Simulace a syntéza

Jan Kořenek, Jiří Matoušek

Brno University of Technology, Faculty of Information Technology
Božetěchova 2, 612 66 Brno
korenek@fit.vutbr.cz

Osnova

- ***Příklad analýzy obvodu***
- Simulace obvodů
- Logická syntéza
- Ukázka práce s vývojovými nástroji

Příklad analýzy obvodu

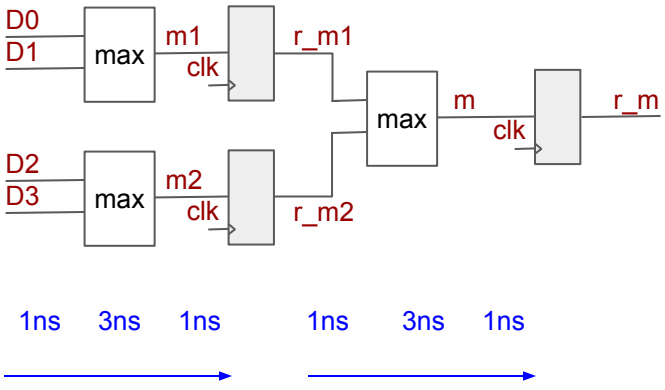
```
Architecture behv of ent_obvodu is
   signal D0,D1,D2,D3     : std_logic_vector(7  downto 0);
   signal m1,m2,m         : std_logic_vector(7  downto 0);
   signal r_m1,r_m2 , r_m : std_logic_vector(7  downto 0);
begin
   m1  <= D0  when (D0>D1)  else D1;
   m2  <= D2  when (D2>D3)  else D3;

   process (CLK, m1)
   begin
       if CLK'event AND CLK='1' then
           r_m1 <= m1;
       end if;
   end process;

   process (CLK, m2)
   begin
       if CLK'event AND CLK='1' then
           r_m2 <= m2;
       end if;
   end process;

  m <= r_m1  when (r_m1>r_m2)  else r_m2;

   process (CLK, m)
   begin
       if CLK'event AND CLK='1' then
           r_m <= m;
       end if;
   end process;

end behv;
```

Analyzujte zdrojový kód a zjistěte:

- Jaké je schéma obvodu?
- Kolik je v obvodu registrů?
- Je zpracování dat D0 až D3 řetězeno do více stupňů?
- Odvoďte maximální frekvenci obvodu, pokud zpoždění všech kombinačních obvodů je 3 ns, zpoždění na vodičích je 1 ns a setup time je 1 ns.

Příklad analýzy obvodu (řešení)

Schéma obvodu



V obvodu jsou celkem tři registry

Zpracování dat D0 až D3 je zřetězeno do dvou stupňové linky.

Maximální frekvence hodin:

$$T_{min} = t_{wire} + t_{KLS} + t_{setup} = (1+1) + 3 + 1 = 6 \text{ ns}$$

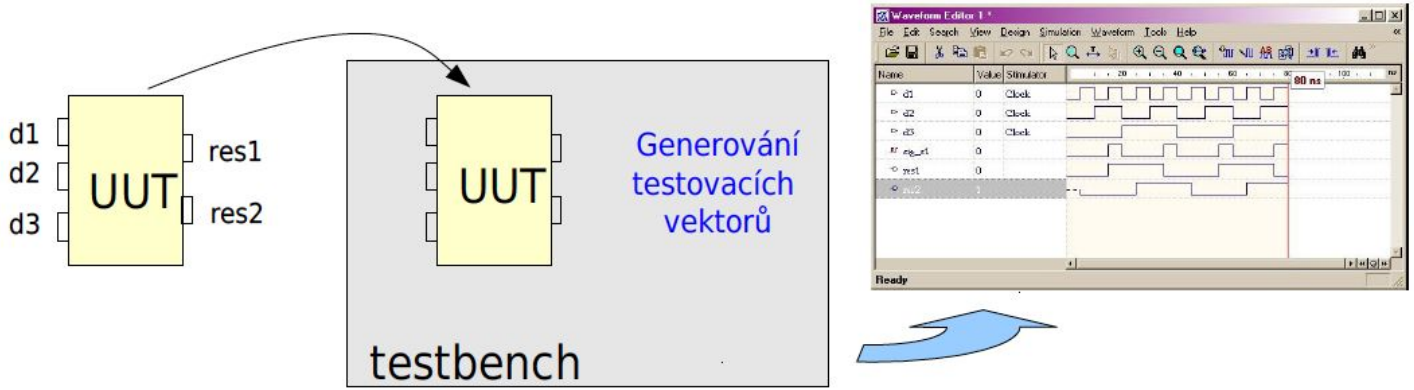
$$f_{max} = 1/T_{min} = \frac{1}{6} = 166{,}7 \text{ MHz}$$

Osnova

- Příklad analýzy obvodu
- ***Simulace obvodů***
- Logická syntéza
- Ukázka práce s vývojovými nástroji

Testbench

- Testování VHDL komponent v prostředí VHDL

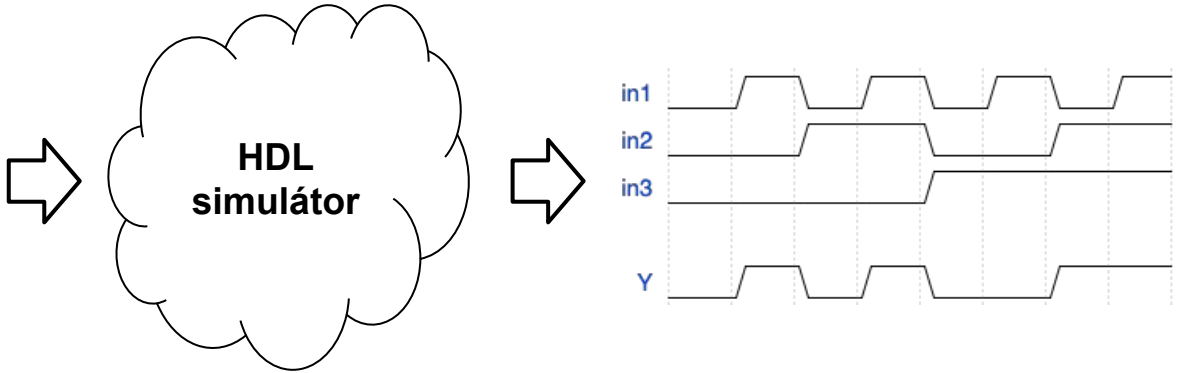


- Testbench obvykle obsahuje
  - Instanci vyvíjené komponenty označenou jako UUT (Unit Under Test)
  - Generátor testovacích vektorů
  - Monitorování a ověřování reakcí UUT

