Simulace a logická syntéza

Jan Kořenek

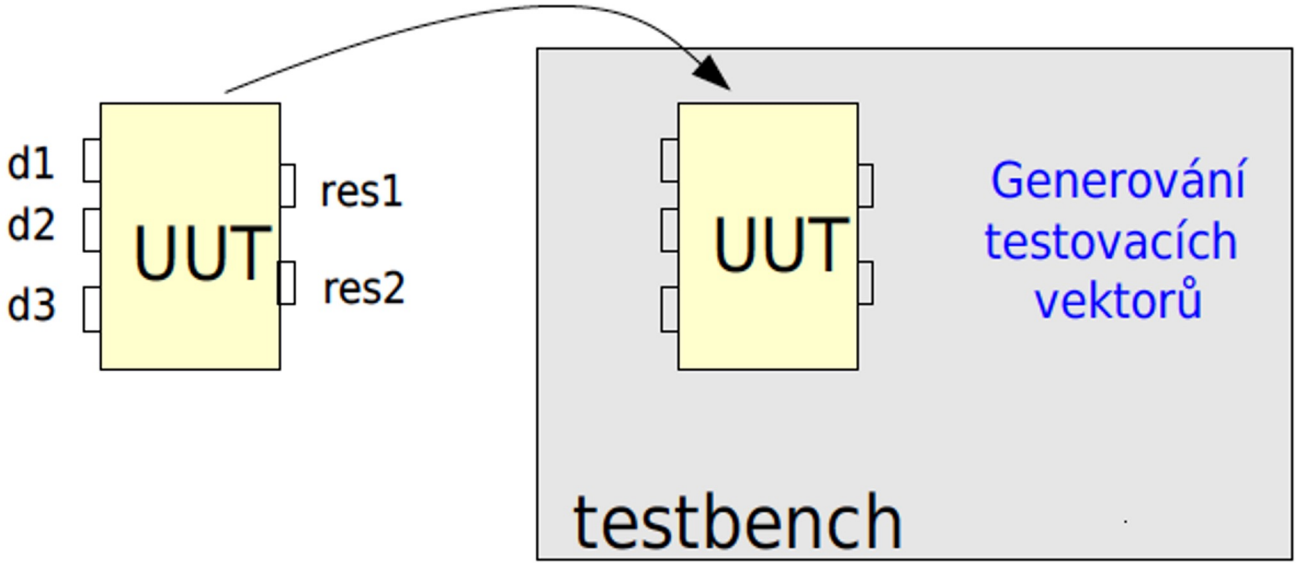
Brno University of Technology, Faculty of Information Technology
Božetěchova 1/2, 612 66 Brno - Královo Pole
korenek@fit.vutbr.cz

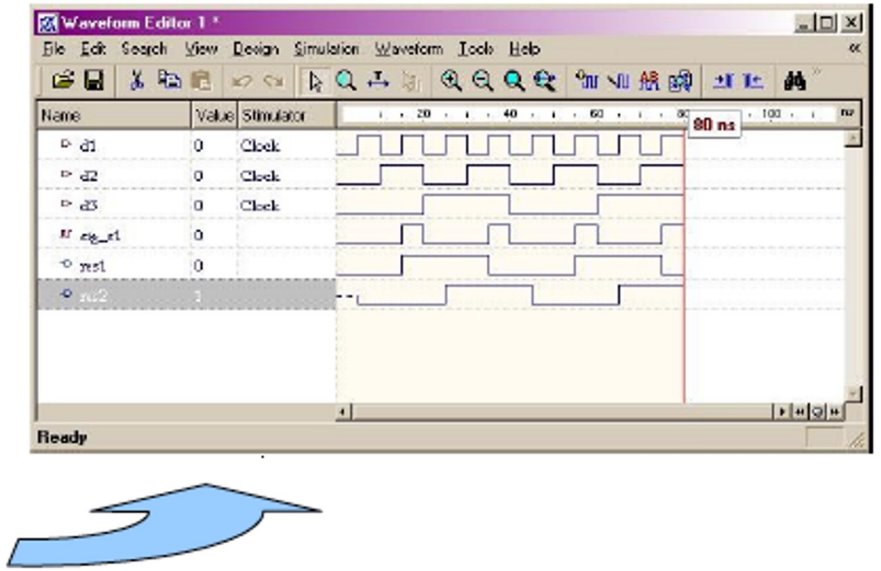
Cíl přednášky

- Pochopit práci se simulátorem obvodů GHDL
- Pochopit princip logické syntézy
  - Co je vstupem?
  - Jaké fáze má logická syntéza?
- Definovat si kritickou cestu v obvodu a její vliv na periodu hodinového signálu.
- Naučit se základní techniky zrychlení obvodů pomocí
  - zřetězeného zpracování (***Pipelining***) a
  - vyvažování zřetězené linky (***Retiming***).
- Jak popisovat číslicové obvody ve VHDL s ohledem na logickou syntézu do ASIC/FPGA.

- HDL popis obvodu je možné modelovat a simulovat
- Simulací myslíme analýzu chování obvodu (jeho vnitřních prvků a výstupů) v závislosti na:
  - vstupních hodnotách (kombinační obvody), nebo
  - počátečním stavu a posloupnosti vstupních hodnot (sekvenční obvody)
- Simulace obvodů:
  - probíhá ve virtuálním prostředí (Testbench) a
  - využívá model obvodu (HDL popis)
- Pro simulaci není třeba realizace obvodu ve fyzickém hardware
  - => významná úspora času

- Testování obvodu implementovaného v HDL v prostředí implementovaném v HDL





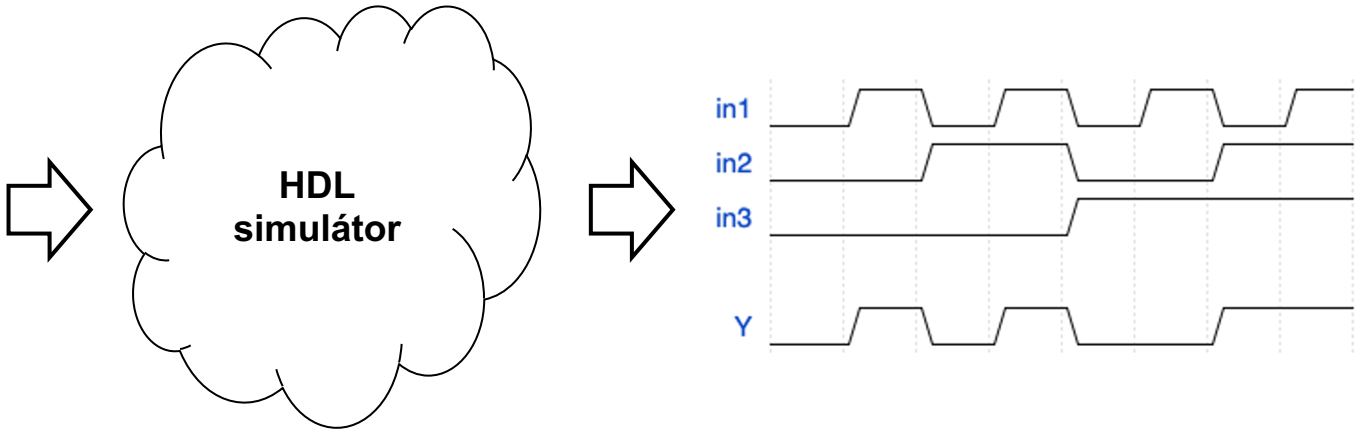
- Testbench obvykle obsahuje
  - Instanci vyvíjené komponenty označenou jako UUT (Unit Under Test)
  - Generátor testovacích vektorů
  - Monitorování a ověřování reakcí UUT

- Nástroj pro automatizovanou analýzu chování obvodu popsaného v HDL

```
entity testbench is
end entity testbench;

architecture arch of testbench is
   signal in1, in2, in3, y: std_logic;
begin
   UUT : entity work.mux
   port map(
      ...
   );

   test : process
   begin
      ...
     wait;
    end process;
end architecture;
```

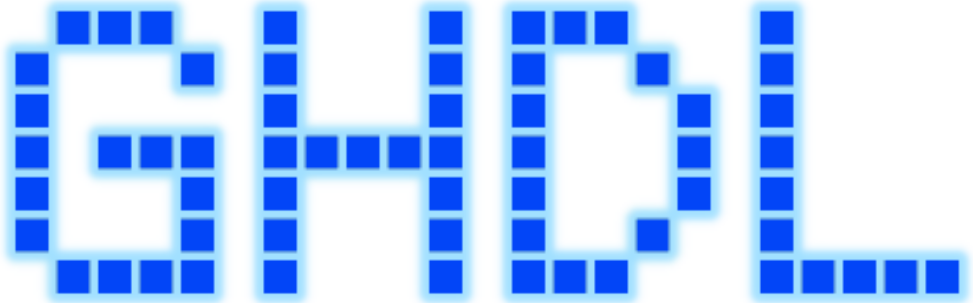


Příklady HDL simulátorů

- Proprietární simulátory
  - ModelSim/Questa (Mentor Graphics)
    - Nejrozšířenější HDL simulátor v rámci vývoje pro FPGA
  - VCS (Synopsis)
  - NCSim (Cadence)
  - ISim, XSim (AMD/Xilinx)
    - Integrované v nástrojích od AMD/Xilinx
- Volně dostupné simulátory pro Verilog
  - Verilator
- Volně dostupné simulátory pro VHDL
  - GHDL

https://en.wikipedia.org/wiki/List_of_HDL_simulators

- GHDL = GNU Hardware Design Language
- Volně dostupný open-source analyzátor, překladač a simulátor jazyka VHDL
  - Plná podpora VHDL 1987, 1993, 2002
  - Částečná podpora VHDL 2008 a 2019
- Implementuje přímý překlad VHDL do strojového kódu
  - nevyužívá se překlad přes jiný jazyk => rychlejší analýza, kompilace, simulace
- Neobsahuje prohlížeč časových diagramů
- Experimentální podpora syntézy VHDL
- Odkazy
  - https://github.com/ghdl/ghdl (zdrojový kód)
  - https://ghdl.github.io/ghdl/ (dokumentace)

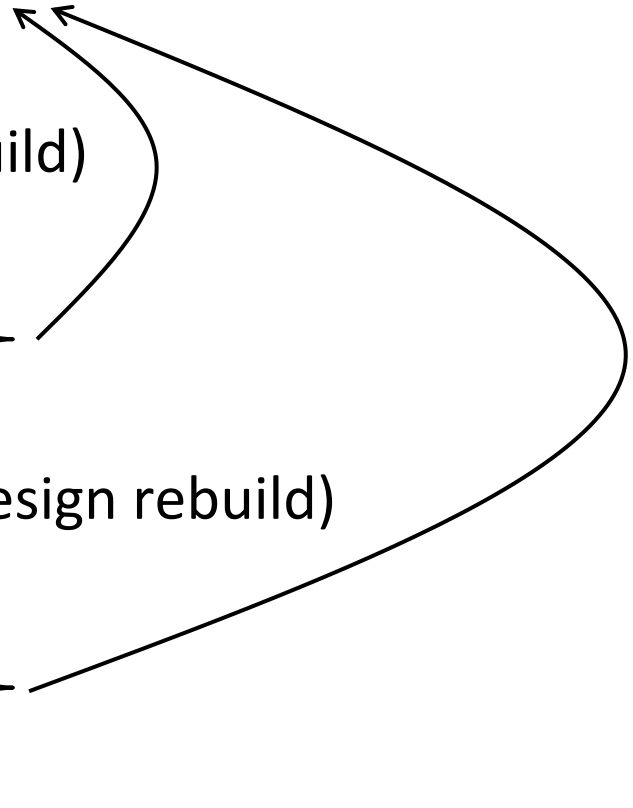


- GHDL aktuálně podporuje tři různé generátory strojového kódu (tzv. backends)
- mcode: vestavěný backend
  - Nejsnadnější použití, ale nejpomalejší simulace
  - Pro simulaci se nevytváří binární soubor
- LLVM: Low-Level Virtual Machine (llvm.org)
  - Kompromis mezi jednoduchostí použití a rychlostí simulace
  - Pro simulaci se vytváří binární soubor
- GCC: GNU Compiler Collection (gcc.gnu.org)
  - Nejsložitější použití, ale nejrychlejší simulace
  - Pro simulaci se vytváří binární soubor
- Nebude-li uvedeno jinak, uvažujeme na dalších slajdech využití vestavěného backendu mcode

- Jednotný formát specifikace parametrů při spouštění GHDL
  - První parametr určuje příkaz, jehož výsledek lze modifikovat pomocí přepínačů

```
ghdl command [options...]
```

- Dva základní způsoby použití GHDL:
  - ruční ošetření závislostí (tzv. design build)
    1. Analýza (analysis) `[-a]`
    2. Zpracování (elaboration) `[-e]`
    3. Simulace (run) `[-r]`
  - automatické ošetření závislostí (tzv. design rebuild)
    1. Načtení (import) `[-i]`
    2. Překlad (make) `[-m]`
    3. Simulace (run) `[-r]`



1. Analýza (analysis)
   - Obdoba překladu zdrojového souboru do objektového souboru

     ```
     ghdl –a <[options...] file...>
     ```
2. Zpracování (elaboration)
   - Obdoba sestavení spustitelného binárního souboru z objektových souborů a knihoven

