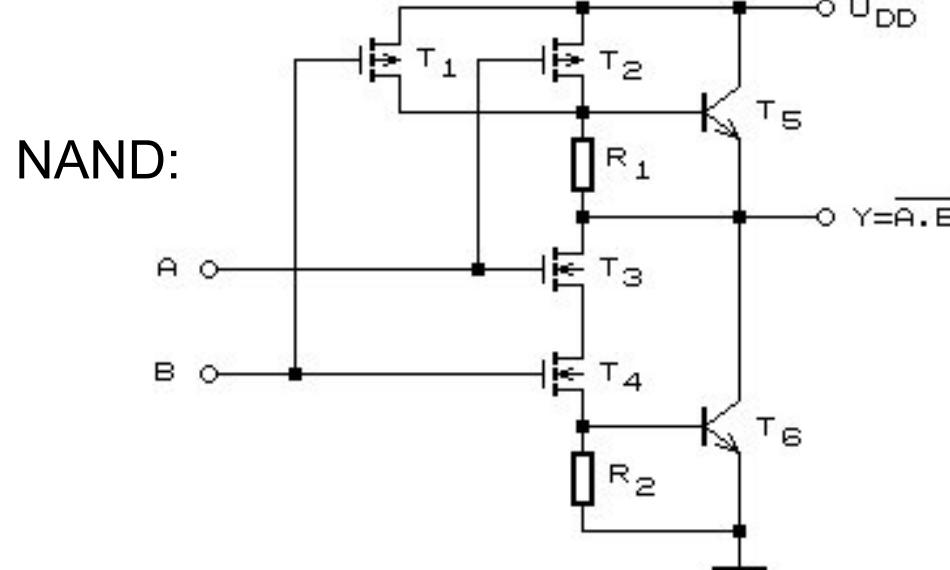
Člen AND-OR-INVERT





Technologie BiCMOS

- velká rychlosť, velký výstupní výkon (vhodné pro RF obvody),
- malá spotřeba v klidovém stavu,
- zpoždění téměř nezávislé na zatížení,
- technologicky náročné (aktuálně 55–130 nm), GaAs.





Snižování napájecího napětí

Trend související s použitou výrobní technologií:

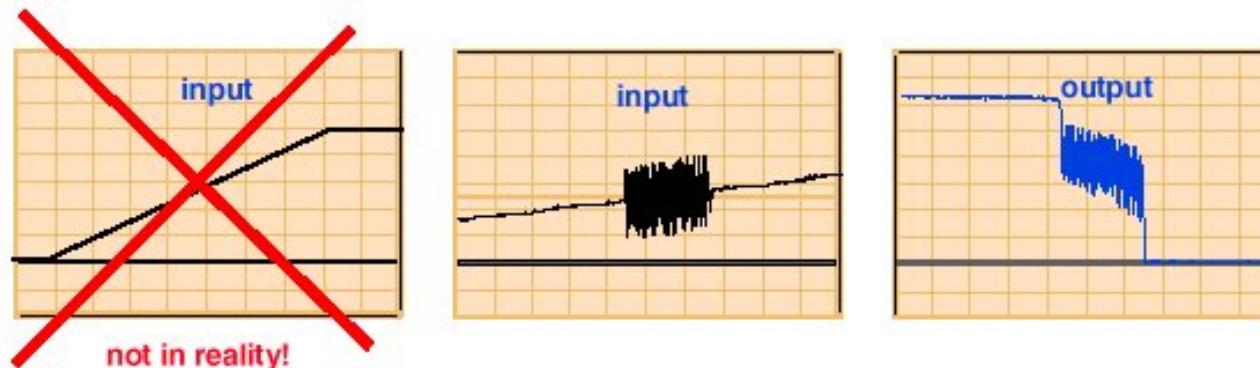
3,3 V (u technologie cca 350 nm), 2,5 V (220 nm), 1,8 V (150 nm),
1,5 V (130 nm), 1,2 V (90 nm), 1,0 V (65 nm), 0,9 V (40 nm),
0,85 V (28 nm).

- Nutné pro snižování ztrátového výkonu (tepla),
- vede na zvyšování počtu napájecích napětí (při zachování kompatibility s okolím),
- zhoršují se dynamické parametry obvodů,
- snižuje se odolnost proti elektromagnetickému rušení,
- na sběrnicích se preferují diferenční napěťové standardy.

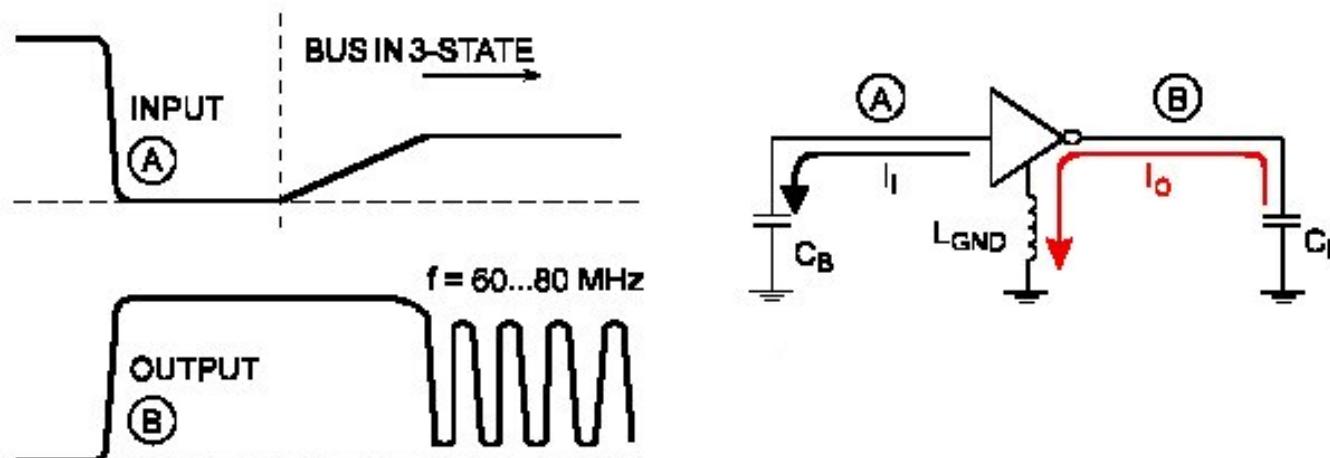


Pomalý vst. signál + vysoká impedance

Pomalu se měnící vstupní signál



Nebezpečí oscilací vlivem třístavových sběrnic





Prodlužování hran (impulsů)

Vlivem zatěžovacích parazitních kapacit (zejména spojů a hradel) dochází k prodlužování hran signálů

$$\text{náboj: } Q = C \cdot \Delta U = I_C \cdot \Delta t$$

Proud potřebný pro nabíjení či vybíjení kapacity C

$$I_C = C \frac{\Delta U}{\Delta t} \quad \Rightarrow \quad \Delta t = C \frac{\Delta U}{I_C} \quad \dots \text{doba náběhu (doběhu)}$$

Příklad: Napěťový skok 2 V, maximální výstupní proud 2 mA, zatěžovací kapacita 10 pF:

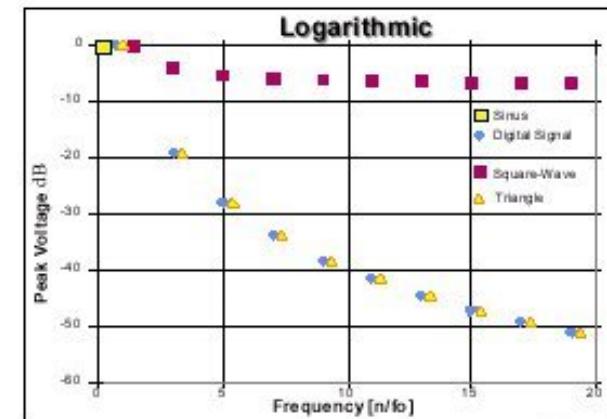
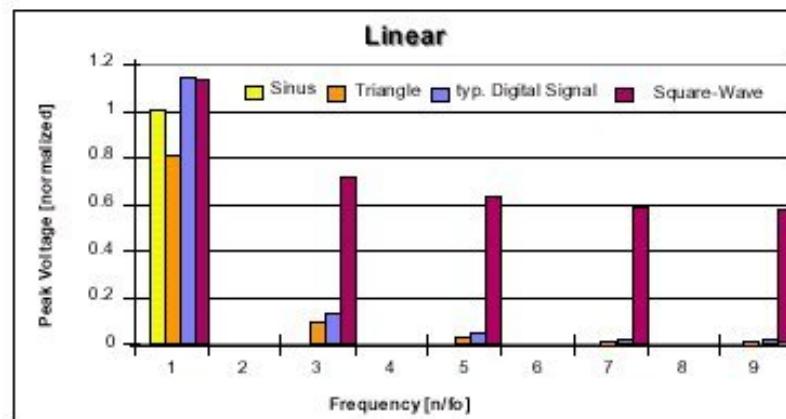
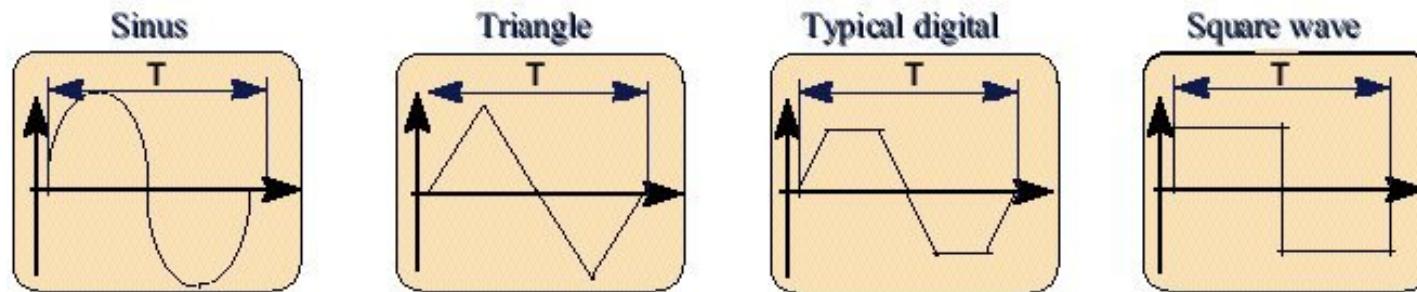
$$\Delta t = 10^{-11} \frac{2}{2 \cdot 10^{-3}} = 10^{-8} s = 10 ns$$



Kmitočtové spektrum čísl. signálů

Každou periodickou funkci lze vyjádřit Fourierovou řadou:

$$u(t) = U_0 + \sum U_{mk} \sin(k\omega_0 t + \varphi_k)$$





Standardy vstupu a výstupu

Single-Ended I/O

LVTTL (3.0V/2.5V/1.8V)

LVCMOS (3.0V/2.5V/1.8V/1.5V/1.2V)

PCI (Peripheral Component Interconnect)

PCI-X (PCI Extended)

Differential I/O

LVDS (mini-LVDS, BLVDS)

SSTL (3V/2V/1.8V) – Series Stub Terminated Logic

HSTL (1.8V/1.5V/1.2V) – High-Speed Transceiver Logic

LVPECL - Low Voltage Positive Emitter Coupled Logic

RSDS – Reduced Swing Differential Signaling

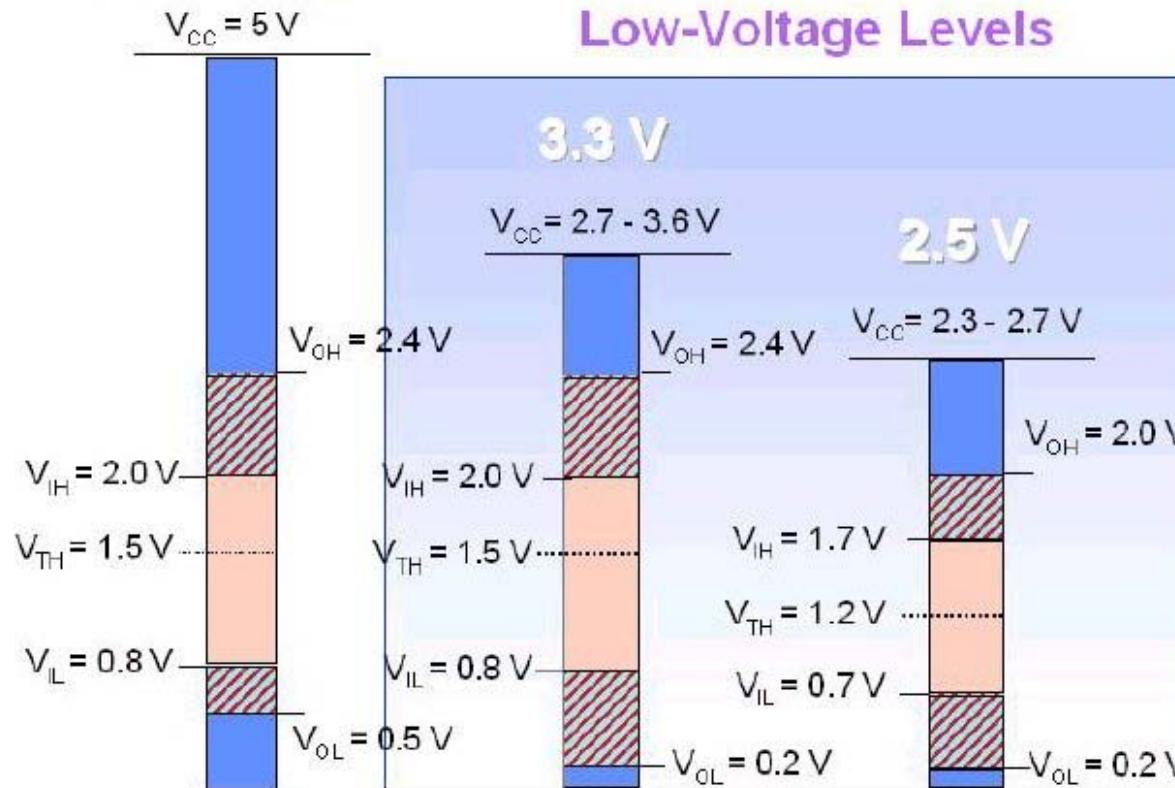
PPDS – Point-to-Point Differential Signaling



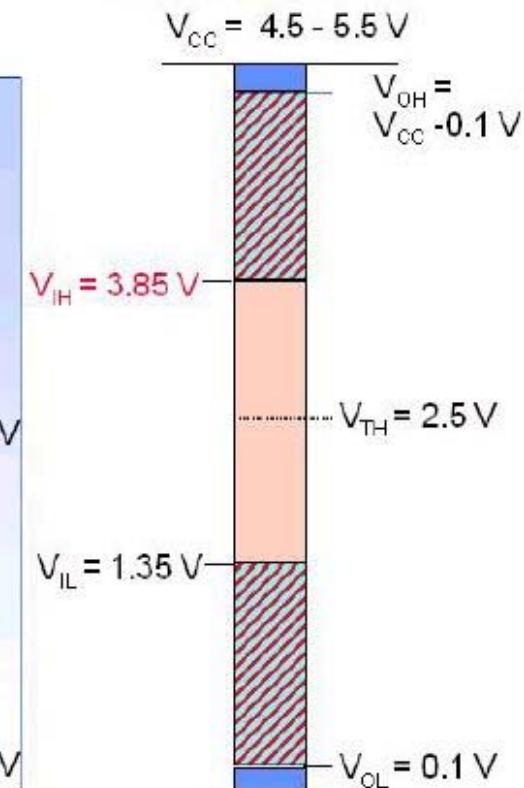
Napěťové úrovně – kompatibilita ?