HDL simulátor

- 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;
```



Simulace obvodů

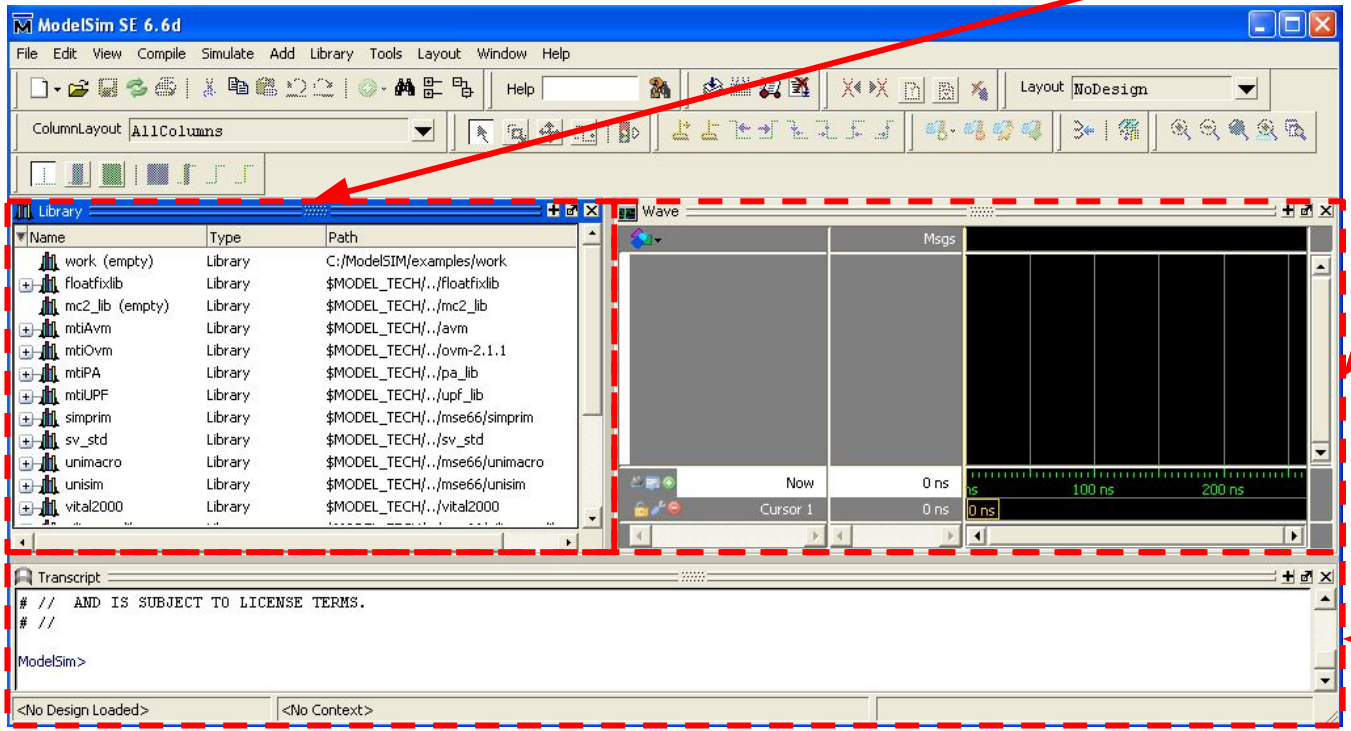
- HDL popis obvodu je možné modelovat a simulovat
- Simulací myslíme ***analýzu chování obvodu*** (jeho vnitřních prvků a výstupů) v závislosti na:
  - vstupních hodnotách (kombinační obvody), nebo
  - počátečním stavu a posloupnosti vstupních hodnot (sekvenční obvody)
- simulace probíhá ve virtuálním prostředí (testbench) a využívá model obvodu (HDL popis)
  - není třeba provádět syntézu => ***významná úspora času***

Příklady HDL simulátorů

- ***ModelSim*** (Mentor Graphics)
  - nejrozšířenější HDL simulátor v rámci vývoje pro FPGA
- VCS (Synopsis)
- NCSim (Cadence)
- *ISim*, *XSim* (Xilinx)
  - integrované v nástrojích od Xilinx

Simulátor ModelSim - základní GUI

- Spouští se příkazem **`vsim`**



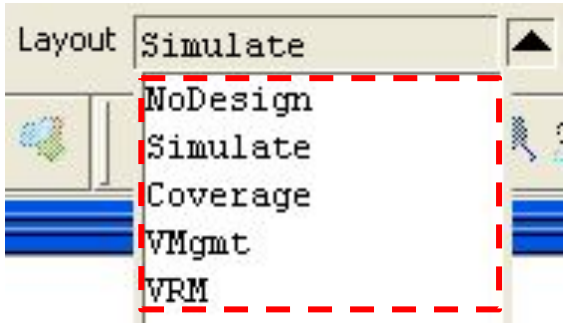
Library
Zdrojové soubory dostupné v simulaci

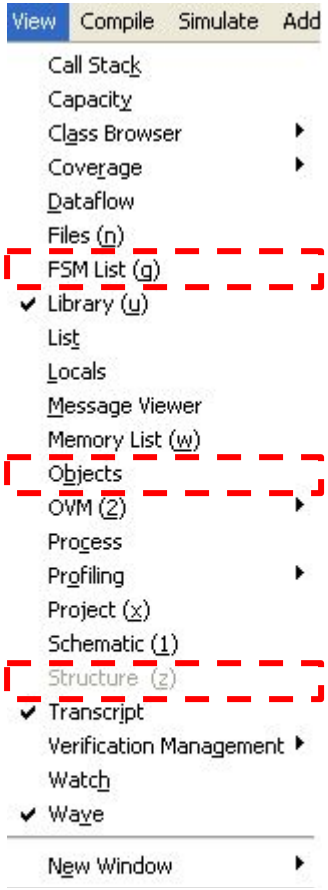
Wave
Okno časového diagramu

Transcript
Okno příkazové řádky

Simulátor ModelSim - přizpůsobení GUI

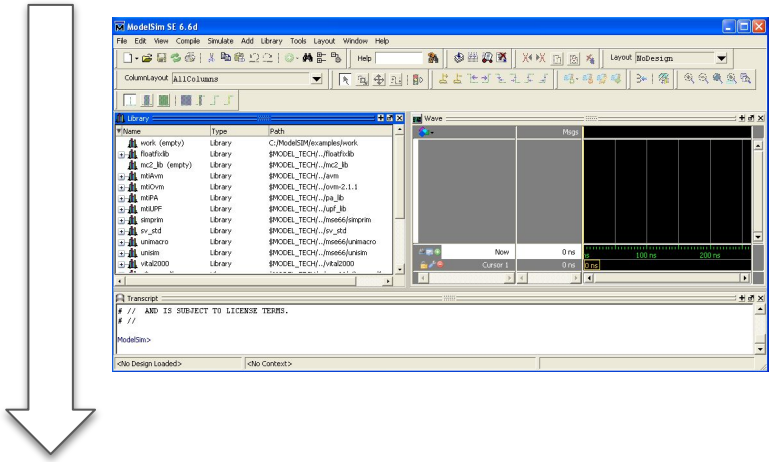
- Další užitečná okna dostupná v menu ***View***
  - FSM List
  - Objects
  - Structure
- Pro organizaci oken lze využít předdefinovaná ***rozvržení***, nebo okna uspořádat ručně





Postup činnosti:

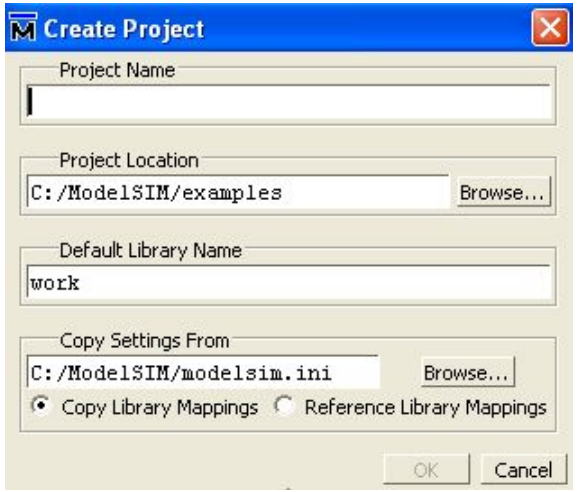
1. Vytvoření projektu
2. Kompilace zdrojových souborů
3. Inicializace simulace
4. Konfigurace časového diagramu
5. Spuštění simulace
6. Analýza výstupů simulace



- Celý postup je možné realizovat prostřednictvím ***GUI*** nebo pomocí ***příkazové řádky***

ModelSim - vytvoření projektu

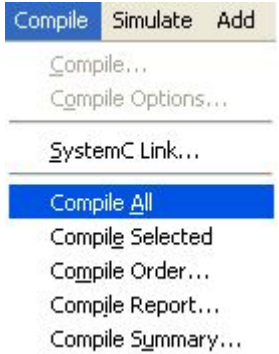
- Pomocí GUI
  - File > New > Project…



- V příkazové řádce:
  - **`project new <homedir> <name>`**
  - **`project addfile <filename>`**

ModelSim - kompilace zdrojových souborů

- Pomocí GUI
  - Compile > Compile All

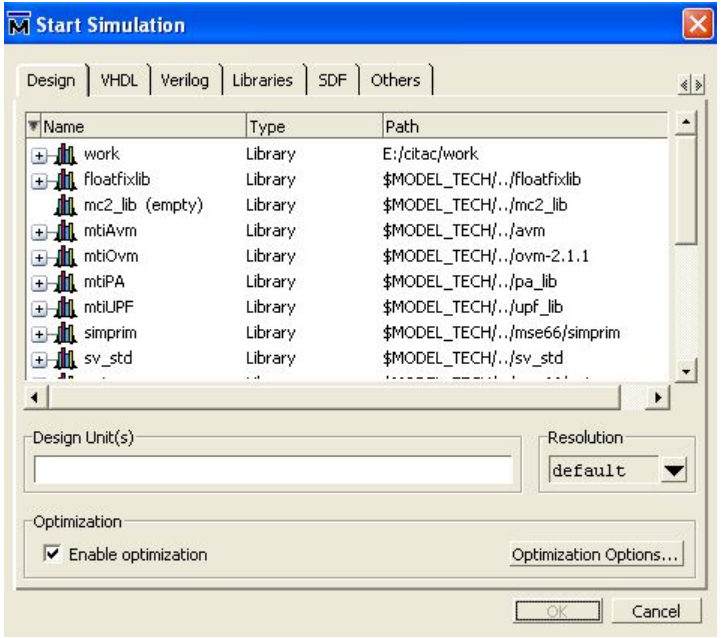