     ```
     ghdl -e <[options...] [library.]top_unit [arch]>
     ```
3. Simulace (run)
   - Obdoba spuštění spustitelného binárního souboru
   - Pozor, `[options...] != [simulation_options...]`

```
ghdl -r <[options...] [library.]top_unit [arch] [simulation_options...]>
```

- V případě backendů LLVM a GCC lze v popisech výše vypustit slovo „obdoba“

1. Načtení (import)
   - Načtení zdrojových souborů a identifikace jednotlivých částí obvodu

     ```
     ghdl –i <[options...] file...>
     ```

2. Překlad (make)
   - Analýza a zpracování načtených zdrojových souborů bez potřeby specifikace závislostí mezi jednotlivými soubory

     ```
     ghdl -m <[options...] [library.]top_unit [arch]>
     ```

3. Simulace (run)
   - Obdoba spuštění spustitelného binárního souboru
   - Pozor, `[options...] != [simulation_options...]`

```
ghdl -r <[options...] [library.]top_unit [arch] [simulation_options...]>
```

GHDL – přepínače příkazu vs. přepínače simulace

- Příkaz pro simulaci (run) umožňuje specifikovat nejen přepínače příkazu `[options...]`, ale i přepínače simulace `[simulation_options...]`
  - Přepínače příkazu musí být od přepínačů simulace striktně odděleny

```
ghdl -r <[options...] [library.]top_unit [arch] [simulation_options...]>
```

- Příklady `[options...]`:
  - `--std=<STANDARD>`
  - `--workdir=<DIR>`
  - `-fsynopsys`
  - `-fexplicit`
  - a mnoho dalších…

- Příklady `[simulation_options...]`
  - -gGENERIC=VALUE
  - --assert-level=<LEVEL>
  - --stop-time=<TIME>
  - --stop-delta=<N>
  - a mnoho dalších …

https://ghdl.github.io/ghdl/using/InvokingGHDL.html#options

https://ghdl.github.io/ghdl/using/Simulation.html#simulation-options

GHDL – export časového diagramu

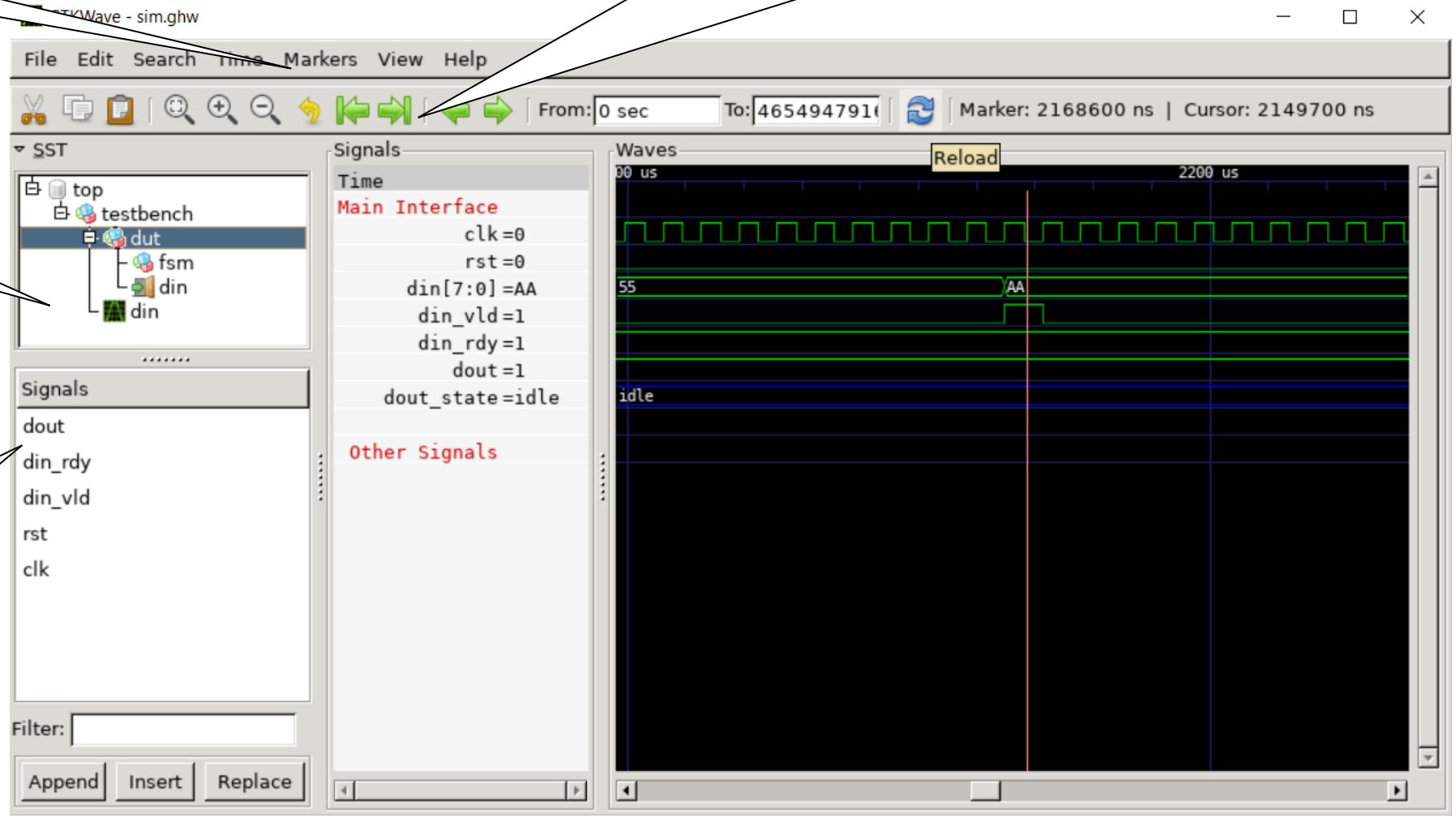
- GHDL neobsahuje prohlížeč časových diagramů, nicméně umožňuje export hodnot signálů v několika formátech
  - VCD (Value Change Dump) `[--vcd, --vcdgz]`
    - Široce rozšířený formát definovaný ve standardu HDL jazyka Verilog
    - Nepodporuje export všech datových typů z VHDL
  - FST (Fast Signal Trace) `[--fst]`
    - Nejúspornější z uvedených formátů
    - Podporou konstrukcí jazyka VHDL ekvivalentní formátu VCD
  - GHW (GHDL Waveform) `[--wave]`
    - Proprietární formát pro export hodnot signálů
    - Podporuje všechny datové typy jazyka VHDL
- Časové diagramy ve všech uvedených formátech je možné zobrazit např. pomocí externího nástroje GTKWave

Zoom v okně signálů

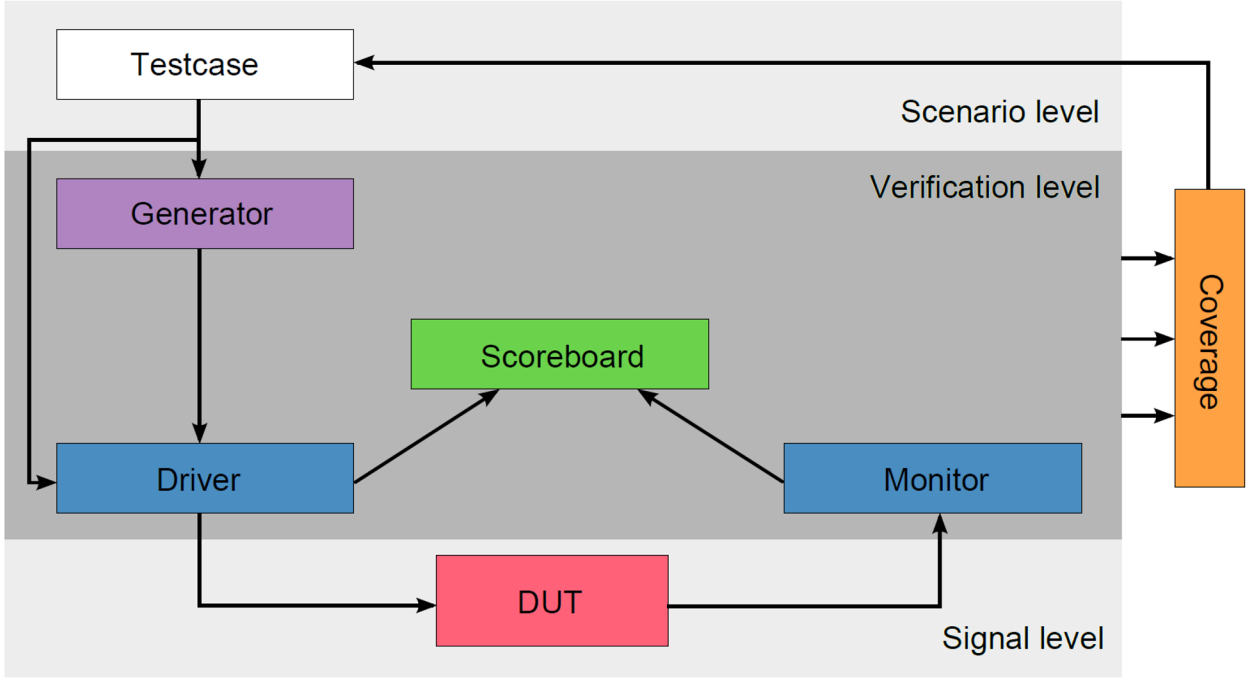
Panel simulačních nástrojů

Struktura obvodu

Signály v rámci vybrané komponenty

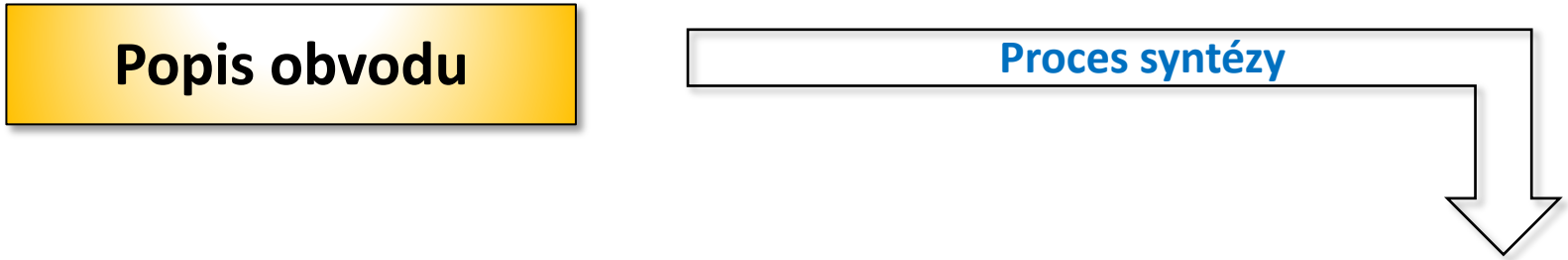


- Simulace rozšířená o pokročilé testovací techniky
  - ➢ Náhodné generování testovacích vektorů podle omezujících podmínek
  - ➢ Analýza pokrytí funkcionality systému
  - ➢ Definice a kontrola invariantních podmínek
  - ➢ Automatické vyhodnocování správnosti výstupních transakcí
- Typicky implementováno s využitím jazyka SystemVerilog
- Využívá se metodika UVM (Universal Verification Methodology)

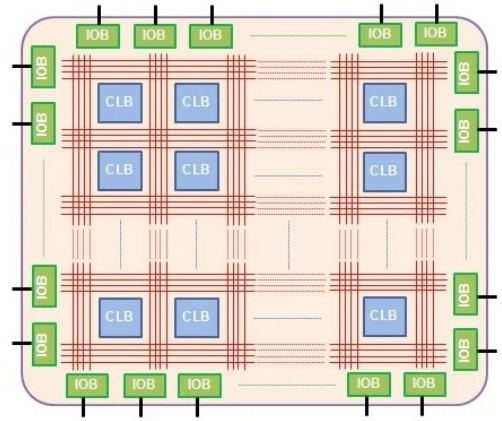


Podrobnosti viz volitelný magisterský kurz
Funkční verifikace číslicových systémů

- Transformace popisu obvodu do konfigurace pro technologii FPGA nebo do podkladů pro vytvoření masky pro výrobu IC/ASIC