Nutno porovnat minimální a maximální hodnoty jednotlivých log. úrovní

TTL Levels



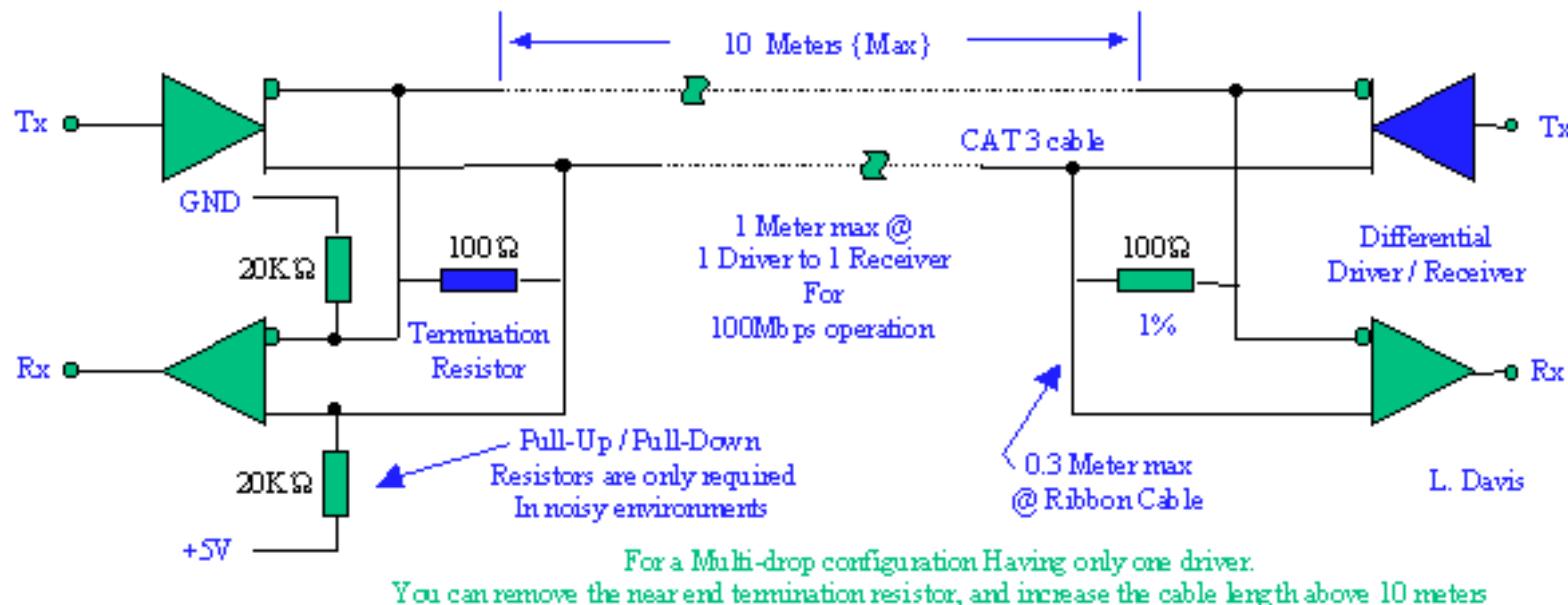
CMOS Levels





Standard LVDS

Low Voltage Differential Signaling – diferenční I/O standard pro vysokorychlostní přenosy (napěťové úrovně 0 ÷ 2,4 V, max. rychlosť 3,125 Gbps, výstupní rozkmit ± 350 mV)
($U_{OL} = 1,07$ V, $U_{IL} = 1,15$ V, $U_{TH} = 1,2$ V, $U_{IH} = 1,25$ V, $U_{OH} = 1,32$ V)

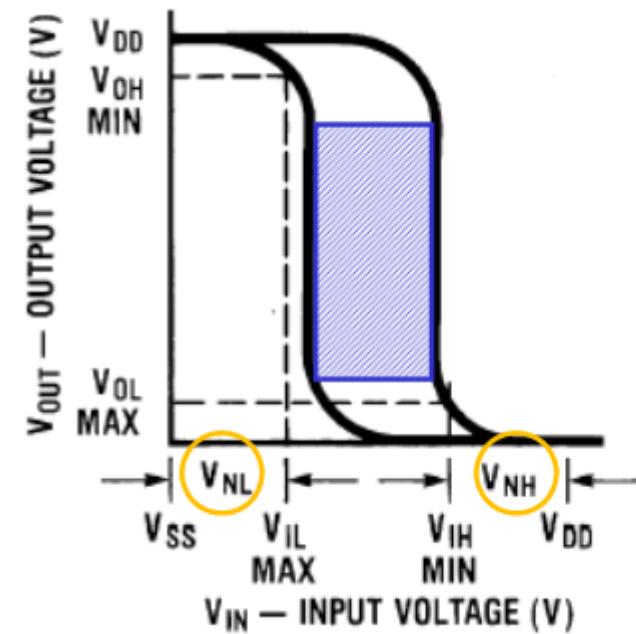




Šumová imunita (odolnost)

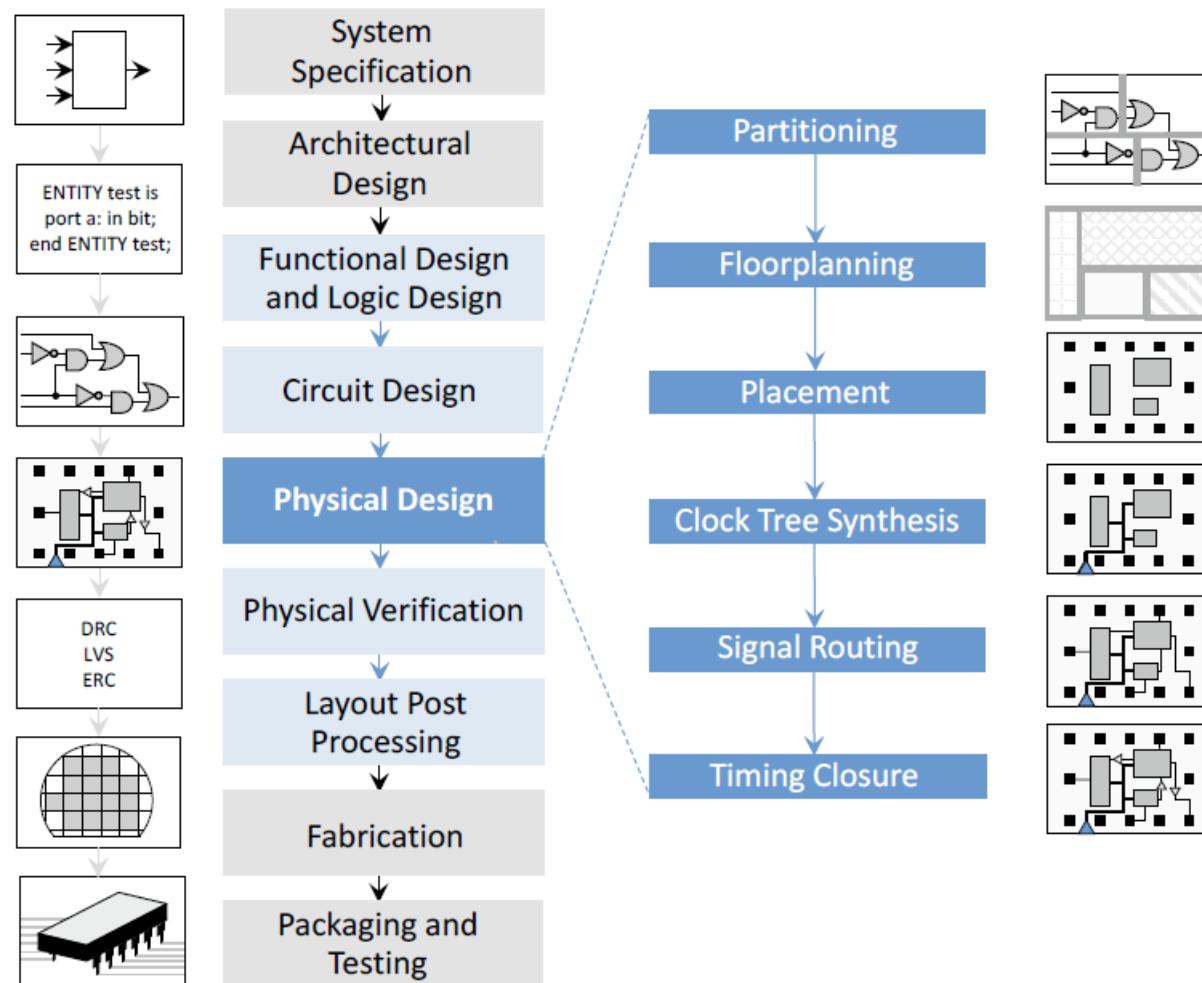
Šumová imunita (velikost vstupního napětí, které nevyvolá změnu na výstupu) je dána tolerančními pásmeny elektrických veličin na vstupu a na výstupu.

U obvodů CMOS je cca 30% U_{DD}





Hlavní fáze návrhu VLSI obvodu

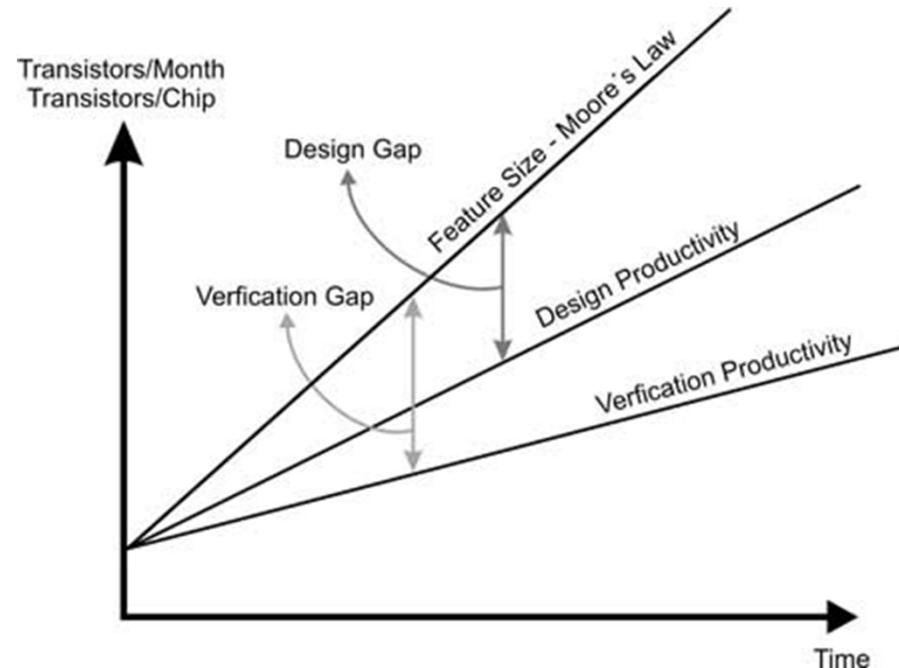




Design gap

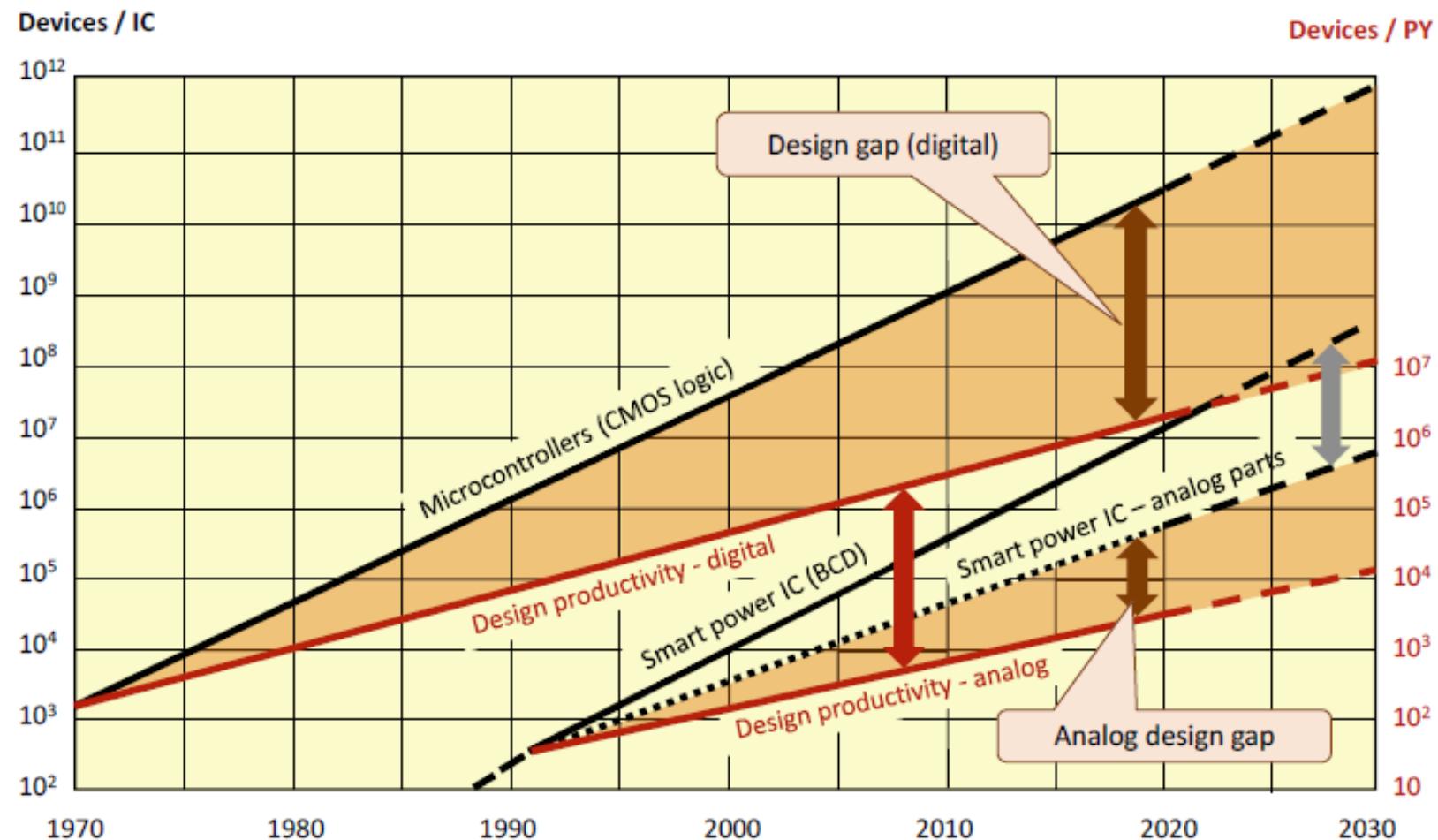
Narůstá rozdíl mezi možnostmi technologie a možnostmi návrhu čipu

- nutný pokrok v návrhovém SW, využití umělé inteligence;
- návrh negativně ovlivňuje nepravidelné struktury (nelze kopírovat);
- malá změna ve vstupních parametrech návrhu způsobuje mnohdy velkou změnu v ploše čipu (časově náročné).





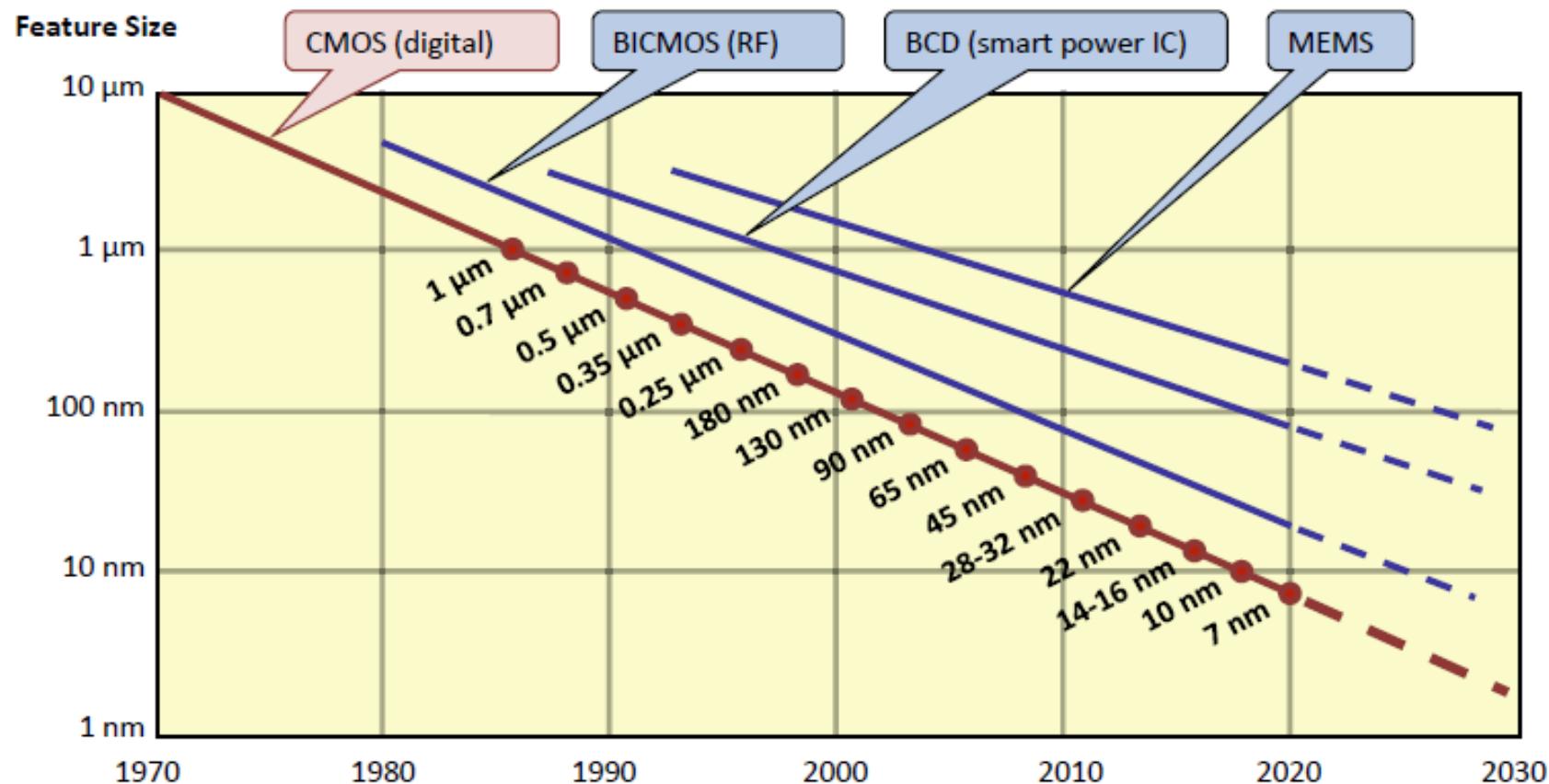
Design gap





Vývoj technologií

Nejmenší vyrobiteLNé prvky v průběhu času v různých technologiích





Specifika systémů VLSI

- Vodiče se chovají jako RC články,
 - pokles strmosti hran,
 - malá rychlosť šíření signálu,
- omezené možnosti odvodu tepla z čipu,
- vodič (sběrnice) - dražší než tranzistor,
- neustálé zhoršování poměru spínací rychlosti tranzistorů a rychlostí přenosu signálů.



