Úvod do VHDL

Obsah

FPGA

- **F**ield-**P**rogrammable **G**ate **A**rray
- hardware charakteristický polem programovatelné logiky (většinou se jedná o jeden čip)
- skládá se zejména z univerzálních LUT (LookUp Table) a registrů (speciální flip-flop), jenž bývají spolu ve stejném logickém bloku, většinou však obsahuje i blokovou paměť
- při dostatku vnitřních prostředků může být konfigurováno tak, že dokáže svým chováním simulovat libovolný digitální systém (třeba i procesor)
- simulovaný digitální systém uvnitř FPGA můžeme chápat jako virtuální hardware
- FPGA čipy i přes svou univerzálnost poskytují vysokou efektivitu
- většinou k dostání spolu s vývojovou deskou, která vedle FPGA čipu většinou implementuje i mnoho V/V periferií, s kterými lze komunikovat prostřednictvím FPGA

VHDL

- **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage
- jazyk sloužící k popisu digitálních systémů (hardware může být též digitálním systémem)
- kód lze použít ke konfiguraci FPGA a popisovaný systém tak simulovat
- používá se zejména k návrhu a simulaci prototypů zákaznických obvodů (ještě před samotnou výrobou) nebo jako levnější alternativa zákaznického obvodu
- silně typovaný (téměř žádné implicitní přetypování)
- využívá koncept modularizace (komponenty)
- implicitně umožňuje popisovat paralelní jevy

VHDL vs C jazyk

- jedná se o dva velmi rozdílné jazyky pro rozdílné účely, a tudíž se to projeví i při návrhu řešení daného problému

VHDL	**C jazyk**
• popisný charakter	• imperativní charakter
• popisuje paralelní jevy	• popisuje jednotlivými kroky algoritmus
• dílčí problémy se řeší komponentou	• dílčí problémy se řeší funkcí
• silně typovaný	• slabě typovaný
• case-insensitive	• case-sensitive
• instancí realizace je konfigurované FPGA, příp. zákaznický obvod	• instancí realizace je běh strojového kódu na procesoru (zejména proces)
• kód se syntetizuje (ovšem často se i říká, že se kompiluje)	• kód se kompiluje

Úlohy

Úlohy jsou konstruovány tak, že každá úloha je zadána na konci probírané látky a slouží k jejímu procvičení. Úloh je celkově 9 a s výjimkou úloh č. 1, 2 a 3 vypracujete ke každé úloze taky její testbench soubor, který bude sloužit k ověření správnosti vašich řešení ve VHDL simulátoru a taky k jejich následné demonstraci.

Pro každou úlohu si vytvoříte samostatnou složku s názvem "uloha**x**", kde **x** je číslo úlohy. V názvu VHDL souborů i v cestě jejich umístění nepoužívejte jiné znaky než znaky anglické abecedy a podtržítko. Vyhnete se tak možným budoucím problémům. Do vytvořených složek pro jednotlivé úlohy budete vkládat veškeré materiály týkající se těchto úloh. Nebude-li vám vyhovovat taková složka jako kořen VHDL projektu, můžete si v ní vytvořit další složku pro takovýto účel.

Pro vytvoření VHDL projektu k dané úloze můžete použít jako kostru nějakou vaši již hotovou úlohu, nebo můžete využít jedné ze dvou zveřejněných koster VHDL:

- **comb_template.vhd** - pro úlohy se zaměřením na kombinační logiku
- **seq_template.vhd** - pro úlohy se zaměřením na sekvenční logiku

Nezapomeňte si však vždy výchozí soubor nejprve zkopírovat a následně přejmenovat tak, aby byl název daného souboru v korespondenci s názvem `entity` výsledného souboru. Nezapomeňte si také poté upravit názvy vstupů, výstupů a signálů, příp. nepotřebné konstrukce smazat.

*Např. pro první úlohu je vhodné využít soubor **comb_template.vhd** a přejmenovat jej na **half_adder.vhd**. Název `entity` pak zřejmě bude `half_adder`.*

Základní konstrukce

- `entity` - definuje rozhraní komponenty (tj. její vstupy a výstupy)
- `architecture` - implementace komponenty, popis jejího chování
  - dělí se na **deklarační část** (před `begin`) a **část paralelních příkazů** (po `begin`)
  - používá vstupy a výstupy z vlastní `entity`
- `std_logic`
  - standardní logický datový typ (z balíku `std_logic_1164`) používaný zejména pro implementaci dvoustavové logiky
  - definuje 9 hodnot, kterých může nabývat (jejich výčet a popis je uveden zde)
  - *"Proč se nepoužívá raději datový typ popisující jen logickou pravdu a nepravdu?"*
    - `std_logic` popisuje reprezentaci stavu (hodnoty) signálů v reálném digitálním systému realističtěji než datové typy popisující pouze dvoustavovou logiku
    - výše zmíněné nalezne široké uplatnění v simulaci systému (testbench)

Základy kombinační logiky

Jako základ kombinační logiky můžeme považovat implementaci základních logických funkcí - binární AND, NAND, OR, NOR, XOR, XNOR a unární NOT. Operátory těchto hradel ve VHDL jsou stejnojmenné a lze je použít v části paralelních příkazů bloku `architecture`.