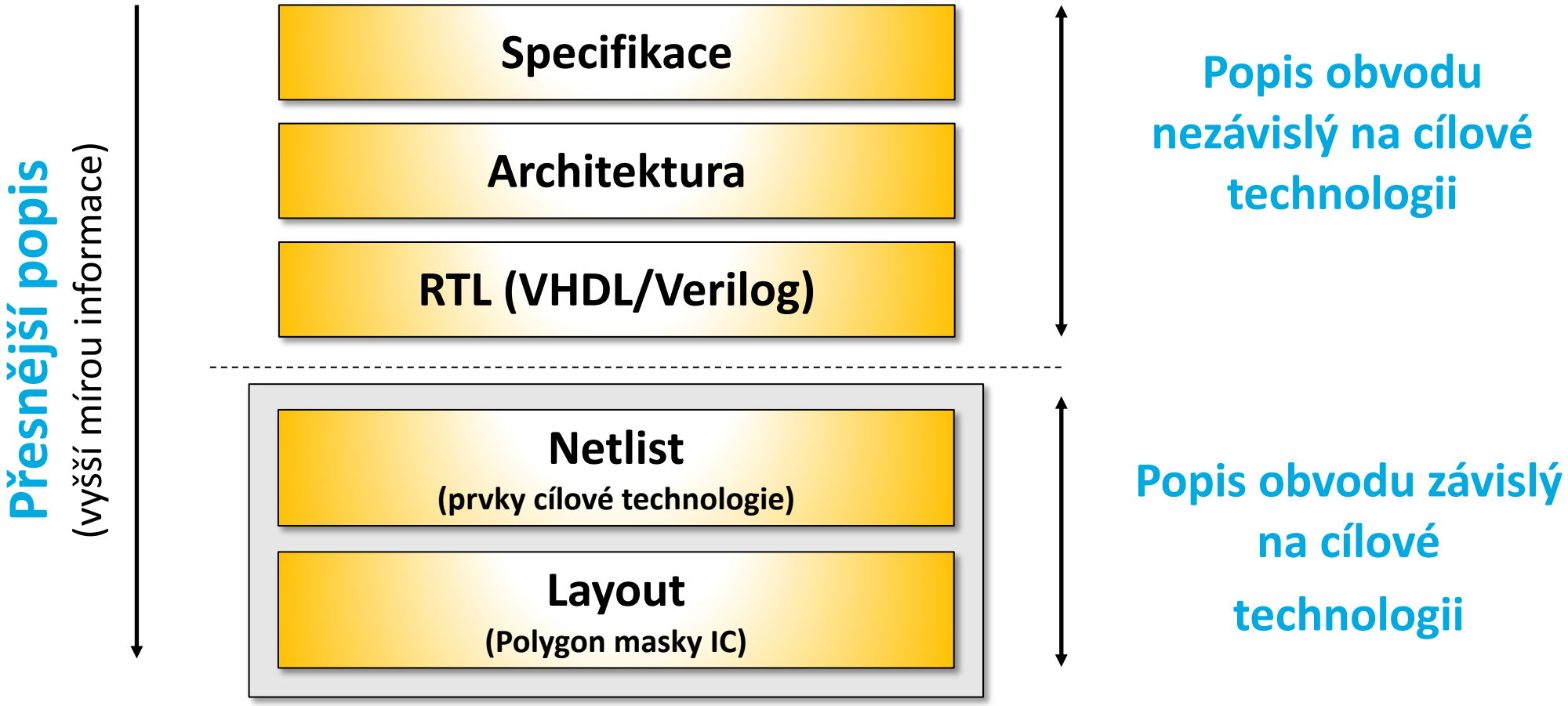


- Obvod je možné popsat na různých úrovní abstrakce (míra detailu)
  - Algoritmus popisující činnost obvodu
  - Zapojení skládající se z obvodových prvků (Registr, čítač, sčítačka, multiplexor, …)



FPGA nebo IC/ASIC

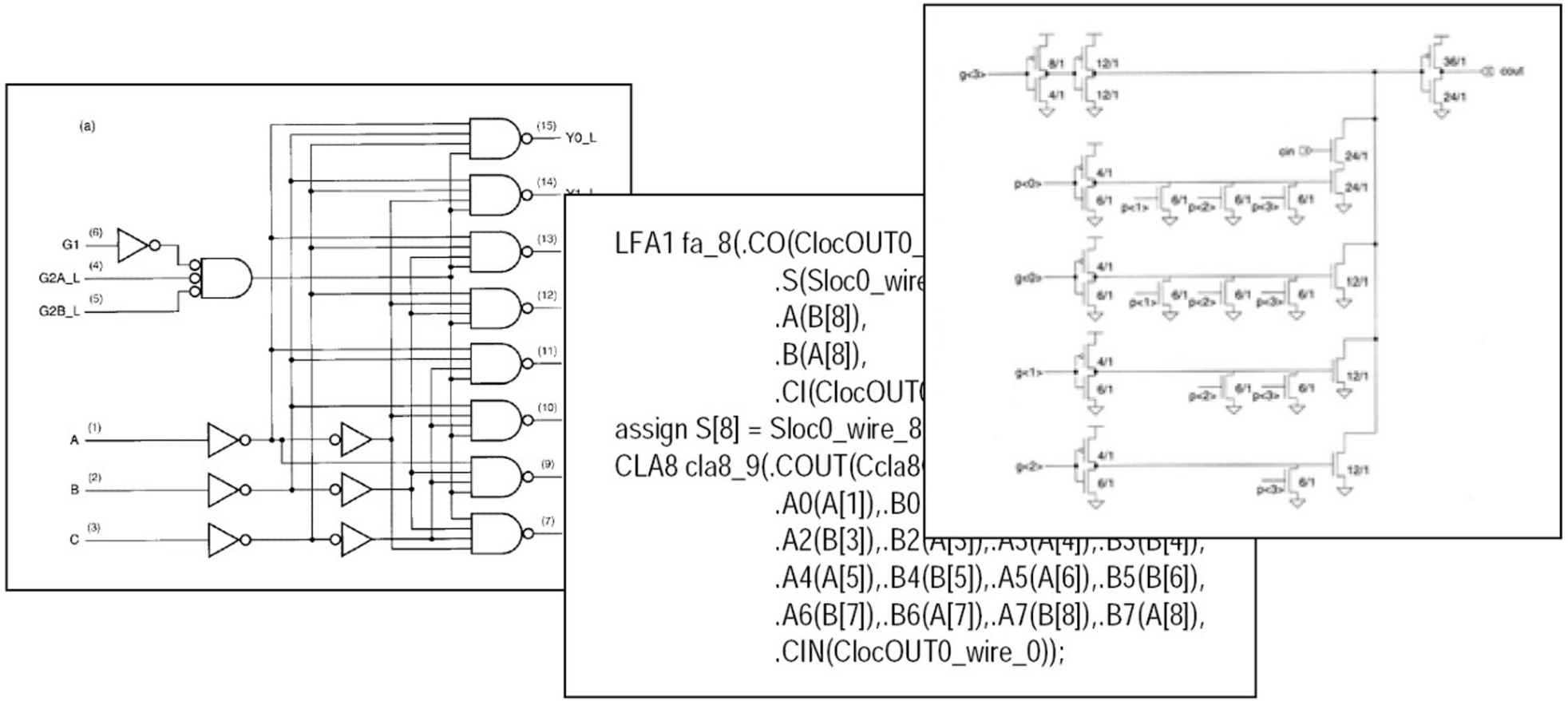
Úrovně popisu číslicových obvodů



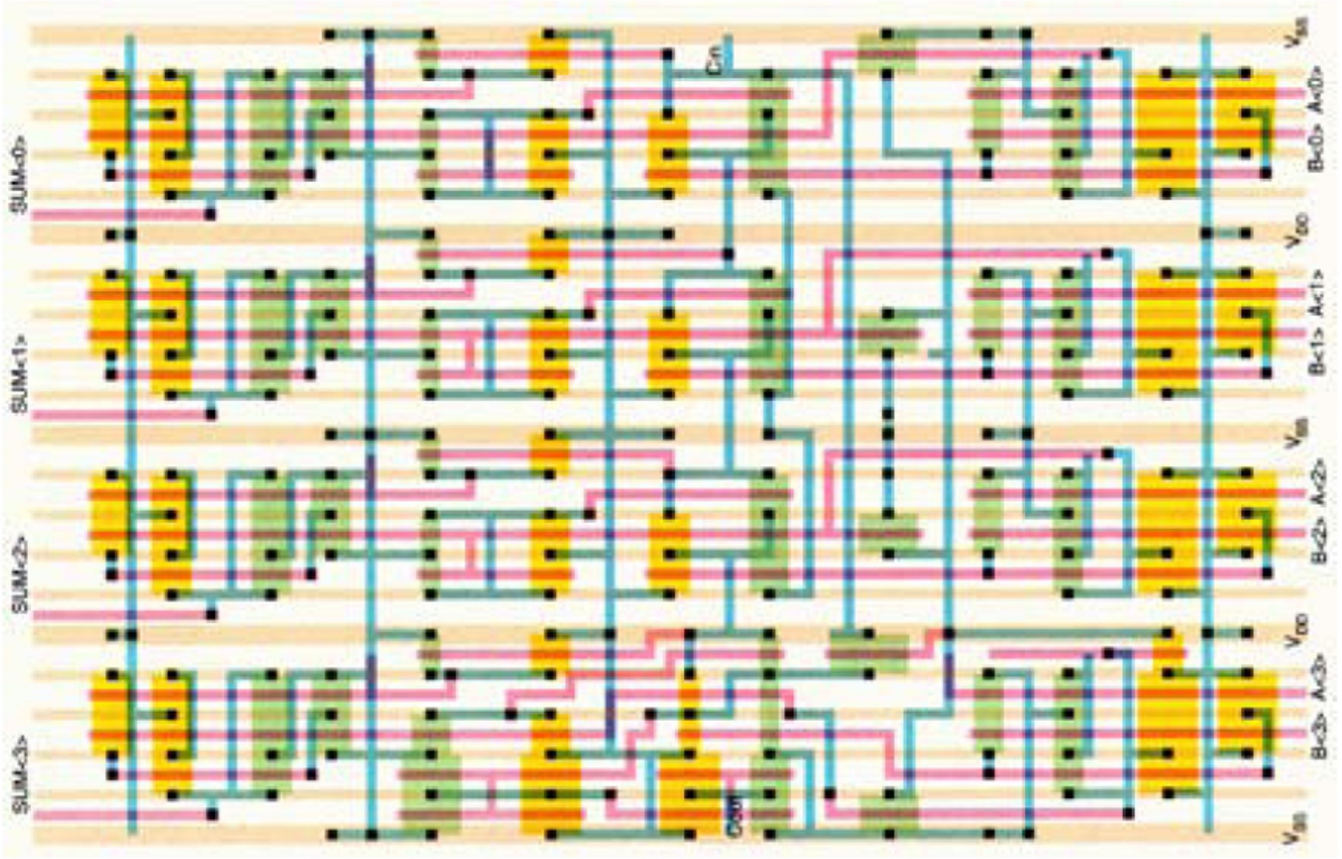
- Popis synchronních obvodů řízených hodinovým signálem
- Oddělení kombinační logiky registry nebo paměťovými prvky
- Chování obvodu řízené hodinovým signálem

Snadná analýza chování i časových parametrů obvodu

- Detailní propojení komponent cílové technologie
  - Schéma pouze z prvků cílové technologie
  - Nejčastěji ve formátu EDIF, ale může být popsán i ve VHDL nebo Verilog



- Podklady pro výrobu čipů v dané cílové technologii
- Geometrické obrazce které odpovídají křemíkové nebo metal-oxidové vrstvě, tvoří základní prvky výsledného integrovaného obvodu



- Syntéza: Automatická transformace mezi různými úrovněmi popisu
  - Vytváří přesnější (jemnější) popis obvodu s cílem dosáhnout parametry zadané uživatelem: ***rychlost***, ***plocha na čipu***, ***spotřeba***, testovatelnost a podobně.
  - Kromě samotného obvodu je vstupem také sada požadavků (***constraints***) specifikovaných uživatelem (perioda hodin, zpoždění propojovacích vodičů, atd.)