Specifika VLSI systémů (pokrač.)

Příklad: α -násobné zmenšení rozměrů

- ⇒ α^2 -krát menší plocha,
- ⇒ α^2 -krát je třeba snížit množství vyzářené energie (rozptyl energie předpokládáme konstantní),
- ⇒ α -násobné snížení napájecího napětí,
- ⇒ nezměněná intenzita elektrického pole (a tím i pohyblivost elektronů),
- ⇒ α -krát menší šířky kanálů tranzistorů,
- ⇒ **α -krát vyšší spínací rychlosť aktivních prvků,**
- ⇒ α -krát nižší délka vodičů, ale α -krát vyšší odpor vodičů na jednotku délky,
- ⇒ snížení parazitních kapacit vodičů i hradel,
- ⇒ α^3 -krát nižší spínací energie budicích prvků (tranzistorů),
- ⇒ **snížení rychlosti šíření signálu po čipu.**



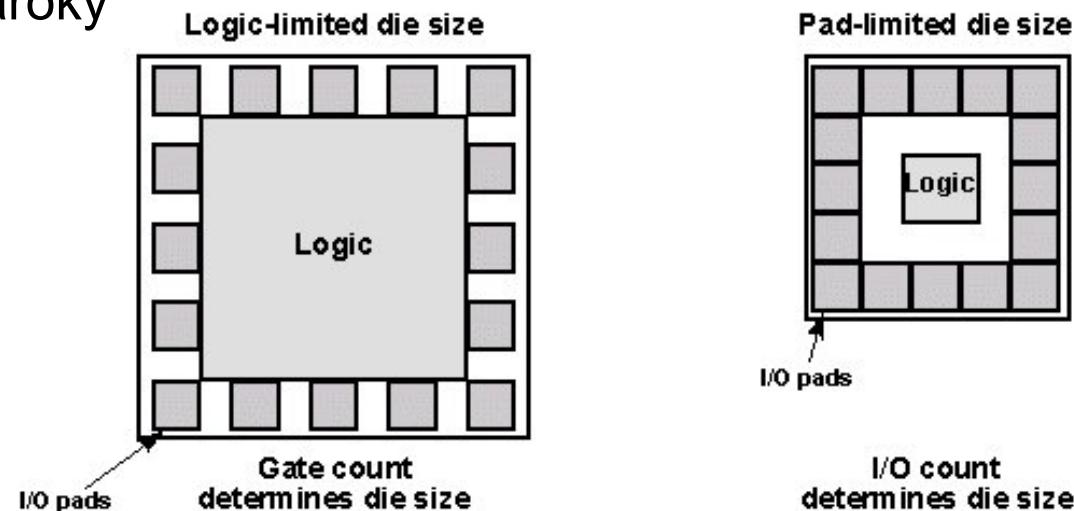
Řešení problému zpoždění signálu

- Technologická
 - snižování parazitních R a C (poly-Si → Al → Cu),
 - zvýšení spínacího výkonu tranzistorů,
 - programovatelná technologie řízení spotřeby (podle požadovaných dynamických vlastností logických buněk).
- Architektonická
 - minimální délky spojů (respektování principu lokality),
 - návrh plně synchronních systémů,
 - dělení rozsáhlých bloků kombinační logiky.



Vnější komunikace (mezičipová)

- omezený počet vývodů (2D struktury),
- rozdílné energetické poměry vně a uvnitř čipu
 - nutné I/O obvody
 - přídavné časové zpoždění
 - prostorové nároky
 - energetické nároky
- problém okraje





Řešení problému vnější komunikace

- Technologická
 - zdokonalení metod pouzdření,
 - maximální využití obvodu čipu,
 - multiplexované vývody, transceivery se serializátory/deserializátory,
 - využití třírozměrných struktur.
- Architektonická
 - požadavky funkční logiky na data jsou v rovnováze s přenosovou kapacitou vývodů,
 - využití výpočetního potenciálu na čipu (vícenásobné využití vstupních dat - princip lokality na systémové úrovni).

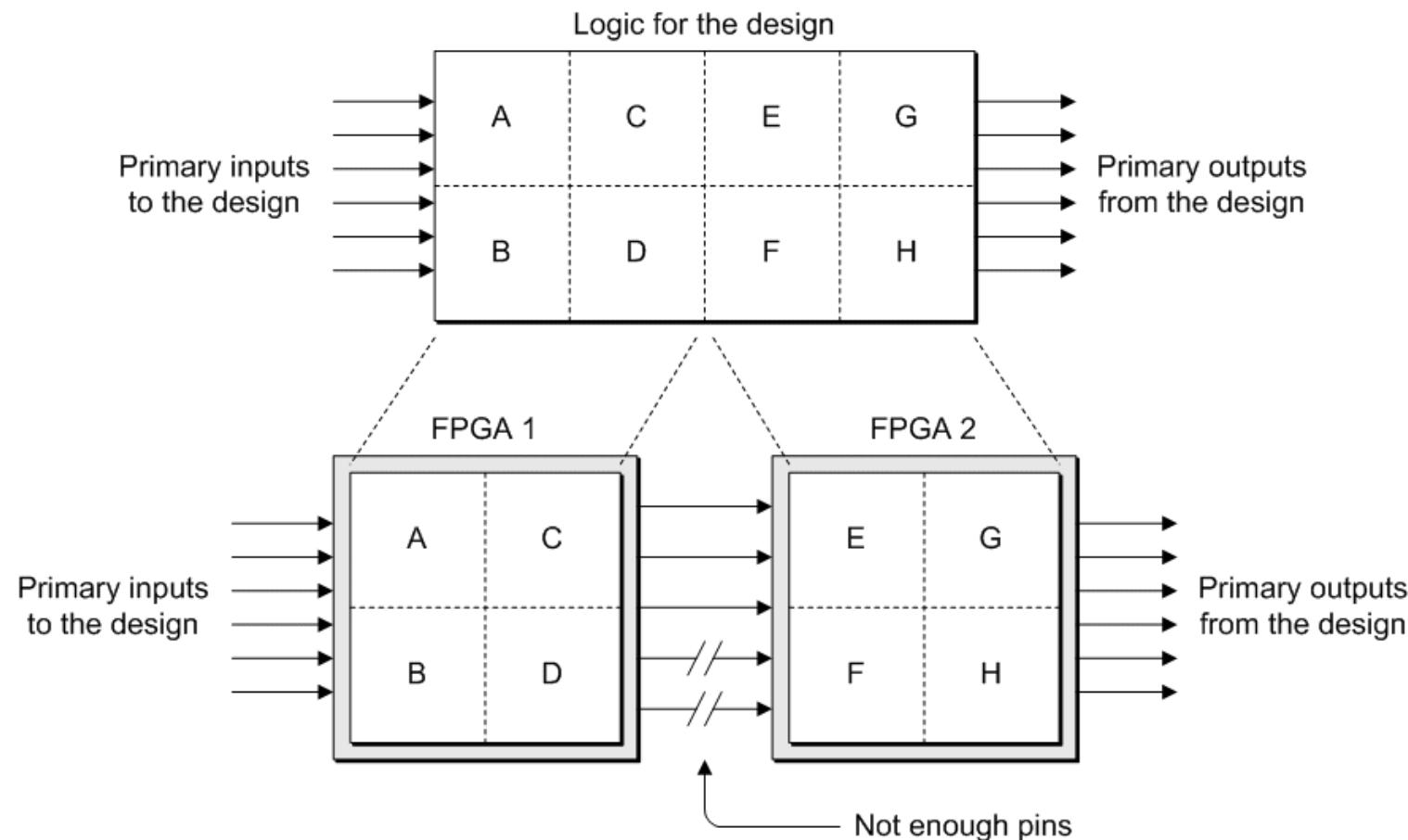


Snižování napájecího příkonu

- **Technologická**
 - snížení napájecího napětí (dáno především rozměry technologie),
 - odpojování nepoužité logiky,
 - programovatelná technologie řízení spotřeby (podle požadovaných dynamických vlastností logických buněk),
 - použití dvouhranových klopných obvodů.
- **Architektonická**
 - výběr algoritmu (paralelní vs. sériové) – množství zabrané logiky,
 - velikost pracovní frekvence,
 - blokování hodinových rozvodů do momentálně neaktivních částí,
 - strmost náběžných/sestupných hran signálů (u I/O buněk),
 - ošetření nepoužitých vstupů.



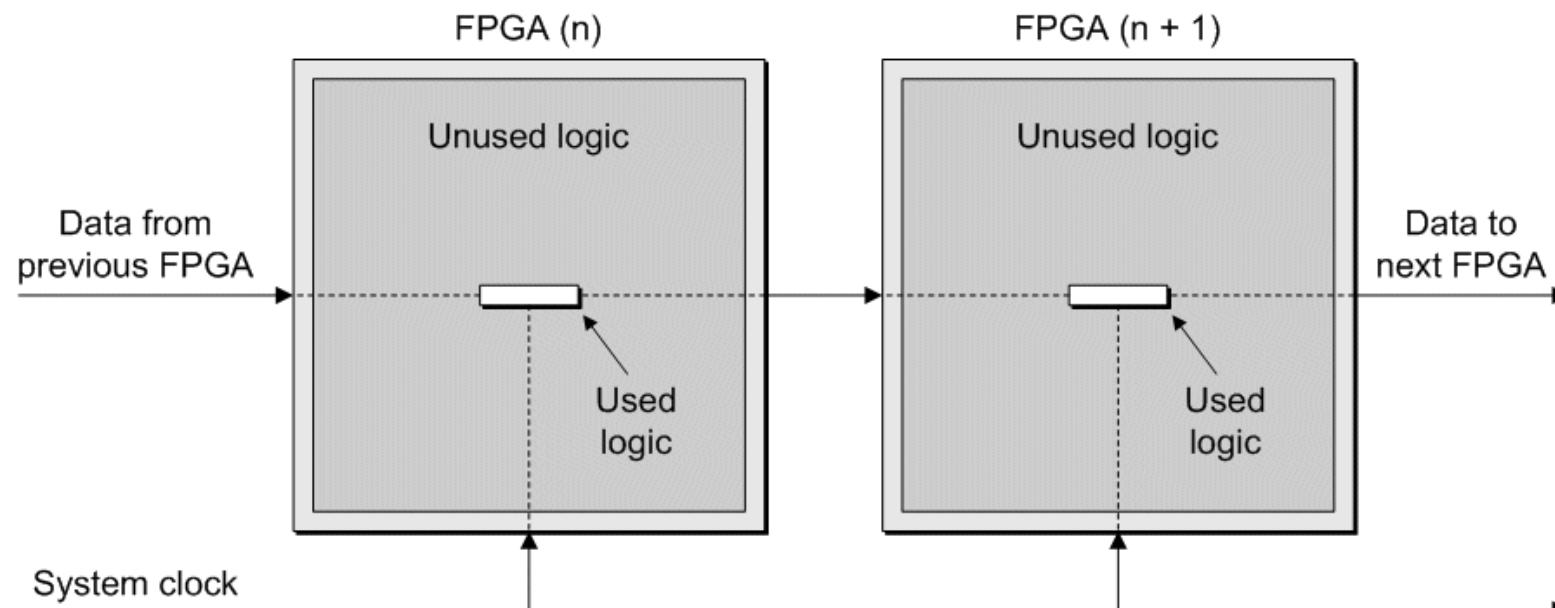
Rozdělení systému na více FPGA





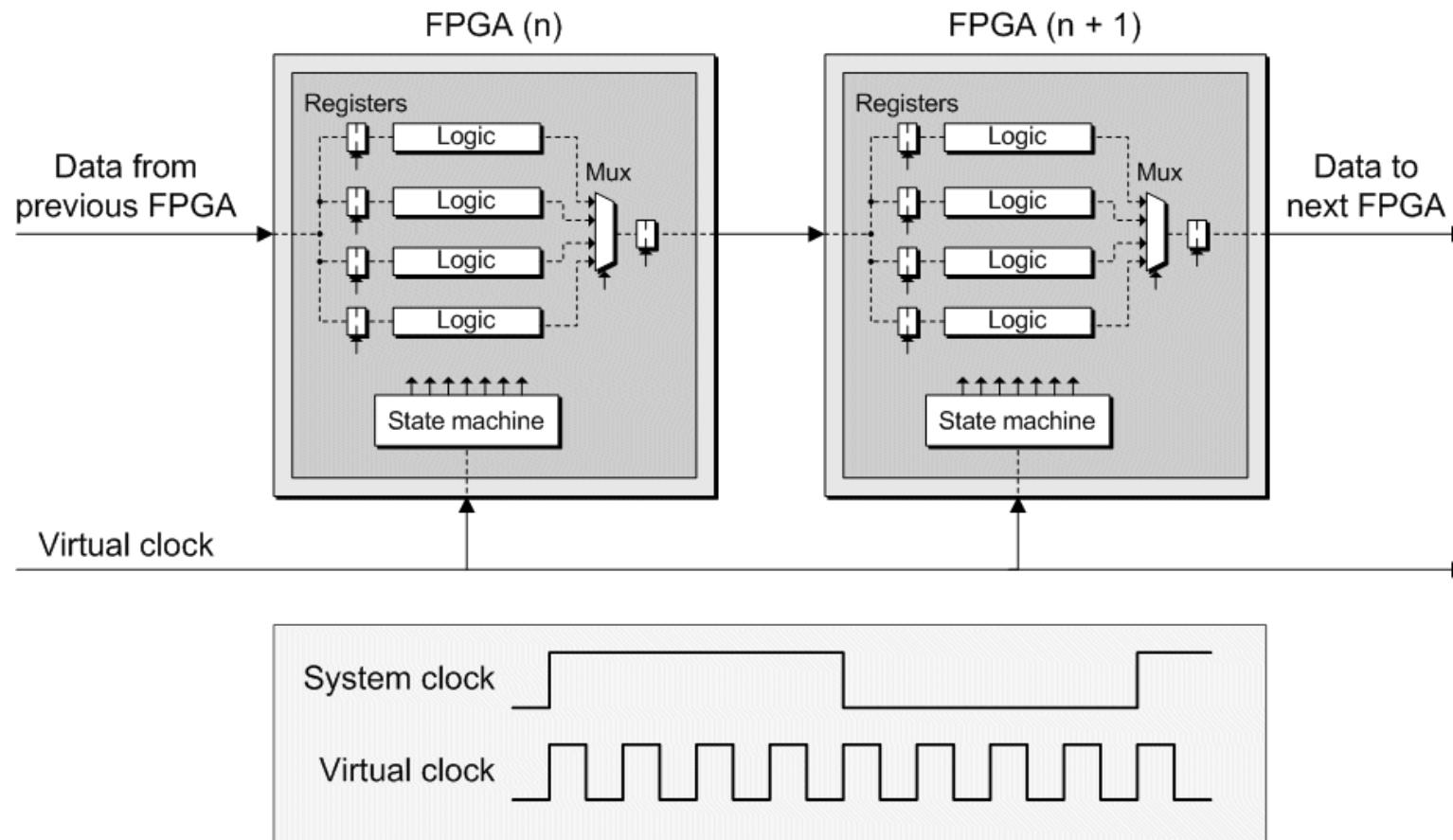
Využitelnost logiky FPGA

Máme-li omezený počet vývodů





Využitelnost logiky FPGA





Protokoly komunikace

Pro zvyšující se potřebu přenosu velkého množství dat se využívají rychlé **sériové diferenční linky** (většinou malý počet vodičů, ale složitější protokol):

- Serial Rapid IO (sRIO)
- PCI Express (v. 1.1 a 2.0),
- Ethernet (0,1 – 1 - 10 Gbps),
- USB (v. 2.0 a 3.x).