- V příkazové řádce:
  - `vcom -work <libpath> <file>` ... (VHDL soubory)
  - `vlog -work <libpath> <file>` ... (Verilog soubory)

- ***Kompilaci je potřeba provést ve správném pořadí!***

ModelSim - inicializace simulace

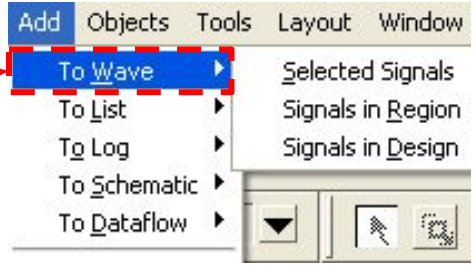
- Pomocí GUI
  - Simulate > Start Simulation…

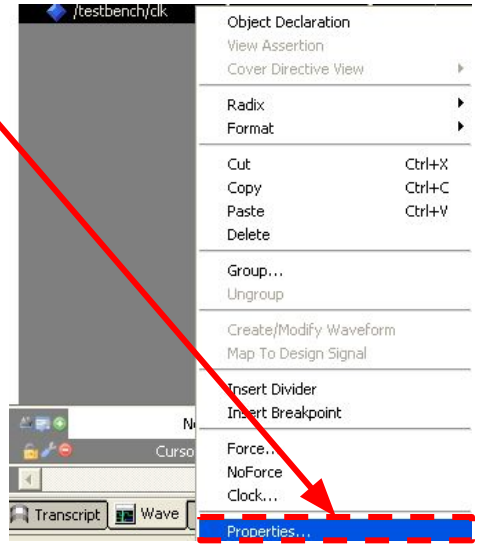


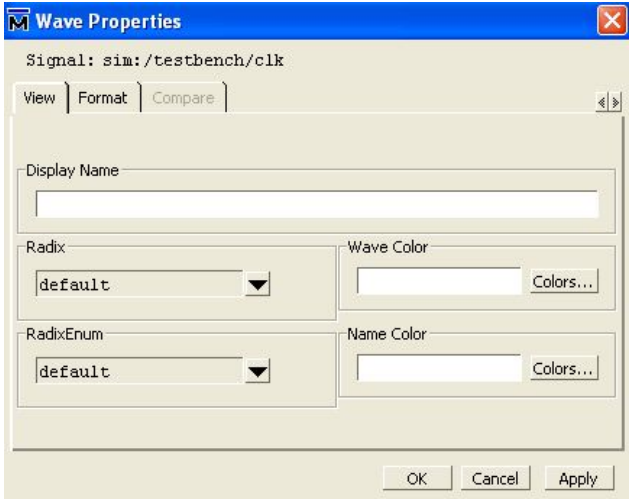
- V příkazové řádce:
  - **`vsim -t <time_step> -lib <libname> <top_level>`**
- Po inicializaci je ještě potřeba ***simulaci spustit*** (viz dále)

ModelSim - konfigurace časového diagramu

- Pomocí GUI
  - Add > To Wave > …
  - Pak můžeme konfigurovat vlastnosti signálu v okně Wave





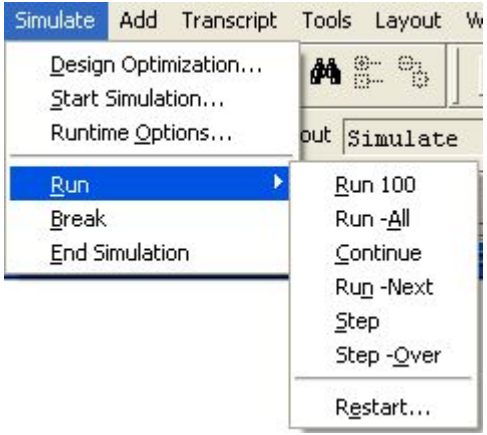


ModelSim - konfigurace časového diagramu

- V příkazové řádce:
  - **`add wave <sigpath>`** s řadou volitelných parametrů
    - **`-color <color_name>`**
    - **`-group <group_name>`**
    - **`-label <name>`**
    - **`-radix <radix_type>`**
    - **`-divider <divider_name>`**

ModelSim - spuštění simulace

- Pomocí GUI
  - Simulate > Run



- V příkazové řádce
  - `run <simulation_time>`

- V rámci simulace jsou k dispozici ***nástroje a techniky*** známé ***ze SW debuggerů*** (breakpoints, krokování, pozastavení a spuštění, …)

ModelSim - analýza výstupů simulace

- V okně časového diagramu (Wave)

- V okně příkazové řádky (Transcript)
  - např. při použití procedur z balíčku ***STD.textio***

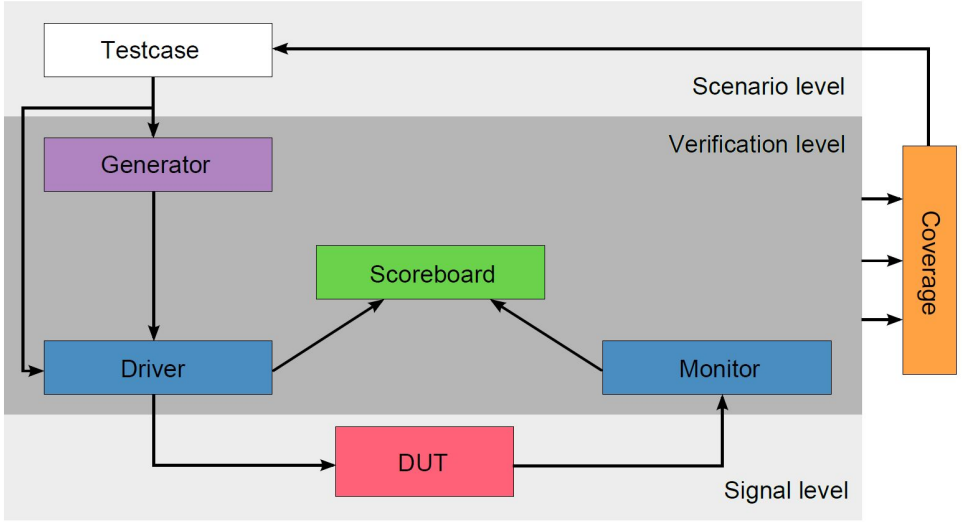
Skriptování simulace v ModelSimu

- Činnost simulátoru ModelSim lze řídit pomocí skriptů v ***jazyce Tcl***
  - de facto standardní skriptovací jazyk pro EDA (Electronic Design Automation) nástroje
- Příkazy uváděné na předchozích slajdech jsou ***validní Tcl příkazy***, i když specifické pro ModelSim
- Pro základní použití často stačí ***sekvence příkazů***
- Spouštění Tcl skriptu
  - `vsim [-c] -do <script>` z terminálu (`-c` … bez GUI)
  - `do <script>` z ModelSimu

Ukázka skriptu pro ModelSim