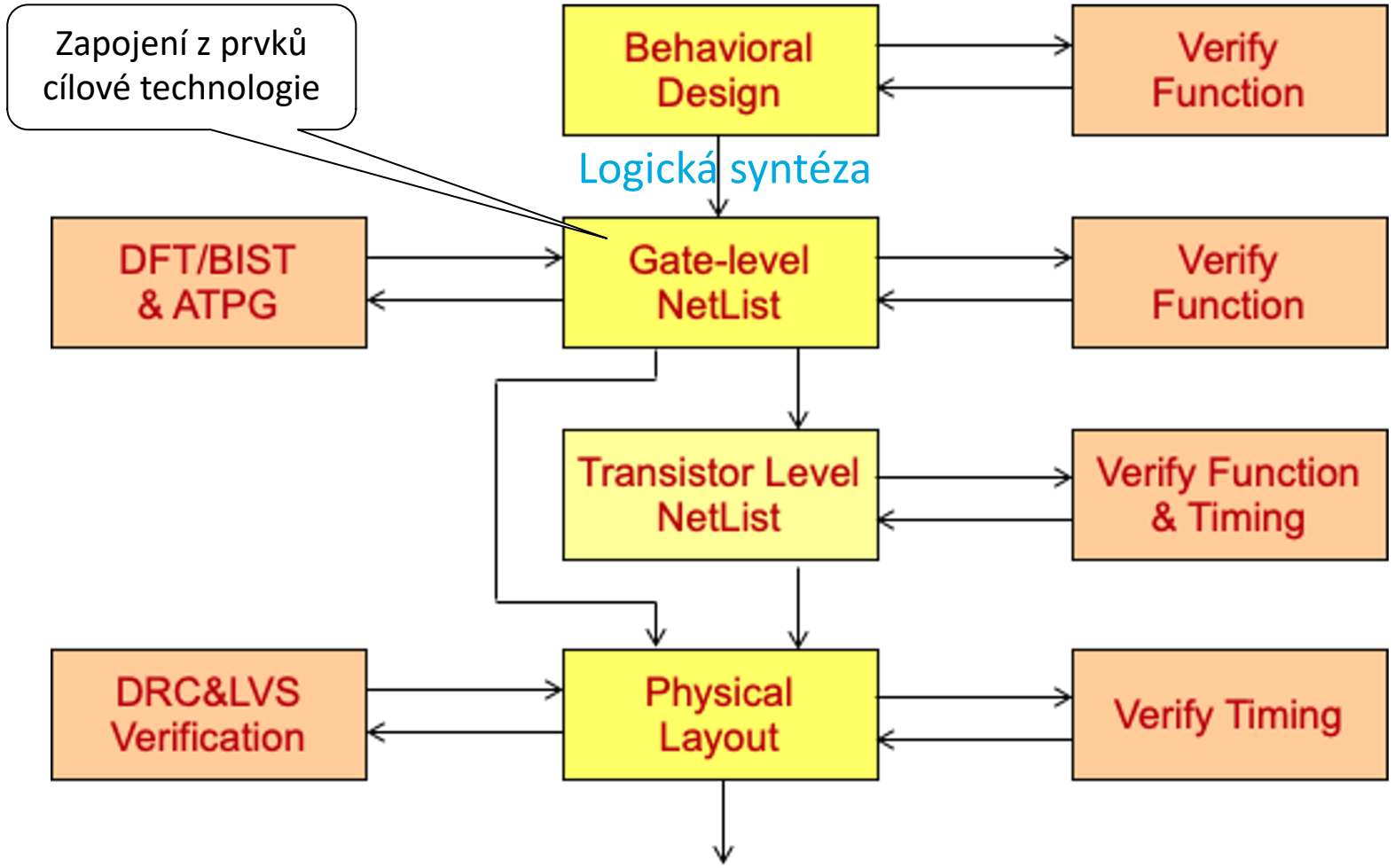
Behaviorální syntéza

Z behaviorálního popisu algoritmu je vytvořena reprezentace na úrovni RT (meziregistrových přenosů)

Logická syntéza

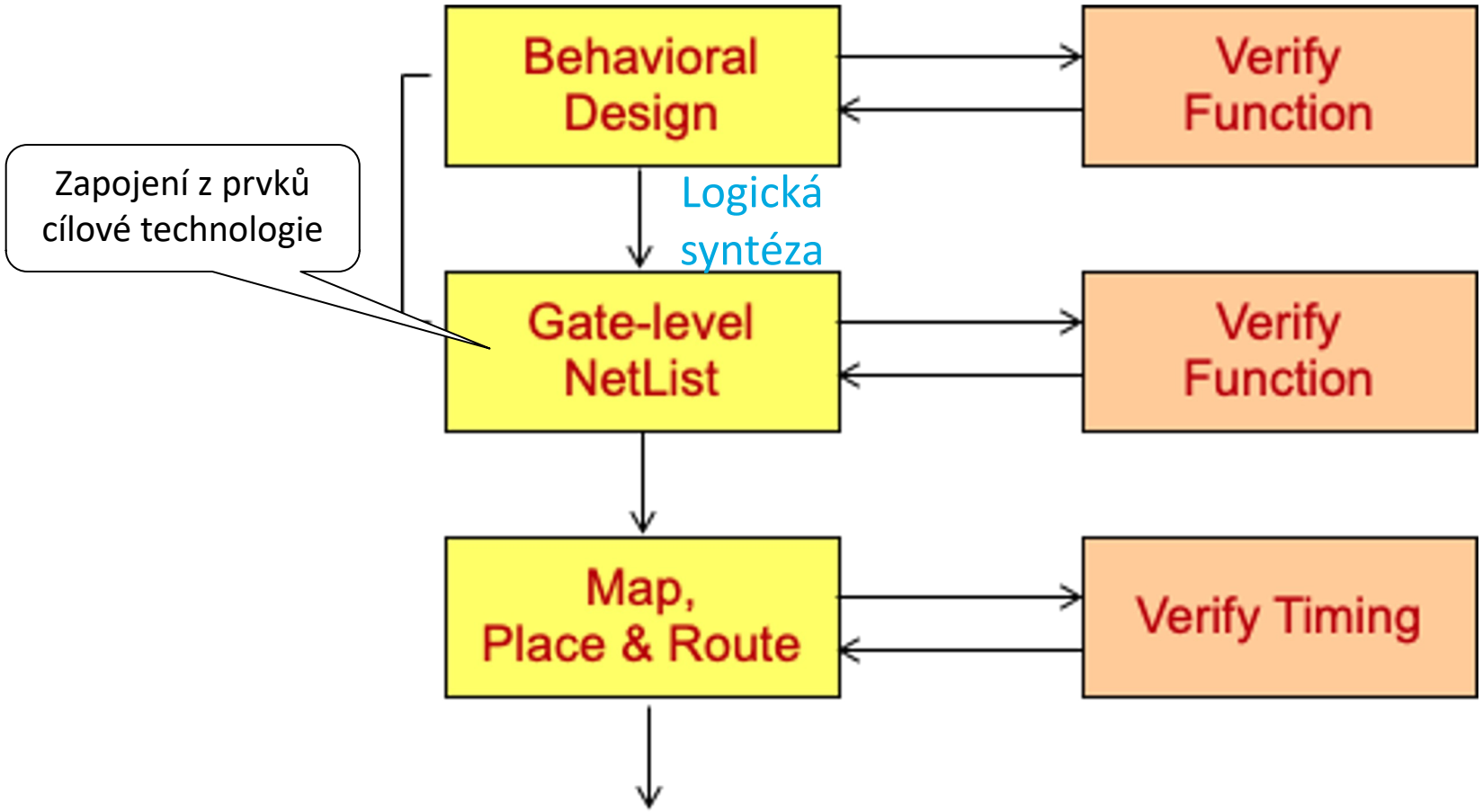
Z HDL popisu na úrovni RT (meziregistrových přenosů) je vytvořen NetList prvků cílové technologie

- Logická syntéza rozpozná prvky cílové technologie, následně jsou odvozeny informací pro tvorbu masky IO



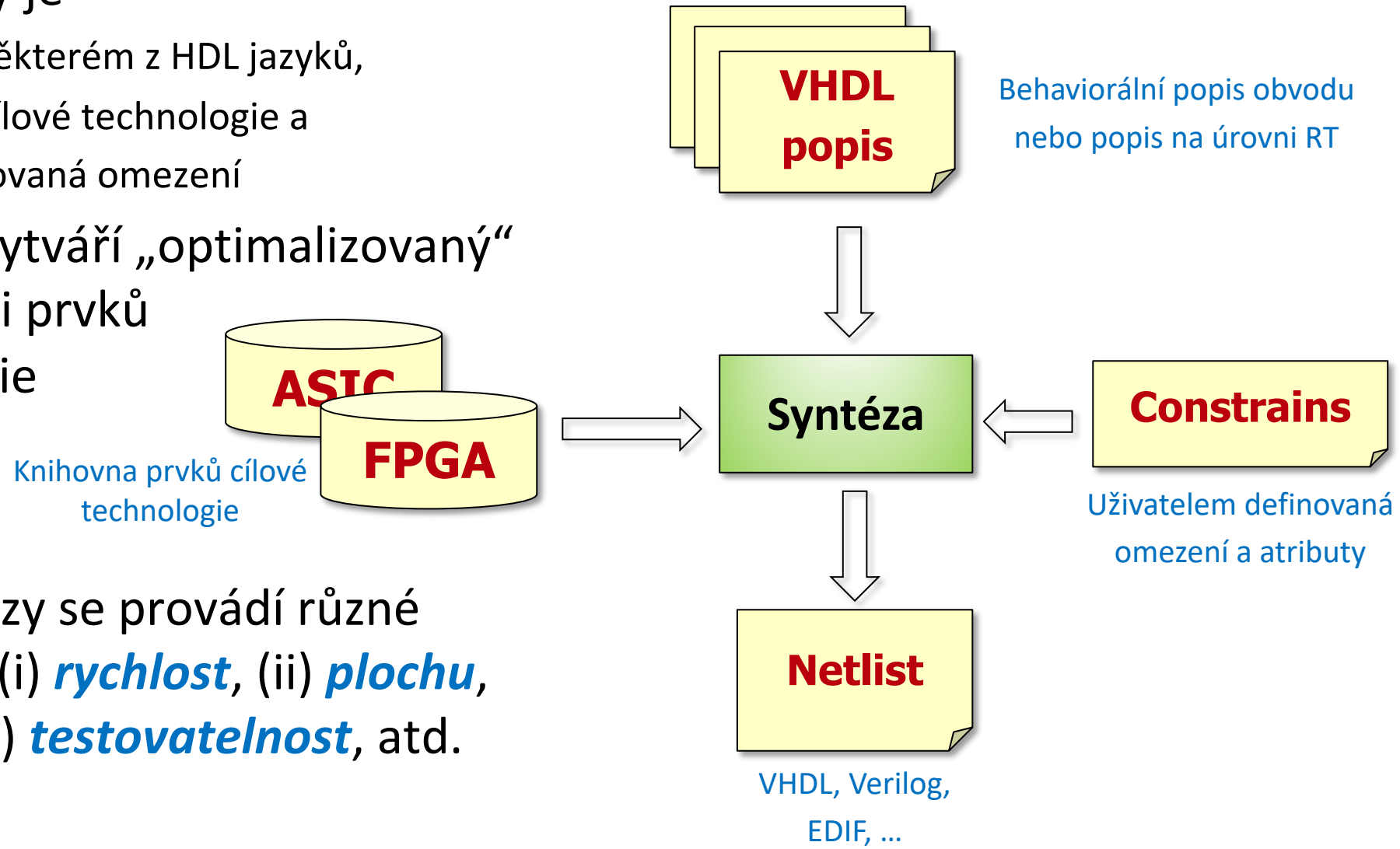
- Nejprve se pouze verifikuje funkce obvodu, v dalších fázích vývojového cyklu se ověřuje také časování obvodu.
- Odhalení chyby funkce nebo nesplněné časování znamená návrat do úvodních fází vývojového procesu

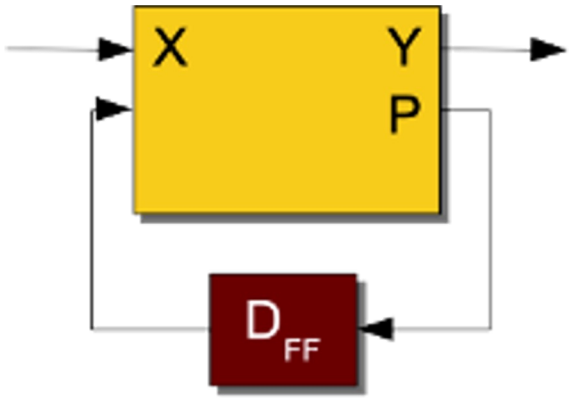
- Logická syntéza rozpozná prvky cílové technologie, následně jsou odvozeny informací pro tvorbu masky IO

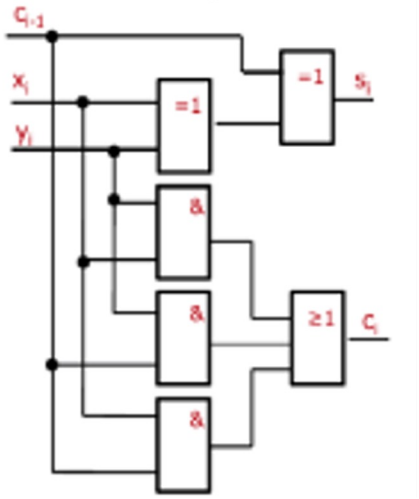


- Nejprve se pouze verifikuje funkce obvodu, v dalších fázích vývojového cyklu se ověřuje také časování obvodu.
- Odhalení chyby funkce nebo nesplněné časování znamená návrat do úvodních fází vývojového procesu

- Vstupem syntézy je
  - popis obvodu v některém z HDL jazyků,
  - knihovna prvků cílové technologie a
  - uživatelem definovaná omezení
- Proces syntézy vytváří „optimalizovaný“ NetList na úrovni prvků cílové technologie
- V průběhu syntézy se provádí různé optimalizace na (i) ***rychlost***, (ii) ***plochu***, (iii) ***spotřebu***, (iv) ***testovatelnost***, atd.