Pro HW podporu rychlých I/O portů jsou FPGA vybaveny **tranceivery** – zajišťují přístup k médiu, jsou vybaveny kodéry a dekodéry, synchronizačními obvody, serializéry a deserializéry či obvody pro řízení úrovní signálů.



Budoucí vývoj technologií

Již cca 50 let platí Moorův zákon (počet tranzistorů na čipu se zdvojnásobuje cca 2x za 1,5 roku).

Nejmenší rozměr struktury se zmenšuje s indexem 0,5/3roky,
Plocha čipu se zvětšuje cca 1,5x za 3 roky.

Zpoždění ve spojích je o 1-2 řády větší než na tranzistoru.

Aplikace vrstevnatých 3D struktur (problémy s chlazením).

Aplikace nových PV sloučenin - na bázi gallia (GaN - nitrid gallia, GaAs – arsenid gallia), karbidu křemíku (SiC), hafnia (Hf), aj.

Přechod od mikroelektroniky, přes nanoelektroniku k molekulární elektronice (jiné fyzikální principy, interdisciplinární charakter).



Analogové zakázkové obvody

Milan Kolář

Ústav mechatroniky a technické informatiky



evropský
sociální
fond v ČR



EVROPSKÁ UNIE



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY



Projekt ESF CZ.1.07/2.2.00/28.0050
**Modernizace didaktických metod
a inovace výuky technických předmětů.**

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ



evropský
sociální
fond v ČR



EVROPSKÁ UNIE



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY



OP Vzdělávání
pro konkurenčeschopnost

Návrh hardwarových komponent

Modernizace didaktických metod a inovace výuky technických předmětů



Analogové HDL jazyky

- slouží k modelování a simulaci
- představují potencionální cestu pro analogovou syntézu (zatím není podporována automatická syntéza)
- popisují různé úrovně abstrakce elektrických i neelektrických systémů
- ve srovnání např. s Pspice je popis složitější
- nákladnější simulační programy
- v současnosti nejrozšířenější analogové HDL:
VHDL-AMS, Verilog-AMS, SystemC-AMS



VHDL-AMS

1999 - první standard pro analogové a smíšené obvody:
revize VHDL-AMS 1076.1 (Analogue & Mixed Signals).

Analogový systém lze popsat soustavou diferenciálních algebraických rovnic (DAR)

- DAR obecně nemají analytické řešení;
- numerickými metodami hledáme co nejpřesnější approximace řešení.

VHDL-AMS poskytuje možnost zápisu DAR, ale nespecifikuje způsob jejich řešení (přenecháno konkrétní implementaci simulátoru)
– je specifikován pouze výsledek.



Nové prvky VHDL-AMS

Pro popis systémů pracujících ve spojitém čase

- Modely (entita + architektura)
- Souběžné výrazy (simultaneous) – pro zápis rovnic
- Typy portů (svorek):
 - signál (klasický digitální – in, out, inout)
 - terminal (analogový uzel + energetické domény)
 - quantity (analogová proměnná)
 - generic (obecná konstanta)
- Tolerance
- A/D interakce
- D/A interakce



Množství (quantity)

nová třída objektů pro reprezentaci neznámých v soustavě DAR
(spojitý časový průběh)

Quantity

- je specifikována svým typem a počáteční hodnotou
- může být deklarována v entitě i architektuře (v deklarační části)
- nelze deklarovat uvnitř struktury „package“
- lze použít jako vstupně-výstupní element v seznamu portů

Implicitní quantity:	Q'dot – derivace
	Q'inter – integrál
	Q'delayed(T) – zpoždění
	Q'ltf – Laplaceova transformace
	Q'ztf – Z transformace



Tradiční systémy

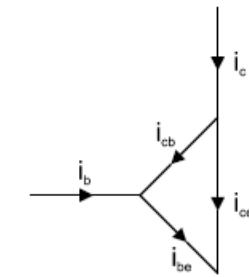
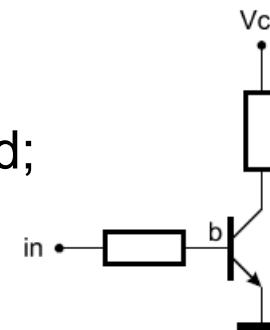
Popis systémů využívajících grafový koncepční model

(popis pomocí Kirchhoffových zákonů)

vrcholy grafu – uzly obvodu;

hrany – větve obvodu, jimiž protéká proud;

dva uzly definují větev (bránu).



Neznámé v rovnicích reprezentují tzv. **branch quantities**

– existují dva druhy:

Across quantities (reprezentují energetické veličiny)

rozdíl potenciálů mezi dvěma vrcholy (U , Θ , p);

Through quantities (představují průtokové veličiny)

odpovídají hranám (proud, tepelný tok, průtok).



Terminály

Nový objekt (např. vývod), který má nějakou přirozenou vlastnost (*nature*) a ke kterému jsou všechny hodnoty dané závislosti vztaženy.

Nature	Across	Through
Electrical	napětí	proud
Thermal	teplota	tepelný tok
Translational	rychlosť	síla
Rotational	úhlová rychlosť	točivý moment
Hydraulic	tlak	objemový průtok



Příklad – quantity, terminály

```
package elektricky_system is
    subtype napeti is REAL ;
    subtype proud is REAL ;
    nature elektro is -- skalární vlastnost elektro
        napeti across
        proud through
        zem reference ;
end package elektricky_system ;
...
terminal t1, t2 : elektro ;
quantity v across i1, i2 through t1 to t2 ;
```



Tolerance

Při řešení soustavy DAR určuje simulátoru, jak velká má být chyba u každé quantity (jak se má blížit nule).

Zavádíme **toleranční skupiny** (*tolerance group*)

– každá quantity i každý výraz náleží do nějaké skupiny.

Není-li tolerance explicitně specifikována, použije se toleranční skupina použité quantity.

```
subtype napeti is REAL tolerance "impl_napeti";
subtype proud is REAL tolerance "impl_proud";
i == iss*(exp(v/vt)-1.0) tolerance "male_napeti";
```



A/D a D/A interakce

Pro A/D a D/A převody

$Q' \text{ABOVE}(E)$... hodnota je TRUE pro $Q > E$ a FALSE pro $Q < E$

Příklad ideálního komparátoru:

```
entity Comparator is
    generic (vthresh : REAL);           -- práh
    port (terminal ain, ref : electrical;
          signal dout : out BOOLEAN);
end entity Comparator;
architecture Ideal of Comparator is
    quantity vin across ain to ref;
begin
    dout <= vin'above(vthresh);
end architecture Ideal;
```



Příklad diody

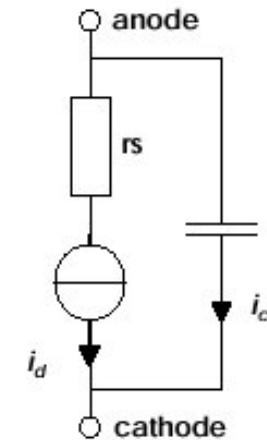
Model diody

$$i_d = i_{ss} \cdot \left(e^{(v_d - rs \cdot i_d) / n \cdot vt} - 1 \right)$$

$$i_c = \frac{d}{dt} \left(t \cdot i_d - 2 \cdot c_j \cdot \sqrt{v_j^2 - v_j \cdot v_d} \right)$$

```
library IEEE, Disciplines;
use Disciplines.electrical_system.all;
use IEEE.math_real.all;
entity Diode is
    generic (iss: REAL := 1.0e-14;
             n, af: REAL := 1.0;
             tt, cj0, vj, rs, kf: REAL := 0.0);
    port (terminal anode, cathode: electrical);
end entity Diode;

architecture Level0 of Diode is
    quantity vd across id, ic through anode to cathode;
    quantity qc: charge;
    constant vt: REAL := 0.0258;      -- thermal voltage
begin
    id == iss * (exp((vd - rs * id) / (n * vt)) - 1.0);
    qc == tt * id - 2.0 * cj0 * sqrt(vj**2 - vj * vd);
    ic == qc'dot;
end architecture Level0;
```





Příklad - pružina

```

use work.types.all;
entity Vibration is
end entity Vibration;

architecture H2 of Vibration is

    quantity x1, x2, xs: displacement;
    quantity energy: REAL;
    constant m1, m2: REAL := 1.00794*1.6605655e-24;
    constant f: REAL := 496183.3;

begin
    x1'dot'dot == -f*(x1 - x2) / m1;
    x2'dot'dot == -f*(x2 - x1) / m2;
    xs == (m1*x1 + m2*x2)/(m1 + m2);
    energy == 0.5*(m1*x1'dot**2 + m2*x2'dot**2 + f*(x1-
    x2)**2);

end architecture H2;

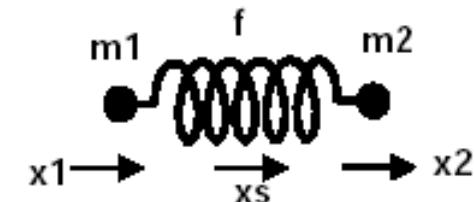
```

$$m_1 \ddot{x}_1 = -f \cdot (x_1 - x_2)$$

$$m_2 \ddot{x}_2 = -f \cdot (x_2 - x_1)$$

$$x_s = (m_1 x_1 + m_2 x_2) / (m_1 + m_2)$$

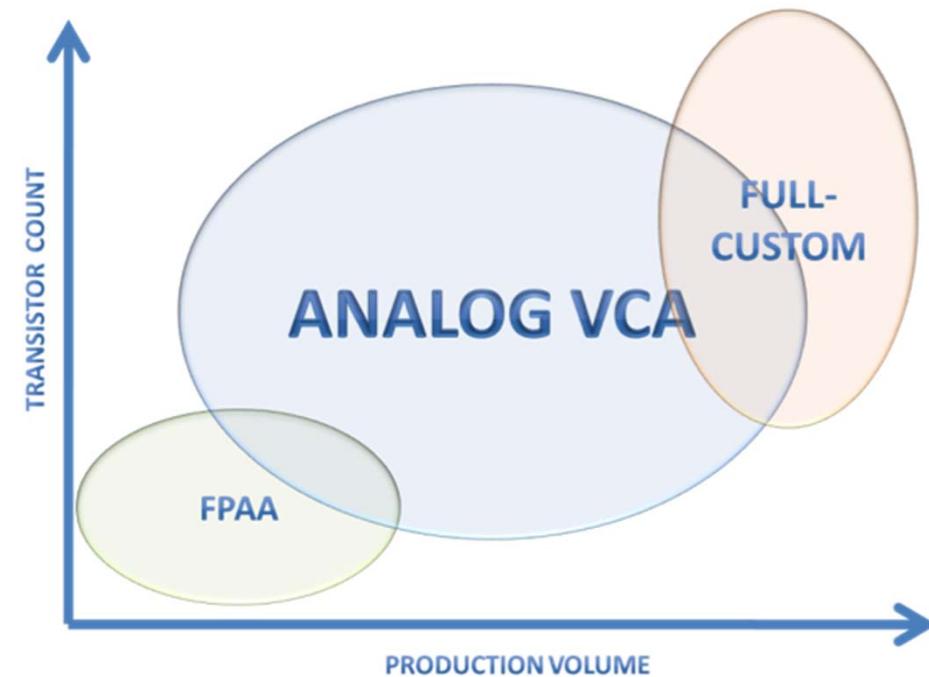
$$Energy = 0.5 \cdot (m_1 \dot{x}_1^2 + m_2 \dot{x}_2^2 + f \cdot (x_1 - x_2)^2)$$





Dělení analogových ASIC

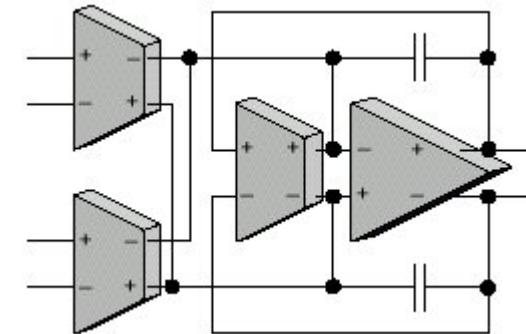
- **Zakázkové (Full-Custom)**
 - parametrizované funkční bloky
 - plně zakázkové IO
- **Via-Configurable Array (polozakázkové)**
 - analogová lineární pole
 - analogová pole obvodů
- **Programovatelné**





Analogové ASIC

- méně rozšířené
- menší hustota integrace
- obtížnější návrh, nutné zkušenosti
- návrh převážně metodou „zdola-nahoru“
- vyšší nároky na přesnost technologie



Specifika analogových technologií

- lepší je tranzistor NPN než PNP
- nelze realizovat induktory
- kapacitory zabírají velkou plochu
- snadná realizace symetrických prvků
- snadnější testování



Prvky analogových IO

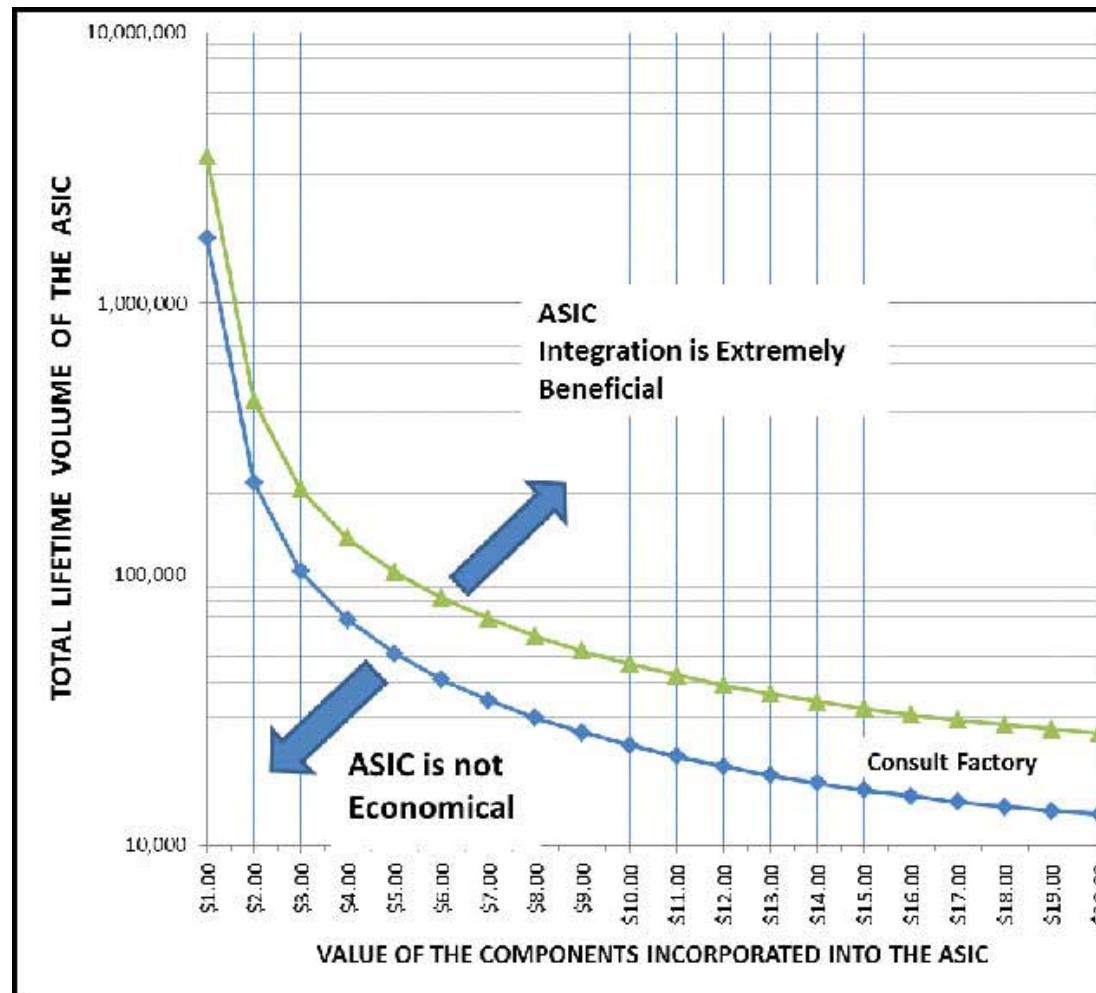
- **rezistory** - meandr izolovaných polovodičových vrstev;
- **kapacitory** - relativně malé hodnoty C;
- **diody** - nejčastěji PN přechod tranzistoru;
- **tranzistory** – NPN, PNP (méně kvalitní);

Funkční bloky:

- zdroje proudu (proudová zrcadla), i jako aktivní zátěž;
- zesilovače (diferenční, napěťové, výkonové);
- napěťové referenční zdroje;
- oddělovací (vazební) obvody.



