Návrh obvodů a jazyk VHDL

Jan Kořenek

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

Osnova

- ***Mapování základních konstrukcí do hardware***
  - Sekvence
  - Selekce
  - Iterace
- Jazyk VHDL
- Analýza popisu obvodů

Proč použít hardware?

- Většinu aplikací je možné realizovat čistě softwarově
- Hardware umožňuje dosáhnout lepších parametrů
  - Vyšší rychlost zpracování a tím i ***vyšší propustnost***
  - ***Nižší spotřeba*** a delší běh aplikace na baterie
  - Rychlejší reakce na nastalou událost, dosažení ***nižší latence***
  - Miniaturizace výsledného řešení
  - …

Softwarové řešení

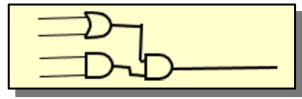
```
void main(){
    ...
    return 0;
}
```

CPU

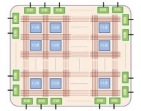
Identifikace kritických částí pro HW zpracování (propustnost, latence, spotřeba, …)

Softwarové řešení s hardwarovou podporou

```
void main(){
    …
    hw_proc(...);
    …
    return 0;
}
```



CPU + HW



FPGA/ASIC

Realizace algoritmu

Příklady algoritmů

- Hledání řetězců
- Šifrování dat
- Filtrace obrazu
- …

Algoritmus

Algorimuts je sestaven pomocí

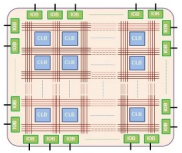
- **Datových struktur** – proměnné, záznamy, pole, lin. seznamy, apod.
- **Řídících struktur** – sekvence, podmínka, iterace

Implementace pomocí programu


```
void main(){
    ...
    return 0;
}
```

- Řízení programem
- Sekvenční zpracování programu na jednom nebo více procesorových jader

Implementace pomocí hardwarové architektury



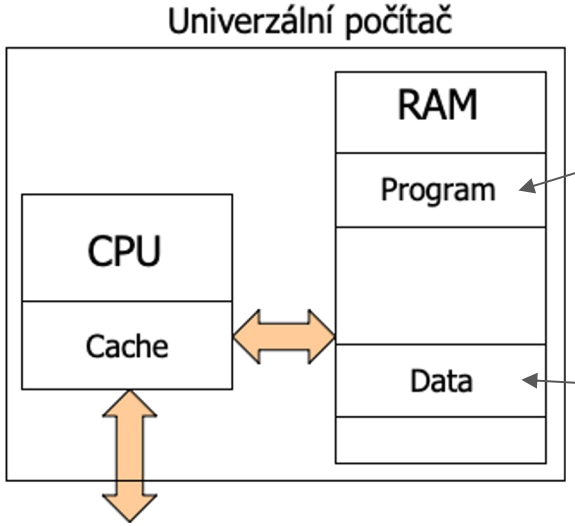
- Chování je dáno zapojením obvodových prvků, které pracují paralelně
- Základní obvodové prvky se liší podle použité technologie – ASIC, FPGA, …

Softwarová realizace algoritmu

- Výpočet běží na ***univerzálním procesoru (CPU)***
- ***Datové struktury*** i předpis programu jsou uloženy ***v paměti RAM***
- Na základě lokality jsou data i program přesouvány mezi pamětí RAM a interní pamětí cache procesoru
- ***Vstupy/Výstupy*** – dodávány skrze V/V zařízení (disk, monitor, porty, apod.)

Algoritmus

```
int a, b, max;
if(a>b)
    max = a;
else
    max = b;
```



Program

```
if(a>b)
    max = a;
else
    max = b;
```

Data

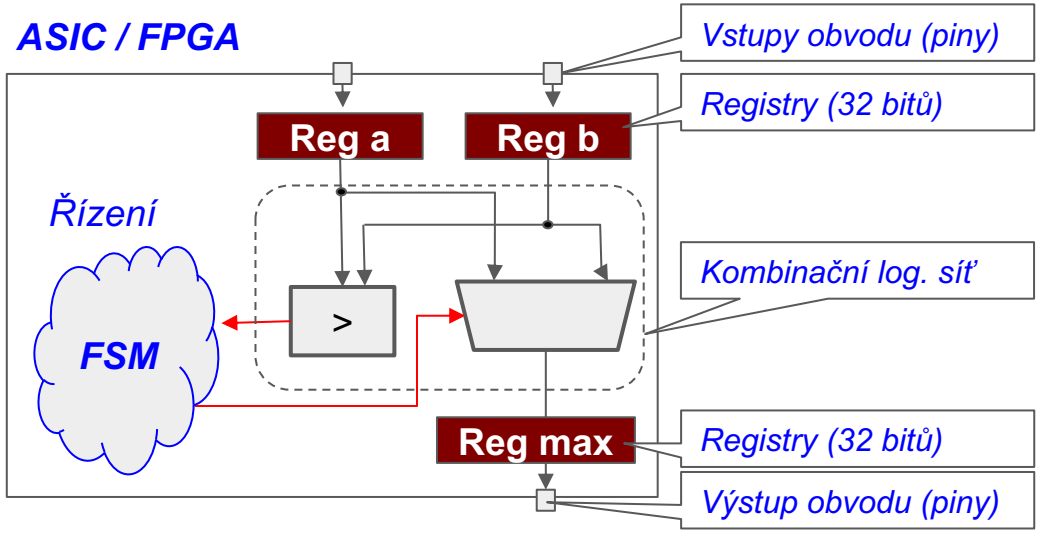
```
int a, b, max;
```

Hardwarová realizace algoritmu

- Výpočet běží na ***specializovaném čipu*** (obvykle ***ASIC*** nebo ***FPGA***)
- ***Datové struktury*** – uloženy v registrech nebo paměťových blocích
- ***Řídící struktury*** – výpočet realizovat skrze ***datovou cestu (funkční jednotky)***, která je ovládána ***řadičem (FSM)***
- ***Vstupy/výstupy*** – dostupné skrze vstupní/výstupní piny obvodu

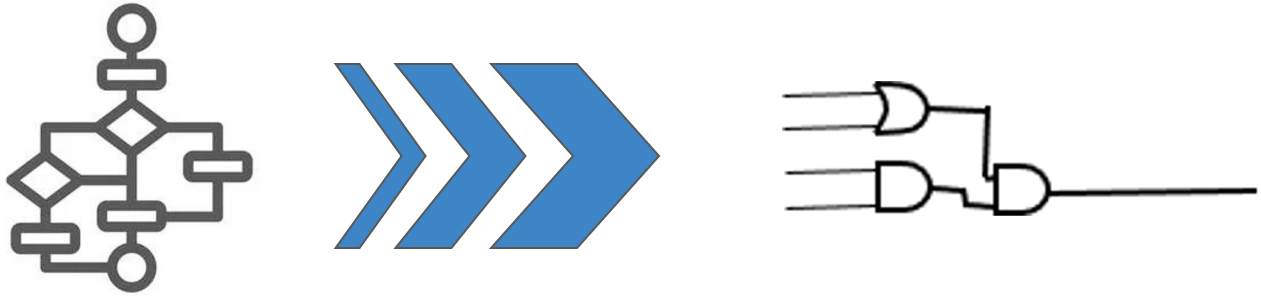
Algoritmus

```
int a, b, max;
if(a>b)
    max = a;
else
    max = b;
```



Od algoritmu k číslicovému obvodu

- Dijkstra dokázal, že libovolný algoritmus lze zaznamenat skrze kombinaci tří řídících struktur:
  - **Sekvence (posloupnost)**
  - **Selekce (podmínka)**
  - **Iterace (cyklus)**
- Pojďme si ukázat, jak lze tyto struktury realizovat v hardware

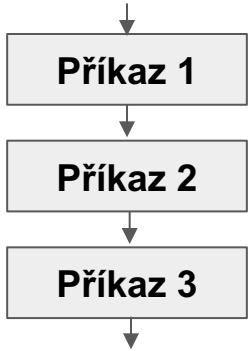


Osnova

- ***Mapování základních konstrukcí do hardware***
  - Sekvence
  - Selekce
  - Iterace
- Jazyk VHDL
- Analýza popisu obvodů

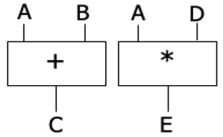
Hardwarová realizace sekvence příkazů

- ***Sekvence příkazů*** je řada za sebou navazujících kroků - jednotlivé kroky jsou definovány pomocí příkazů
- Jak spouštět sekvenci příkazů v HW?



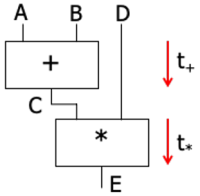
Příklad

```
C = A + B;
E = A * D;
```



Příklad

```
C = A + B;
E = C * D;
```



- Datová závislost
  - Dvojice příkazů je ***datově závislá***, pokud ***vstupní parametr jednoho příkazu závisí na výstupu druhého příkazu*** nebo naopak. Jinak jsou příkazy datově nezávislé.

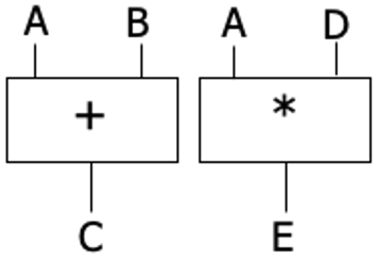
Vykonání sekvence příkazů (bez datové závislosti)

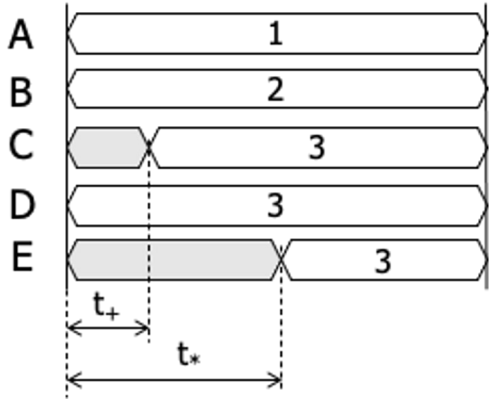
- Doba pro vykonání datové nezávislých příkazů je dána maximem dob jednotlivých příkazů (bloků)

Příklad

Bez datové závislosti

```
C = A + B;
E = A * D;
```





- Celková doba výpočtu **$t = max(t_+, t_*)$**
- V reálném obvodu je nutné uvažovat také zpoždění vodičů
- Vstupní *signály A, B, D musí být stabilní* po celou dobu výpočtu ***t***
- Dokud se nedokončí výpočty, není na výstupech *C* a *E* správná hodnota

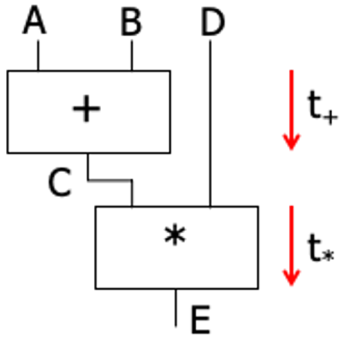
Vykonání sekvence příkazů (s datovou závislostí)

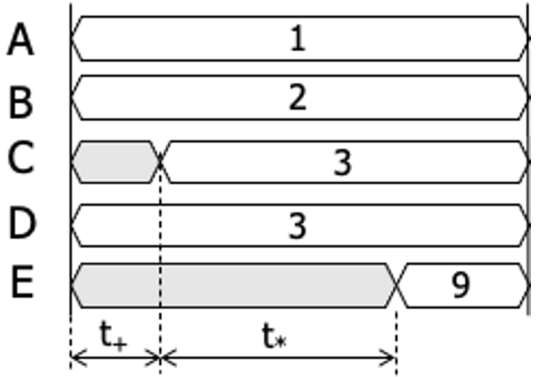
- Doba pro vykonání datové závislých příkazů je dána součtem dob jednotlivých příkazů (bloků)