```
# Compile VHDL files
vcom -work work counter.vhd
vcom -work work counter_tb.vhd

# Start simulation
vsim -t 1ps -lib work testbench

# Add signals and dividers to waveform
add wave -divider "Signaly rozhrani"
add wave -color Yellow            -label CLK     /testbench/uut/clk
add wave -color Orange            -label RST     /testbench/uut/rst
add wave -divider ""
add wave                          -label CE      /testbench/uut/ce
add wave                          -label UP      /testbench/uut/up
add wave -divider ""
add wave              -radix uns -label COUNTER /testbench/uut/counter

# Run simulation
run 1 us
```

- 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



podrobnosti viz volitelný magisterský kurz
Funkční verifikace číslicových systémů

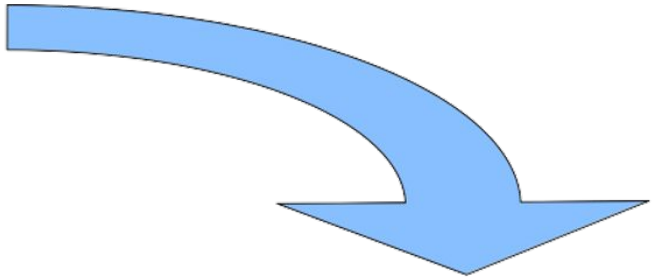
Osnova

- Příklad analýzy obvodu
- Simulace obvodů
- ***Logická syntéza***
- Ukázka práce s vývojovými nástroji

Proč potřebujeme logickou syntézu?

Transformace popisu obvodu do konfigurace FPGA nebo masky pro výrobu IC/ASIC

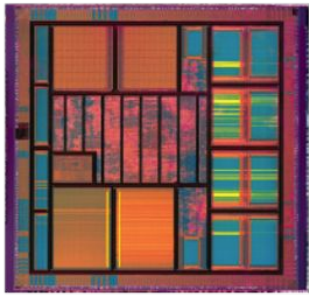
Popis obvodu



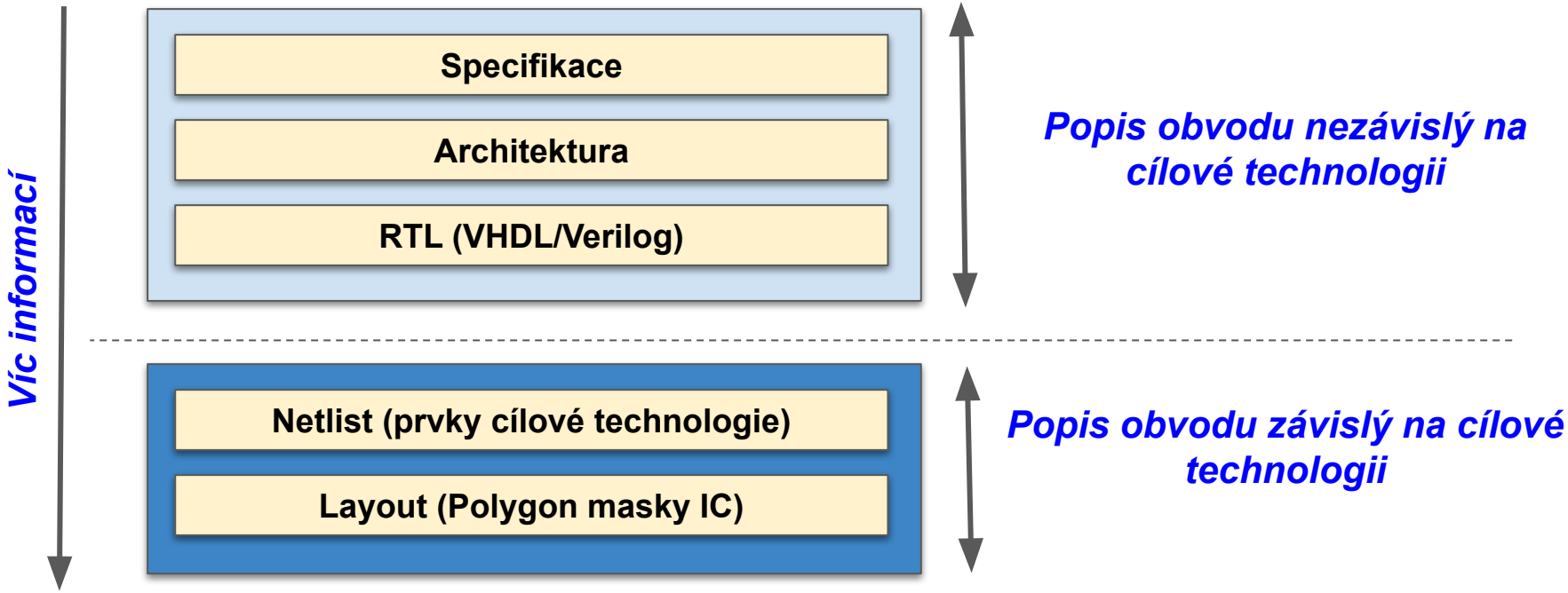
Obvod mohou být popsány na různých úrovních abstrakce

FPGA

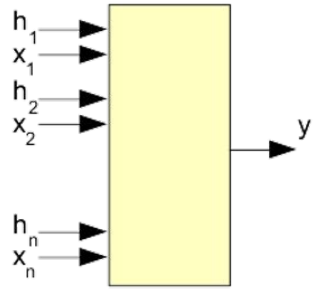
ASIC



Proč potřebujeme logickou syntézu?

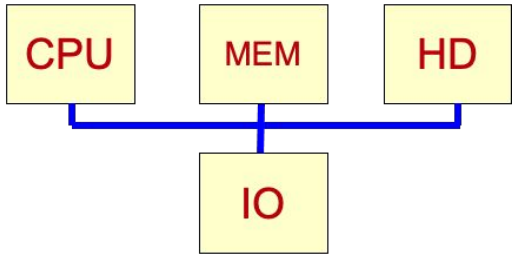


- ***Specifikace***
  - Definuje co má z uživatelského pohledu obvod dělat
  - Cíle návrhu: rychlost, spotřeba, cena, rozměry, etc.



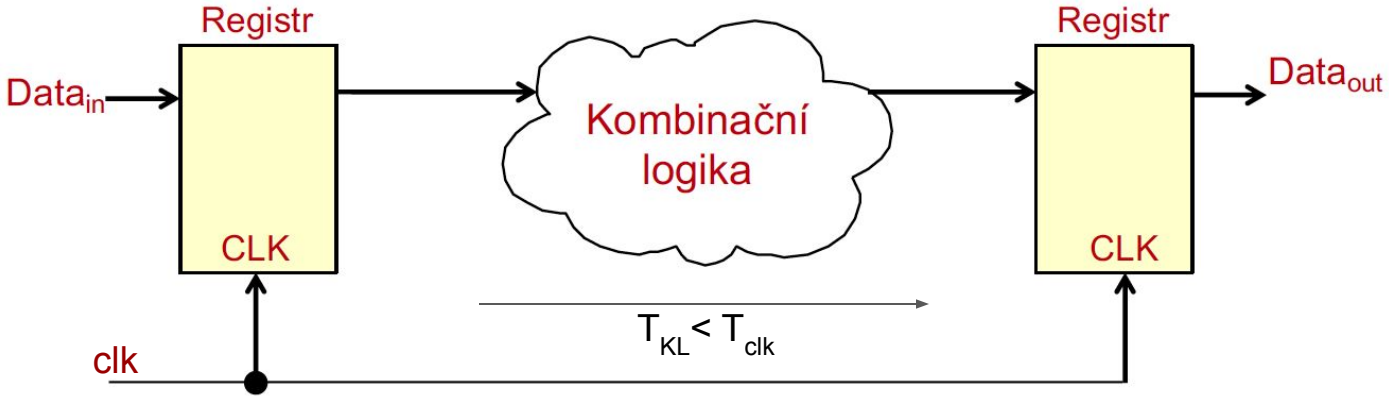
$$y = \sum x_i * h_i$$