Výhodnost analogových ASIC



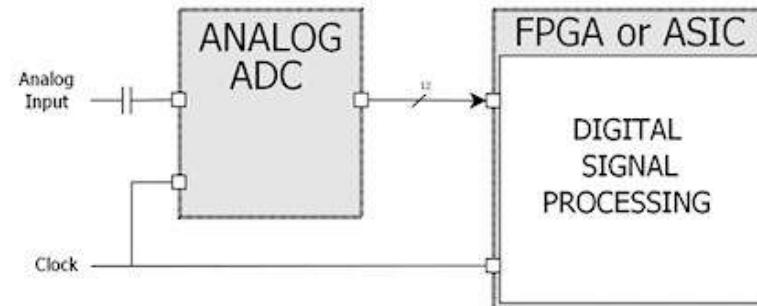


A/D převodníky a FPGA

Implementace A/D převodníku s FPGA

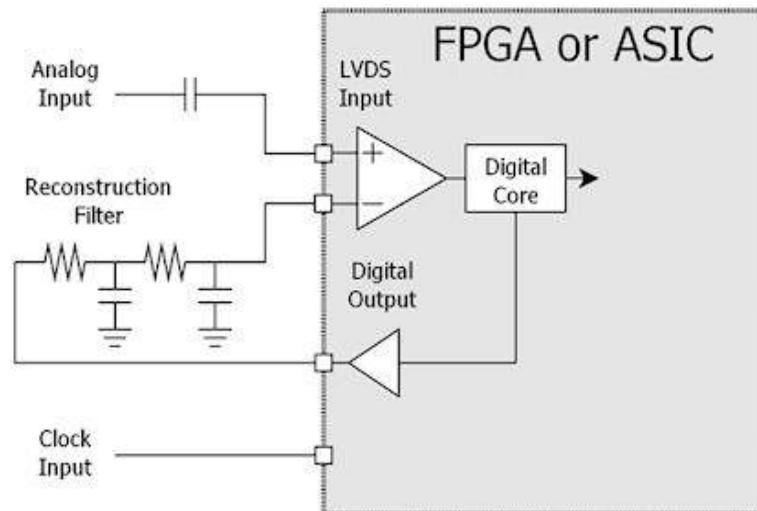
Tradiční řešení s vnějším ADC

- dobré parametry (rychlé, výkonné)
- další IO
- potřeba více pinů



Zaintegrovaný komparátor

- kompaktní řešení
- zabere navíc jen 2 piny
- zvýšení MTBF
- snazší a rychlejší implementace

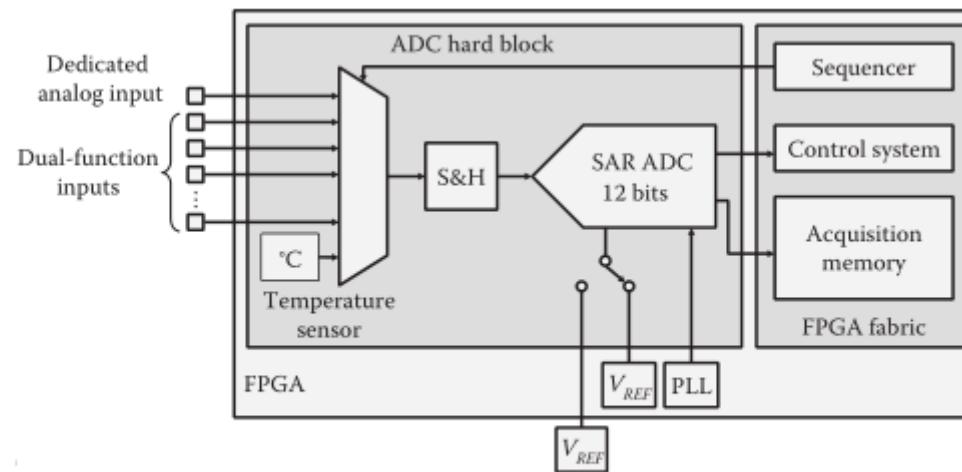




A/D převodník v MAX 10

12bitový A/D převodník v FPGA Intel MAX 10

Vkládá se do návrhu jako IP core (makroblok)



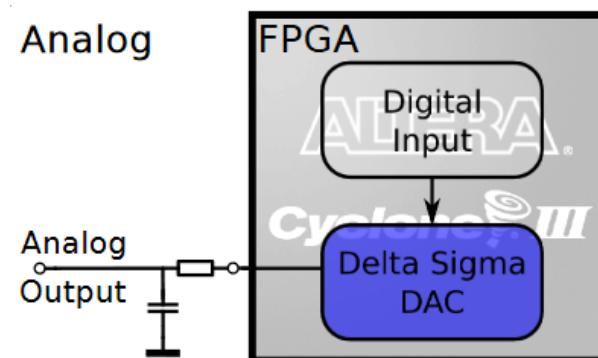


D/A převodníky a FPGA

Implementace není příliš častá.

Řeší se např. nepřímými převodníky – nejčastěji principu sigma-delta modulace

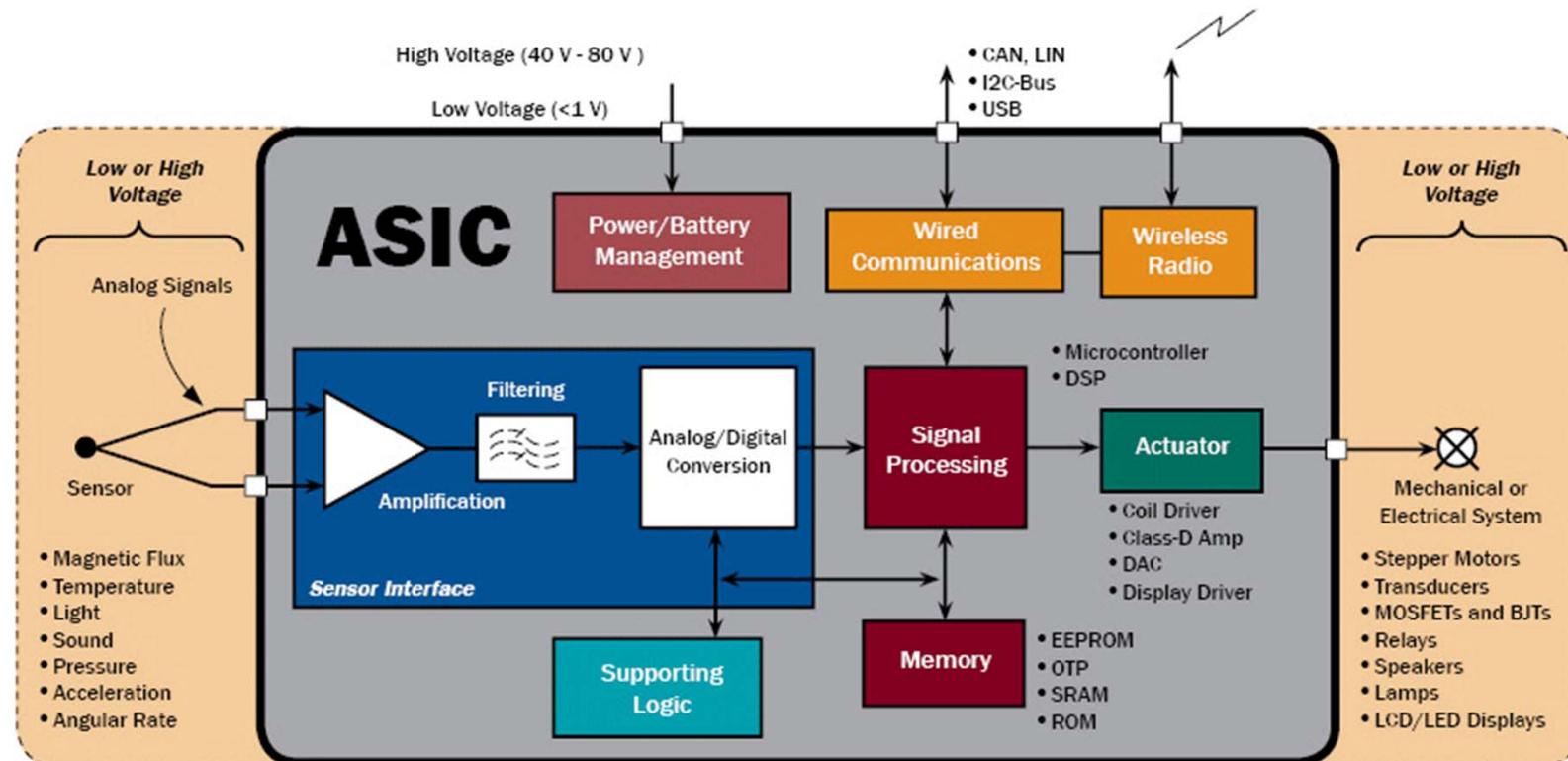
- snadná integrace číslicovými technologiemi (relativně složité zapojení)
- vysoká přesnost a linearita





Typický ASIC – smíšený A/Č

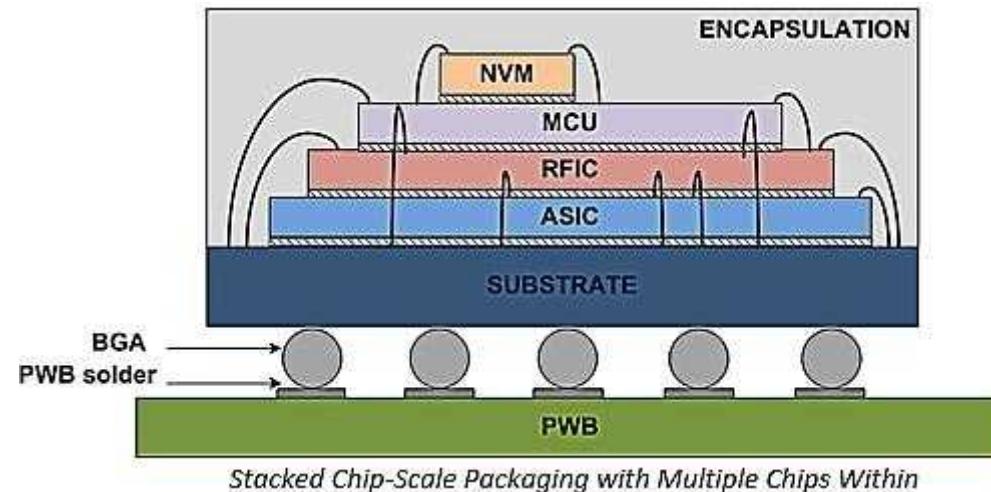
Úprava analogového signálu, digitální zpracování a komunikace





Vícevrstvé čipy

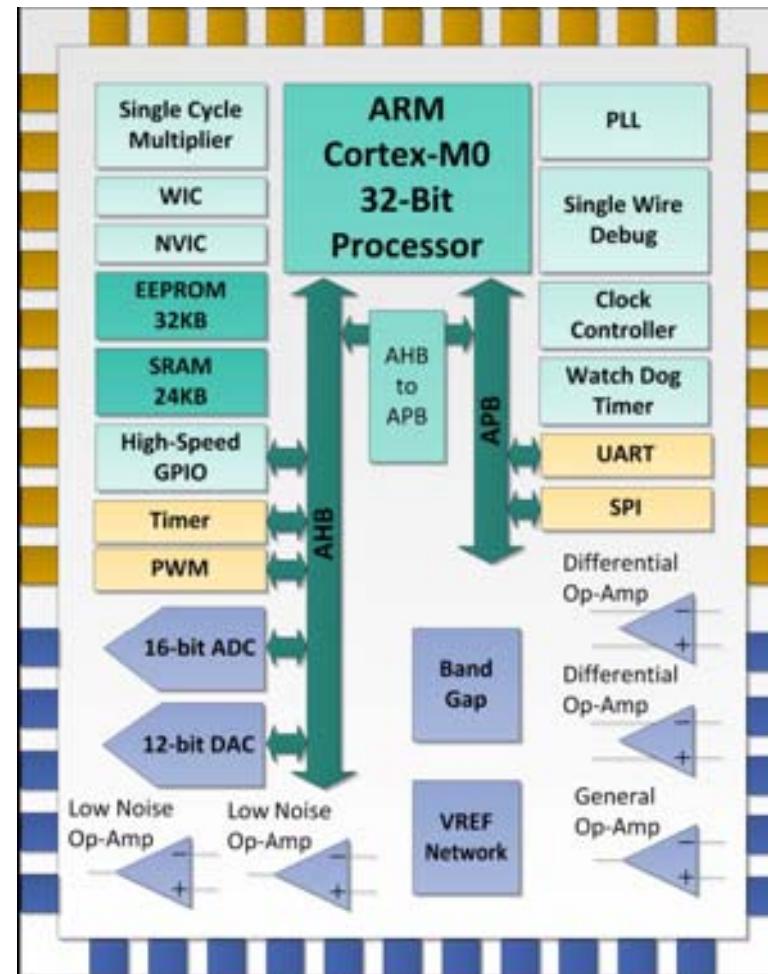
Používání více čipů v pouzdře ve vrstvách
(NVM – NonVolatile Memory, RFIC - Radio Frequency IC,
MCU – MicroController Unit)





Jiná varianta smíšeného A/Č čipu

SoC firmy Triad Semiconductor
(integrovaný 32bitový ARM Cortex)

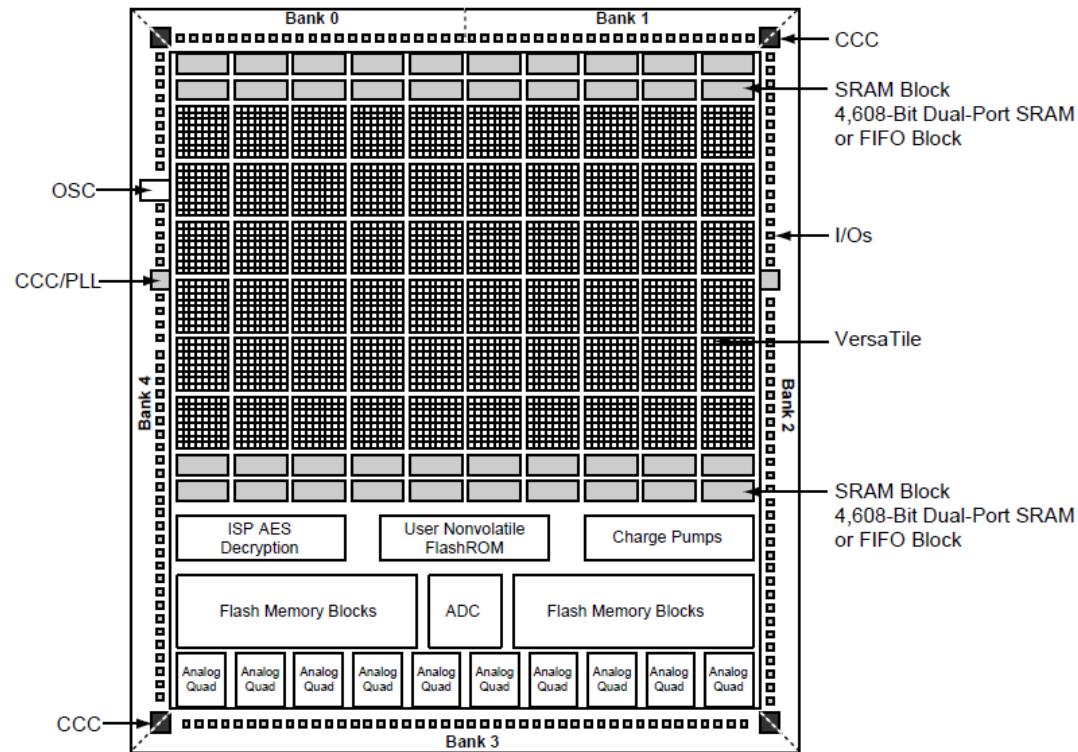




FPGA s analogovými bloky

Řada Fusion (Microsemi):

- ADC (8-/10-/12-bitové s postupnou aproximací),
- DAC (12-bitové sigma-delta),
- komparátory,
- monitory teploty,
- monitory napětí,
- monitory proudu,
- RC oscilátor,
- krystalový oscilátor,
- čítač reálného času.