Příklad:

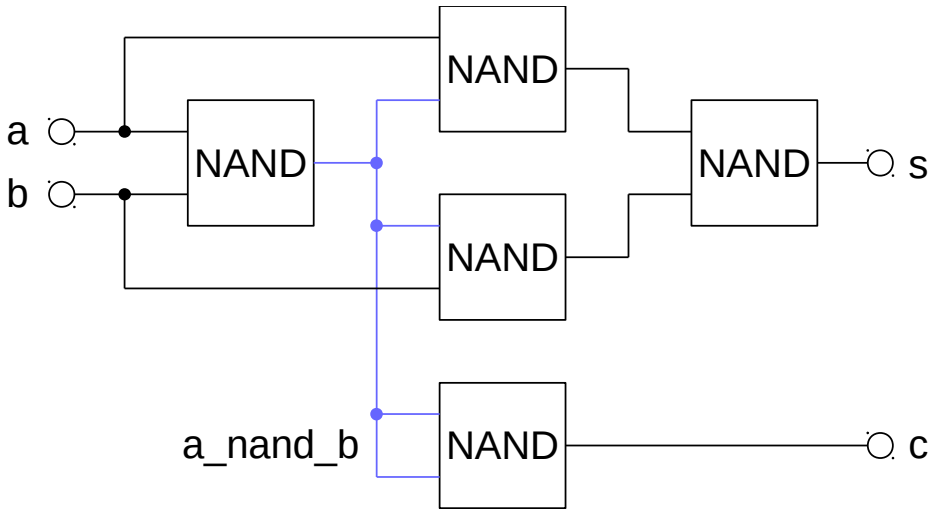
```
library ieee;
use ieee.std_logic_1164.all;


entity equivalence is
  port(
    a: in std_logic;
    b: in std_logic;

    y: out std_logic -- u posledniho uvedeneho portu se strednik nepise
  );
end equivalence;


architecture equivalence_arch of equivalence is
begin

  y <= a xnor b; -- pevne prirazeni

end equivalence_arch;
```

Úloha č. 1 - poloviční sčítačka

Poloviční sčítačka patří mezi úplný základ téměř jakéhokoli hardwaru, který nějakým způsobem pracuje s aritmetikou. Proto v první úloze budete navrhovat právě tento jednoduchý kombinační obvod. Nejsou kladena žádná omezení na použité operátory. Výsledný obvod také načrtněte.

Signál jako identifikátor vodiče

Pokud shlédneme schéma poloviční sčítačky implementované pomocí hradel NAND, které je zobrazeno níže, snadno dojdeme k názoru, že by bylo vhodné, kdybychom mohli vytvořit "identifikátor na vodič". Právě k tomuto účelu lze použít datový objekt `signal`, který spolu s identifikátorem a datovým typem deklarujeme hned do bloku `architecture` ještě před `begin`, tedy do deklarační části architektury. Je zřejmé, že do signálu, který je identifikátor na vodič, smíme přiřadit hodnotu pouze jednou (na jednom místě v části paralelních příkazů bloku `architecture`).



V tomto případě by bylo vhodné mít identifikátor na modře zvýrazněný vodič na výše uvedeném obrázku. Výsledný kód za použití `signal` pro vytvoření identifikátoru `a_nand_b` na modře vyznačený vodič výše uvedeného schématu by tak mohl vypadat následovně:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4
5  entity half_adder_nand is
6    port(
7      a: in std_logic;
8      b: in std_logic;
9
10     s: out std_logic;
11     c: out std_logic
12   );
13 end half_adder_nand;
14
15
16 architecture half_adder_nand_arch of half_adder_nand is
17
18   signal a_nand_b: std_logic; -- identifikator a datovy typ
19
20 begin
21
22   a_nand_b <= a nand b;
23
24   s <= (a nand a_nand_b) nand (a_nand_b nand b);
25
26   c <= a_nand_b nand a_nand_b; -- negace a_nand_b
27
28 end half_adder_nand_arch;
```

Datový objekt `signal` nemusí popisovat jen proměnné skalárních veličin jako je `std_logic`. Později se dostaneme i k jiným datovým typům a tyto datové typy lze také použít s datovým objektem `signal`. Můžeme tak později třeba vytvořit i sběrnici.

Nutno ještě jednou připomenout, že se snažíme popisovat chování obvodu, nikoli kroky, které sekvenčním prováděním označíme za algoritmus. To znamená, že můžeme bez problému prohodit třeba obsahy řádků 22 a 26 a kód půjde syntetizovat a bude popisovat stále stejný obvod.

Na používání klíčového slova `signal` jako identifikátoru vodiče můžeme pohlížet jako na obdobu funkce z procedurálního programování na úrovni dané architektury, kde namísto psaní kódu celé funkce tam, kde je ji potřeba, ji raději jednou napíšeme a pak už jen voláme. Zkrátka se tak vyhneme "DRY (Don't Repeat Yourself) kódu", který ne vždy musí být syntetizérem rozpoznán a ušetříme tak explicitně hardwarové prostředky a velmi pravděpodobně i samotnou práci syntetizéru.

Úloha č. 2 - poloviční sčítačka pomocí NOR

Tato úloha navazuje na předchozí; budete implementovat taktéž poloviční sčítačku. Nicméně v této úloze se bude implementovat tento obvod pouze za použití `nor` operátorů a musíte využít znalosti spojené s používáním `signal` jako identifikátoru vodiče. Výsledný obvod taktéž načrtněte a zaznačte vodič, na který budete vytvářet identifikátor signálem.

Je doporučeno nejprve se pokusit obvod nakreslit a pak teprve začít s jeho popisem ve VHDL, aby jste měli představu o tom, pro který vodič budete používat `signal`.

***Nápověda**: Následující konstrukce v části paralelních příkazů v `architecture` bloku je korektní:*