Příklad

Datová závislost

```
C = A + B;
E = C * D;
```



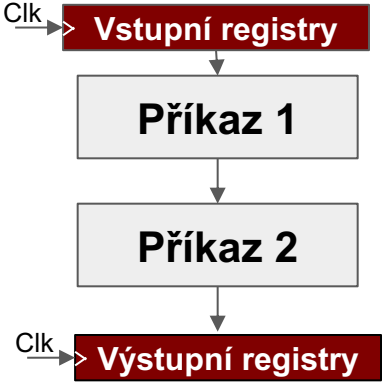


- Celková doba výpočtu $\mathbf{t = t_+ + t_*}$
- V reálném obvodu je nutné uvažovat také zpoždění vodičů
- Vstupní *signály A, B, D musí být stabilní* po celou dobu výpočtu ***t***
- Dokud se nedokončí výpočty, není na výstupu *E* správná hodnota

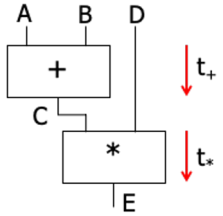
Zpracování sekvence příkazů (synchronní návrh)

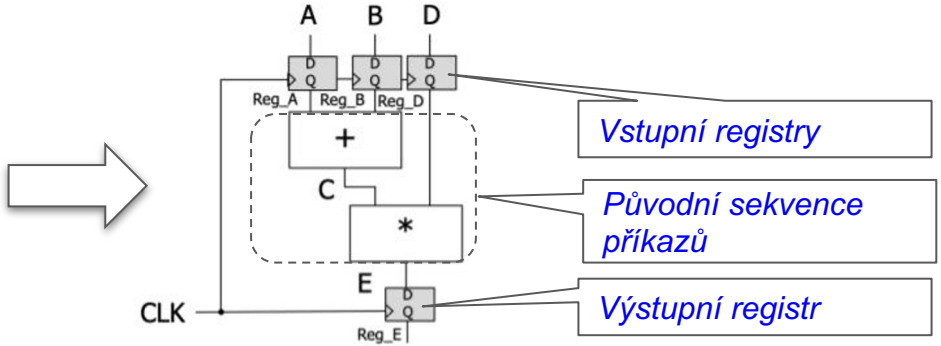
- Jak udržet vstupní parametry stabilní po celou dobu výpočtu?
- Kdy bude vypočítaný výsledek?
- Jak ve správný okamžik uložit výsledek?

- Synchronní návrh umožňuje zajistit správné časování obvodu
  - Všechny vstupní signály jsou stabilní po celou periodu hodin
  - V průběhu výpočtu můžou být na výstupech zákmity. Správná hodnota je vzorkována až v okamžiku náběžné hrany hodin, kdy už jsou na výstupu ustálené hodnoty.



Jaká musí být perioda hodin CLK?





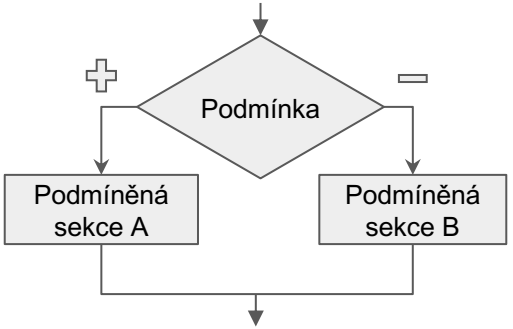
Osnova

- ***Mapování základních konstrukcí do hardware***
  - Sekvence
  - Selekce
  - Iterace
- Jazyk VHDL
- Analýza popisu obvodů

Selekce

- Pomocí selekce je možné ovládat průběh vykonávání jednotlivých příkazů algoritmu
- Existují různé varianty
  - Selekce s jednou podmíněnou sekcí (if)
  - Výběr mezi dvojicí podmíněných sekcí (if else)
  - Výběr mezi několika podmíněnými sekcemi (switch)
- ***Realizace na úrovni software***
  - (i) Nejprve je vyhodnocena podmínka a
  - (ii) až potom jsou vykonány příkazy ve zvolené sekci
- ***Jakým způsobem je možné udělat HW realizaci?***

if(Podmínka) A else B;



- ***Realizace na úrovni hardware***
  - Všechny podmíněné sekce je možné vykonávat paralelně včetně vyhodnocení podmínky
  - Na závěr se pomocí multiplexoru vybírá požadovaný výstup z podmíněné sekce, která byla vybrána na základě podmínky

Příklad

absolutní hodnota z A a B

```
if (A > B)
   abs = A-B;
else
   abs = B-A;
```

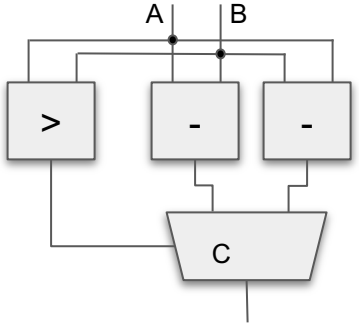
Podmínka

Sekce B

Sekce A

Paralelní výpočet

Dvě funkční jednotky



Doba výpočtu

$t = \max(t_{>};t_{-}) + t_{mx}$

HW Zdroje

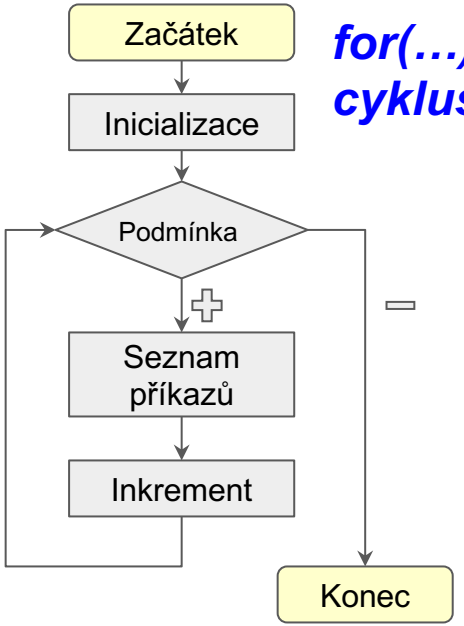
- ➔ 2x odčítačka,
- ➔ 1x multiplexor,
- ➔ 1x komparátor

Osnova

- ***Mapování základních konstrukcí do hardware***
  - Sekvence
  - Selekce
  - Iterace
- Jazyk VHDL
- Analýza popisu obvodů

Iterace

- Umožňuje opakování určité části algoritmu pomocí cyklu
- Varianty cyklů
  - ***for*** s pevným počtem opakování (iterací)
  - ***while*** a ***do while*** s proměnným počtem opakování (iterací)
- Z čeho se cyklus skládá?
  - Inicializační část
  - Podmínka ukončení cyklu
  - Seznam příkazů
  - Inkrement



for(…) cyklus

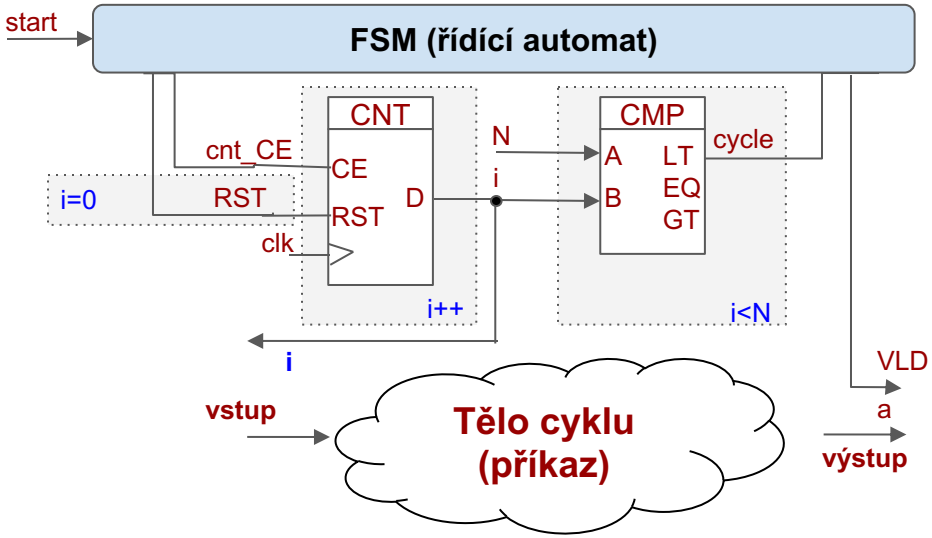
Příklad pro smyčku FOR

```
for(inicializace; podmínka; inkrement)
{ seznam příkazů }
```

- Jak cyklus realizovat na úrovni HW?

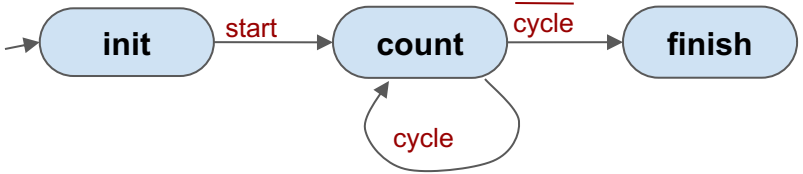
- Zpracování dat v cyklu

```
for (i=0;i<N;i++){
   příkaz;
}
```



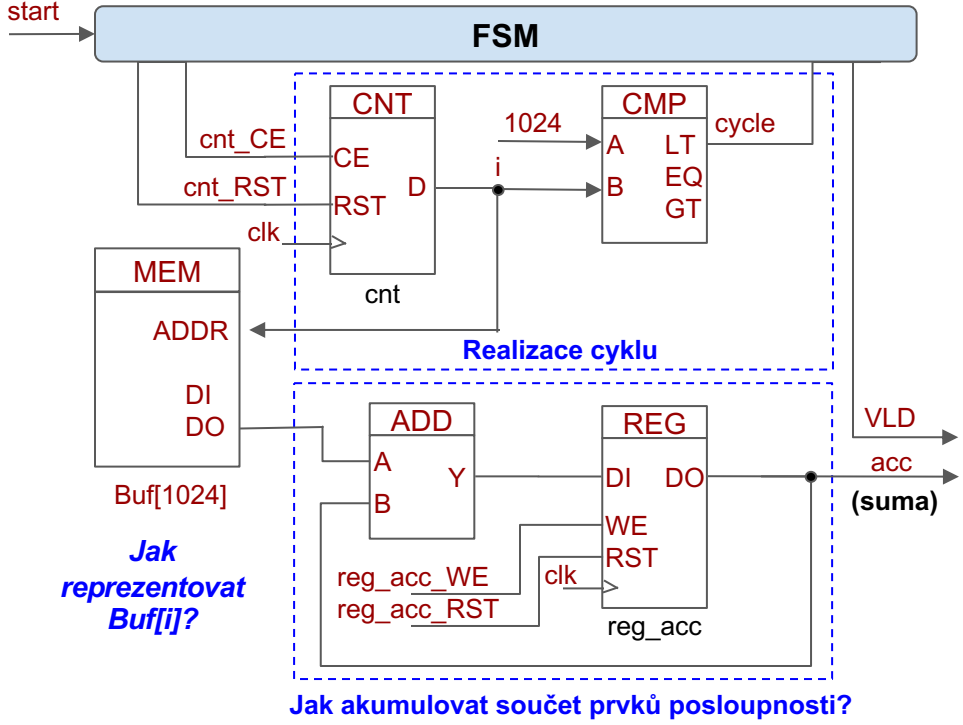
Činnost obvodu

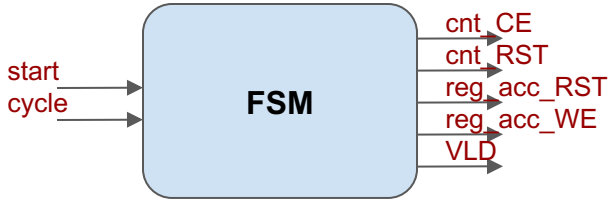
- Před začátkem cyklu je potřeba ve stavu ***init*** inicializovat proměnou cyklu *i (i=0)*. Čítač je inicializován signálem reset *(RST=1)*
- Cyklus je startován signálem start *(start=1)*, kdy obvod přechází do stavu ***count***
- Ve stavu ***count*** v každém cyklu hodin (tiku) je inkrementován čítač *(i++)* signálem *cnt_CE* a je provedeno tělo cyklu
- Komparátor kontroluje podmínku ukončení cyklu *(i<N)* a v případě nesplnění podmínky je cyklu ukončen - přechod do stavu ***finish***