Programovatelná analogová pole

Rozdílná terminologie (zavedená převážně výrobcí):

- **FPAA – Field Programmable Analog Array**
- FPAD – Field Programmable Analog Device
- FPMA – Field Programmable Mixed Analog Digital Array
- EPAC – Electrically Programmable Analog Circuit
- TRAC – Totally Reconfigurable Analog Circuit

Často kombinace analogově-číslicových obvodů:

- AD a DA převodníky, napěťové reference, oscilátory, PWM obvody, generátory zpoždění, zesilovače, násobičky;
- programovatelná konfigurace bloků nebo propojovací sítě.



evropský
sociální
fond v ČR



EVROPSKÁ UNIE



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY



OP Vzdělávání
pro konkurenční
schopnost

Návrh hardwarových komponent

Modernizace didaktických metod a inovace výuky technických předmětů



Programovatelné analogové funkce

Obvody využívající ke změně parametrů digitální programovatelné potenciometry, ovládání přes I2C, SPI nebo inkrementace, resp. dekrementace.

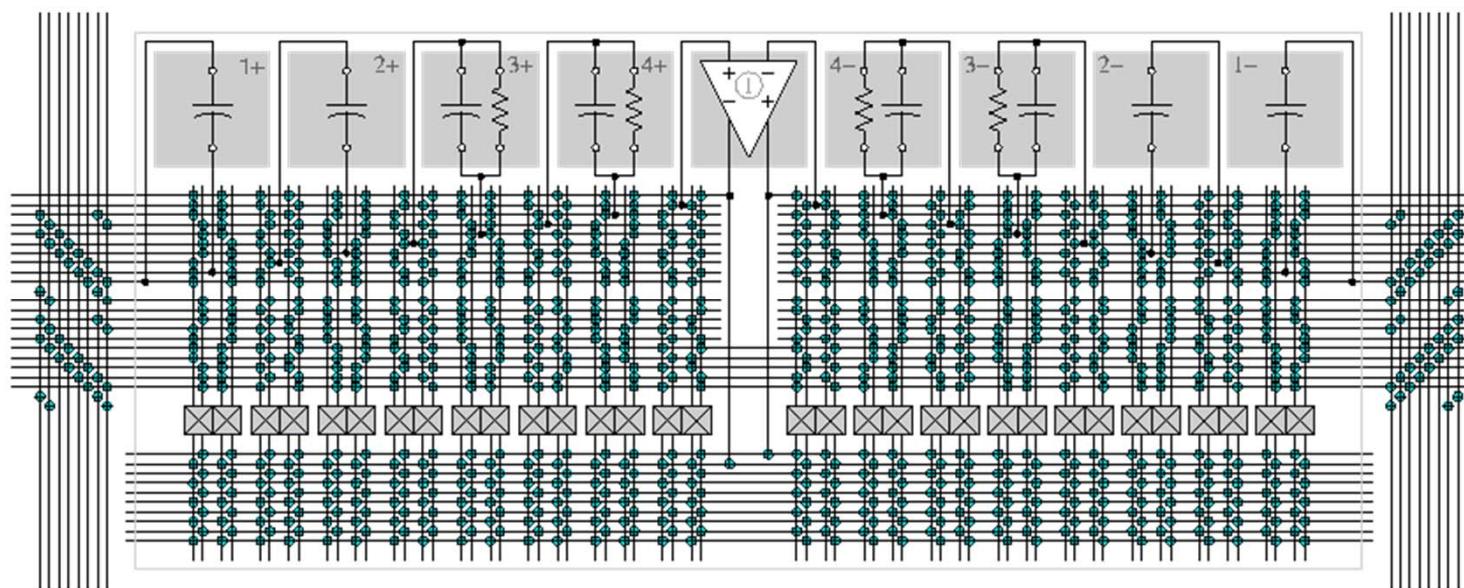
Typické programovatelné obvody:

- napěťové zesilovače,
- přístrojové zesilovače,
- oscilátory (časovače),
- analogové filtry,
- napěťové regulátory,
- zdroje proudu,
- převodníky I/U,
- Schmittovy klopné obvody.



Programovatelná analogová pole

Obecně se skládají z konfigurovatelných analogových bloků
(Configurable Analog Block) – nejčastěji OZ a programovatelná
kapacitní a odporová pole.





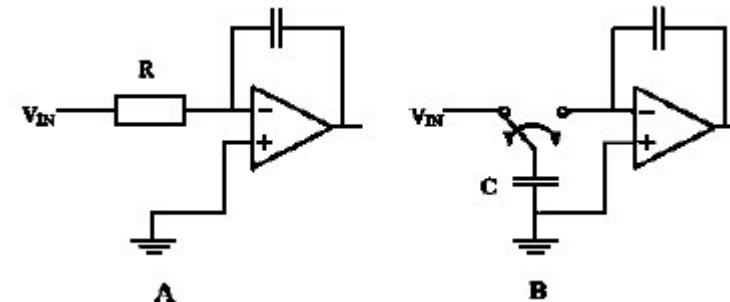
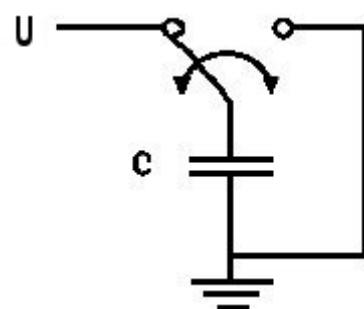
Spínané kondenzátory

Pomocí spínače a kondenzátoru realizujeme odpory

$$R = U / I = U / (\Delta Q / \Delta T) = \Delta T / C = 1 / f \cdot C$$

f ... frekvence přepínání spínače

C ... kapacita kondenzátoru

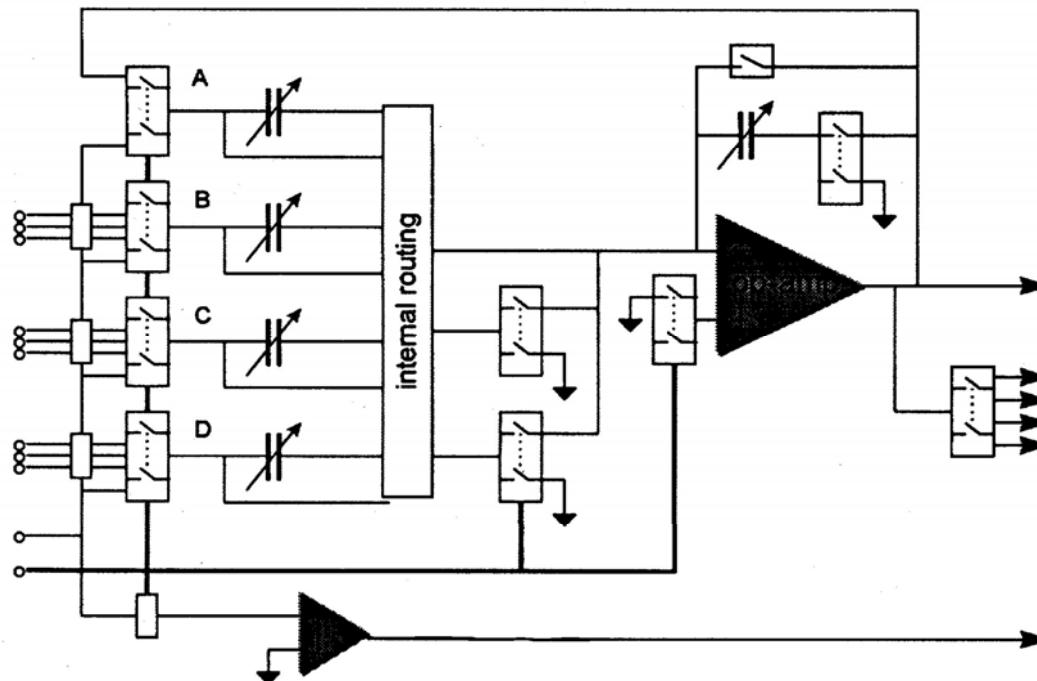




Programovatelná analogová pole

MPAA020 (Motorola) – 20 CAB

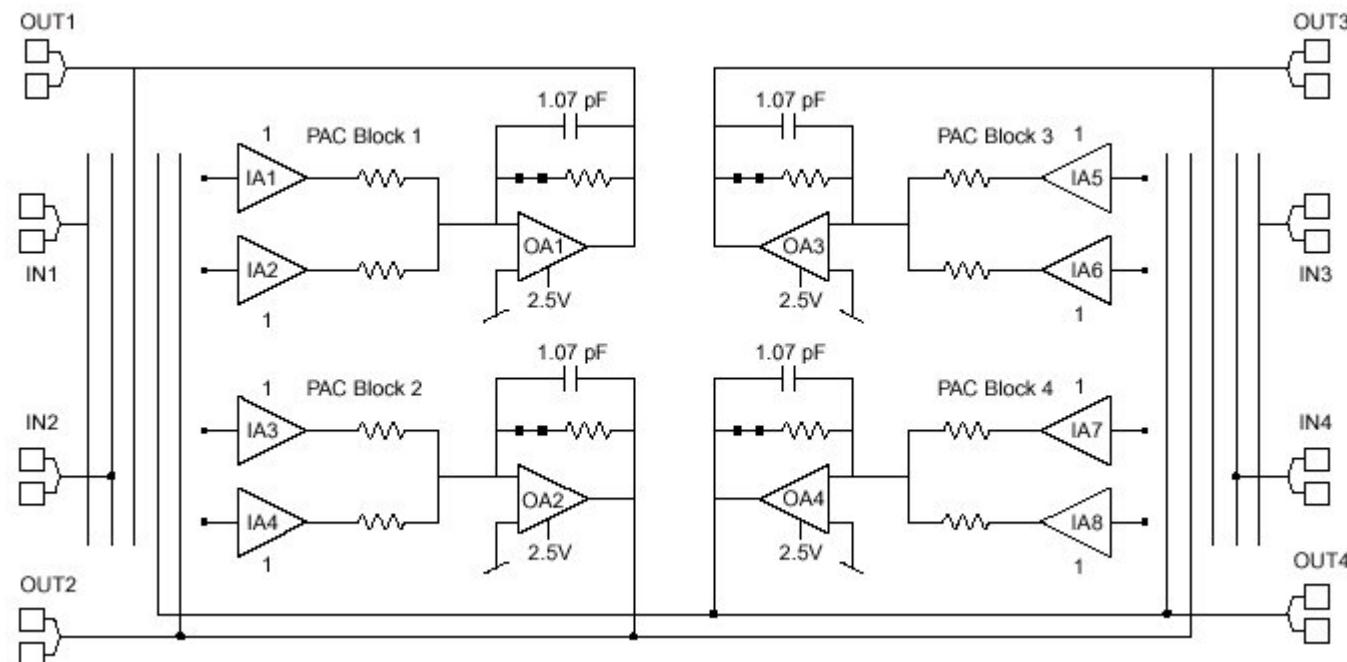
AN220E04 (Anadigm) – 4 CAB (konfigurace v ext. EEPROM)
místo odporů – spínané kondenzátory





Programovatelná analogová pole

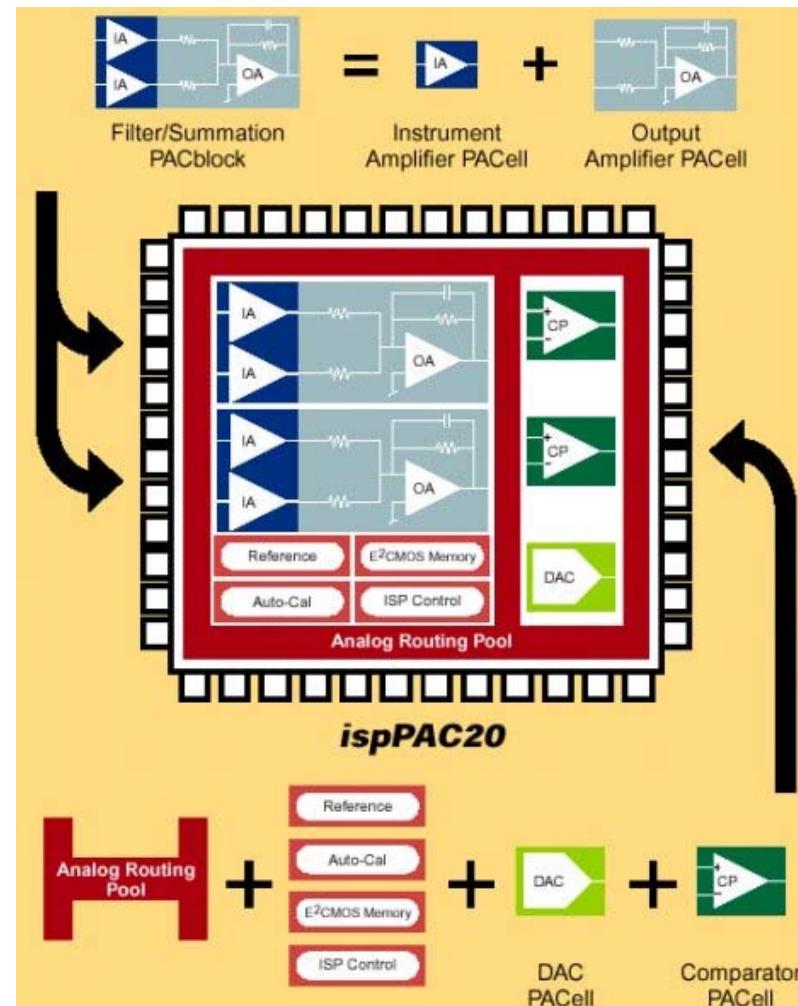
ispPAC10 (Lattice) – programovatelné v systému





Programovatelná analogová pole

ispPAC20 (Lattice)

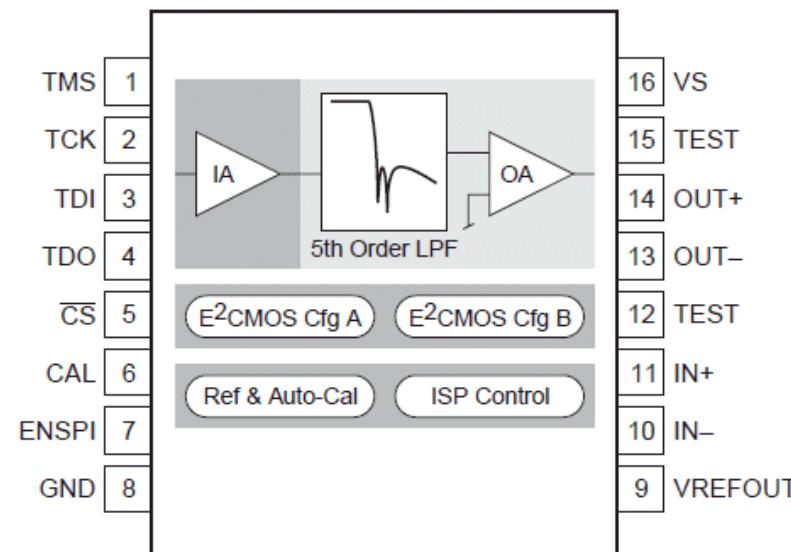




Programovatelný analogový filtr

ispPAC 80 (Lattice)

- duální dolní propust 5. řádu
- mezní frekvence 50–750 kHz
- různé typy approximace útlumové charakteristiky
- programování přes JTAG
- EEPROM technologie