```
output <= sig_output;
```

Kde `sig_output` je deklarován jako `signal` uvnitř deklarační části architektury, `output` jako výstupní port architektury módu `out` a jsou stejného datového typu.

Komponenty

Komponenta (příp. modul) je základním stavebním prvkem jazyka VHDL, dokonce jsme již dvě vytvořili - první a druhá úloha. Umožňuje nám uplatňovat principy modularizace v návrhu řešení větších projektů. Komponentou tedy může být dvojice `entity` a její libovolné implementace v podobě `architecture` ze stejného zdrojového souboru. Na naší úrovni VHDL však budeme na komponentu pohlížet jako na celý zdrojový soubor, který bude vždy obsahovat jednu `entity` a k ní přiřazenou právě jednu `architecture`. Deklarace komponenty se provádí v deklarační části `architecture` komponenty vyšší úrovně, jedná se tedy o zapouzdřování.

Pro demonstraci používání komponent můžeme použít jednoduchý tester. Bude reprezentován souborem **tester.vhd**, který bude top-level entitou (tj. komponenta, kterou by neměla žádná jiná komponenta reálně deklarovat, dá se však nastavit manuálně). Uvnitř bude deklarovat `test_unit` komponentu a definuje (příp. vytvoří) její dvě instance. Tester bude předávat oběma `test_unit` identické vzorky dat a na výstupu bude signalizovat chybu hodnotou `'1'` v případě, že výsledky z obou `test_unit` nejsou shodné. V opačném případě je na výstupu hodnota `'0'`. Následuje kód:

test_unit.vhd:

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


entity test_unit is
  port(
    a: in std_logic;
    b: in std_logic;

    y: out std_logic
  );
end test_unit;


architecture test_unit_arch of test_unit is
begin

  -- nejaka implementace pracujici s a i b, vystup smeruje do y
  <implementace>

end test_unit_arch;
```

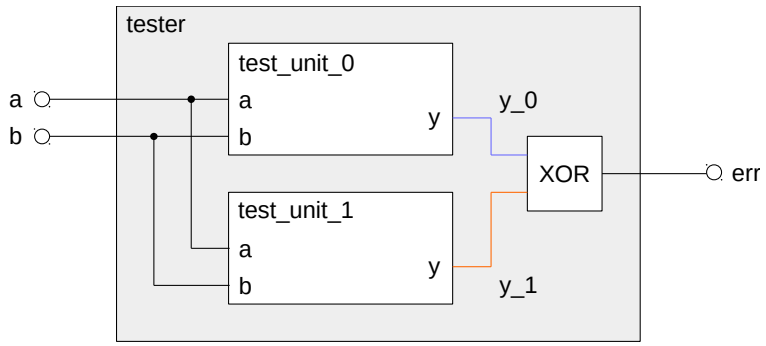
tester.vhd:

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


entity tester is
  port(
    a: in std_logic;
    b: in std_logic;

    err: out std_logic
  );
end tester;
```

`architecture` je definována na další straně.

```
architecture tester_arch of tester is

  component test_unit is -- deklarace komponenty
    port(
      a: in std_logic;
      b: in std_logic;

      y: out std_logic
    );
  end component;

  signal y_0: std_logic;
  signal y_1: std_logic;

begin

  err <= y_0 xor y_1; -- neshodnost vysledku => indikace chyby


  -- definice (vytvoreni) dvou instaci deklarovane komponenty
  test_unit_0: test_unit
  port map(
    a => a,
    b => b,

    y => y_0
  );

  test_unit_1: test_unit
  port map(
    a => a,
    b => b,

    y => y_1
  );

end tester_arch;
```

S výhodou opět používáme `signal` jako identifikátor na vodič. Následuje obrázek, který naznačuje strukturu modularizace demonstrovaného příkladu (identifikátory jsou na modrý a oranžový vodič).



Úloha č. 3 - úplná sčítačka

Vaším úkolem v této úloze bude využít již dokončené úlohy č. 1 (nejprve zkopírovat) a rozšířit ji o soubor **full_adder.vhd**, který se později stane top-level entitou. Tento soubor bude popisovat úplnou sčítačku za použití dvou instancí komponenty `half_adder`. Nezapomeňte, že úplná sčítačka má tři **vstupy** a dva **výstupy**. Načrtněte i její schéma s použitými moduly.

Základy simulace

Chceme-li náš VHDL projekt (třeba i jeden VHDL soubor) zkontrolovat syntetizérem pro jeho správnost, provádíme jeho syntézu. Budeme-li chtít zkontrolovat kompatibilitu našeho VHDL projektu s určitým FPGA, musíme nejprve úspěšně provést zmíněnou syntézu a poté můžeme spustit proces nazývaný implementace. Pokud i implementace skončí úspěšně, můžeme spustit proces generování bitového proudu (bitstreamu), kterým lze FPGA konfigurovat (naprogramovat) tak, aby svým chováním popisovalo náš VHDL kód. Vskutku dlouhý proces.

Mnohdy se spíše stává, že nepotřebujeme položit instanci našeho popisu do FPGA, ale stačil by nám nějaký softwarový nástroj, který by umožnil simulovat instanci našeho popisu. Simulovaný kód bychom pak mohli jen syntetizovat a vyhnout se tak procesu implementace a generování bitstreamu, jenž mohou pro větší FPGA a větší designy trvat opravdu dlouho. Právě k takovému účelu můžeme použít simulaci, která i mimo jiné slouží k podrobné analýze hodnot sledovaných signálů (zejména vstupy a výstupy nějaké komponenty) v závislosti na čase. Jazyk VHDL svou syntaxí přímo podpořil vznik simulací. Má v sobě totiž zabudované konstrukce, které se k tomu účelu používají. To znamená, že i chování samotné simulace budeme psát ve VHDL.

Omezíme se na standardní způsob řešení simulace - simulace jednotlivých komponent. Ideou takového postupu je vytvořit novou komponentu se stejným jménem doplněnou o "`_tb`" (Test Bench). Taková komponenta bude mít prázdnou `entity`. Např.:

```
entity or_gate_tb is
end or_gate_tb;
```

V její architektuře pak budeme deklarovat komponentu, jejíž vstupní porty a výstupní porty mají podléhat simulaci. Pak v ní deklarujeme signál pro každý vstupní port a výstupní port takové deklarované komponenty se stejným jménem a stejným datovým typem. V deklarační části architektury takové testbench komponenty to pak může vypadat následovně:

```
component or_gate is
  port(
    a: in std_logic;
    b: in std_logic;

    y: out std_logic
  );
end component;

signal a: std_logic;
signal b: std_logic;

signal y: std_logic;
```

V části paralelních příkazů poté definujeme jednu instanci již deklarované komponenty a její porty mapujeme na připravené signály. V našem příkladě to pak bude vypadat následovně:

```
or_gate_0: or_gate
port map(
  a => a,
  b => b,

  y => y
);
```

"Proč se to dělá takto? Co kdybychom raději simulovali rovnou požadovanou komponentu?"

- pak bychom nemohli posílat hodnoty na její vstupní porty v různých časech simulace, ve kterých pak i analyzujeme hodnoty výstupních portů

Poté definujeme blok `proces` pro popis samotného průběhu simulace. Blok `proces` bude v pozdějších kapitolách detailně vysvětlen. Do té doby budeme na něj nekorektně pohlížet jako na program, který se spustí na začátku simulace a kterým ji budeme řídit. Úkolem takového procesu pak bude v jednotlivých časových okamžicích posílat hodnoty na vstupní porty, podle kterých pak budeme kontrolovat očekávané hodnoty na výstupních portech. Poslední část architektury našeho příkladu by tak mohla vypadat následovně:

```
sim: process
begin

  -- pocatecni inicializace vstupnich portu
  a <= '0';
  b <= '0';
  -- mezi jednotlivymi kroky budeme cekat vzdy 10 ns
  wait for 10 ns;

  -- zmenime hodnotu vstupu a na '1' a b zustava na '0'
  -- v tuto chvili by se mel i vystup y zmenit na '1'
  a <= '1';
  wait for 10 ns;

  a <= '0';
  b <= '1';
  wait for 10 ns;

  a <= '1';
  wait for 10 ns;

  -- pokud nechceme, aby se proces opakoval, umistime na konec "wait;"
  wait;

end process sim;
```

Jak vidíme v procesu výše, vyzkoušeli jsme všechny kombinace logických hodnot pro vstupy. Vždy to ovšem nemusí být v našich silách. V takovém případě se snažíme provést simulaci pro očekávané hodnoty a přinejmenším ještě pro známé kritické kombinace vstupních hodnot.

Naše příkladová komponenta `or_gate_tb` by tedy nyní byla připravena pro simulaci. Simulaci (v podobě grafu) můžeme provést podle testebench souboru téměř v jakémkoliv vývojovém prostředí VHDL, dokonce i online (vyžaduje však registraci).

Soubor komponenty **or_gate_tb.vhd** je zveřejněn a můžete jej používat jako šablonu pro testebench soubory vašich komponent budoucích úloh. Od následující úlohy je totiž testebench soubor jejich povinnou součástí. Budete na něm demonstrovat funkcionalitu vašich popisů.

Úloha č. 4 - simulace úplné sčítačky

Tato úloha bude zaměřená pouze na testbench soubor. Zkopírujte si úlohu č. 3 a vytvořte pro její top-level `entity` testbench soubor, který bude použit pro simulaci. Vzhledem k jednoduchosti obvodu na vstupech testované komponenty simulujte všechny kombinace logických hodnot.

Vektorový logický datový typ

Vektorová podoba datového typu `std_logic` je `std_logic_vector`. Tento datový typ musí mít v době syntézy kódu známý rozsah vektoru. Jedná se tedy o obdobu staticky alokovaného pole skalárních proměnných typu `std_logic`. Jeho rozsah lze zadat dvěma způsoby:

- `std_logic_vector(x downto y)`

  nebo
- `std_logic_vector(y to x)`

Pro vektory se nicméně omezíme pouze na variantu `std_logic_vector(x downto y)`, kde je nejméně významný bit nejvíce vpravo a má index `y`. I z toho důvodu se deklarace proměnné tohoto datového typu uvádí následovně: `std_logic_vector(x downto 0)`, kde `x` + 1 je celkový počet prvků (příp. bitů) vektoru a indexovat začínáme od `0` zprava.

Budeme-li chtít pracovat s **celým vektorem**, použijeme pouze jeho identifikátor: `vector_name`, kde `vector_name` je datového typu `std_logic_vector`. Logické operace mezi dvěma vektory stejné délky jsou definovány jako operace mezi jejich jednotlivými bity.

Chceme-li pracovat pouze s **jedním bitem**, jeho hodnotu obdržíme následovně: `vector_name(x)`, kde `x` (příp. lze nahradit i za `x downto x`) je index požadovaného bitu v rozsahu daného vektoru.

A chceme-li snad pracovat s **více než jedním bitem**, ovšem ne se všemi, použijeme konstrukci `vector_name(x downto y)`, kde `x` >= `y` a `x` i `y` jsou v rozsahu daného vektoru.

Pro **sloučení vektorů do jednoho** musí být součet délek takových vektorů roven délce vektoru, do kterého sloučení směřujeme, např.: `vector_8 <= vector_3 & vector_4 & vector_1;`

Kombinační podmíněné přiřazení

Pokud budeme chtít na výstup našeho modulu přiřadit pro jednotlivé vstupní hodnoty různé funkce, může se kombinační podmíněné přiřazení hodit. Nebo můžeme taky chtít mít na výstupu modulu pro nějakou podmnožinu vstupních hodnot nějakou funkci a pro zbytek hodnot funkci rozdílnou.

Uvedeme si příklad komponenty, která má na **vstupu** 4bitový vektor `data` a na **výstupu** bitovou proměnnou `is_power`, která nám signalizuje hodnotou `'1'`, že vstupní vektor `data` reprezentuje v binární soustavě číslo, které je libovolnou celočíselnou mocninou čísla 2. V opačném případě je na výstupu hodnota `'0'`. Existují dva základní způsoby, jakými lze popsat požadované chování komponenty. Tyto dva základní způsoby si ukážeme. Oba používají shodné rozhraní (`entity`):

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4
5  entity power_indicator is
6    port(
7      data: in std_logic_vector(3 downto 0);
8
9      is_power: out std_logic
10   );
11 end power_indicator;
```

`architecture` je definována pro každý ze způsobů zvlášť na další straně.