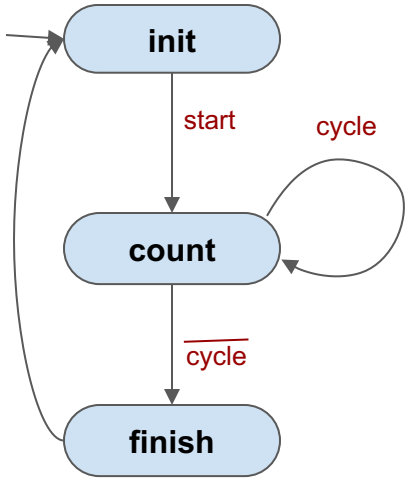
Příklad smyčky FOR: sečtení prvků posloupnosti

```
for (i=0;i<1024;i++)
   acc=acc+Buf[i];
```





Stavový automat




```
init:
     cnt_RST=1;
     reg_acc_RST=1;
```

```
count:
     cnt_CE=1;
     reg_acc_WE=1;
```

```
finish:
     VLD=1;
```

Pozn: Řízení signálů na základě aktuálního stavu. Pokud není některý výstup ve stavu nastaven, má nulovou hodnotu.

Příklad smyčky FOR: sečtení prvků posloupnosti

```
for (i=0;i<1024;i++)
   acc=acc+Buf[i];
```

- Vstupní pole ***Buf[1024]*** je uloženo v paměti RAM nebo ROM
- Před začátkem výpočtu je potřeba inicializace, která je řešena stavem ***init***:
  - Vynulovat proměnnou cyklu ***i*** signálem (cnt_RST = 1)
  - Vynulovat obsah registru ***reg_acc*** signálem (reg_acc_RST = 1)
- Ve stavu ***count*** probíhá tělo cyklu a v každém cyklu (tiku) hodinového signálu:
  - je inkrementovaná proměnná cyklu ***i*** pomocí čítače (cnt_CE = 1). Hodnota čítače současně slouží jako adresa do paměti ROM, odkud se přečte prvek pole indexovaný proměnou ***i***
  - Prvky posloupnosti čtené z paměti jsou postupně akumulovány v registru ***reg_acc***
- Celý cyklus končí v okamžiku nesplnění podmínky cyklu ***i < 1024*** signál (cycle = 0), obvod přechází do stavu ***finish*** a v registru ***reg_acc*** je výsledek

Rozbalení smyčky do paralelního zpracování

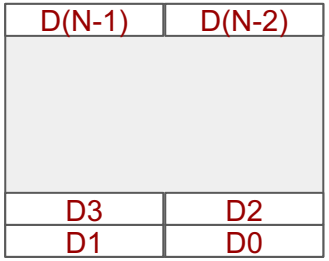
```
for (i=0;i<N;i++)
   a = a + Buf[i];
```

N řádků (iterací)

Paralelní zpracování těla smyčky

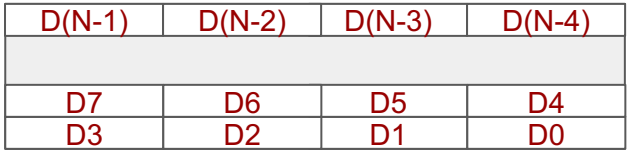
```
for (i=0;i<N/2;i++){
   a = a + Buf[i].D0
         + Buf[i].D1;
}
```

N/2 řádků (iterací)



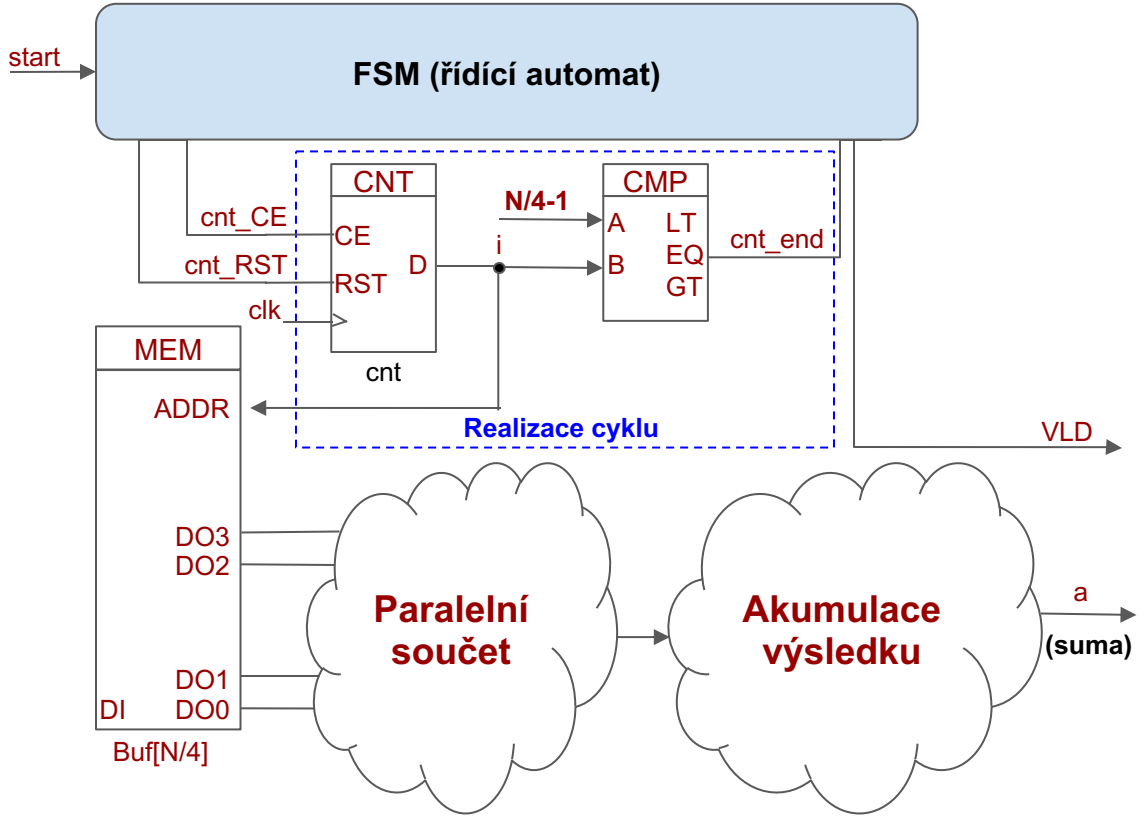

```
for (i=0;i<N/4;i++){
   a = a + Buf[i].D0;
   a = a + Buf[i].D1;
   a = a + Buf[i].D2;
   a = a + Buf[i].D3;
}
```

N/4 řádků (iterací)

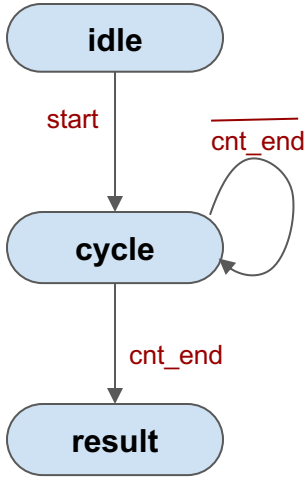


Paralelní čtení dat → paralelní zpracování dat → redukce počtu cyklů

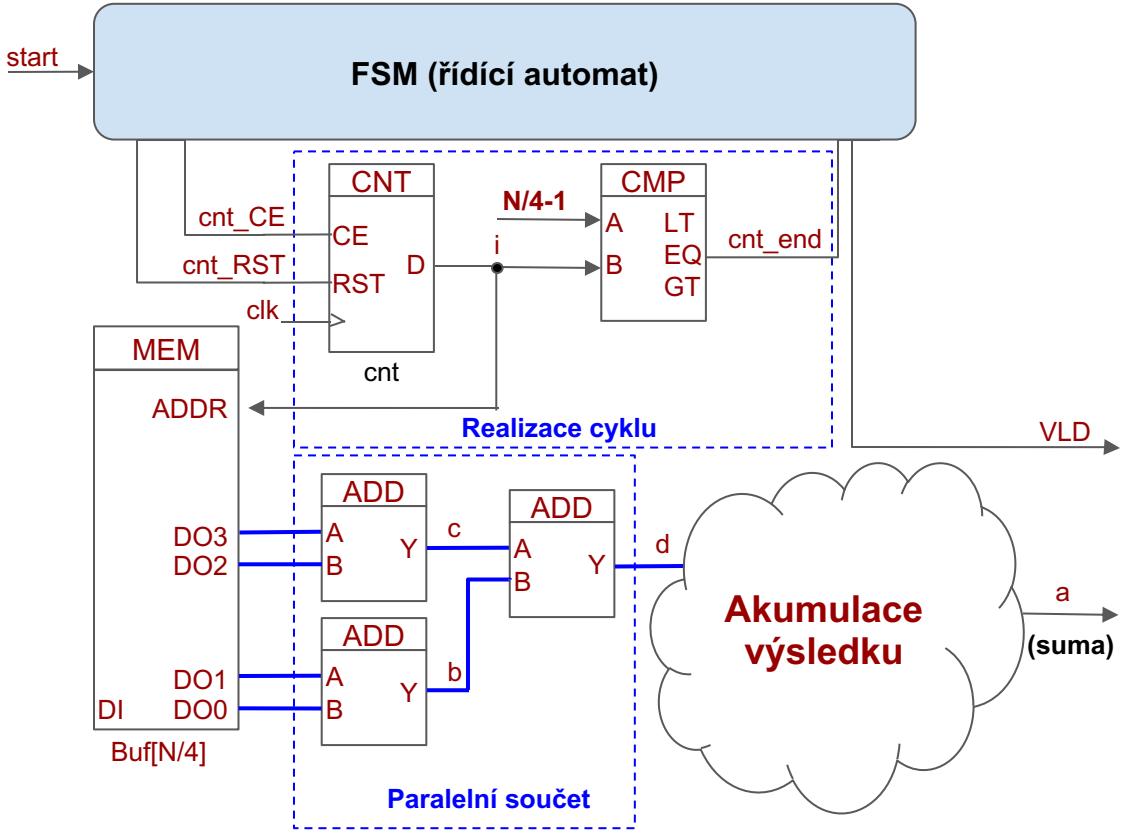
Iterace - příklad rozbalení smyčky



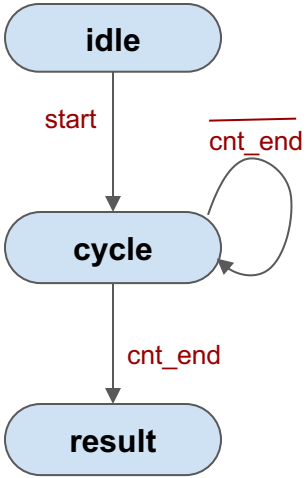

```
for (i=0;i<N/4;i++){
   a=a+Buf[i].D0;
   a=a+Buf[i].D1;
   a=a+Buf[i].D2;
   a=a+Buf[i].D3;
}
```