- ***Architektura***
  - Interní struktura, jak má obvod fungovat.
  - Propojení dílčích bloků – specifikace bloků.



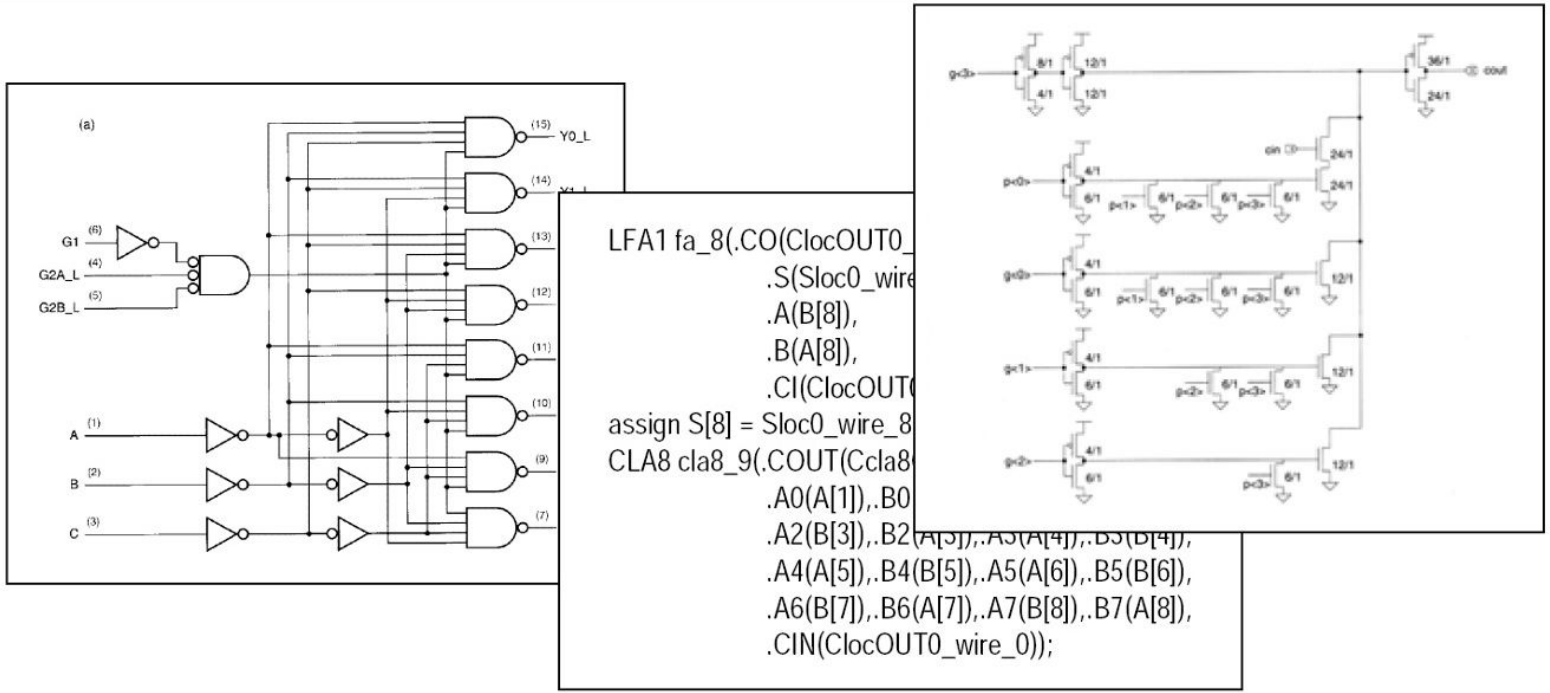
Popis obvodu na úrovni RT

- Popis synchronních obvodů řízených hodinovým signálem
- Oddělení kombinační logiky registry (paměťové prvky)
- Chování obvodu řízené hodinovým signálem



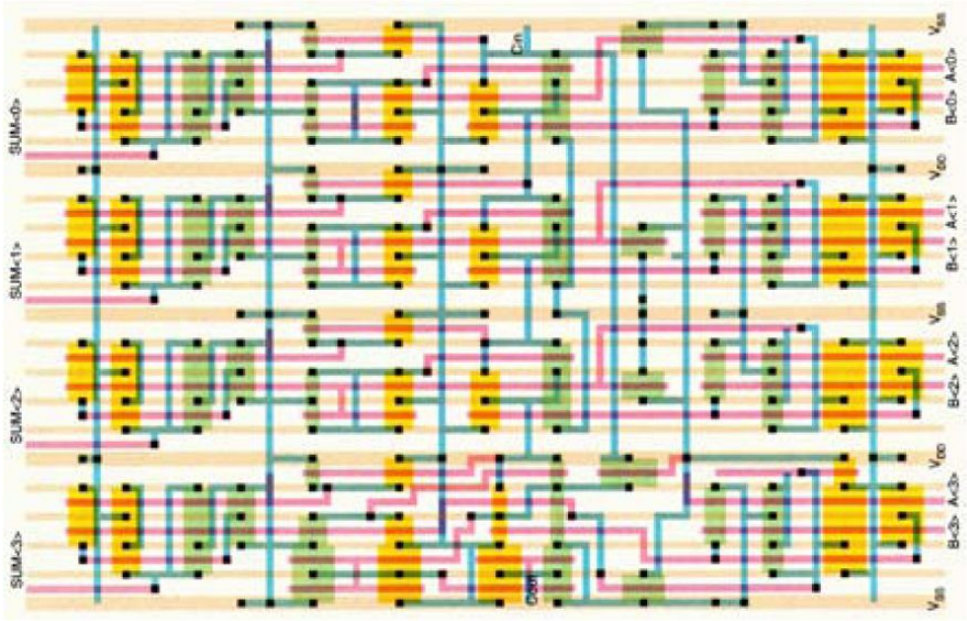
Snadná analýza chování i časových parametrů obvodu

- Detailní propojení komponent cílové technologie

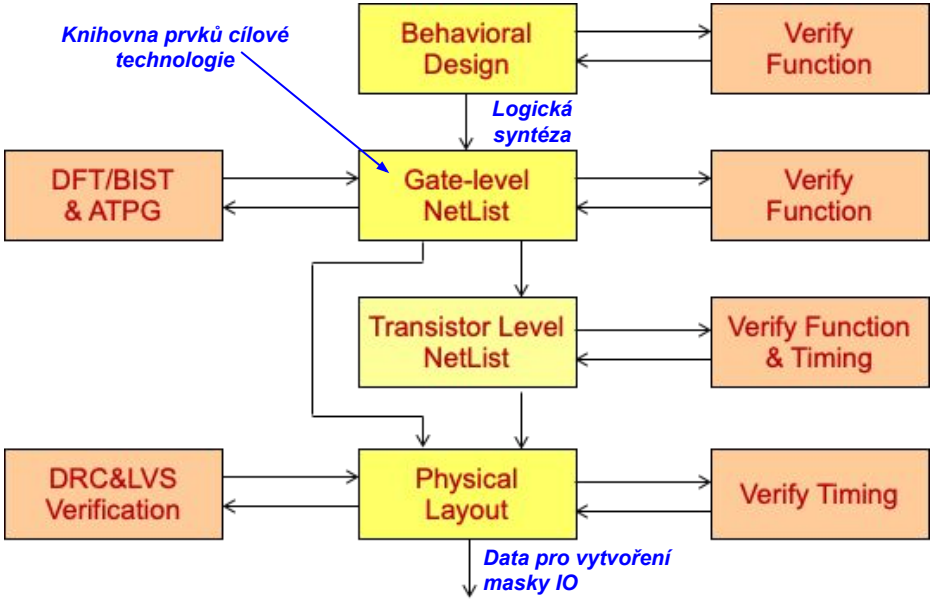


Layout

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



Proces vývoje u technologie IC/ASIC

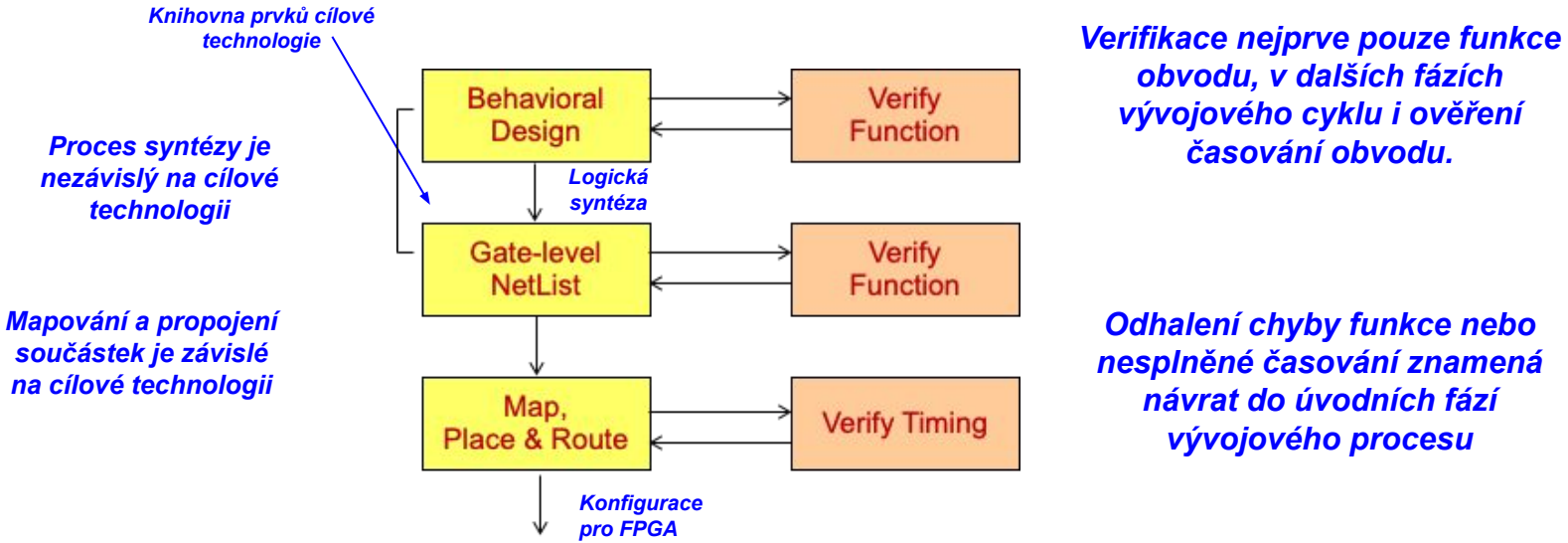


Verifikace nejprve pouze funkce obvodu, v dalších fázích vývojového cyklu i ověření časování obvodu.

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

Proces vývoje u technologie FPGA



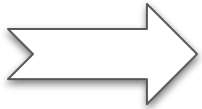
- Rozpoznání prvků cílové technologie a jejich mapování do FPGA
- Výsledkem procesu je konfigurační soubor pro FPGA

Syntéza

- ***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)



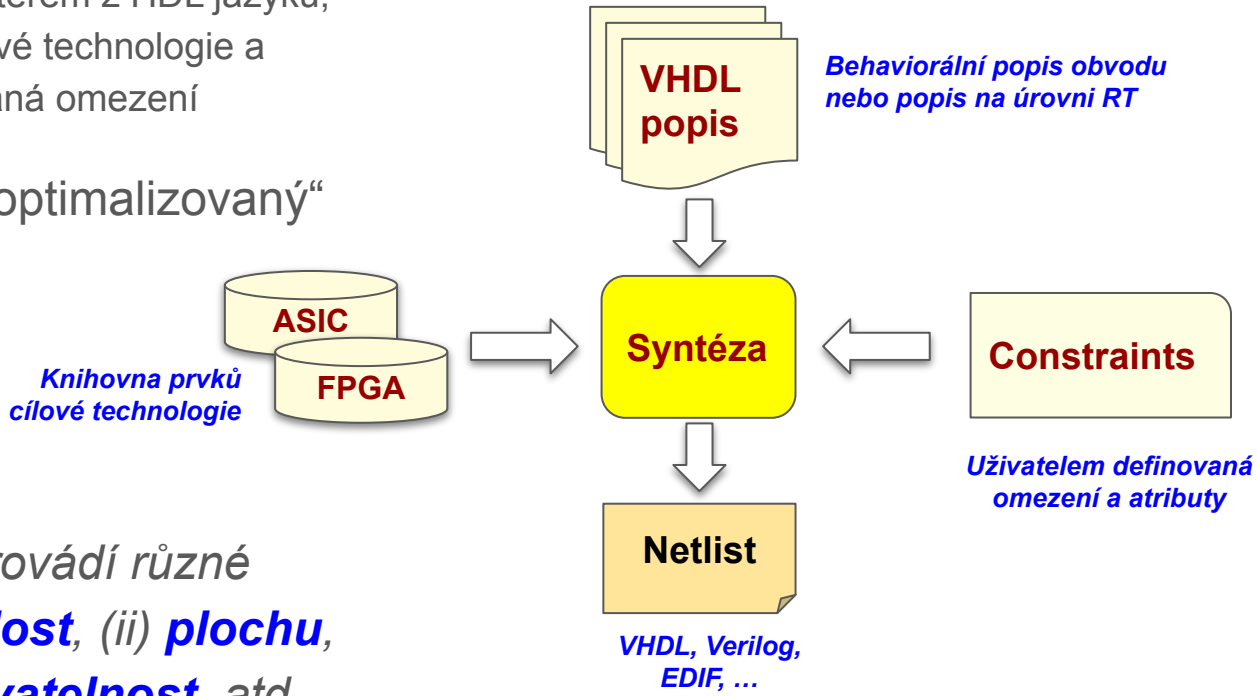
Logická syntéza

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

Vstupem syntézy je

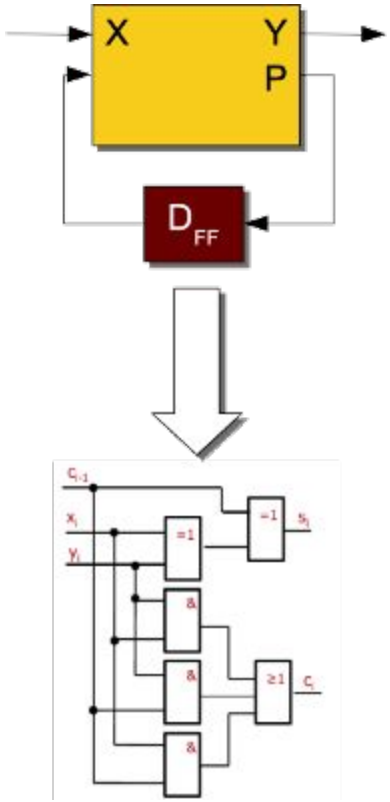
- popis obvodu v některém z HDL jazyků,
- knihovna prvků cílové technologie a
- uživatelem definovaná omezení

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.*

Co je logická syntéza?

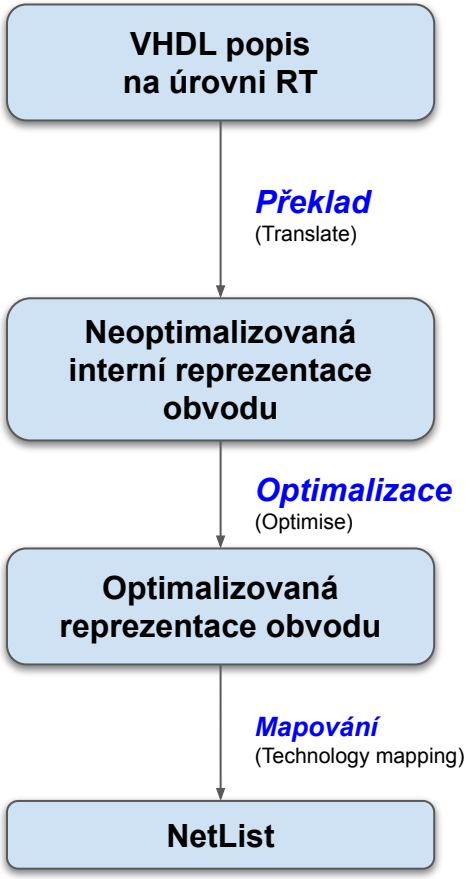


- **Vstup:** ***Finite State Machine (FSM)***