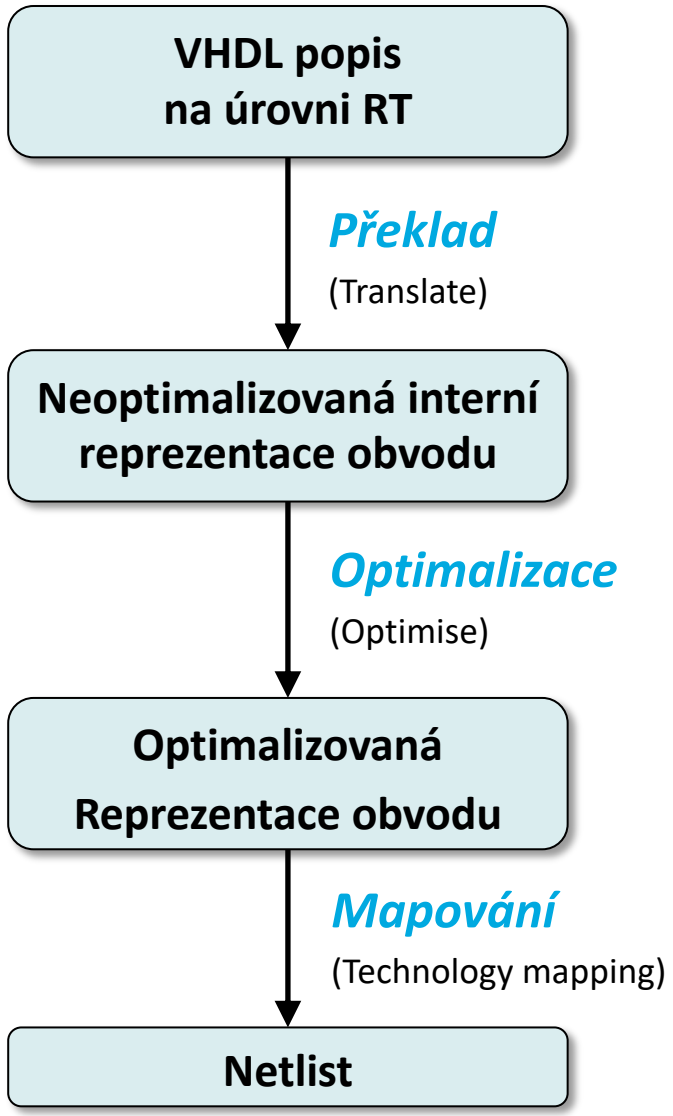




- Vstup: ***Finite State Machine (FSM)*** definovaný jako (X,Y,Z,P,V), kde
  - ***X*** je vstupní abeceda
  - ***Y*** je výstupní abeceda
  - ***Z*** je množina vnitřních stavů
  - ***P : X x Z → Z*** je přechodová funkce
  - ***V : X x Z → Y*** je výstupní funkce
- Výstup: ***Obvod (G,W)***, kde
  - ***G*** je množina prvků cílové technologie (hradla, registry a podobně) a
  - ***W*** je množina propojů mezi prvky cílové technologie G

Syntéza probíhá typicky ve třech základních krocích:

1. ***HDL popis obvodu je převeden do interní reprezentace syntézního nástroje***
   Na základě analýzy zdrojového kódu je celý kód převede na zapojení z primitivních hradel AND, OR, Invertor, registry typu flip-flop a podobně. Interní reprezentace odpovídá zdrojovému kódu, není nijak optimalizována
2. ***Boolovské optimalizace interní reprezentace***
   Pomocí teorémů Boolovy algebry jsou minimalizovány reprezentace logických funkcí. Výsledkem je optimalizovaná interní reprezentace obvodu.
3. ***Mapování do hradel cílové technologie***
   S využitím knihovny obsahující prvky cílové technologie je provedeno mapování zapojení obvodu na zapojení těchto prvků. Výsledkem je NetList.



- Optimalizace logických výrazů

$$F(x,y,z) = \bar{x}\cdot\bar{y}\cdot z + \bar{x}\cdot y\cdot z + x\cdot y\cdot\bar{z}$$
$$= \bar{x}\cdot z\cdot(y+\bar{y}) + x\cdot y\cdot\bar{z}$$
$$= \bar{x}\cdot z + x\cdot y\cdot\bar{z}$$

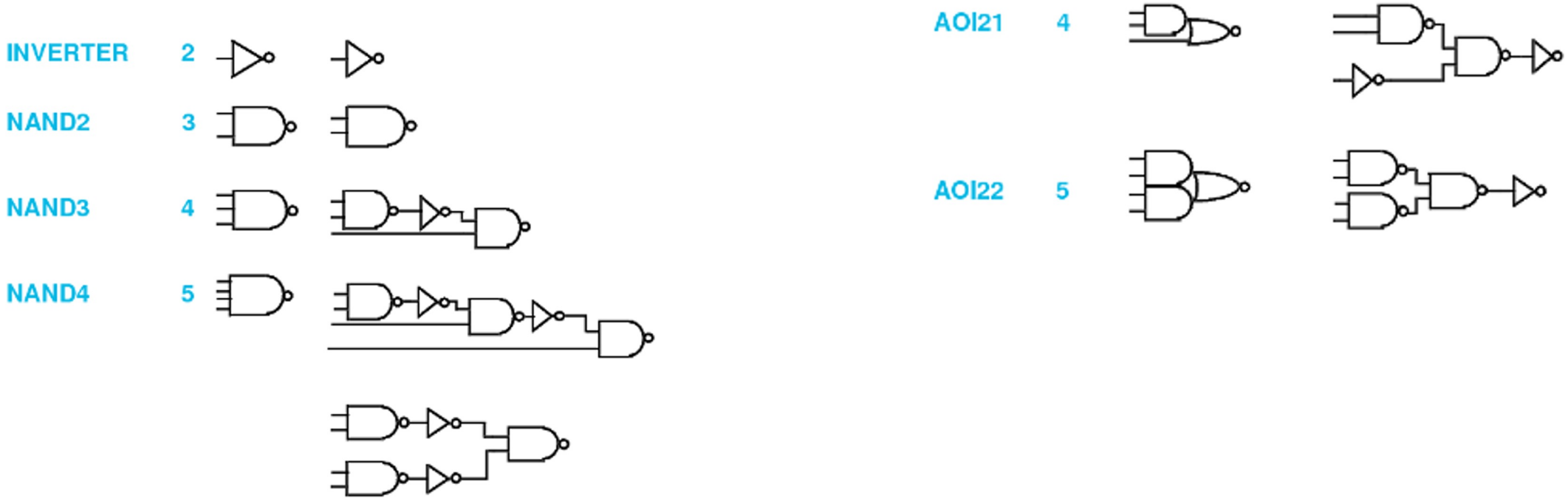
- Techniky minimalizace kombinační logické sítě:
  - Algebraická pomocí Boolovy algebry
  - Graficky pomocí map
  - Algoritmicky

Cíl mapování: Vytvořit interní reprezentace obvodu NetList, který je složen pouze z prvků cílové technologie

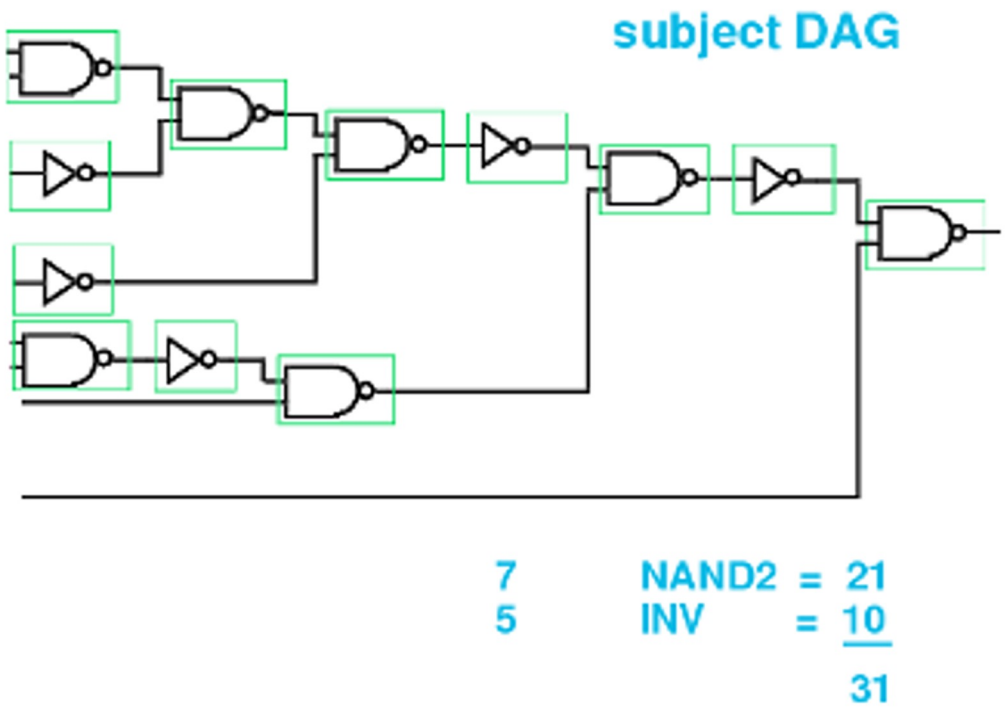
- Vstupem je ***interní reprezentace obvodu ve formě acyklického orientovaného grafu (DAG)*** a ***knihovna prvků cílové technologie***
- DAG graf obsahuje jako uzly binární operace AND a hrany mohou být označeny tak, aby znázorňovaly buď přímé nebo negované vstupy
- Knihovna prvků cílové technologie popisuje funkci pomocí DAG AIG grafu
- Algoritmus hledá postupně v DAG grafu podgrafy reprezentující prvky cílové technologie, snaží se najít pokrytí vstupního grafu prvky tak, aby cena implementace byla co nejmenší
- Výstupem je NetList (zapojení) obsahující pouze prvky cílové technologie

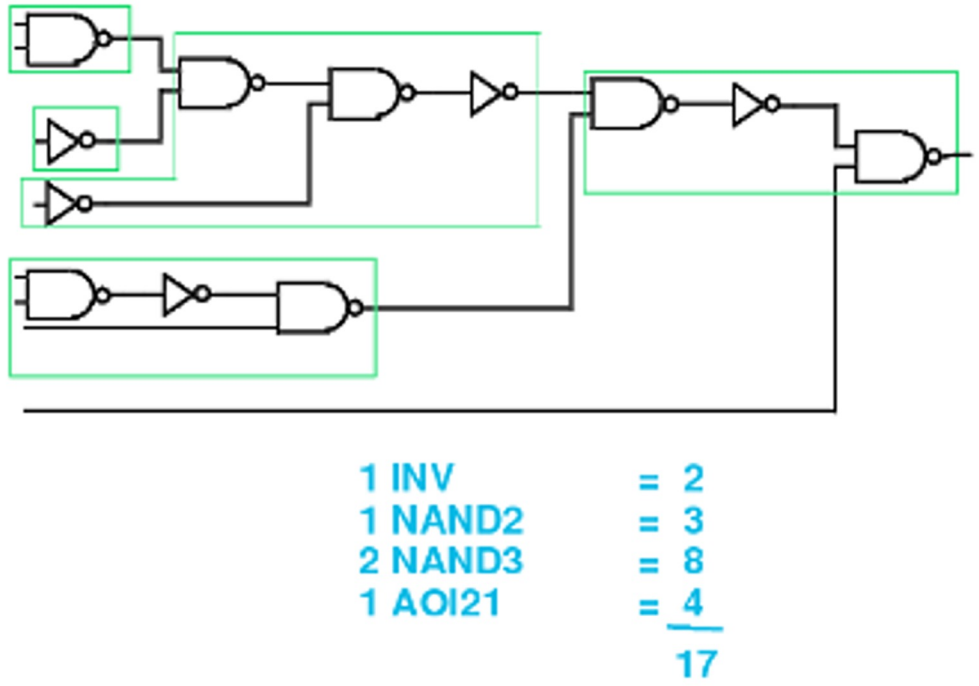
(3) Mapování do cílové technologie (Příklad)

- Reprezentace prvků cílové technologie pomocí DAG grafů složených pouze z prvků dvou-vstupý NAND a Invertor



(3) Mapování do cílové technologie (Příklad)





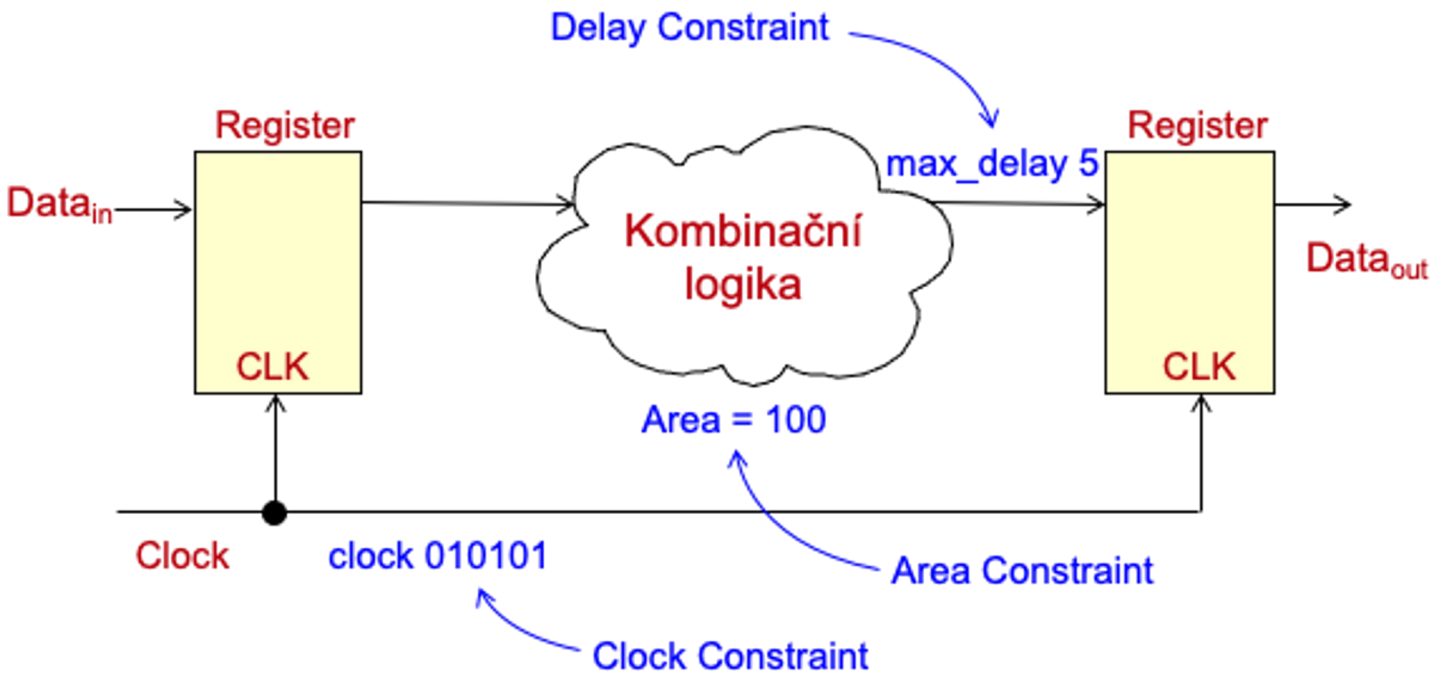
- Dva příklad pokrytí obvodu prvky cílové technologie
- Různé mapování má různou cenu implementace
- Cena může zohledňovat nejen plochu na čipu, ale i rychlosti zpracování