Následuje **první** verze s použitím prostého `when`:

```
14 architecture power_indicator_arch_0 of power_indicator is
15 begin
16
17   -- = je operator rovnosti, /= je operator nerovnosti
18   is_power <= '1' when data = "0001" or data = "0010" or
19                        data = "0100" or data = "1000"
20               else '0';
21
22 end power_indicator_arch_0;
```

Syntaxe konstrukce `when` je velmi jednoduchá:

```
nejaky_signal <= hodnota_0 when logicky_vyraz else hodnota_1;
```

Syntaxe jazyka také umožňuje použít více dvojic `hodnota_n when logicky_vyraz_n else` na více řádcích. Je i velmi tolerantní k vlastnímu odřádkování v kódu (např. vizte řádek č. 19).

Druhou možností je použití konstrukce with-select:

```
25 architecture power_indicator_arch_1 of power_indicator is
26 begin
27
28   -- | znaci, ze podminka muze byt splnena vice hodnotami
29   with data select is_power <=
30     '1' when "0001" | "0010" | "0100" | "1000",
31     '0' when others;
32
33 end power_indicator_arch_1;
```

Tento zápis je rozhodně přehlednější a mnohdy i požaduje méně hardwarových prostředků. Většinou se totiž převádí na jednoduchý multiplexer. Zápis with-select dodržuje následující syntaxi:

```
with selektor select cil <=
  cilova_hodnota_0 when hodnota_selektoru_0,
  cilova_hodnota_1 when hodnota_selektoru_1,
  ...
  cilova_hodnota_n when others;
```

Nesmíme však zapomenout na větev `when others`, protože ta definuje chování systému v nevyužitých hodnotách v předchozích podmínkách `when`. I vzhledem k tomu, že se pro implementaci dvoustavové logiky používá `std_logic`, musíme větev `when others` téměř vždy definovat, protože `std_logic` může nabývat až 9 hodnot (alespoň pro simulaci) a my postihujeme především kombinace hodnot dvou - `'0'` a `'1'`. Obdobně to platí i pro první verzi s `else`, kde namísto `when others` je poslední přiřazení bezpodmínečné, tj. bez `when` (např. vizte řádek č. 20).

Statické parametry komponenty

Je-li potřeba předat komponentě nějaké statické informace v podobě vstupní konstanty, může se k tomu využít klíčové slovo `generic`. Používá se jako první blok na rozhraní komponenty, velmi podobně jako `port`, avšak neuvádí se mód (`in`, `out`, ...) u jeho položek, protože jsou pouze vstupní. Tyto statické informace lze použít i jako parametr šířky vektorů, kde má `generic` velmi široké uplatnění (nikoli však v samotném bloku `generic`). Umožňuje nám to tedy psát více generický (=> univerzální) kód a zlepšuje celkovou jeho čitelnost. Je důležité ještě poznamenat, že `generic` je pouze informace pro preprocesor syntézy a ve výsledném designu přímo nevystupuje.

Příklad:

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


entity shift_left is
  generic(
    -- cele cislo x takové, ze 2 <= x <= 64,
    -- jinak error/warning pri synteze
    WIDTH: integer range 2 to 64 := 16 -- inicializace
  );
  port(
    a: in std_logic_vector(WIDTH - 1 downto 0);

    y: out std_logic_vector(WIDTH - 1 downto 0)
  );
end shift_left;


architecture shift_left_arch of shift_left is
begin

  y <= a(WIDTH - 2 downto 0) & '0'; -- slouceni (spojeni) vektoru

end shift_left_arch;
```

V bloku `generic` inicializujeme výchozí hodnotu, která může být ve vyšší úrovni v deklaraci, příp. až v definici instance této komponenty nahrazena za jinou (při mapování v `generic map`).

Proměnná pro nás nového datového typu `integer` funguje velmi podobně jako int v C jazyku. Je možné ji použít i v samotné implementaci komponenty, syntetizér pro ni sám zajistí poskytnutí patřičných zdrojů. Nicméně je velmi důležité mu v tom pomáhat omezením rozsahu klíčovým slovem `range`. Proměnné typu `std_logic` a `std_logic_vector` jsou však svou strukturou mnohem blíže reálnému fungování obvodů, proto se na implementaci používají nejčastěji.

Úloha č. 5 - zámek

Vaším úkolem v páté úloze bude navrhnout kombinační obvod, kterým bude jednoduchý zámek. Tento zámek bude mít jeden vektorový **vstup** `key` typu `std_logic_vector` délky 4 a jeden skalární **výstup** `unlocked` typu `std_logic`. Dále bude mít jeden parametr v bloku `generic` s názvem `KEYHOLE` typu `std_logic_vector` délky 8. Parametr `KEYHOLE` inicializujte na nějakou hodnotu, jinak vám nepůjde kód syntetizovat.

Celý zámek pak bude fungovat tak, že na výstupu `unlocked` bude `'1'`, pokud je `key` obsažen v libovolném poli 4 za sebou jdoucích prvků vektorového parametru `KEYHOLE` (rotace ze začátku na konec vektoru se neuvažuje). V opačném případě bude na tomto výstupu `'0'`. Pro představu:

Nezapomeňte na testbench soubor vašeho úkolu. Testujte v něm vstupní hodnoty, které si myslíte, že by mohly mohly být důležité při demonstraci vašeho řešení. V příštích úlohách už nebudete upozorňováni. Testbench soubory tedy vždy automaticky vytvářejte (pro top-level `entity`).

Vlastní balíky a datové typy

Balíkem ve VHDL rozumíme především sbírku konstant, funkcí a vlastních datových typů, kde položky balíku mají mezi sebou úzký vztah. Jeho obsah by neměl mít k žádné komponentě lokální charakter.

Aby se balík mohl použít, musí se umístit do nějaké knihovny. Ihned po vytvoření by měl být balík implicitně umístěn ve virtuální lokální knihovně (většinou `work`). Vývojové prostředí by mělo umožnit snadnou tvorbu vlastních knihoven v rámci projektu.

Následuje velmi zjednodušený příklad demonstrující použití vlastního balíku z vlastní knihovny:

i2c_master.vhd:

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

library i2c; -- zavadime vlastni knihovnu
use i2c.sim_t.all; -- balik obsahujici datove typy pro simulaci


entity i2c_master is
  generic(
    sim_en: boolean := true -- simulace povolena
  );
  port(
    clk: in std_logic; -- jedna se o sekvencni obvod, trochu predbihame
    rst: in std_logic;
    --
    scl: inout std_logic;
    sda: inout std_logic;

    sim: out i2c_master_sim_t -- datovy typ pro simulaci i2c_master
  );
end i2c_master;


architecture i2c_master_arch of i2c_master is
begin

  -- implementace i2c na strane mastera s realizaci zapisu do sim
  <implementace>

end i2c_master_arch;
```