  definovaný jako (X,Y,Z,P,V), kde
  - **X** je vstupní abeceda
  - **Y** je výstupní abeceda
  - **Z** je množina vnitřních stavů
  - **P : X x Z → Z** je přechodová funkce
  - **V : X x Z → Y** je výstupní funkce
- **Výstup:** ***Obvod (G,W), kde***
  -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

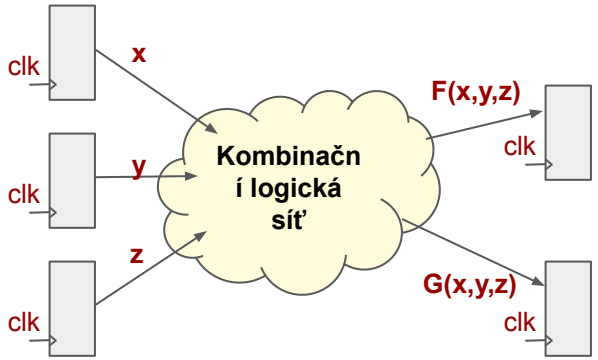
Fáze syntézy

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 teoremů 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

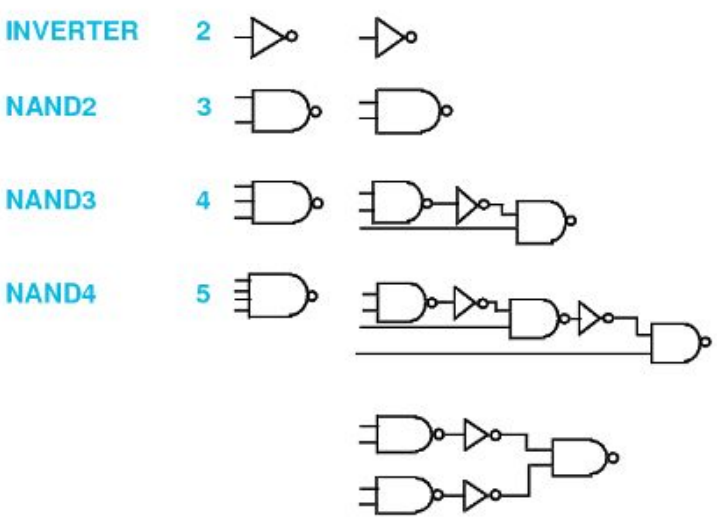
(3) Mapování do cílové technologie

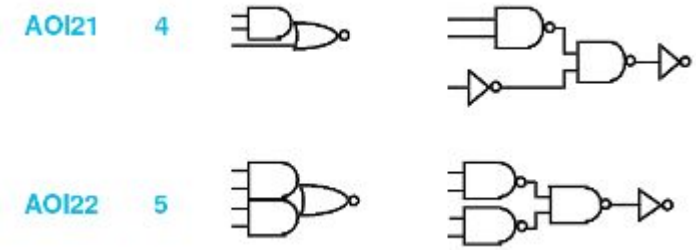
Cíl mapování: Vytvořit interní reprezentace obvodu NetList, který je složen pouze z prvků cílové technologie

- Vstupem je ***interní reprezentace obvodu ve formě acyklického orientovaného DAG grafu*** 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

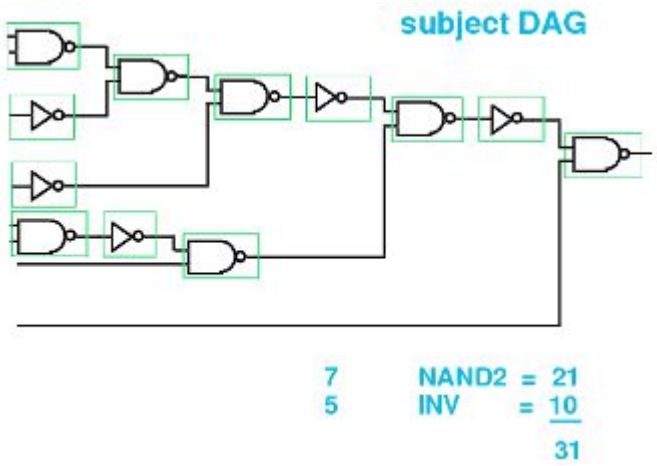
(3) Mapování do cílové technologie (Příklad)

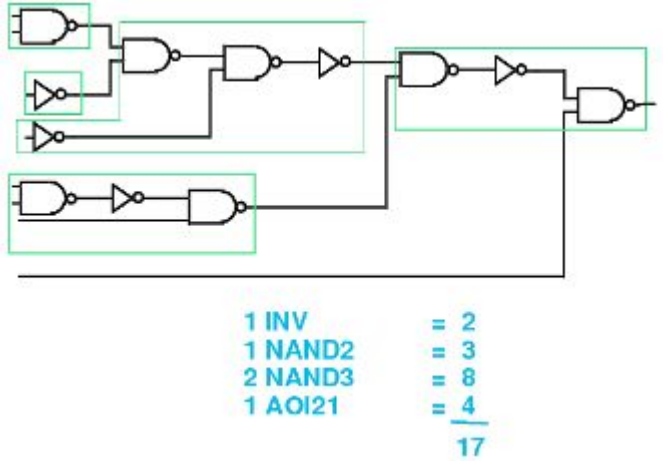
Reprezentace prvků cílové technologie pomocí DAG grafů složených pouze z prvků dvou-vstupý NAND a Invertor





(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

Knihovna prvků cílové technologie

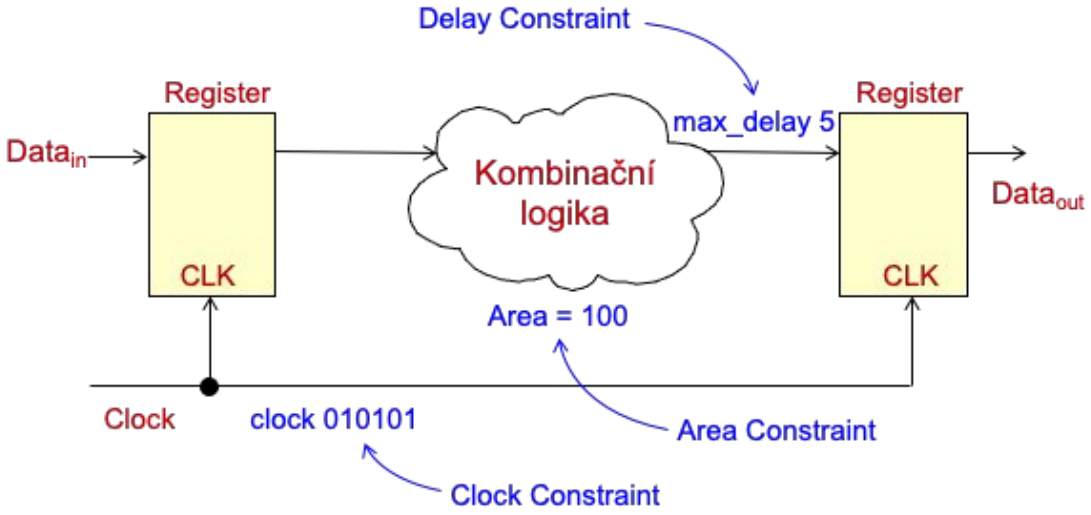
- Obsahuje informace o prvcích cílové technologie zahrnující
  - Logickou funkci ve formě DAG grafu
  - Zabranou plochu na čipu
  - Časování průchodu signálu ze vstupu na výstup
  - Omezení na fanout
  - Časová omezení

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"
   }
 }
}
}
```