- Obsahuje informace o prvcích cílové technologie zahrnující
  - Logickou funkci ve formě DAG grafu
  - Zabranou plochu na čipu
  - Časování průchodu signálu ze vstupu na výstup
  - Omezení na fanout
  - Časová omezení

Příklad pro hradlo AND

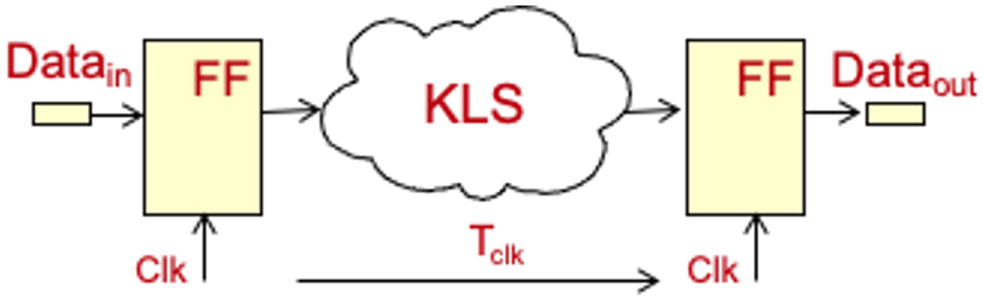
```
Library (x,y,z) {
cell (and2) {
 area : 5;
 pin (a1, a2) {
   direction   : input;
   capacitance : 1;
 }
 pin (o1) {
   direction   : output;
   function    : "a1 * a2";
   timing () {
      intrinsic_rise : 0.37;
      intrinsic_fall : 0.56;
      rise_resistance: 0.1234;
      fall_resistance: 0.4567;
      related_pin    : "a1 a2"
   }
 }
}
}
```

- Constraints se používají pro řízení optimalizace a mapování
- V současnosti jsou nástroji podporované uživatelská omezení zohledňující ***časování***, ***spotřebu***, ***velikost*** nebo ***umístění*** obvodu nebo části obvodu.

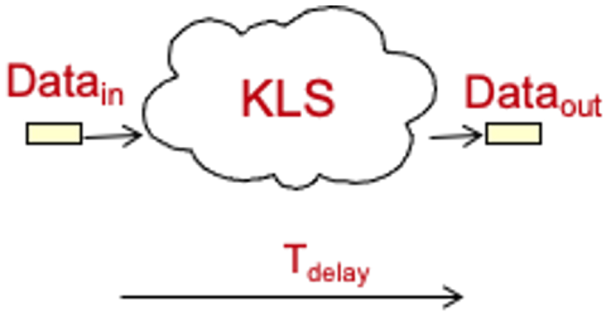


- ***Timing constraints*** (časová omezení) se používají pro nastavení maximálního zpoždění pro vybrané cesty v obvodu
  - Nutí proces optimalizace a mapování ke splnění definovaných časů (zpoždění, frekvence, atd.)
  - Dosažení správných časových parametrů je jednou z nejtěžších úloh při návrhu obvodů pro technologii ASIC nebo FPGA

Clock constraint



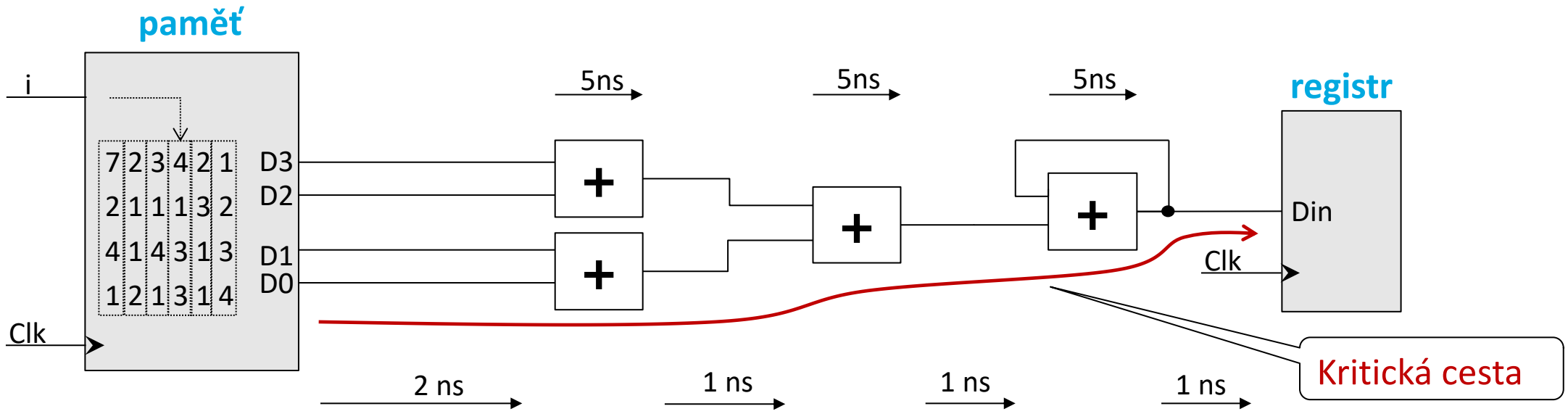
Pad to Pad constraint



Syntézní nástroje automaticky kontrolují, jestli jsou splněna omezení zadaná uživatelem

- ***Kritická cesta*** – posloupnost logických prvků a propojů tvořící největší zpoždění mezi dvojicí registrů.

- Příklad: Součet všech dat uložených v paměti, součet po čtyřech vzorcích

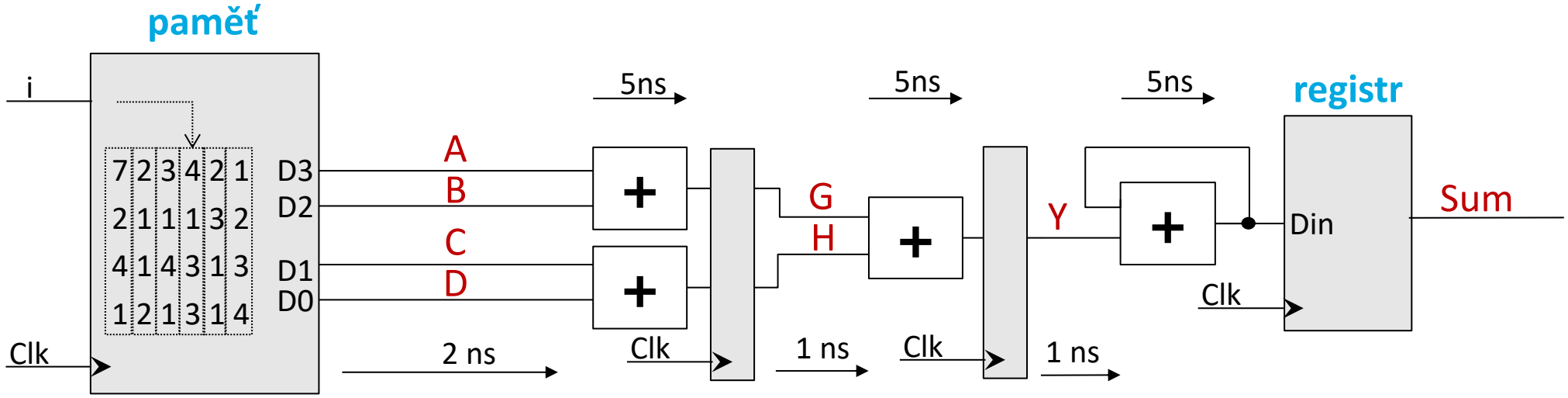


- Jak určím kritickou cestu a spočítám její délku? **T=2+5+1+5+1+5+1 = 20 ns**
- Jaká bude maximální dosažitelná frekvence obvodu? **f=1/T = 1/20ns = 50 MHz**
- Jak dlouho bude trvat výpočet při zpracování 6 vzorků dat adresovaných pomocí vstupu i při maximální frekvenci? **$t_{výpočtu}$= 6 * 20 ns = 120 ns**

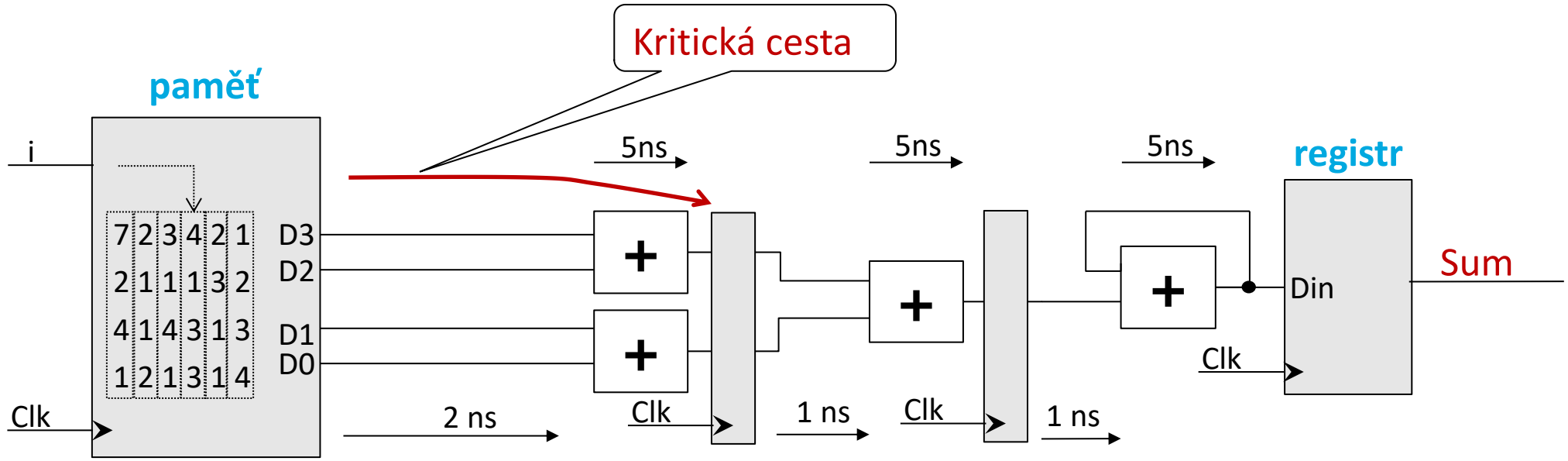
Zřetězená linka (analogie z výroby)

- Jak zpracování více vzorků dat zrychlit?
- Můžeme se inspirovat výrobními linkami se sériovou výrobou.
  - Výroba každého nového auta je rozdělena do spousty malých operací
  - Současně se na lince vyrábí více aut, každé auto je v jiné fázi výrobního procesu
- Důsledek
  - Celá výroba jednoho auta sice trvá stejně dlouho, ale
  - máme vyšší produkci aut za jednotku času

Zřetězené zpracování (Pipelining)



	A, B, C, D	G, H	Y	Sum
T0: 0 ns	1, 2, 3, 4			0
T0: 5 ns	2, 3, 1, 1	3, 7		
T0: 10 ns	4, 1, 3, 3	4, 2	10	
T0: 15 ns	3, 1, 4, 1	5, 6	6	10
T0: 20 ns	2, 1, 1, 2	4, 5	11	16
T0: 25 ns	7, 2, 4, 1	3, 3	9	27