sim_t.vhd:

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


package sim_t is -- definice baliku

  type i2c_master_sim_t is record
    ack_error: std_logic;
    byte_out: std_logic_vector(7 downto 0);
  end record;

end sim_t;
```

V tomto příkladu jsme použili klíčové slovo `type`, kterým zavádíme vlastní datový typ. Vychází z datového typu `record`, kterým říkáme, že se jedná o výčet dvojic identifikátorů a datových typů. Přístup k položkám `record` se provádí následovně: `type_identifier.record_item_name`. Pro výše uvedený příklad: `sim.ack_error <= '1';` - tímto přesměrujeme do proměnné `sim` typu `i2c_master_sim_t` do položky `ack_error` hodnotu `'1'`. Čtení je obdobné.

Konstanty

Konstanty jsou elegantní způsob, jak systém udělat více univerzální. Jinak tomu nebude ani ve VHDL. Konstantu definujeme datovým objektem `constant`. Konstanty lokálního charakteru můžeme umístit do `architecture` nebo `process` do jejich deklaračních částí. Konstanty rozsáhlejšího charakteru se umísťují do balíků (popsány výše). Bude-li nějaká komponenta potřebovat konstantu z balíku, importuje si daný balík a může konstantu používat. Velmi často se toho využívá třeba při tvorbě testebench komponenty, ve které importujeme balíky, jenž nám umožní přiřazovat hodnoty na vstupy simulované komponenty pomocí identifikátorů konstant.

Kdyby např. ve výše uvedeném kódu v souboru **i2c_master.vhd** byla potřeba v bloku `architecture` konstanta `RST_SIM_MASK` datového typu `std_logic_vector(7 downto 0)` a my bychom věděli, že tato konstanta se bude dát použít i jinde než v komponentě `i2c_master`, jednoduše upravíme soubor **sim_t.vhd** do následující podoby:

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


package sim_t is

  -- inicializace hodnotou "10101100"
  constant RST_SIM_MASK: std_logic_vector(7 downto 0) := "10101100";

  type i2c_master_sim_t is record
    ack_error: std_logic;
    byte_out: std_logic_vector(7 downto 0);
  end record;

end sim_t;
```

Aritmetické operace

Jazyk VHDL přímo v sobě aritmetické operace pro `std_logic_vector` nezahrnuje (je to celkem pochopitelné, když i samotný datový typ `std_logic_vector` je zaváděn až balíkem `std_logic_1164`). Pokud je však potřeba aritmetické operace implementovat pro tento datový typ, je vhodné využít již existujících balíků. Ačkoli se většina požadovaných balíků nachází v knihovně `ieee`, ne všechny balíky v ní jsou, jak si později ukážeme, vydávány institutem IEEE.

Pro implementaci aritmetických operací budeme používat balík `numeric_std` z knihovny `ieee` od institutu IEEE. Mnoho hardwarových designérů však stále používá balík `std_logic_arith` z knihovny `ieee`, který svou popularitou i svým názvem připomíná standardní balík. Pravdou však je, že autorská práva k tomuto balíku vlastní Synopsys, Inc. a nejedná se tak o standardní balík, který by byl vydán institutem IEEE. Takový balík bohužel není pouze jediný.

Následující upozornění je nejspíš na místě: ačkoli práce s balíkem `std_logic_arith` a podobnými balíky může být celkově pohodlnější (méně přetypování, jednodušší a přímočařejší konstrukce), nenechte se zmást tím, že jeho používání je vám jen ku prospěchu.

Pokud existují, vždy preferujte standardní knihovny!

Součet lze pro datový typ `std_logic_vector` s importovaným balíkem `numeric_std` provést docela jednoduše:

```
vysledek <= std_logic_vector(unsigned(operand_a) + unsigned(operand_b));
```

"Docela jednoduše? Kde je ta jednoduchost?"

- jazyk VHDL je silně typovaný a tohle je jen náznak toho, co to znamená
- potřebujeme mít přehled nad naším kódem na nejnižší úrovni a pokud si snad můžeme dovolit vytvořit abstrakci nějakých jeho částí, měli bychom k tomu využívat koncepty, které jsou pro daný jazyk charakteristické (jednoduchost je tedy i v síle tohoto zápisu)

"Dobrá, co to tedy všechno znamená?"

- nejprve potřebujeme oba operandy, které jsou datového typu `std_logic_vector` stejné šířky, přetypovat na datový typ `unsigned`, se kterým přirozeně pracují funkce `numeric_std` balíku, jako je i součet
- výsledek obdržíme taktéž v datovém typu `unsigned`, proto je ještě potřeba jej přetypovat zpátky na `std_logic_vector`

"Nezpomaluje přetypování výsledný systém?"