Uživatelem definovaná omezení (constrains)

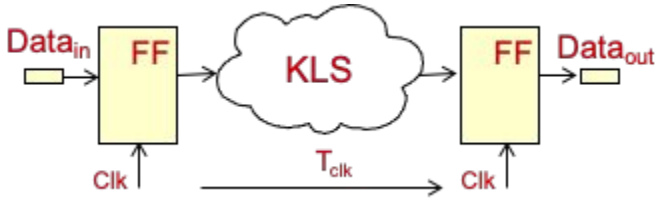
- ***Constraints*** se používají pro řízení optimalizace a mapování
- V současnosti jsou nástroji podporované uživatelská omezení zohledňující ***časování***, ***spotřebu***, ***velikost*** nebo ***umístění*** obvodu nebo části obvodu.



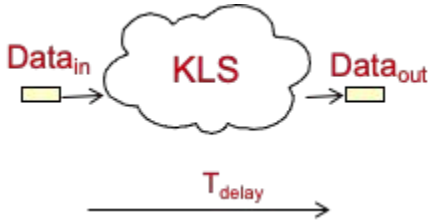
Časová omezení (timing constraints)

- ***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



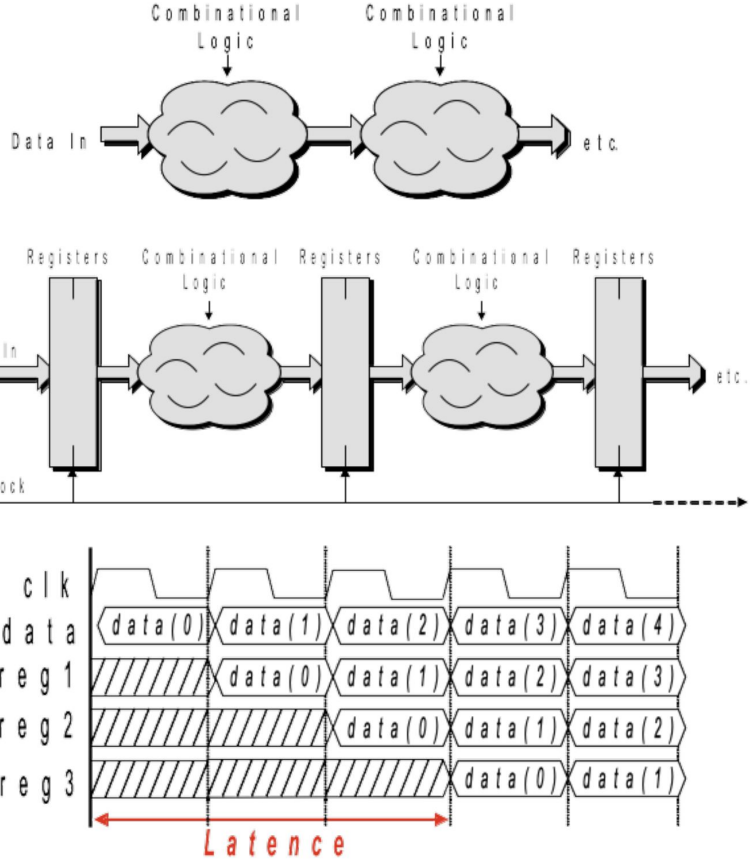
Syntézní nástroje automaticky kontrolují, jestli jsou splněna omezení zadaná uživatelem

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

Zřetězené zpracování (Pipelining)

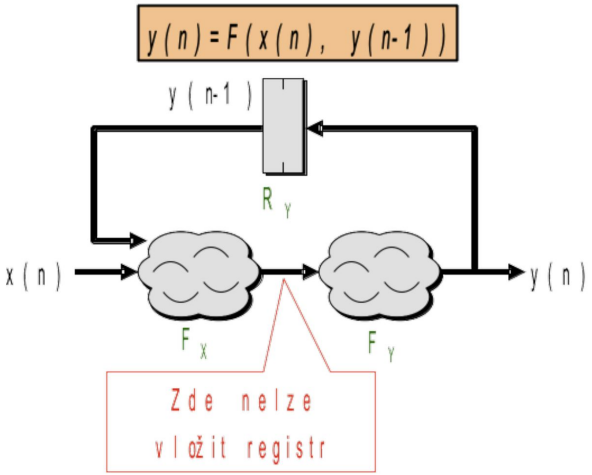
Základní princip:

- 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***



Zřetězené zpracování (Pipelining)

- ***Pipeline nelze použít v obvodech, kde je použita zpětná vazba!***
- Vložením registru vzniká zpoždění, které naruší zpětnou vazbu (vzniká problém s časováním)



Správná funkce obvodu

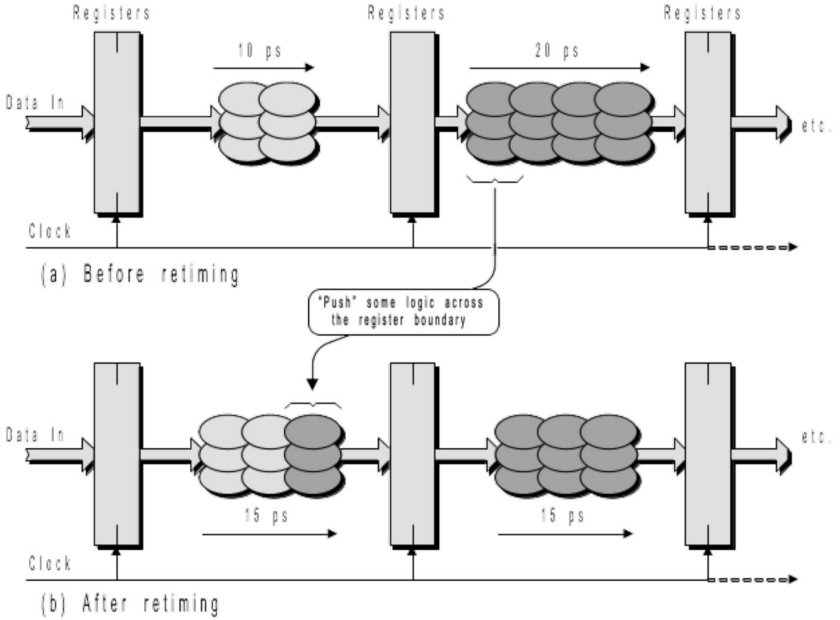
X	X_0	X_1	X_2
Y	F_{XY0} Y_0	F_{XY1} Y_1	F_{XY2} Y_2
R Y	-	Y_0	Y_1

Po vložení registru

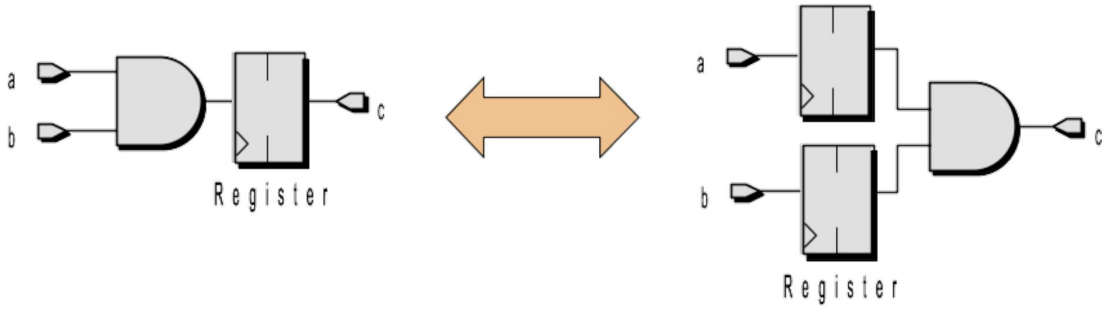
X	X_0 F_{X0}	X_1 F_{X1}	X_2 F_{X2}
R F_X	.	F_{X0}	F_{X1}
Y	.	F_{XY0} Y_0	F_{XY1} Y_1
R Y	-	-	Y_0