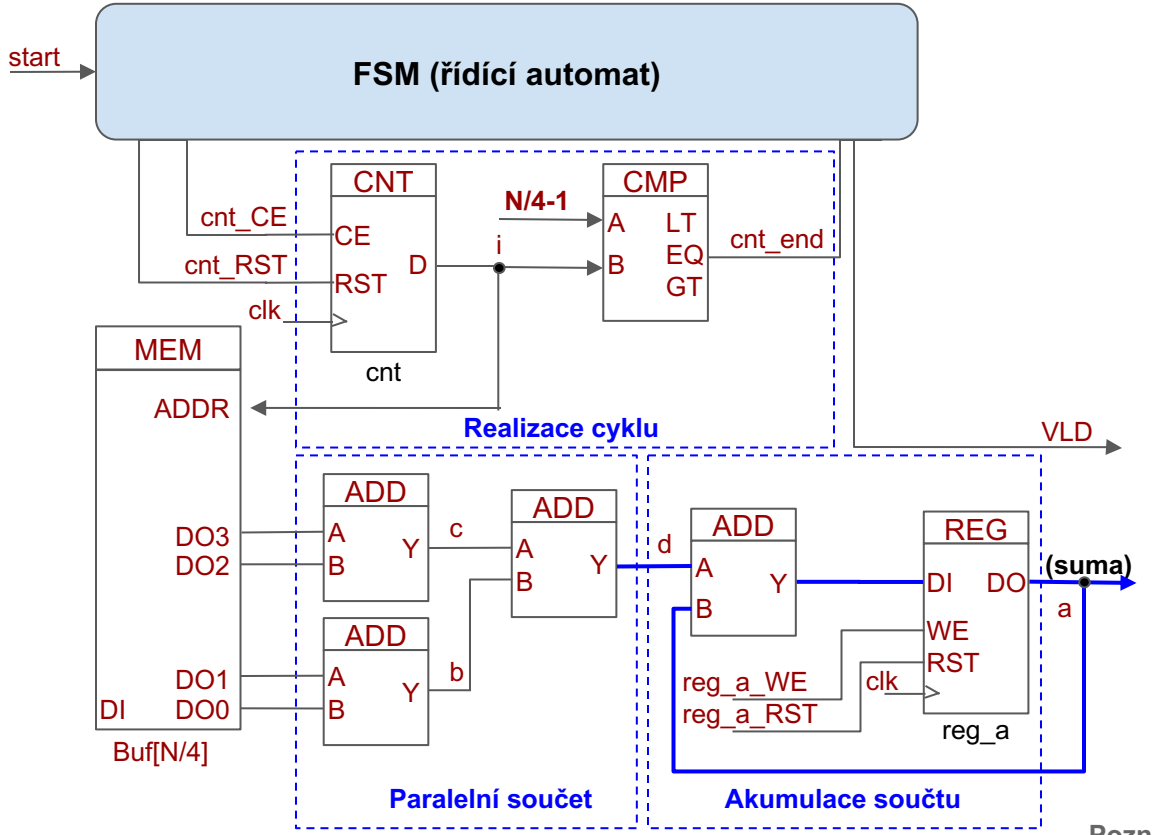
Iterace - příklad rozbalení smyčky



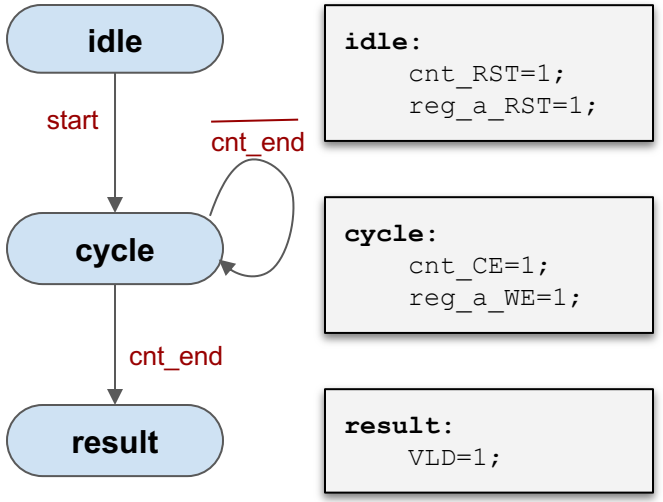

```
for (i=0;i<N/4;i++){
   b=Buf[i].D0+Buf[i].D1;
   c=Buf[i].D2+Buf[i].D3;
   d=b+c;
   a=a+d;
}
```



Iterace - příklad rozbalení smyčky

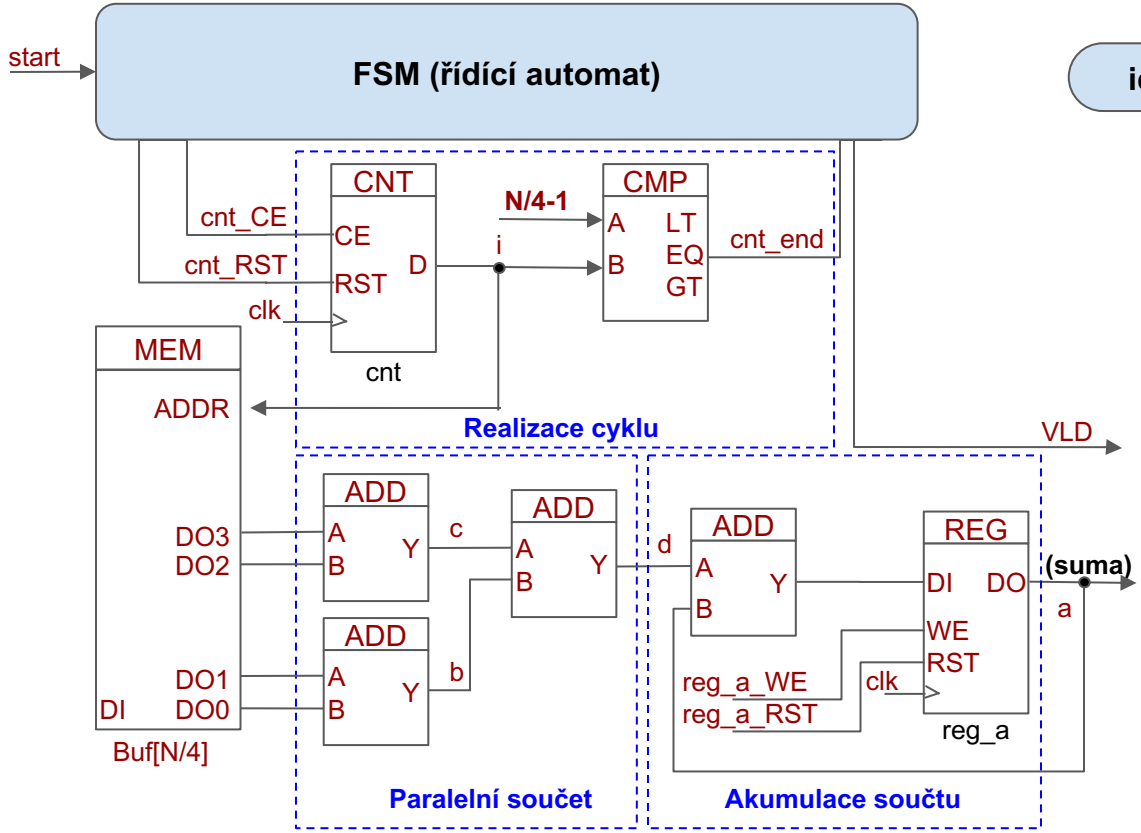


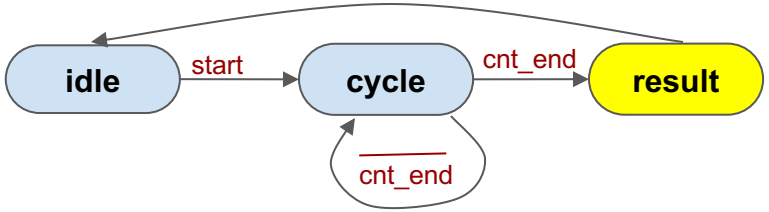

```
for (i=0;i<N/4;i++){
    b=Buf[i].D0+Buf[i].D1;
    c=Buf[i].D2+Buf[i].D3;
    d=b+c;
    a=a+d;
}
```



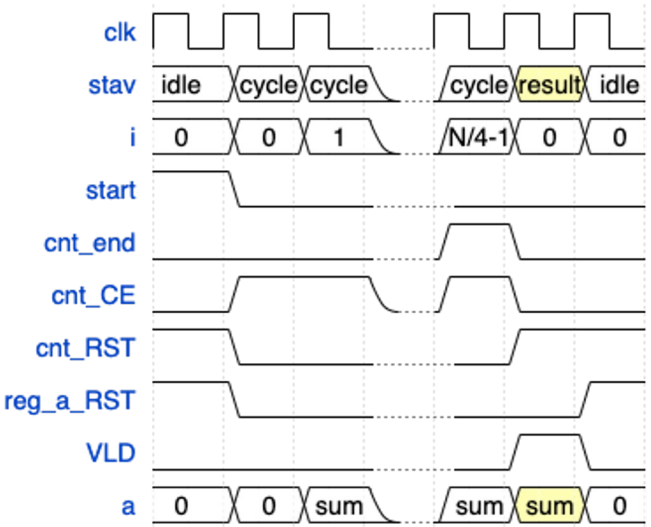
Pozn: Řízení signálů na základě aktuálního stavu. Pokud není některý výstup ve stavu nastaven, má nulovou hodnotu.