- ani náhodou, přetypováním pouze říkáme syntetizéru, jak my proměnnou interpretujeme
- častěji se stává, že použitím balíků z knihoven se dosáhne vyšší efektivity systému oproti vlastnímu řešení (lepší využití hardwarových zdrojů, vyšší maximální frekvence)

Rozdíl a ostatní aritmetické operace se pomocí balíku `numeric_std` implementují obdobně.

Balík `numeric_std` nám umožňuje i implementovat parametrické **bitové posuny** nad datovým typem `std_logic_vector`. Zabudovanou implementaci bitových posunů přímo ve VHDL nepoužívejte, může přivodit neočekávané chování, jak uvádí několik zdrojů. Bitový posun doleva (a obdobně i doprava) se tedy za použití `numeric_std` implementuje následovně:

```
vysledek <= std_logic_vector(shift_left(unsigned(operand), pocet));
```

Kde `pocet` je datového typu `integer` a udává o kolik pozic se bude posouvat doleva `operand` datového typu `std_logic_vector`, který musí být opět předem přetypován na `unsigned`.

"Co když budu chtít místo `pocet` datového typu `integer` použít vlastní `std_logic_vector`?"

- datový typ `std_logic_vector` lze jednoduše převést na `integer` pomocí `numeric_std` a využití to nalezne i jinde, než v bitovém posunu, implementace vypadá následovně:

```
to_integer(unsigned(nas_std_logic_vector));
```

Tímto pak lze nahradit `pocet` a zvládneme implementovat parametrický bitový posun podle námi zadaného vektoru. Vektor `nas_std_logic_vector` může být i částí nějakého většího vektoru.

Za zmínku ještě stojí, že většina čísel, které ve VHDL něco definují, třeba index pro čtení bitu z vektoru, se dá nahradit výše uvedenou konstrukcí - použitím `std_logic_vector` jako index.

Úloha č. 6 - aritmeticko-logická jednotka

Jelikož už máte dostatek znalostí v návrhu kombinačních obvodů, v této úloze se pustíte do něčeho obtížnějšího. Bude to aritmeticko-logická jednotka (dále už jen ALU) jednoduchého 8bitového procesoru. ALU bude mít na **vstupu** dva operandy (`operand_a` a `operand_b`) a operační kód (`op_code`), **výstupem** bude pouze výsledek (`result`). Oba operandy i výstup výsledku mají stejnou šířku jako procesor a jsou datového typu `std_logic_vector`. ALU bude umět provést 8 aritmeticko-logických operací (definovány níže) mezi vstupními operandy a výsledek operace pak bude propagovat na `result`. Operaci, která se bude provádět, určuje hodnota `op_code`. Vstup operačního kódu si navrhněte celý sami podle uvážení (použijte však znalostí již probraných látek).

Dále budete implementovat i vlastní knihovnu, kterou pojmenujete `cpu_lib`. Tato knihovna bude obsahovat balík `alu_const`, který si importujete do své ALU komponenty i jejího testbench souboru. Balík `alu_const` bude popisovat výčet konstant pro operační kód, aby byl VHDL kód celkově lépe čitelný. Vytvoříte tím i nějaké rozhraní pro toho, kdo importuje `alu_const` balík do své komponenty na vstupu operačního kódu. Rozsáhlejší využití by to zřejmě mělo, kdyby se `alu_const` importoval do komponenty vyšší úrovně (kterou však implementovat nebudete). Konstanty z tohoto balíku musíte používat i ve vlastním kódu.

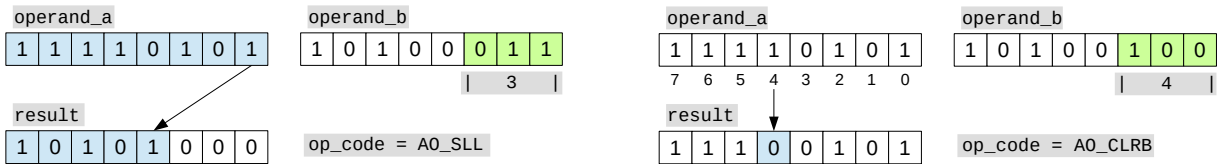
Výčet operací ALU a jejich zápis konstantou:

- negace logického součinu - `AO_NAND` (vzniklo ze zkratky ALU Opcode NAND)
- negace ekvivalence - `AO_XOR`
- logický posun o n pozic doleva - `AO_SLL`
- logický posun o n pozic doprava - `AO_SRL`
- nastavení hodnoty n-tého bitu na `'0'` - `AO_CLRB`
- nastavení hodnoty n-tého bitu na `'1'` - `AO_SETB`
- aritmetický součet - `AO_ADD`
- aritmetický rozdíl - `AO_SUB` (`operand_a` - `operand_b`)

Operační kódy `AO_SLL`, `AO_SRL`, `AO_CLRB` a `AO_SETB` se odvolávají na n-tý bit/n pozic. Všechny tyto operace se provádí nad prvním operandem a zmíněné n je určeno podle třech spodních bitů druhého operandu (aritmeticky nám to dává rozsah 0 - 7) a použito následovně:

- v případě `AO_CLRB` a `AO_SETB` se ve výsledku tato hodnota projeví jako index do pole bitů prvního operandu s tím, že se na tomto indexu provede patřičná operace
- v případě `AO_SLL` a `AO_SRL` tato hodnota reprezentuje o kolik bitů se provede bitový posun

Příklad:



***Nápověda**: Pro implementaci `AO_CLRB` a `AO_SETB` je vhodné si vytvořit bitovou masku.*

Kombinační proces

Proces již instinktivně používáme při popisu chování simulací našich komponent na základě naší nekorektní dočasné definice jako programu. Nyní si však detailně vysvětlíme, co proces je (konkrétně kombinační proces) a proč to vlastně obdoba programu není. Pamatujte, že v procesu pro simulace se většinou používají odlišné konstrukce než v procesu pro popis chování navrhovaného obvodu.