Retiming

- ***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***



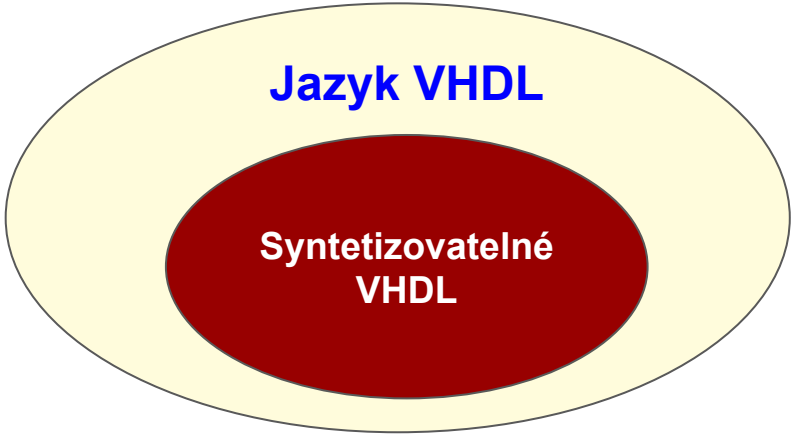
- ***Základní pravidlo metody Retiming***
  - registr na výstupu kombinačního obvodu může být přesunut před tento prvek, pokud je aplikován na všechny jeho vstupy
  - podobným způsobem registr může být přesunut ze vstupu kombinačního obvodu na výstup, pokud je přesunut ze všech jeho vstupních portů
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



VHDL konstrukce při syntéze

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

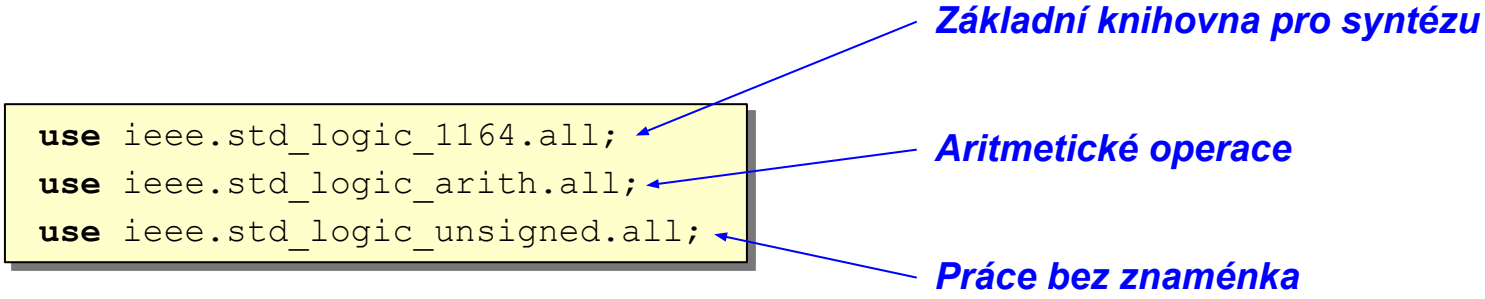
- Problematické konstrukce
  - Čtení nebo zápis ze souboru
  - Rozsah smyček a generických instancí musí být konstantní
  - Inertní nebo transportní zpoždění definované pomocí příkazu wait
  - Příkazy pro ověřování funkce komponent – assert, report, ...



Ne všechny konstrukce napsané ve VHDL jsou podporovány syntézními nástroji!

Knihovny a standard pro syntézu

- 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
- …

Nežádoucí vznik Latch registrů

- ***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)

Nekompletní sensitivity list

- ***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!

Dynamická délka iterace a příkaz generate

- 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

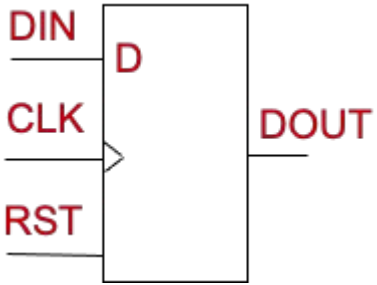
Převod binárního čísla do kódu 1 z n

Šablony obvodů pro logickou syntézu

- ***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.

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;
```

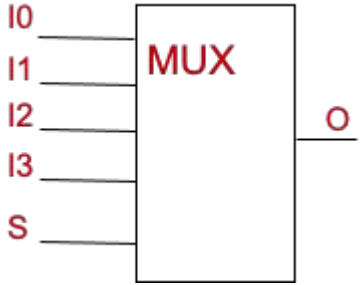


CLK (clock) – hodinový vstup
RST(reset) – asynchronní reset
DIN (data in) – data přivedená na vstup registru
DOUT (data output) – hodnota uložená v registru

Asynchronní reset
(není synchronizován s hodinovým signálem)

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

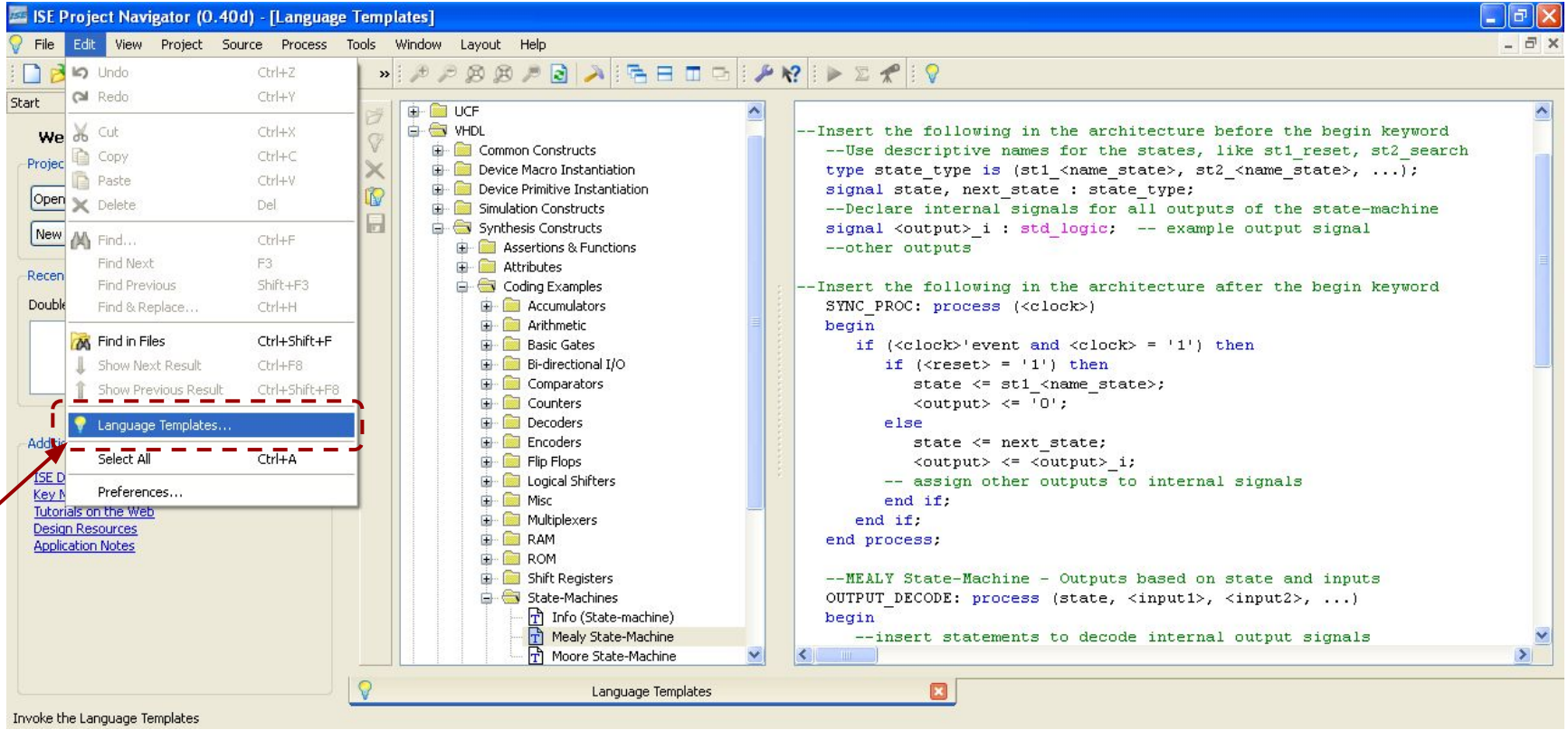
Automat (Finite State Machine)

```
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;
```

Další syntetizovatelné šablony



Konec přednášky

Děkuji za pozornost