Iterace - příklad rozbalení smyčky





Časový diagram



Iterace - rozbalení smyčky

- Rozbalením smyčky se zápis algoritmu stává delším, avšak snižuje se režie spojená s vyhodnocením podmínky cyklu
- Rozlišujeme:
  - Částečné rozbalení smyčky
  - Úplné rozbalení smyčky

Příklad

Rozbalení smyčky s 10 iteracemi

```
int a[10];
int acc = 0;

for(i=0; i<10; i++)
   acc = acc + a[i];
```

Nerozbalená smyčka

```
int a[10];
int acc = 0;

for(i=0; i<10; i+=2){
   acc = acc + a[i];
   acc = acc + a[i+1];
}
```

Částečně rozbalená smyčka

```
int a[10];
int acc = 0;

acc = acc + a[0];
acc = acc + a[1];
acc = acc + a[2];
…
acc = acc + a[9];
```

Úplně rozbalená smyčka

Rozbalení smyčky do paralelního zpracování

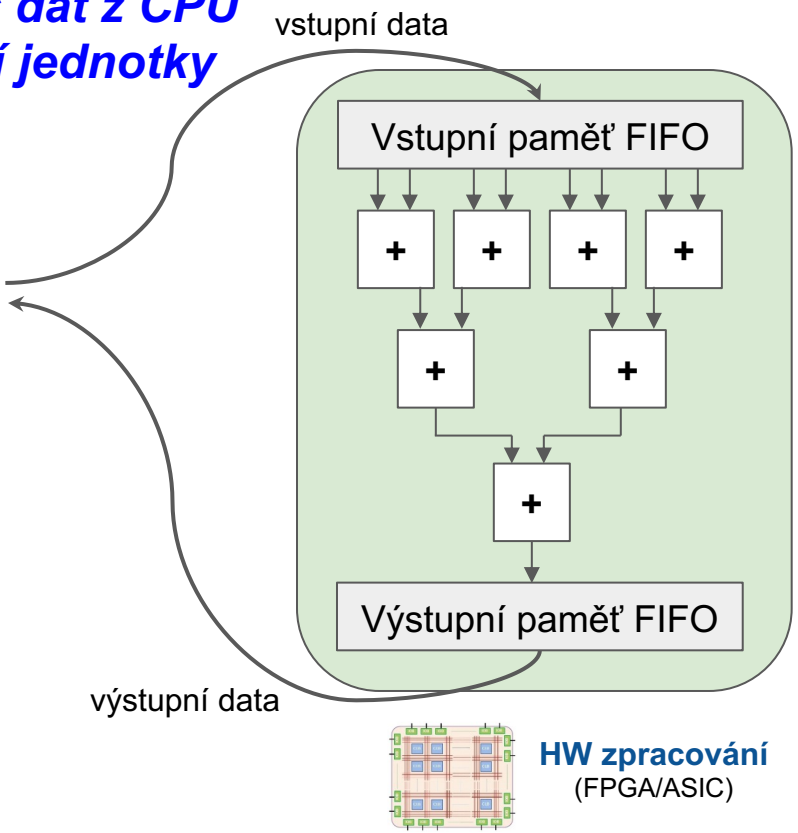
- ***Rozbalení smyčky umožňuje vykonání obecně N iterací cyklu paralelně*** (v jednom kroku).
- Musí být splněno několik předpokladů:
  - Fixní počet iterací smyčky, počet iterací nelze měnit v průběhu výpočtu.
  - Pro N paralelních iterací rozbalené smyčky ***musí být k dispozici všechna vstupní data***.
  - ***Mezi i a i+1 iterací není datová závislost:*** Jinými slovy není potřeba v další iteraci smyčky výsledek z předcházející iterace.
- Pokud jsou vstupní data uložena v paměti, musíme při rozbalení smyčky zvýšit propustnost paměti.
  - Zvýšení datové šířky čtených dat.
  - Více paralelních pamětí nebo portů paměti.
  - Násobná frekvence rozhraní pro přístup do paměti.

Zpracování dat v akceleračních jednotkách

Rychlý přenos dat z CPU do akcelerační jednotky

```
void main(){
    …
    hw_proc(...);
    …
    return 0;
}
```

Softwarové zpracování
(CPU)



HW zpracování
(FPGA/ASIC)

- Velké bloky dat často pouze prochází zpracováním v akcelerační jednotce
- FIFO paměti pouze vyrovnávají rychlosti příjmu, vysílání a zpracování dat (FIFO - first in first out)
- Příklady akcelerace:
  - Šifrování a komprese dat
  - Filtrace obrazu
  - Vyhledávání řetězců
  - Algoritmy strojového učení a umělé inteligence

Osnova

- Mapování základních konstrukcí do hardware
  - Sekvence
  - Selekce
  - Iterace
- ***Jazyk VHDL***
- Analýza popisu obvodů

Jak popsat číslicový obvod?

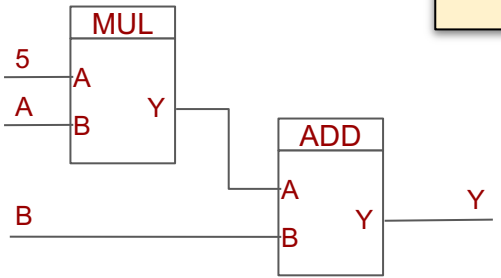
- Slovně
  - Slovní vyjádření toho co má obvod dělat je pro člověka přirozené, avšak vyrobit podle něj obvod není možné

„Navrhněte (číslicový) obvod, který spočítá sumu všech členů zadané posloupnosti“

- Matematicky
  - V současné době nejsou vhodné nástroje, které by umožnily automatizovaně bez úzké asistence člověka (návrháře) mapování do fyzického hardware

$$S = \sum_{i=1}^{N} V_i$$

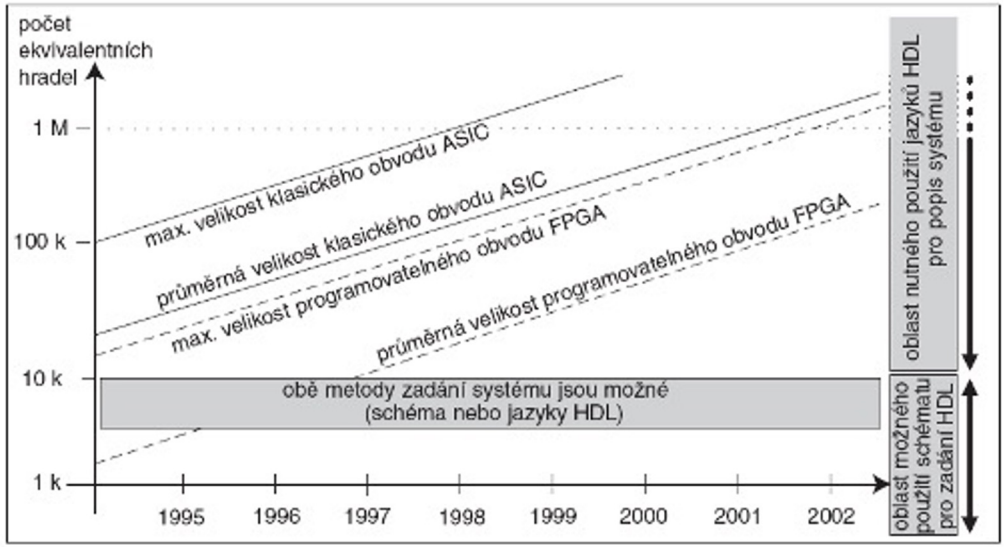
- Graficky pomocí schématu
  - Funkčních bloky a jejich propojení
  - Pro velké obvody pracné a nepřehledné



- Programovacím jazykem
  - Lze vytvořit popis chování obvodu v programovacím jazyku
  - Vhodné zejména pro velké obvody

Popis číslicových obvodů

- Grafická reprezentace pomocí logického schématu
- Zvyšující se složitost číslicových zařízení vedla ke vzniku HDL (Hardware Description Language) jazyků



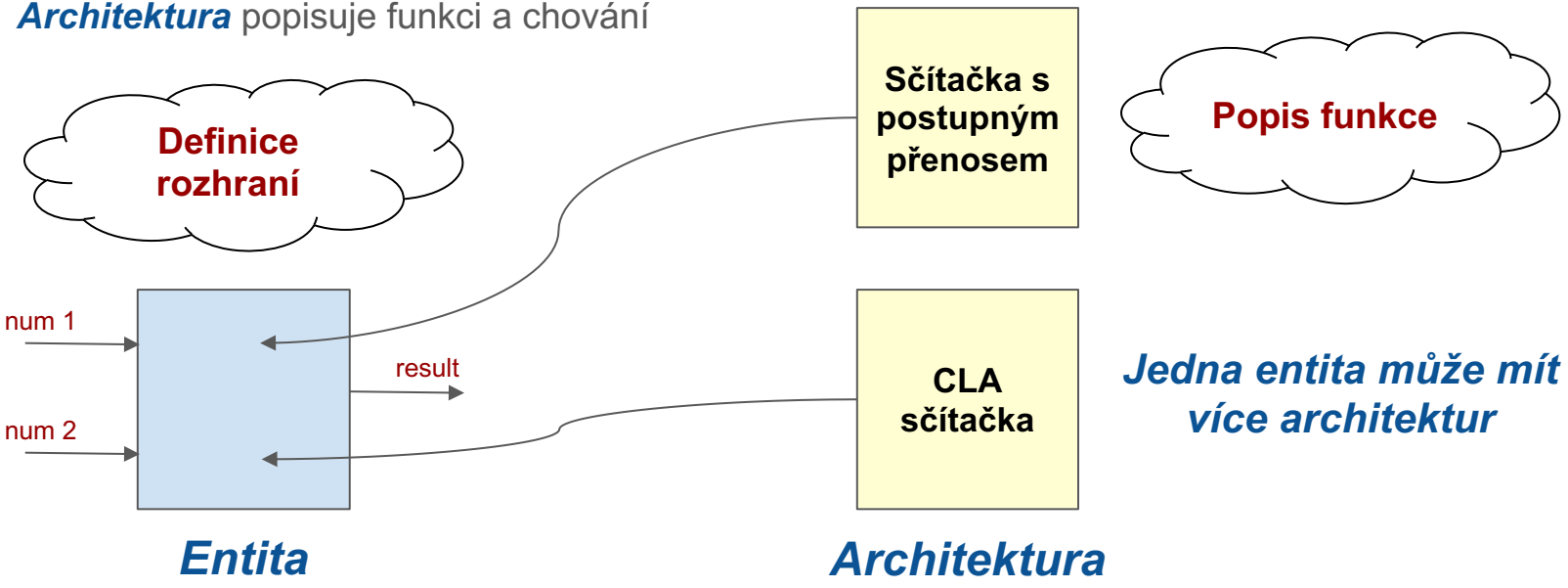
Rostoucí složitost FPGA vyžaduje nové metody návrhu

HDL jazyky

- Na rozdíl od schématu návrhář popisuje funkci obvodu
  - Zařízení je možné modelovat a simulovat
  - Proces syntézy umožňuje transformovat HDL popis do prvků cílové technologie – syntéza je proces analogický kompilaci používané u programovacích jazyků
- V praxi se používají zejména jazyky VHDL a Verilog
  - Oba jazyky mohou být použity pro mapování popisu do cílové technologie (logická syntéza)
  - VHDL dominuje v Evropě, Verilog v USA

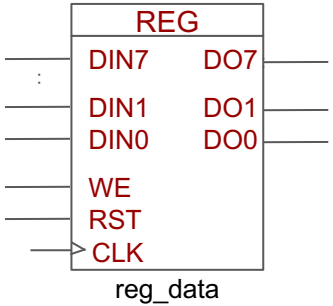
Popis číslicového zařízení ve VHDL

- VHDL popisuje číslicová zařízení a jednotlivé části zařízení pomocí ***komponent***
  - ***Entita*** definuje rozhraní komponenty
  - ***Architektura*** popisuje funkci a chování



Entita

- Popisuje rozhraní mezi komponentou a okolím
- Rozhraní komponenty se skládá ze signálů rozhraní (***port***) a generických parametrů (***generic***)
- Signály rozhraní mohou být podle směru šíření dat v módu ***IN***, ***OUT*** nebo ***INOUT***



Příklad:

```
entity register is
  generic (DATA_WIDTH : integer :=8
  );
  port (CLK     : in  std_logic;
        RST     : in  std_logic;
        WE      : in  std_logic;
        DIN     : in  std_logic_vector(DATA_WIDTH-1 downto 0);
        DO      : out std_logic_vector(DATA_WIDTH-1 downto 0);
  );
```