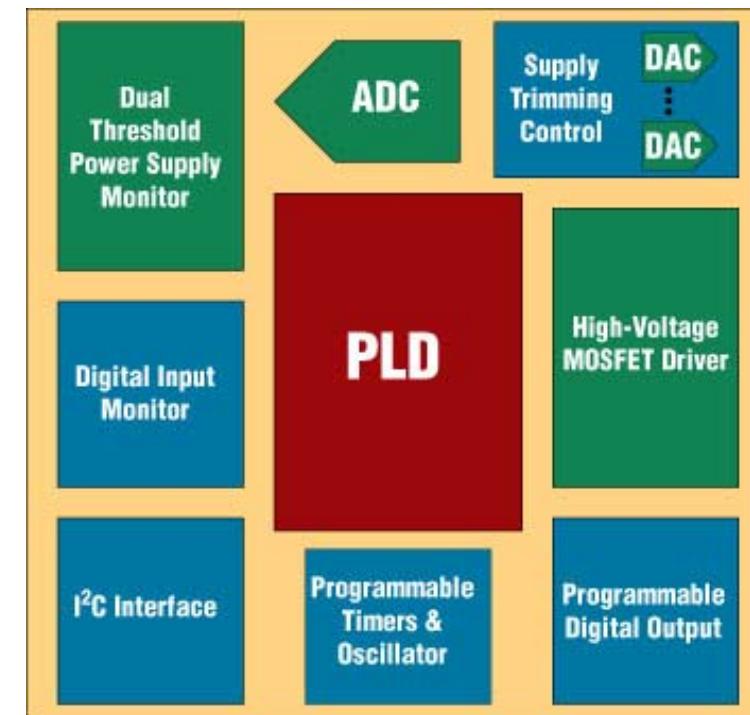




Programovatelný power manager

POWR1220AT8 (Lattice)

- monitoruje až 12 napájecích zdrojů
- 20 digitálních řídicích výstupů
- 8 analogových výstupů
- CPLD (48 makroobrůnek)
- 4 časovače (od 32µs do 2s)
- MOSFET budiče
- I²C rozhraní
- programování přes JTAG

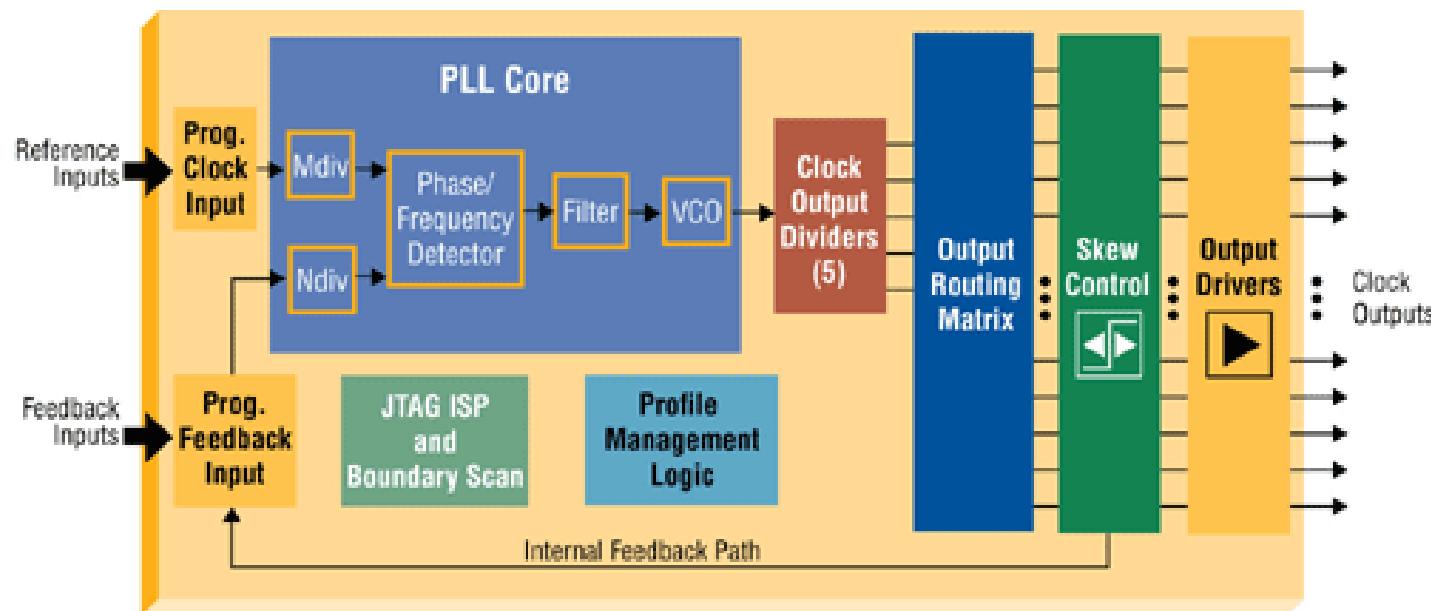




Generátor hodinových signálů

ispClock5600A (Lattice)

- vstupní/výstupní frekvence – 8 až 400 MHz
- až 20 programovatelných výstupů (různé nap. standardy)
- až 5 časových domén
- programování přes JTAG





TECHNICKÁ UNIVERZITA V LIBERCI
**Fakulta mechatroniky, informatiky
a mezioborových studií**



Aritmetické operace v hardwaru

Milan Kolář

Ústav mechatroniky a technické informatiky



evropský
sociální
fond v ČR



EVROPSKÁ UNIE



MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY



Projekt ESF CZ.1.07/2.2.00/28.0050
**Modernizace didaktických metod
a inovace výuky technických předmětů.**

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ



Aritmetické operace

Při číslicovém zpracování signálů se nejčastěji používá součet a součin – pokud používáme k popisu úroveň RTL (nebo vyšší), syntetizační nástroje použijí DSP bloky (nemusí to být vždy optimální).

Algoritmy aritmetických operací se v podstatě liší mírou paralelizace (potřebou logických buněk) a počtem hodinových taktů při N -bitovém datovém slovu, a s tím související spotřebou energie.

Zjednodušeně je možné dělit na sériové, paralelní, zřetězené.

V jazyce VHDL podpora v knihovnách (numeric_std, std_logic_arith, std_logic_signed, std_logic_unsigned).



Zobrazování čísel

Čísla v hardwaru je třeba ukládat do logických obvodů (registrů, pamětí, ...), příp. přenášet po sběrnicích
⇒ užívání dvojkové soustavy (polyadická soustava o základu $z = 2$):

$$A = a_{n-1} \cdot z^{n-1} + \dots + a_0 \cdot z^0 + a_{-1} \cdot z^{-1} + \dots + a_{-m} \cdot z^{-m}$$

Polyadické soustavy zobrazují pouze nezáporná čísla
⇒ k zobrazování záporných čísel používáme transformace (*číselné kódy*); nejčastější:

- přímý se znaménkem
- inverzní
- doplňkový
- aditivní



Řádová mřížka

Rozdělení na celou a zlomkovou část označujeme jako **řádová mřížka**, číslo zapsané v mřížce je *slovo*.

Nelze-li zapsat číslice v nejvyšších řádech, mluvíme o *přeplnění* (přetečení, overflow), v nejnižších řádech o *ztrátě přesnosti*



$n-1$... nejvyšší řád řádové mřížky (celá část má velikost n bitů)

$-m$... nejnižší řád řádové mřížky (zlomková část má velikost m bitů)

N ... délka řádové mřížky (počet obsažených řádů) $N = n + m$

$\varepsilon = 2^{-m}$... jednotka řádové mřížky (nejmenší zobrazitelné číslo)

$M = 2^n$... modul řádové mřížky (nejmenší číslo, které již v řádové mřížce není zobrazitelné)



Formáty nezáporných čísel

Čísla bez znaménka (**unsigned**)

- *Obecný formát:*

$$U = u_{n-1} \cdot 2^{n-1} + \dots + u_0 \cdot 2^0 + u_{-1} \cdot 2^{-1} + \dots + u_{-m} \cdot 2^{-m}$$

Rozsah zobrazitelných čísel: $0 \leq U \leq (2^N - 1) \cdot 2^{-m}$

- *Celočíselný formát* (tj. $m = 0$, $N = n$):

$$U = u_{n-1} \cdot 2^{n-1} + \dots + u_0 \cdot 2^0$$

- *Zlomkový (fraction) formát* – řádová čárka se umísťuje těsně za nejvyšší bit, který je řádu 2^0 ($n = 1$, $N = m+1$)

$$U = u_0 \cdot 2^0 + u_{-1} \cdot 2^{-1} + \dots + u_{-m} \cdot 2^{-m}$$



Přímý kód se znaménkem

také „přirozený kód“

absolutní hodnota čísla se znaménkovým bitem; $0 \sim (+)$, $1 \sim (-)$

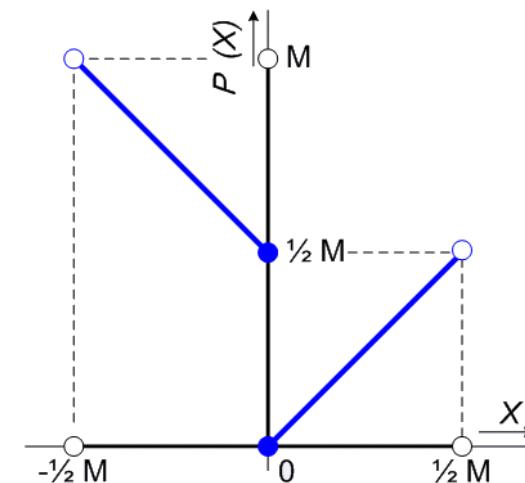
$$P(X) = X \quad \text{pro } X \geq 0$$

$$P(X) = |X| + \frac{1}{2}M \quad \text{pro } X \leq 0$$

- složitá realizace aritmetických operací
nejprve třeba otestovat znaménko
pak se použije algoritmus operace
(sčítání, odečítání)

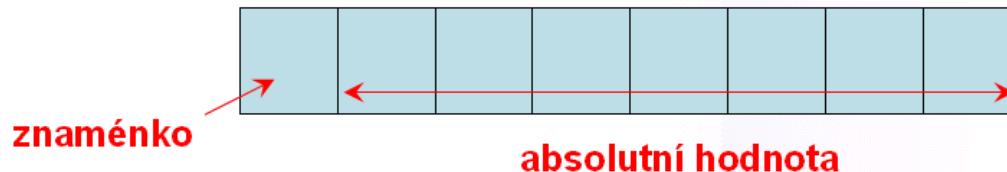
- nevýhodou jsou dvě reprezentace nuly
(nutno ošetřit)

$$-\frac{1}{2}M \leq X < \frac{1}{2}M$$





Přímý kód se znaménkem

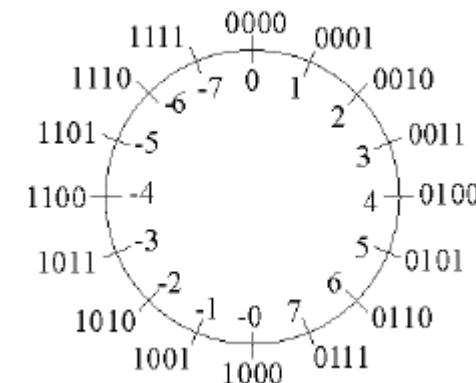


Např. pro 4-bitové slovo (n-bitové)

kladná čísla: 0 ... 7 ($2^{n-1}-1$)

záporná čísla: -7 ... 0

dvě nuly: 0000 a 1000





Inverzní kód

Vychází z jednotkového doplňku (one's complement)

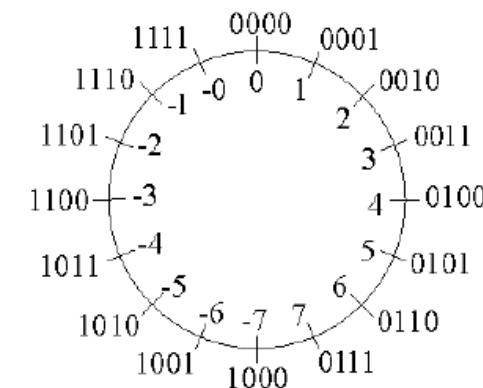
$$I(X) = X \text{ pro } X \geq 0$$

$$I(X) = M - \varepsilon + X \text{ pro } X \leq 0$$

- opět problém dvou nul (0000 a 1111) $-\frac{1}{2}M \leq X < \frac{1}{2}M$
- obtížnější realizace aritmetických operací
- vznikne negací bitů daného slova
- MSB bit má opět charakter znaménka

Např. $+0,8125 \sim 0,1101$

$-0,8125 \sim 1,0010$





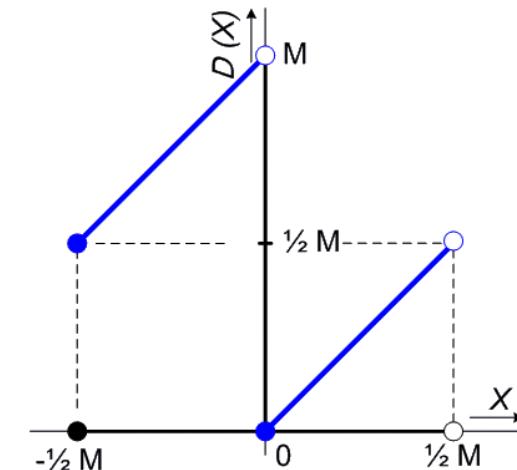
Doplňkový kód

Vychází z dvojkového doplňku“ (two's complement)

$$D(X) = X \text{ pro } X \geq 0$$

$$D(X) = M + X \text{ pro } X < 0$$

- nejvyšší bit má opět charakter znaménka
(nenese informaci o hodnotě)
- $-\frac{1}{2}M \leq X < \frac{1}{2}M$
- vznikne přičtením ε k inverznímu kódu
- max. záp. číslo nemá kladný ekvivalent
- algoritmus odečítání je stejný jako sčítání
(sečtou se obrazy a ignoruje se přenos)
- nejpoužívanější





Kódování čísel se znaménkem

Doplňkový kód – chápeme jako **signed** (se znaménkem):

$$D(X) = S = -s_{n-1} \cdot 2^{n-1} + \sum_{i=-m}^{n-2} s_i \cdot 2^i \quad \begin{array}{l} \text{platí pro celý rozsah } X \\ -\frac{1}{2}M \leq X < \frac{1}{2}M \end{array}$$

- Celočíselný formát (tj. $m = 0$, $N = n$):

$$S = -s_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} s_i \cdot 2^i$$

- Zlomkový (*fraction*) formát (tj. $n = 1$, $N = m+1$)

$$S = -s_0 + \sum_{i=-m}^{-1} s_i \cdot 2^i$$

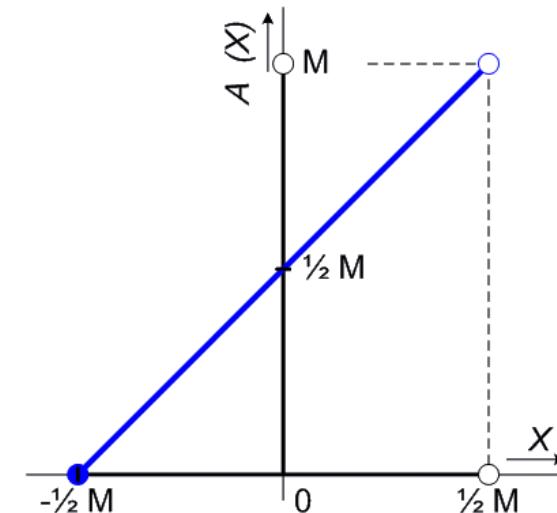


Aditivní (posunutý) kód

Kód s posunutou nulou – k číslu připočteme známou konstantu K :

$$A(X) = X + K$$

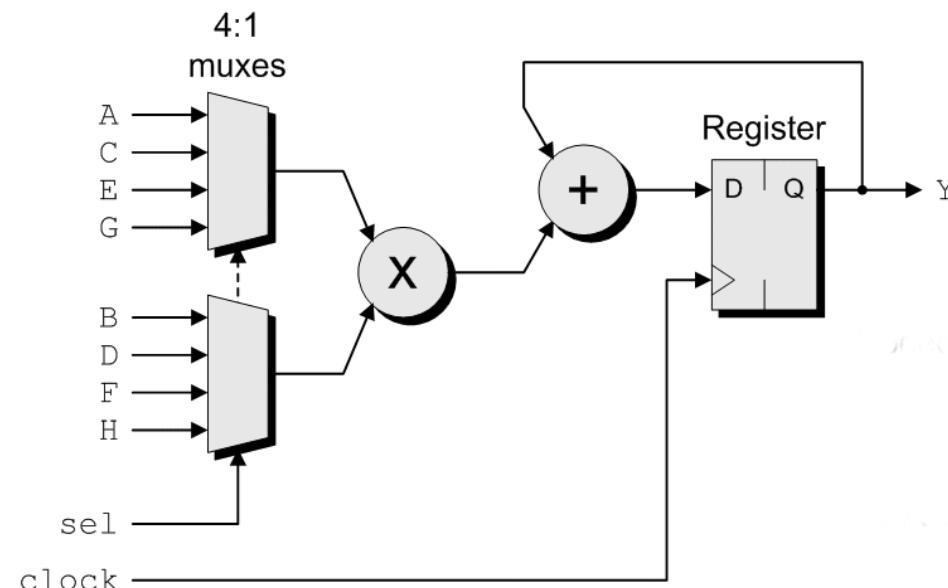
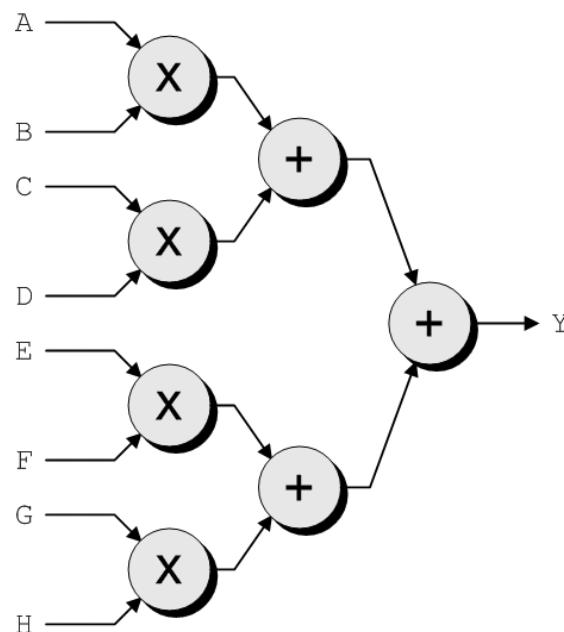
- zachovává relace <
- především se používají 2 varianty:
 $\Rightarrow K = 2^{n-1}$, tj. $\frac{1}{2}M$ (sudý aditivní kód)
 $K = 2^{n-1} - \epsilon$ (lichý aditivní kód)
- složitější realizace násobení (nejprve je třeba odečíst známou konstantu)
- převod do doplňkového kódu provedeme negací nejvyššího bitu (pro sudý aditivní kód)
- používá se pro reprezentaci exponentu reálných čísel





DSP operace

Kompromis mezi rychlostí a plochou;



někdy preferujeme rychlosť i před přesností.