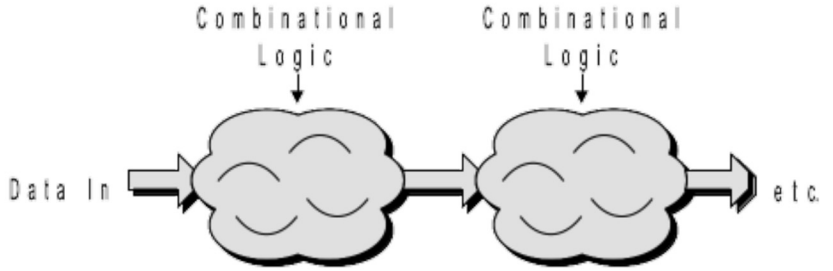
- Jak určím kritickou cestu a spočítám její délku? **T=2+5 = 7 ns**
- Jaká bude maximální dosažitelná frekvence obvodu? **f=1/T = 1/7ns = 142,9 MHz**
- Jak dlouho bude trvat výpočet při zpracování 6 vzorků dat adresovaných pomocí vstupu i při maximální frekvenci? **$t_{výpočtu}$= 3 * 7 ns + 6 * 7ns = 63 ns**

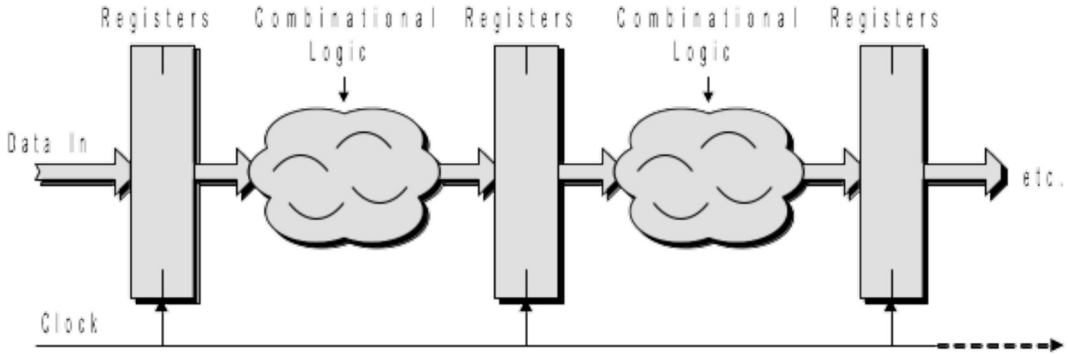
Doba než projde 1. vzorek linkou

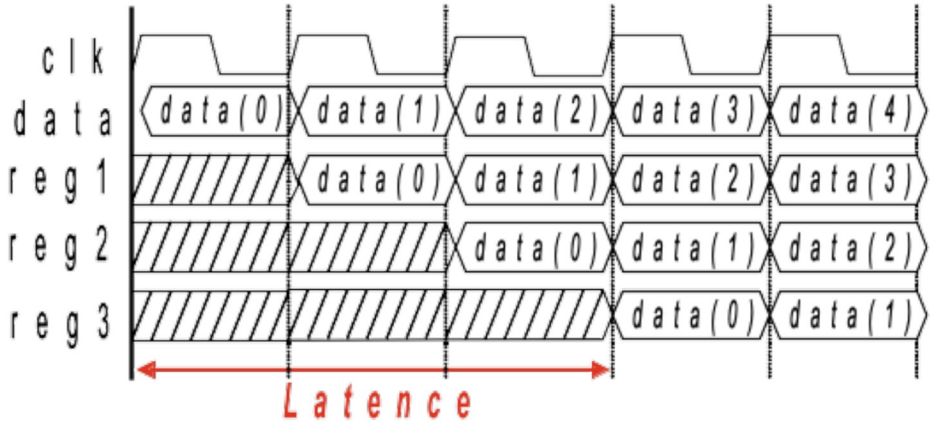
Všechny vzorky na výstupu

Základní princip:

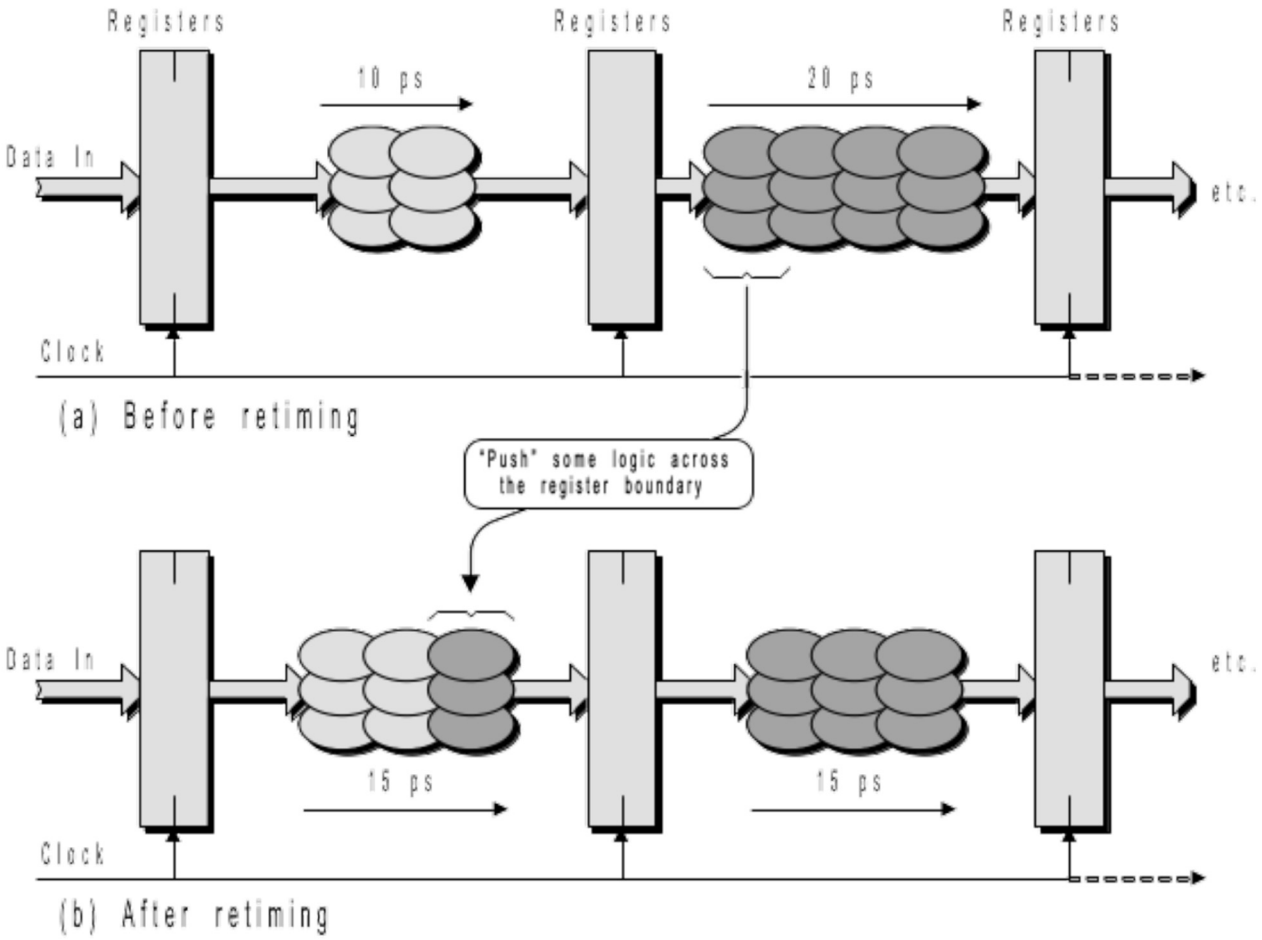
- Větší úsek kombinační logiky se rozdělí na několik menších částí
- Mezi jednotlivé části se vloží registry zajišťující zřetězení
- Po vložení ***k registrových stupňů*** lze dosáhnout až ***k+1 násobné zrychlení*** výpočtu, v každém taktu hodinového signálu je k dispozici jeden výsledek
- ***Latence obvodu je ale k-krát vyšší!***
- ***Důležité je, aby bylo zpoždění obvodů v jednotlivých stupních vyvážené***



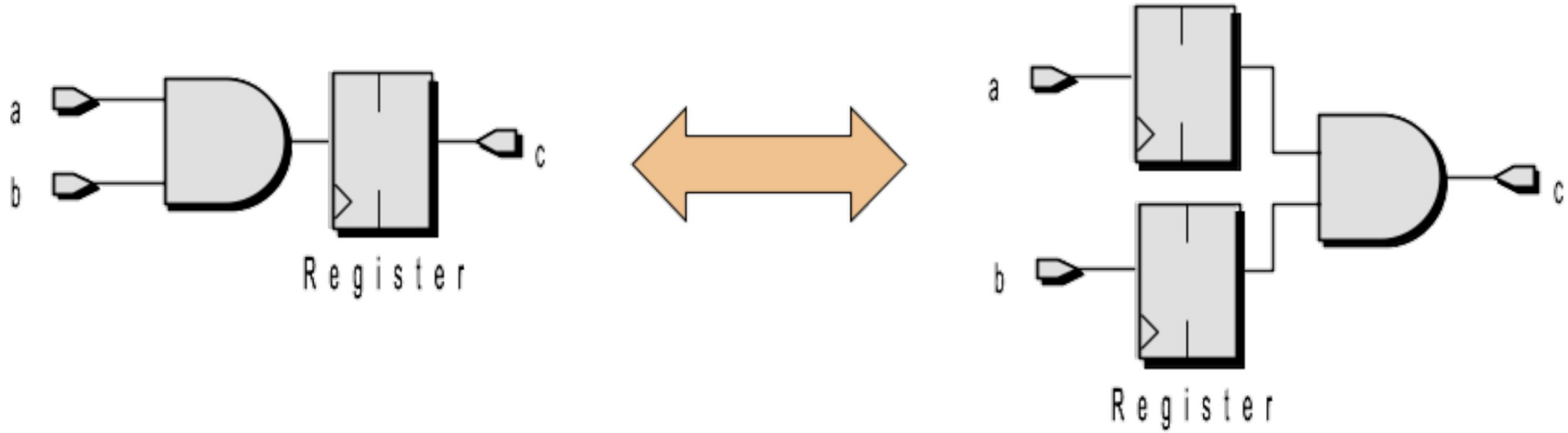




- ***Metoda založena na přeuspořádání kombinační logiky mezi jednotlivými stupni zřetězené architektury***
- Vede na ***vyvážení kombinační cesty mezi registry*** a možnost aplikovat vyšší pracovní frekvenci na celkový obvod
- Podle typu obvodu může metoda vést na zrychlení v řádu desítek procent výkonu
- ***Dostupná obvykle ve formě volitelné optimalizace v syntézních nástrojích***

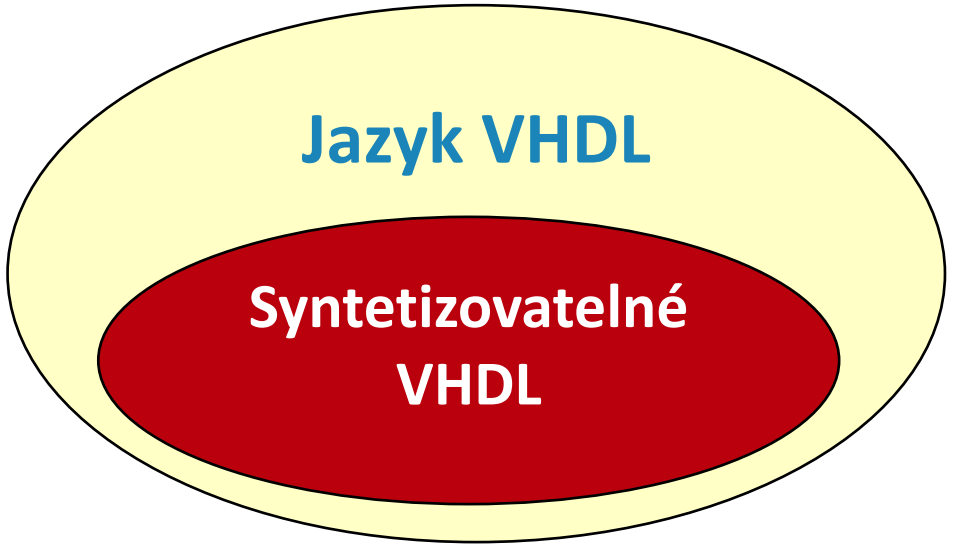


- ***Základní pravidlo metody Retiming***
  - registr na výstupu kombinačního obvodu může být přesunut před tento prvek, pokud je aplikován na všechny jeho vstupy
  - podobným způsobem registr může být přesunut ze vstupu kombinačního obvodu na výstup, pokud je přesunut ze všech jeho vstupních portů
  - funkce obvodu zůstává v rámci této transformace zachována



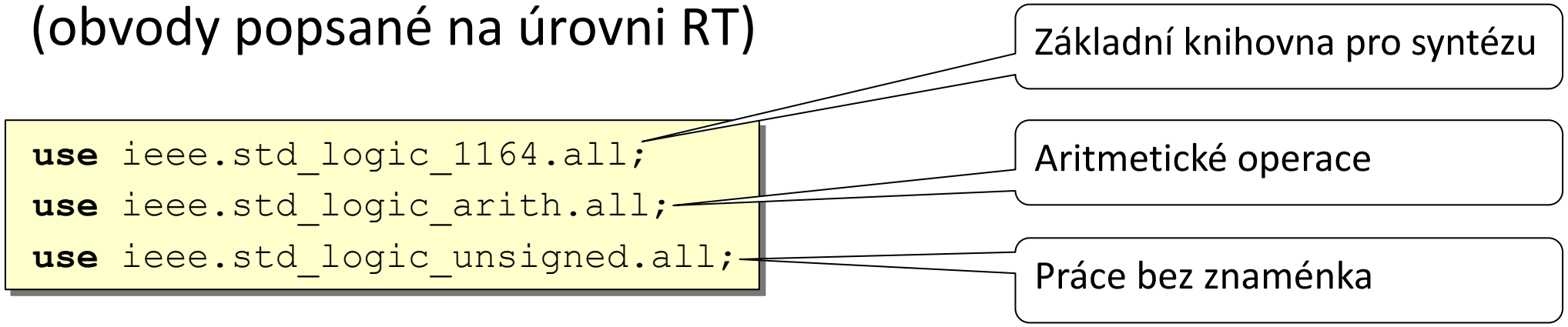
- Syntéza dokáže pracovat pouze s podmnožinou jazyka VHDL - ***Syntetizovatelné VHDL***

- Problematické konstrukce
  - Čtení nebo zápis ze souboru
  - Rozsah smyček a generických instancí musí být konstantní
  - Inertní nebo transportní zpoždění definované pomocí příkazu wait
  - Příkazy pro ověřování funkce komponent – assert, report, ...