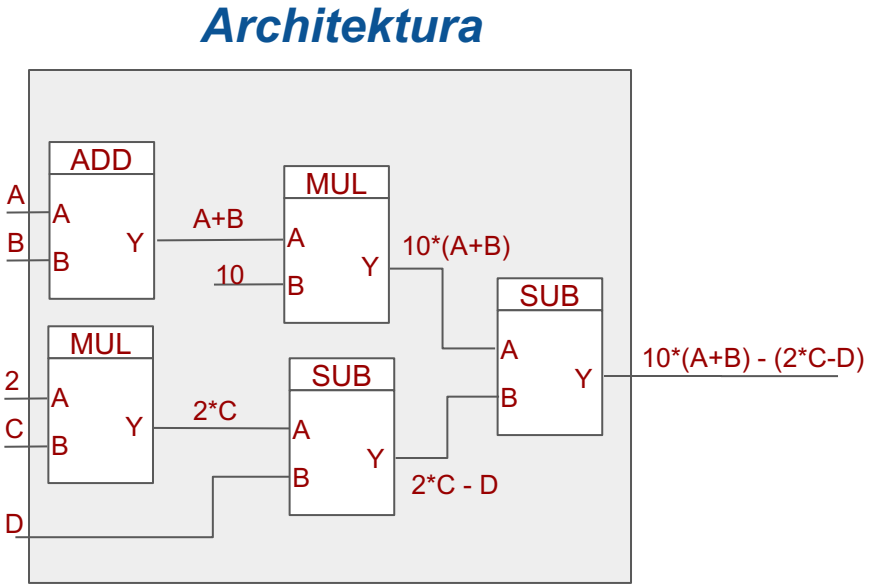
Generické parametry

Signály rozhraní

Architektura

- Definuje chování nebo strukturu komponenty
- ***Architektura je vždy svázána s entitou***, která definuje rozhraní komponenty (interakci s okolím)
- Každá komponenta může být popsána na úrovni ***struktury***, ***chování*** nebo ***dataflow popisu***
- Různé způsoby popisu je možné kombinovat

Jak popsat architekturu číslicových obvodů?



Každá komponenta neustále generuje výstup na základě vstupů

Komponenty pracují zcela paralelně
(nezávisle na sobě)

- Popis architektury obvodu musí reflektovat implicitně paralelní zpracování v hardware

Syntax

```
Architecture arch_name of ent_name is
    -- Deklarační část
begin
    -- Sekce paralelních příkazů
end architecture name;
```

Jméno entity
určuje signály rozhraní

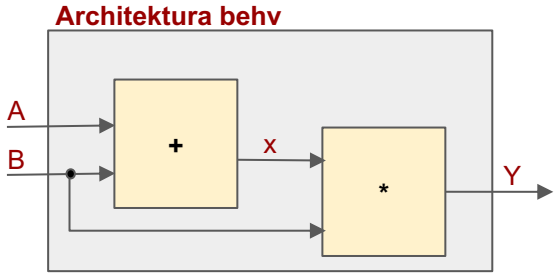
Deklarační část architektury
je vyhrazena pro pro deklaraci signálů, konstant nebo typů použitých uvnitř architektury.

- Součástí sekce paralelních příkazů mohou být instance komponent nebo procesy vzájemně propojené signály
  - ***Behaviorální popis*** – architektura je složena z jednoho nebo více procesů
  - ***Strukturní popis*** – architektura obsahuje instance již implementovaných komponent
- V praxi se behaviorální a strukturní popis často kombinují

Behaviorální popis

- Architektura je složena z jednoho nebo více procesů
- Proces je program, který určuje, jak se mají nastavit výstupní signály v závislosti na změnách vstupních signálů
- Z popisu nemusí být zřejmá hardwarová realizace

```
Architecture behv of ent_obvodu is
   signal x : std_logic_vector(7 downto 0);
begin
   process (A,B)
   begin
      x <= A + B;
   end process;
   process (x,B)
   begin
      Y <= x * B;
   end process;
end behv;
```



Proces přepočítá výstup vždy při změně signálu na senzitivity listu

Dataflow popis

- Modeluje datové závislosti vstupů a výstupů
- Zkrácený zápis chování pomocí paralelních příkazů v architektuře

Přiřazovací příkaz

```
Y <= NOT (A AND B);
```

Podmíněný přiřazovací příkaz

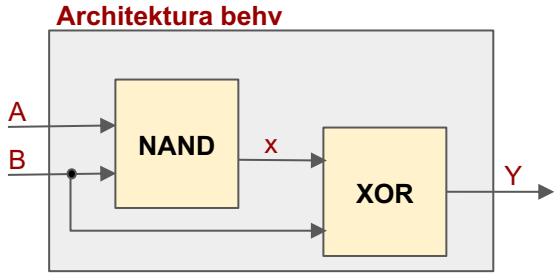
```
Y <= B when (A='1') else '0';
```

Výběrový přiřazovací příkaz

```
with S select    Y <= A when "0",
                      B when "1";
```

Příklad

```
architecture dataflow of ent_obvodu is
   signal x : std_logic;
begin
      x <= not (A AND B);
      Y <= (x XOR B);
end behv;
```



Proces ve VHDL

Syntax

```
name: process (sensitivity list)
    declarations
begin
    sequential statements
end process name;
```

Seznam vstupních signálů

Kdykoliv se změní signál na sensitivity listu, je spuštěn proces a vypočítají se nové hodnoty signálů

Deklarační část procesu

je vyhrazena pro pro deklaraci proměnných, konstant nebo typů použitých uvnitř procesu

Sekvenční příkazy

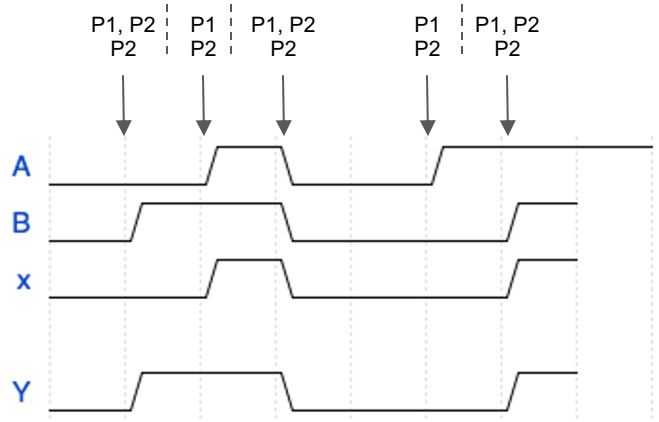
Program, který popisuje chování dané komponenty nebo její části. Na základě vstupních signálů a vnitřních proměnných program vypočítá hodnoty výstupních signálů

- Proces popisuje chování celé komponenty nebo pouze její části
- Architektury může obsahovat více procesů komunikujících vzájemně pomocí signálů

Sensitivity list procesu

- ***Sensitivity list procesu definuje seznam vstupních signálů***, které ovlivňují některý z výstupů procesu
- ***Pokud má process sensitivity list, vykoná se pouze při změně některého ze signálu na sensitivity listu*** tak, aby se vždy při změně vstupu generovaly nové výstupy
- Simulace je pak rychlá, reaguje pouze na změny vstupních signálů

```
Architecture behv of ent_obvodu is
   signal x : std_logic;
begin
   P1: process (A,B)
   begin
      x <= not (A AND B);
   end process;
   P2: process (x,B)
   begin
      Y <= (x OR B);
   end process;
end behv;
```



Řídící struktury v procesu

- Podmíněné vykonání příkazů (*if* ... *then* ...)

```
IF <condition> THEN <statements> END IF;
```

- Podmíněné vykonání příkazů s alternativou (*if* ... *then* ... *else* ... nebo *if* ... *then* ... *elsif* ...)

```
 IF                  <condition> THEN  <statements>
[ELSIF    <condition> THEN  <statements>]
[ELSE                                 <statements>]
 END IF;
```

- Výběr více příkazů (*case* ...)

```
CASE <expression> IS WHEN <value1> =>  <statements>
                    [WHEN <value2> =>  <statements>]
                    [WHEN <value3> =>  <statements>]
END CASE;
```

Řídící struktury v procesu (cykly)

- Cykly umožňující opakované vykonání sekvence příkazů
  - *while* ... *do* ...

```
WHILE <condition> LOOP <statements> END LOOP;
```

  - *for* ... *loop* ...

```
FOR <range> LOOP <statements> END LOOP;
```

- Příkazy pro přerušení běhu smyčky
  - *NEXT* – skok do další iterace

```
next when <condition>;
```

  - *EXIT* – ukončení celé smyčky

```
exit when <condition>;
```

Příklad procesu pro součet jedniček

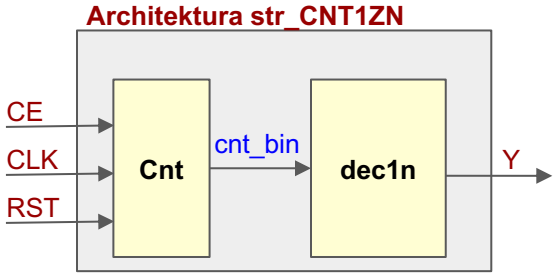
Součet jedniček v bitovém vektoru bus_in

```
process(bus_in)
    variable count : std_logic_vector(3 downto 0);
begin
   count   := "0000";
   for i in 0 to 15 loop
      if bus_in(i) = '1' then
         count := count + '1';
      end if;
   end loop;
   N_ONE <= count;
end process;
```

- Při každé změně signálu ***bus_in*** se vyvolá proces, sečtou se jedničky ve vektoru ***bus_in*** a výsledek se uloží do ***N_ONES***.
- Pro akumulaci počtu jedniček je využita proměnná

Strukturní popis: instance entity

- ***Mám k dispozici komponenty a vytvářím zapojení***
- Je nutné specifikovat knihovnu, ve které se entita nachází
- Připojení signálů na porty pořadím nebo přiřazením jména
- Nepřipojené signály se označují klíčovým slovem open



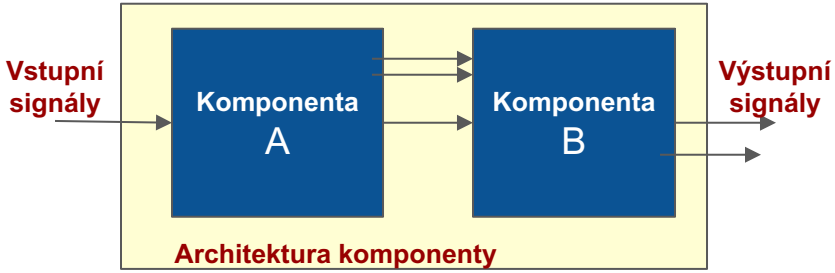

```
entity CNT1ZN is
   port (
      CLK   : in  std_logic;
      RST   : in  std_logic;
      CE    : in  std_logic;
      DO    : out std_logic_vector(7 downto 0)
   );
end CNT1ZN;
architecture str_CNT1ZN of CNT1ZN is
signal cnt_bin : std_logic_vector(2 downto 0);
begin
   cnt_i: entity work.cnt
   port map ( CLK=>CLK, RST=>RST, CE=>CE,
              DOUT => cnt_bin);
   dec1n_i: entity work.dec1n
   port map ( addr=>cnt_bin, y=>DO);
end struct;
```

```
entity cnt is
port (
   CLK    : in  std_logic;
   RST    : in  std_logic;
   CE     : in  std_logic;
   DOUT   : out std_logic_vector(2 downto 0)
end cnt;
```

```
entity dec1n is
    port (
        addr: in  std_logic_vector(2 downto 0);
        y   : out std_logic_vector(7 downto 0)
    );
 end dec1n;
```

Signály ve VHDL

- Signály slouží pro komunikaci mezi komponenty nebo procesy
- Mohou být implementovány pomocí *vodiče* nebo *sběrnice* (více vodičů)



- Signálu je možné přiřadit libovolný datový typ, který definuje charakter přenášených hodnot
  - Pro reprezentace 1 bitového propoje se používá typ ***std_logic***
  - Pro popis sběrnic se používá typ ***std_logic_vector(N downto 0)***. Šířka sběrnice je definována šířkou pole.
- Příklady popisu sběrnice:

```
std_logic_vector(7 downto 0);
std_logic_vecotr(0     to 7);
```

7 je MSB bit

0 je MSB bit

Deklarace signálů

- Vnitřní signály komponenty jsou deklarovány v deklarační části architektury (signály lze použití pouze v rámci architektury)

```
Architecture name of processor is
    -- Deklarační část
begin
    -- Sekce paralelních příkazů
end architecture name;
```

Zde je možné deklarovat vnitřní signály architektury

Signál nelze deklarovat uvnitř procesu!!!