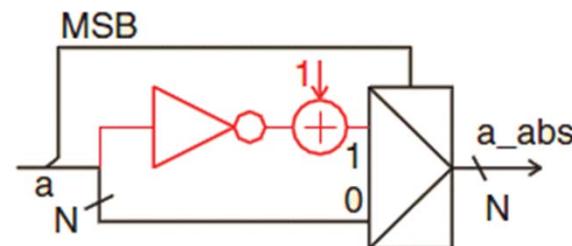
Absolutní hodnota

$$D(-a) = M + (-a) = 2^n - a - \varepsilon + \varepsilon = NOT(a) + \varepsilon$$

Chceme-li invertovat znaménko, invertujeme všechny bity a přičteme jedničku (u celých čísel).

ve VHDL: `a_inv <= -a;` nebo `a_inv <= not(a) + 1;`

`a_abs <= unsigned(a) WHEN a(N-1) = '0' ELSE unsigned(NOT(a))+1;`





Sčítacíky

Sčítacíky (a odečítacíky) – poloviční (half), plné (full)

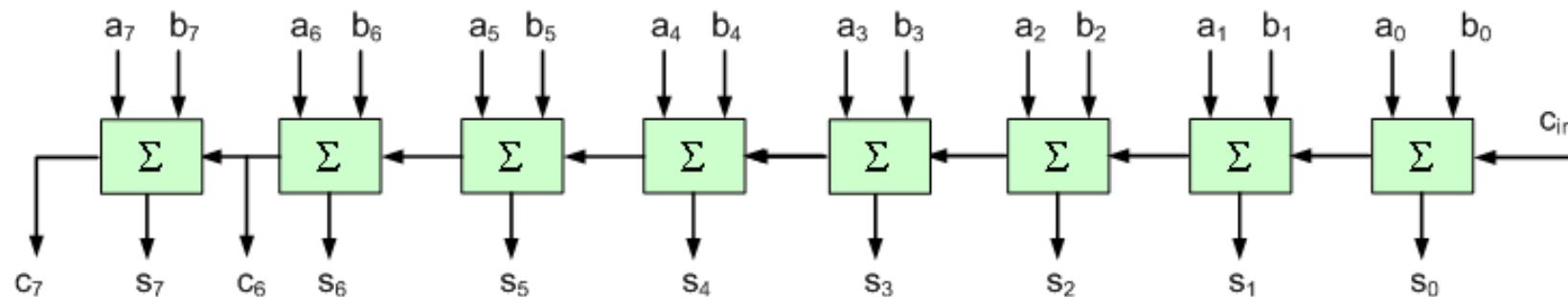
Nejjednodušší je sériová sčítadla – úplná sčítadla s klopným obvodem pro uchování přenosu – potřeba N hodinových taktů.

Pokud ve VHDL zapíšeme: $c \leq a + b$;

vznikne nejčastěji *paralelní sčítadla se sériovým přenosem*

(ripple carry adder) – při velkém N raději „roztrhat“

=> sériově-paralelní sčítadla



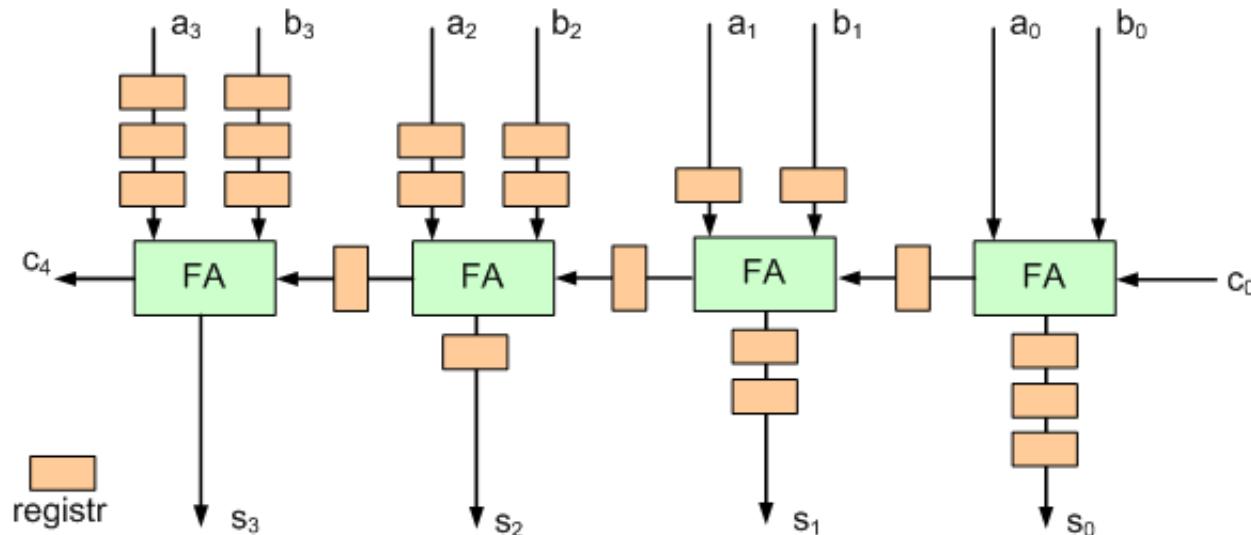


Sčítačky (pokračování)

Jazyky HDL neřeší přetečení – lépe rozšíření na $N+1$ bitová čísla

$$c \leq (a(N-1) \& a) + (b(N-1) \& b);$$

Sčítačka s proudovým zpracováním – výsledek získáme za N hodinových taktů (a dále každý takt);
 FA možné nahradit i vícebitovou sčítačkou



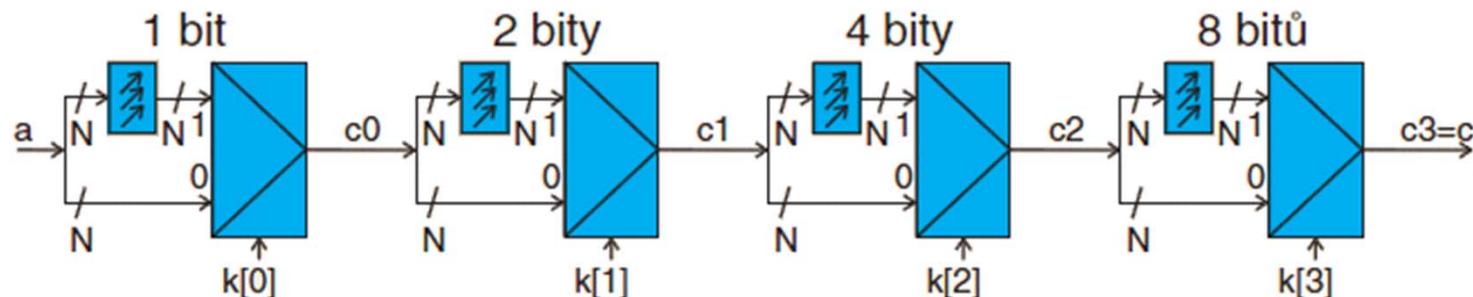


Násobičky

Nejjednodušší varianta – násobení celočíselnou mocninou dvou

=> provádí se pouze posuv

Barrel shifter pro posuv čísla (násobení $2^0, 2^1, \dots, 2^{15}$).



Při násobení dvou N -bitových čísel má výsledek $2N$ bitů.

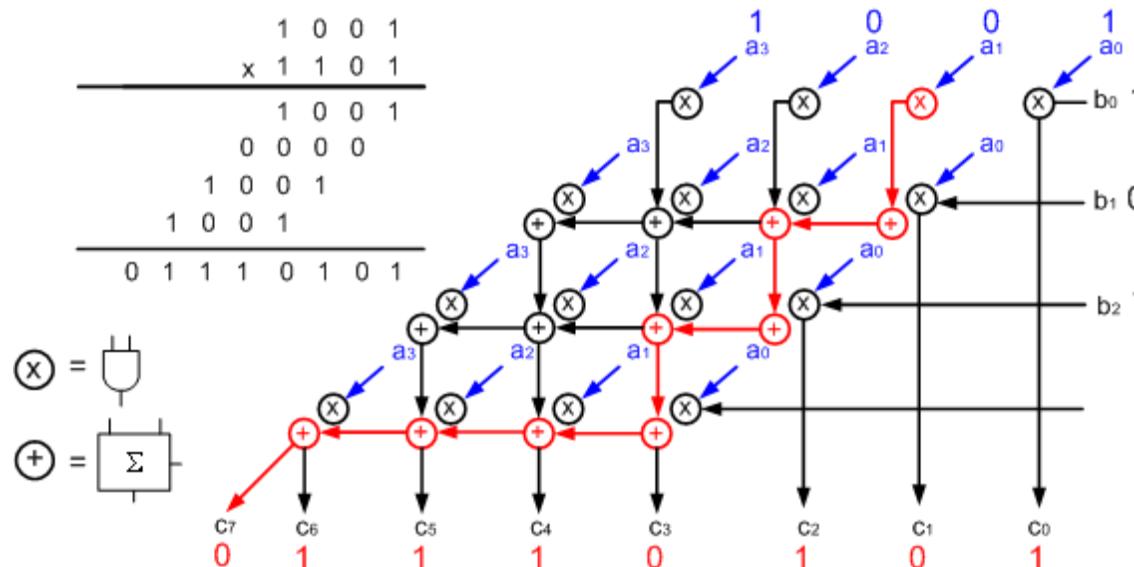
Jednoduché řešení je postupné sčítání (velmi pomalé).



Násobičky (pokračování)

Paralelní násobička – vychází z písemného algoritmu,

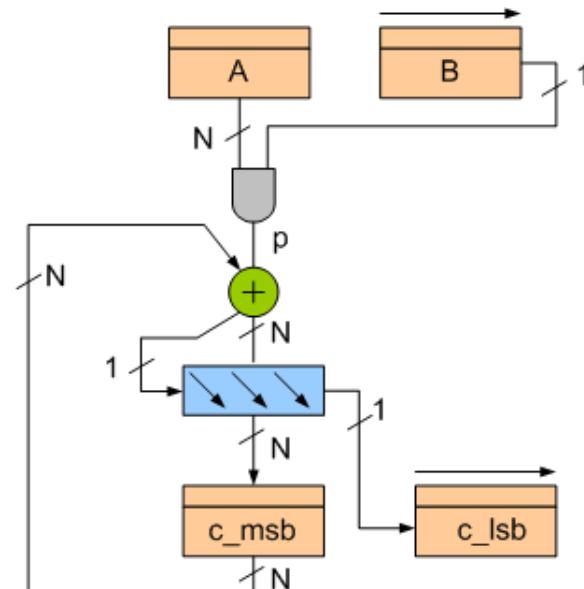
- kombinační obvod z polosčítaček a log. součinů,
- rychlé, ale zabírá množství logiky,
- tato násobička vzniká při zápisu ve VHDL: $c \leq a * b$;
- pokud jsou v FPGA bloky DSP, preferují se (pro větší N).





Násobičky (pokračování)

Sériová násobička – jednoduchá, ale relativně pomalé (N taktů), vychází opět z písemného algoritmu.



Pro slova s velkým N používáme sériově-paralelní varianty nebo násobičky s proudovým zpracováním.



Děličky

Dělení je relativně pomalá a HW náročná operace.

Speciální případy:

Dělení celočíselnou mocninou dvou => posuv;

- zleva nasouváme nuly (unsigned) nebo kopii znaménka.

Potřebujeme-li dělit známou *konstantou*, převedeme na násobení její převrácenou hodnotou.

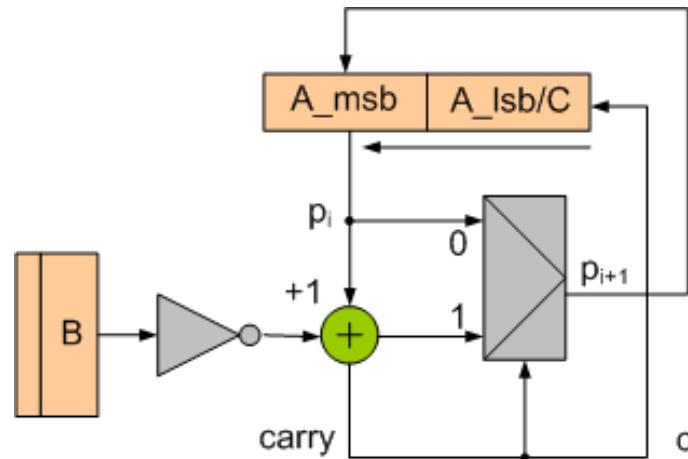
Nejjednodušší obecné řešení je *postupné odečítání* – pomalé.

Jinou obecnou možností je převod na *násobení převrácenou hodnotou* – problém je získání převrácené hodnoty.



Děličky (pokračování)

Sériová dělička s nezápornými operandy – vychází z algoritmu písemného dělení



Matematické knihovny ve VHDL neposkytují podporu dělení.



Druhá odmocnina

Často používaná operace, většinou počítáme celočíselně;
má-li odmocněnc 2 N bitů, výsledek má N bitů.

Řada algoritmů – nejjednodušší, ale časově náročné:

Postupné hledání: for $y = 1$ to A

```
    if  $A - y^*y \leq 0$  then return( $y-1$ );  
    end;
```

Metoda bisekce (půlení intervalu) – počáteční odhad výsledku,
spočítáme mocninu, porovnáme, zvolíme spodní nebo horní
polovinu původního intervalu.

Rekurentní algoritmy:

$$Y_0 = X, \quad Y_{i+1} = 0,5 \cdot (X/Y_i + Y_i), \quad \text{kde } X \text{ je vstupní hodnota}$$



Goniometrické funkce

Použití v algoritmech čísl. zpracování signálu, v generátorech průběhů, v počítačové grafice aj.

Záleží, zda potřebujeme generovat spojitý průběh nebo libovolnou hodnotu (bez předešlé historie).

- Použití tabulky (paměť ROM) – stačí 1 kvadrant, náročné na plochu, rozlišení lze zlepšit lineární interpolací;
- Použití mocninných řad (Taylorův rozvoj aj.);
- Metoda CORDIC – iterační metoda založená na rotaci vektoru okolo počátku o předem známé úhly.



Logaritmus, exponenciální funkce

Méně časté, občas logaritmické transformace operandů před výpočtem
=> násobení a dělení převádí na sčítání a odečítání, obecnou
mocninu a odmocninu na násobení a dělení.

Algoritmy: na principu approximace,
rekurentní (konvergentní řady).