Ne všechny konstrukce napsané ve VHDL jsou podporovány syntézními nástroji!

- Byl definován standard IEEE 1076.6 – 1999 pro logickou syntézu (obvody popsané na úrovni RT)



- Cílem standardu bylo definovat syntaxi a sémantiku umožňující ve VHDL snadno popisovat obvody vhodné pro logickou syntézu
- ***Definuje podmnožinu VHDL, která by měla být podporována všemi nástroji pro syntézu***

Problémy při logické syntéze

- Nežádoucí vznik Latch registrů
- Neúplný sensitivity list
- Proměnná délka smyčky nebo příkazu generate
- …

- ***Inferring latches*** – nežádoucí vznik registrů typu Latch

  Pokud popisujeme kombinační logickou síť (KLS) a není pro nějakou kombinaci vstupu definována hodnota výstupu, vytvoří na výstupu obvodu nežádoucí registr typu Latch. Latch registr umožní pro neošetřenou kombinaci vystavit na výstup předcházející hodnotu.

```
process (sel, a, b, c)
begin
   case sel is
     when "00" => mux_out <= a;
     when "01" => mux_out <= b;
     when "10" => mux_out <= c;
     when others => null;
   end case;
end process;
```

Neošetřené kombinace "11" na vstupu sel

Implementace vytvoří Latch registr

```
process (sel, a, b, c)
begin
   mux_out <= a;
   case sel is
     when "00" => mux_out <= a;
     when "01" => mux_out <= b;
     when "10" => mux_out <= c;
     when others => null;
   end case;
end process;
```

Správně napsaný kód vytvoří KLS (multiplexor)

- ***Incomplete sensitivity list*** – na senzitivity listu procesu nejsou umístěny všechny vstupní signály procesu

  Pokud neuvedeme některé ze vstupních signálů na sensitivity list, dostáváme rozdílné chování obvodu v simulaci a v hardware. Chyba se špatně ladí. Syntéza problém indikuje hlášením “Incomplete sensitivity list”.

```
process (sel)
Begin
   mux_out <= a;
   case sel is
      when "00" => mux_out <= a;
      when "01" => mux_out <= b;
      when "10" => mux_out <= c;
      when "11" => mux_out <= d;
      when others => null;
   end case;
end process;
```

Chybí signály a, b, c, d

Syntéza vytvoří korektní multiplexor, protože analyzuje datové závislosti a funkci v rámci procesu. V simulaci nicméně nebude obvod reagovat na změny signálů a, b, c, d.

Různé chování obvodu v simulaci a po syntéze!

- Rozsah cyklu ***for*** nebo příkazu ***generate*** musí být znám v okamžiku syntézy
- Dynamický rozsah cyklů nebo podmíněné cykly ***do/while*** vedou na vytvoření dynamicky se měnícího obvodu → nelze syntézou vytvořit
- Řada syntézních nástrojů podporuje v definici rozsahu pouze typ integer

Příklad použití cyklu FOR

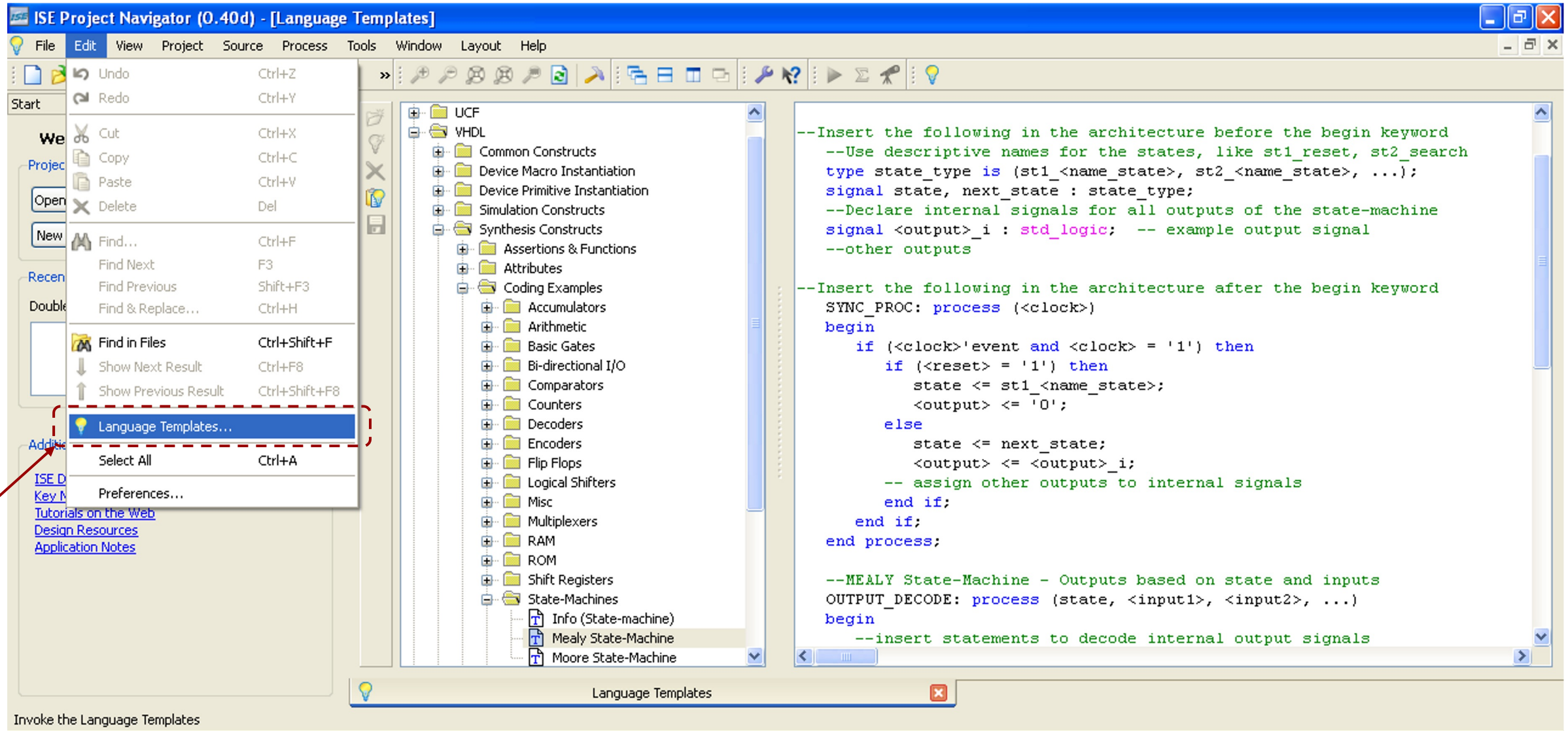
```
process(cnt_bin)
begin
   DO  <= (others => '0');
   for i in 0 to 7 loop
      if (conv_std_logic_vector(i, 3) =
          cnt_bin) then
         DO(i) <= '1';
      end if;
   end loop;
end process;
```

Konstantní rozsah cyklu FOR

Převod binárního čísla do kódu 1 z n

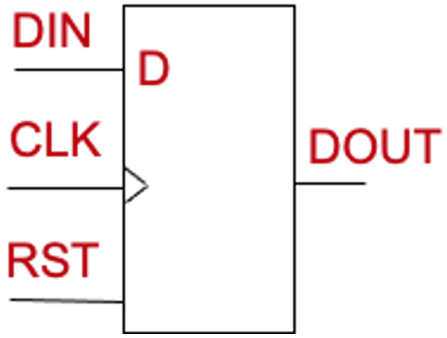
- ***Language templates*** – syntézní nástroje definují šablony kódu jako vhodné implementace základní komponent, kód vhodný pro syntézu
- Použití šablon dává vývojáři přesnou kontrolu nad syntézním nástrojem a současně poskytuje prostor pro optimalizace
- Popisují základní číslicové obvody jako jsou
  - registry a čítače,
  - multiplexory, demultiplexory,
  - komparátory, dekodéry,
  - různé typy pamětí,
  - automaty (FSM),
  - atd.

Další syntetizovatelné šablony



Registr s asynchronním signálem Reset

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dffx is
port (
   CLK   : in  std_logic;
   RST   : in  std_logic;
   DIN   : in  std_logic;
   DOUT  : out std_logic );
end dffx;
architecture behav of dffx is
begin
   process (CLK,RST)
   begin
        if (RST = '1') then
        DOUT <= '0';
        elsif (CLK'event and CLK = '1') then
        DOUT <= DIN;
        end if;
   end process;
end  behav;
```



CLK (clock) – hodinový vstup

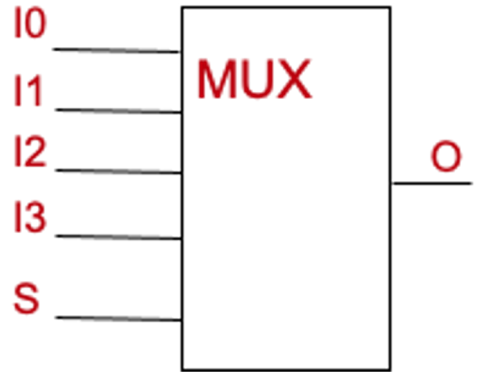
RST(reset) – asynchronní reset

DIN (data in) – data přivedená na vstup registru

DOUT (data output) – hodnota uložená v registru

Multiplexor

```
library ieee;
use ieee.std_logic_1164.all;
entity Mux is
port(    I3: in  std_logic_vector(2 downto 0);
         I2: in  std_logic_vector(2 downto 0);
         I1: in  std_logic_vector(2 downto 0);
         I0: in  std_logic_vector(2 downto 0);
         S : in  std_logic_vector(1 downto 0);
         O : out std_logic_vector(2 downto 0));
end Mux;
architecture behv1 of Mux is
begin
   process(I3,I2,I1,I0,S)
   begin
      case S is
         when "00" =>      O <= I0;
         when "01" =>      O <= I1;
         when "10" =>      O <= I2;
         when "11" =>      O <= I3;
         when others =>    O <= I0;
         end case;
   end process;
 end behv1;
```



I0, **I1**, **I2**, **I3** – přepínané vstupy

S – řídící vstup

Y – výstup

Automat (Finite State Machine)

```
proc_cstate : process (CLK, RST, next_state)
begin
     if RST = '1' then
             cur_state <= s_idle;
     elsif CLK'event AND CLK='1' then
             cur_state <= next_state;
     end if;
end process proc_cstate;
```

```
output_logic : process (cur_state)
begin
     DNEXT <= '0';
     ACK   <= '0';
     case cur_state is
     when s_data =>
             ACK   <= '1';
     when s_next =>
             DNEXT <= '1';
     when others =>
             null;
     end case;
end process output_logic;
```

```
nstate_logic : process (cur_state, RQ, DRDY)
begin
     next_state <= s_idle;
     case cur_state is
     -- ----- stav IDLE -----
     when s_idle =>
             if RQ='1' then
                     next_state <= s_wait;
             else
                     next_state <= s_idle;
             end if;
     -- ----- stav WAIT -----
     ...
     -- ----- stav DATA -----
     ...
     -- ----- stav NEXT -----
     when s_next =>
             next_state <= s_idle;
     when others =>
             null;
     end case;
end process nstate_logic;
```

Kontrolní otázky

- Jak byste popsali proces logické syntézy?
  - Co je vstupem a výstupem?
  - Jak se řeší v rámci logické syntézy optimalizace a jak funguje mapování na prvky cílové technologie?
  - Co určují uživatelská omezení (constrains)?
- Co určuje maximální dosažitelnou frekvenci obvodu?
- Jak se určí kritická cesta v obvodů?
- Dokázat určit pro zadaný obvod kritickou cestu a maximální frekvenci.
- Jaké znáte techniky zrychlení obvodů?
- Doplnit do obvodu registry, vytvořit zřetězenou linku, zkrátit kritickou cestu a zvýšit tak maximální dosažitelnou frekvenci obvodu.
- Jak popisovat základní logické prvky ve VHDL tak, aby proces syntézy obvodové prvky dokázal bez problému rozpoznat a optimalizoval.

Děkuji za pozornost!