- Deklarace signálu

```
signal <jmeno> : <typ> [:= imp_hodnota];
```

  - Jméno jednoznačně identifikuje signál, typ definuje charakter přenášených dat
  - Signálu je možné nastavit počáteční (implicitní) hodnotu

Koncept nastavení signálů

- Nastavení signálu se provádí až v okamžiku pozastavení / uspání procesu
- Pokud je v procesu spuštěno více příkazů přiřazení hodnoty do signálu, je provedeno pouze poslední přiřazení, a to až v okamžiku dokončení procesu (pozastavení / uspání).
  - ***Příkaz přiřazení pouze naplánuje akci přiřazení nové hodnoty do signálu***, obsah signálu ale zůstává do pozastavení / uspání procesu nezměněn
  - Samotná ***změna hodnoty signálu se provede až v okamžiku pozastavení / uspání procesu***, kdy zpracování programu uvnitř procesu dojde do konce procesu

```
example : process (A, B)
begin
    C <= A;
    C <= B+1;
end process example;
```

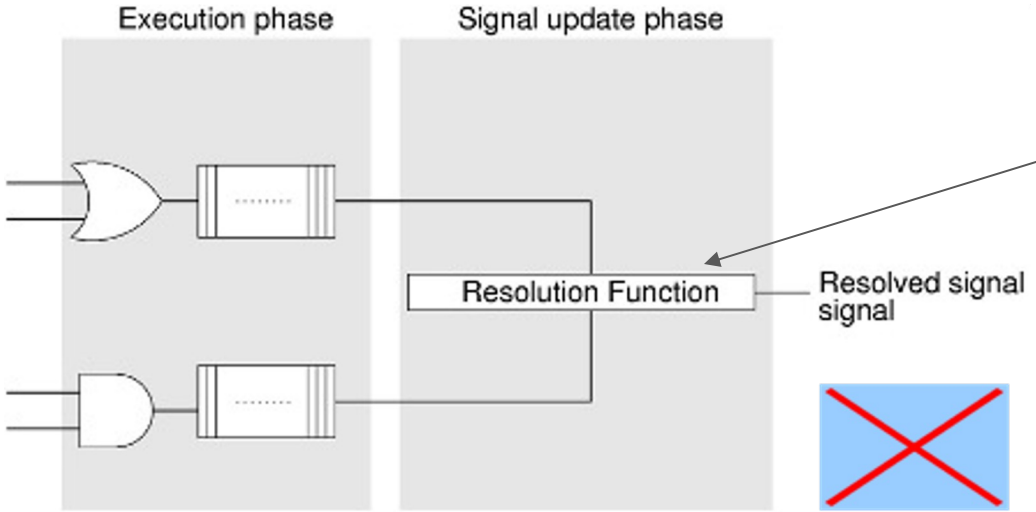
Naplánováno přiřazení hodnoty A do signálu C

Přepsání plánu přiřazení do signálu C: místo A se přiřazení změní na hodnotu B+1

Pozastavení / uspání procesu a provedení posledního přiřazení signálu C na novou hodnotu (B+1)

Nastavení signálu z více zdrojů

- Pokud má signál jediný zdroj, je jednoduché určit hodnotu signálu
- Signály jsou nastavovány z více zdrojů v řadě aplikací – například datová sběrnice počítače
  - Co se stane v případě, že je nastavován signál z více zdrojů současně?
  - Jaké hodnoty bude výsledný signál nabývat?



Rezoluční funkce definuje prostřednictvím tabulky hodnotu signálu v případě, že je nastavován signál ze dvou různých zdrojů současně

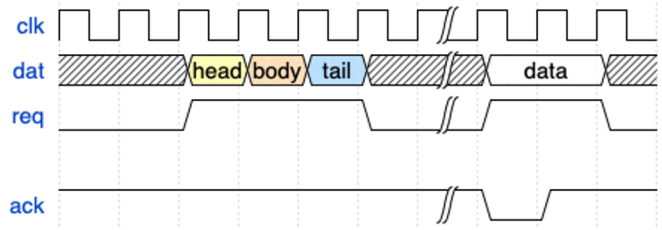
Dvě hodnoty jsou na reprezentaci výsledku málo!

Jaký bude výsledek rezoluční funkce – hodnota '0' nebo '1' ?

Atributy signálů

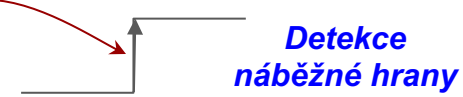
- Každý signál má svůj v průběh v čase, nikoliv pouze aktuální hodnotu
- ***Historie signálů je ve VHDL přístupná pomocí atributů***
- Syntaxe použití atributu

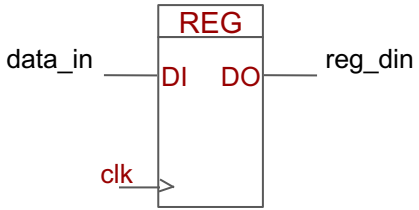
```
<jmeno_signalu>'<jmeno_atributu>
```



- Příklady používaných atributů
  - ***event*** – boolean – atribut je hodnoty true v případě změny hodnoty signálu
  - ***last_value*** – hodnota signálu před poslední změnou hodnoty
- Používá se zejména při ***detekci náběžné hrany hodin***

```
process (clk, data_in)
begin
    if clk'event AND clk='1' then
        reg_din <= data_in;
    end if;
end process;
```

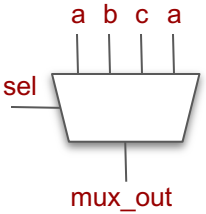




Sekvenční vs Kombinační obvod ve VHDL

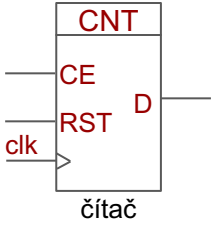
Kombinační obvod
(multiplexor)

```
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;
```



Sekvenční synchronní obvod
(čítač)

```
process (clk,RST,CE)
begin
   if (RST = '1') then
      DOUT <= (others => '0');
   elsif (clk'event and clk = '1') then
      if CE='1' then
         D <= D + '1';
      end if;
   end if;
end process;
```



- ***Sekvenční obvod obsahuje synchronizaci na náběžnou hranu hodin (clk'event and clk=1)***
- Kombinační obvod vypočítává změnu výstupních signálů na základě vstupu bez synchronizace (není využit hodinový signál)

Signály v procesu

- Signály slouží pro komunikaci mezi procesy – připojení ***vstupů a výstupů procesů***
- Vlastnosti signálů
  - Signály nemohou být deklarovány uvnitř procesů
  - ***Přiřazení signálů je fyzicky provedeno až v okamžiku pozastavení / uspání procesu***. Dokud není proces uspán, má signál původní hodnotu
  - ***V procesu je provedeno pouze poslední přiřazení signálů***. Ostatní přiřazení jsou nejsou realizovány.
- V procesu je možné přechodně uchovat hodnotu pomocí proměnných (variable)

Proměnné

- Proměnné mohou být deklarovány a použity pouze v procesu
- Příklad deklarace a použití proměnné

```
Process (b)
variable a : integer := 3;
begin
   a := a + 2;
   output <= a * b;
end process;
```

Deklarace může obsahovat ***jméno***, ***typ*** a ***implicitní hodnotu***

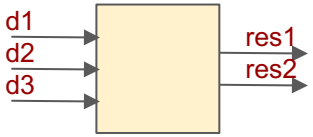
Signál output je nastaven již podle aktualizované proměnné ***a*** ***(output <= 5 * 4)***

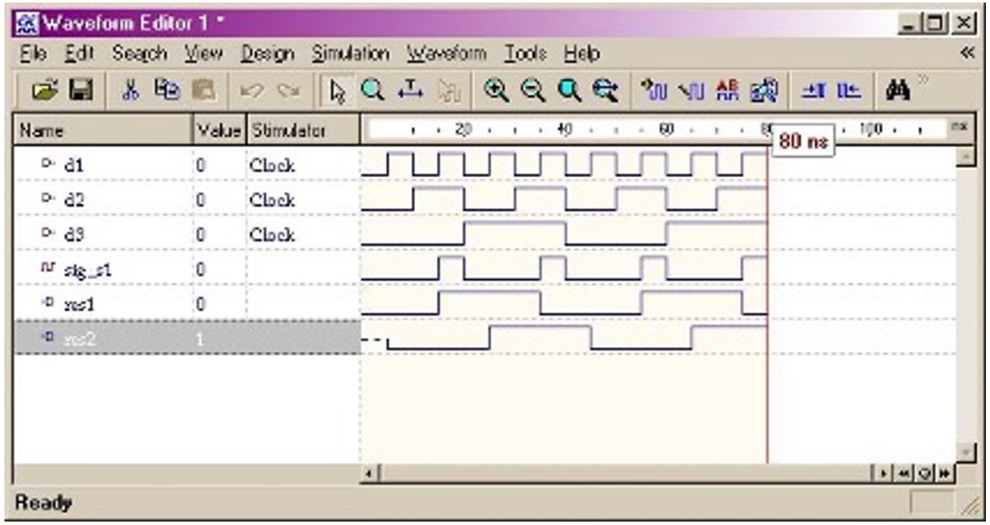
Přiřazení do proměnné se provede okamžitě ***(a = 3 + 2 = 5)***

Hodnota proměnné zůstává zachována i po uspání procesu

Signál vs. Proměnná v procesu

```
entity sig_var is
port(d1, d2, d3:    in  std_logic;
     res1, res2:    out std_logic);
end sig_var;

architecture behv of sig_var is
signal sig_s1: std_logic;
begin

  proc1: process(d1, d2, d3)
    variable var_s1: std_logic;
  begin
     var_s1 := d1 and d2;
     res1 <= var_s1 xor d3;
  end process;

  proc2: process(d1,d2,d3)
  begin
     sig_s1 <= d1 and d2;
     res2 <= sig_s1 xor d3;
  end process;
end behv;
```





Rozdíly mezi výstupy res1 a res2

Komentáře, znaky, řetězce

- Komentář je uvozen dvojicí znaků **--**

```
-- Toto je komentář
```

- Znak nebo bit se zadává pomocí apostrofů

```
sig_bit <= '1';
```

- Řetězec nebo bitový vektor se zadává pomocí uvozovek

```
sig_bit_vector <= "0001";
```

- Jména signálů a proměnných musí začínat písmenem a dále mohou obsahovat písmena a číslice

Deklarace rozšiřujících datových typů

- Výčtový datový typ

```
TYPE muj_stav IS (reset, idle, rw, io);
signal stav: muj_stav;
...
stav <= reset;       -- nelze stav <="00";
...
```

- Pole

```
TYPE data_bus IS ARRAY (0 TO 15) OF BIT;
variable x: data_bus;
variable y: bit;
y := x(12);
```

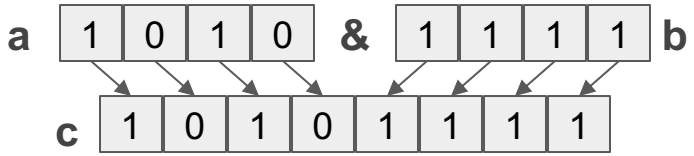
Práce na úrovni bitů

- Uvažujme deklaraci signálů

```
signal a, b: bit_vector (3 downto 0);
signal c   : bit_vector (7 downto 0);
signal clk : bit;
```

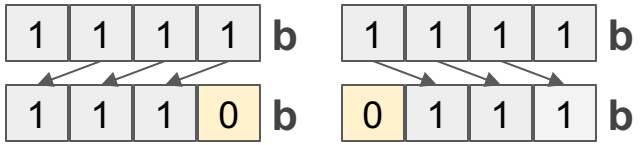
- Konkatenace signálů:

```
c <= a & b;
```



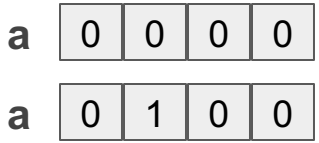
- Bitový posun doleva a doprava

```
b <= b(2 downto 0) & '0'; -- posun doleva
b <= '0' & b(3 downto 1); -- posun doprava
```



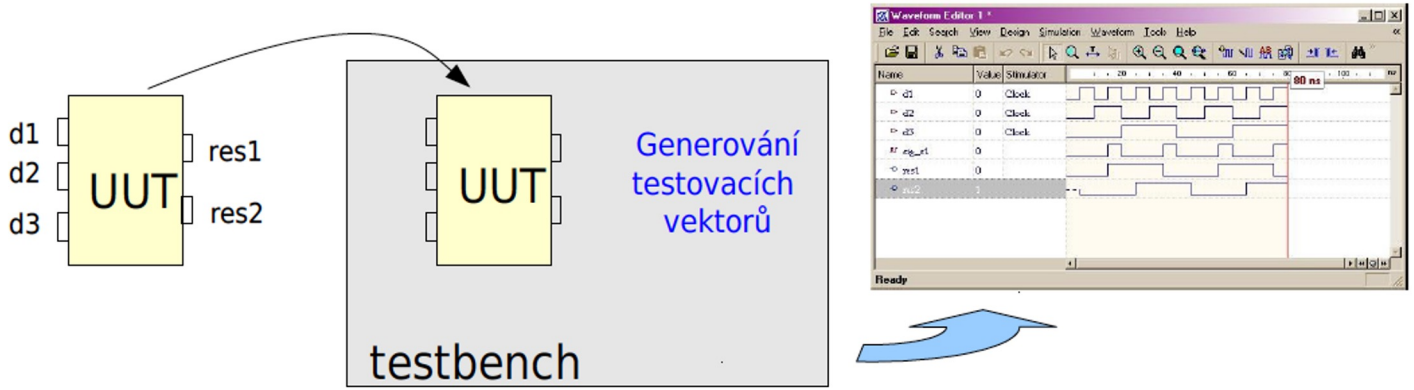
- Agregace

```
a <= (others => '0');          -- vše do nuly
a <= ('0', '1', others=>'0'); -- MSB = "01"
```



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

Základní prvky testbench

```
entity testbench is
end entity testbench;

architecture  testbench_arch of testbench is

    signal a, b : std_logic;
    signal ...

begin
    UUT : entity work.and_gate
    port map( ...
              ...
            );

    test : process
    begin
        a <= ...;
        b <= ...;
        wait for ...;
        ...
        wait for ...;
        ...

        wait;
    end process test;
end architecture testbench_arch;
```

Rozhraní entity Testbench

Testovací vektory připojené k UUT

Instance UUT (připojení testovacích vektorů)

Postupné přikládání testovacích vektorů

Příkaz WAIT

- ***Proces může být pozastaven příkazem WAIT*** v kterékoliv fázi zpracování procesu → je možné ***definovat posloupnost kroků při testování komponent***
- Proces obsahující sensitivity list nemůže obsahovat příkaz WAIT
- ***Sensitivity list je možné převést na příkaz WAIT on***

```
muj_process: process (A, B)
begin
   příkaz 1;
   příkaz 2;
   příkaz 3;
end process muj_process;
```

```
muj_process: process
begin
   WAIT on A, B;
   příkaz 1;
   příkaz 2;
   příkaz 3;
end process muj_process;
```

Příkaz WAIT

- ***WAIT FOR*** čas – pozastaví proces na specifikovaný čas

```
wait for 10 ns;
wait for clk_per/2;
```

- ***WAIT UNTIL*** podmínka - pozastaví proces dokud není podmínka pravdivá

```
wait until Clk='1';
wait until CE and (not RESET);
```

- ***WAIT ON*** sensitivity list - pozastaví proces dokud není detekována nějaká událost na některém ze signálu uvedených na senzitivity listu

```
wait on Enable;
```

Příklad Testbench

```
 entity and_gate is
    port (
        A : in  std_logic;
        B : in  std_logic;
        Y : out std_logic
    );
end entity and_gate;

architecture  behavioral of and_gate is

begin
    p_and : process (A, B)
    begin
        Y <= A and B;
    end process p_and;

end architecture behavioral;
```

Test všech kombinací na vstupu hradla AND

```
entity testbench is
end entity testbench;

architecture  tb_arch of testbench is
    signal a, b, y : std_logic;
begin
    UUT : entity work.and_gate
    port map( A => a,
              B => b,
              Y => y
    );
    test : process
    begin
        a <= 0; b <= 0;
        wait for 10 ns;
        a <= 0; b <= 1;
        wait for 10 ns;
        a <= 1; b <= 0;
        wait for 10 ns;
        a <= 1; b <= 1;
        wait;
    end process test;
end architecture tb_arch;
```

Assertion

- Automaticky ověří při simulacích platnost zadané podmínky
  - Výsledky jsou zobrazeny ve Waveform dialogu nebo jsou zapsány do logu v rámci simulace
- ***Asssertion*** se skládá ze tří částí
  - ***assert*** – specifikuje podmínku, která by měla být splněna
  - ***report*** – definuje zprávu, která se zobrazí v případě nesplnění podmínky
  - ***severity*** – určuje jak se simulátor zotaví z chybové situace (general note, warning, system failure)
- Příklad použití

```
assert (left>right)
    report   "Right number greater than left number"
    severity warning;
```

Osnova

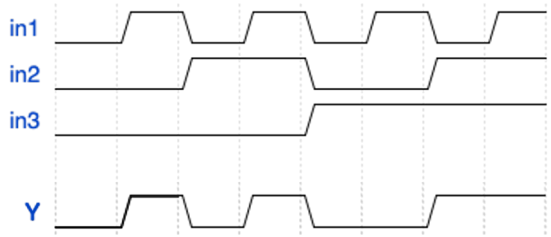
- Mapování základních konstrukcí do hardware
  - Sekvence
  - Selekce
  - Iterace
- Jazyk VHDL
- ***Analýza popisu obvodů***

Analýza popisu obvodu

- Vytvoření schématu architektury
  - Vytvoření základního schématu tak, že ***každý proces tvoří jednu HW jednotku*** (blok) a **signály HW jednotky** (procesy) **vzájemně propojují**
  - Vstupní signály HW bloků (procesů) jsou dány sensitivity listem, výstupní signaly odpovídají signálům nastavovaným v rámci procesů
- Analýza průchodu signálu obvodem
  - Schéma propojení procesů (HW jednotek) modeluje vzájemné datové závislosti → schéma můžeme využít při analýze šíření změn signálů obvodem
  - Pro každou změnu signálu na vstupu procesu je nutné přepočítat výstupy a nové změny signálu propagovat do dalších procesů podle schématu propojení (datových závislostí)
- Automatizovaně pomocí simulátoru (ModelSim, vsim, …)
  - Simulace časových průběhů pro kontrolu správné funkce obvodu

Příklad: Rozpoznání obvodu popsaného ve VHDL

```
process(in1, in2, in3)
begin
   case in3 is
      when '0' => y <= in1;
      when '1' => y <= in2;
      when others => null;
   end case;
end process;
```

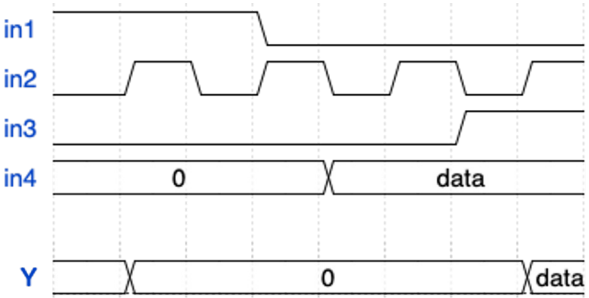


Vstupy ….in1, in2, in3 …

Výstupy ….y……

Jaký základní prvek proces ve VHDL popisuje?

```
process(in1, in2, in3, in4)
begin
   if in1='1' then
      out1='0';
   elsif (in2'event and in2='1') then
      if in3='1' then
         out1=in4;
      end if;
   end if;
end process;
```



Vstupy …in1, in2, in3, in4 …

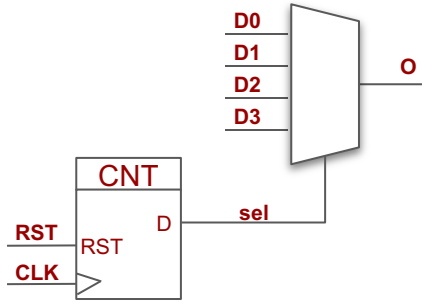
Výstupy …out1……

Jaký základní prvek proces ve VHDL popisuje?

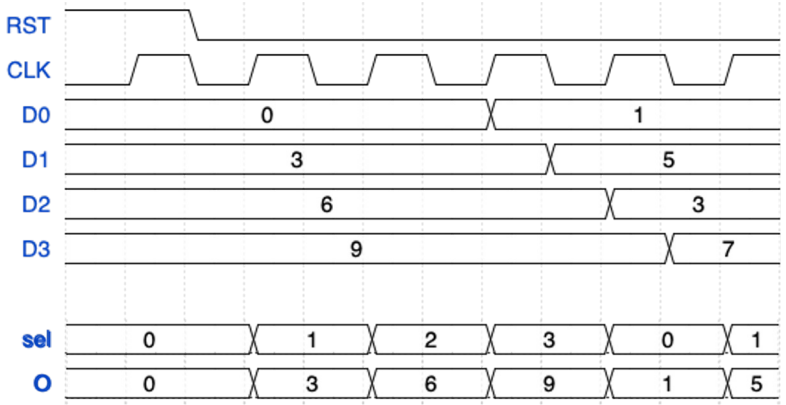
Příklad analýzy obvodu

```
Architecture behv of ent_obvodu is
   signal sel : std_logic_vector(1 downto 0);
begin

   process (CLK, RST)
   begin
       if RST = '1' then
           sel <= "00";
       elsif CLK'event AND CLK='1' then
           sel <= sel + '1';
       end if;
   end process;

   process(D3,D2,D1,D0,sel)
   begin
      case sel is
         when "00" =>     O <= D0;
         when "01" =>     O <= D1;
         when "10" =>     O <= D2;
         when "11" =>     O <= D3;
         when others =>   O <= D0;
      end case;
   end process;

end behv;
```

Schéma zapojení



Časový průběh



Konec přednášky

Děkuji za pozornost