Kombinační proces je tedy speciální konstrukce jazyka VHDL, která se umísťuje (stejně jako proces obecně) do části paralelních příkazů architektury, tj. za `begin`, který náleží `architecture`. Pro jeho vytvoření používáme klíčové slovo `process`. Vytvořený proces se dělí na dvě části: deklarační část (před `begin`) a část sekvenčních příkazů (po `begin`). `process` bloků může být v rámci jedné architektury libovolný počet. Existují však významná omezení, která potenciálně velký počet těchto `process` bloků redukují: nesmíme zapisovat do signálu, který je deklarován v deklarační části `architecture` bloku, příp. výstupního portu, z více než jednoho procesu a zároveň nesmí být do tohoto signálu/portu v části paralelních příkazu dané architektury přiřazena žádná hodnota. Pokud si dokážeme představit, jak se tyto konstrukce implementují v hardwaru, tato omezení jsou celkem pochopitelná. Blok `process` dodržuje následující syntaxi:

```
nazev: process(<citlivostni_seznam>)
  -- deklaracni cast
begin
  -- cast sekvencnich prikazu
end process nazev;
```

`nazev` je nepovinný, je však doporučený a my jej používat budeme. Kdybychom náhodou nevěděli, jaký název procesu zvolit, můžeme použít `main`.

`<citlivostni_seznam>` je seznam signálů a vstupních portů oddělených "`,`". Idea procesu je jednoduchá - kdykoli se změní hodnota jakéhokoli signálu nebo vstupního portu uvedeného v tomto seznamu, tak se vykoná část sekvenčních příkazů.

V **deklarační části** procesu můžeme proměnné lokálního charakteru deklarovat jako datový objekt `variable`, což je obdobou deklarace signálu v deklarační části architektury. Nicméně v procesu můžeme i nadále pracovat se signály definovanými v téže architektuře. A právě způsob přístupu k proměnným objektu `variable` a `signal` se v **části sekvenčních příkazů** procesu zásadně liší:

- zápis do proměnné objektu `variable` má v rámci vykonávání procesu okamžitou platnost a provádí se operátorem "`:=`", např. `int := irq and ie;`
  - *způsob práce s `variable` proměnnými uvnitř procesu může trochu připomínat styl imperativního programování (odtud pojem část sekvenčních příkazů)*
- všechny zápisy do proměnných objektu `signal` mají účinek až na konci procesu (všechny zápisy se provedou "na konci najednou") a provádí se standardně operátorem "`<=`", je-li zaznamenáno více zápisů v procesu do jednoho signálu, uplatní se poslední platný z nich

Jako **první příklad** si můžeme ukázat ekvivalentní zápis architektury našeho prvního příkladu na kombinační logiku z kapitoly Základní kombinační prvky. Implementace architektury zmíněného příkladu byla následující:

```
y <= a xnor b;
```

Ekvivalentní zápis výše uvedeného pomocí kombinačního procesu by mohl vypadat následovně:

```
main: process(a, b)
begin

  y <= a xnor b;

end process main;
```

Vykonávání části sekvenčních příkazů procesu může být větveno. Pro **jednoduché větvení** se k tomuto účelu používají klíčová slova `if`, `elsif` a `else`, která doprovází `then` a `end if`. Jejich syntaxe je následující:

```
-- zavorky mezi if a then nejsou povinne, ale zlepsuji citelnost kodu
if (logicky_vyraz_0) then
  <sekvencni_prikaz(y)_0>
-- elsif a else vetve nejsou obecne v if bloku procesu povinne
elsif (logicky_vyraz_1) then
  <sekvencni_prikaz(y)_1>
else
  <sekvencni_prikaz(y)_2>
end if;
```

Pro **rozsáhlejší a více strukturované větvení** se používá `case` blok. Jedná se o obdobu with-select konstrukce používané v sekci paralelních příkazů bloku `architecture`. Stejně jako v bloku with-select se i zde může místo jednotlivých hodnot selektorů použít operátor `|` a umožnit tím reagovat stejně na více hodnot. Pro příklad vizte níže řádek č. 4. Syntaxe `case` bloku je následující:

```
1  case selektor is
2    when hodnota_selektoru_0 =>
3      <sekvencni_prikaz(y)_0>
4    when hotnota_selektoru_1 | hodnota_selektoru_2 =>
5      <sekvencni_prikaz(y)_1_2>
6    ...
7    -- tato vetev je obecne povinna, nicmene nemusi nic vykonat
8    when others =>
9      -- pouzijeme "NULL;", pokud nechceme vykonat nic
10     <sekvencni_prikaz(y)_n>
11 end case;
```

Důležité je ještě poznamenat, že tato pro nás nová klíčová slova se mohou používat pouze uvnitř procesu v jeho části sekvenčních příkazů. V procesu se i proto nemůže používat většina konstrukcí z části paralelních příkazů architektury (jmenovitě především konstrukce `when` a with-select blok).

Jako další příklad kombinačního procesu si můžeme uvést proces, který se bude chovat jako převodník ze `std_logic_vector` do `integer`. Je asi vhodné poznamenat, že se nejedná o obdobu funkce pro přetypování, zde se však jedná o popis kombinačního obvodu, který by mimo jiné tímto převodem mohl zpomalovat výsledný systém (na rozdíl od funkcí pro přetypování). Proces bude **číst** 1bitový signál `en` (ENable) a 2bitový vektorový signál `binary`. Oba tyto signály jsou datového typu standardní logiky. **Zapisovat** bude do `unsigned_decimal` datového typu `integer`. Pokud bude na signálu `en` hodnota `'1'`, tak proces bude převádět vstup `binary` přirozeně v binární soustavě do jeho beze znaménkové podoby v desítkové soustavě na výstup `unsigned_decimal`. Pokud bude na `en` hodnota `'0'`, výstup `unsigned_decimal` bude mít hodnotu `-1`. Možná podoba takto definovaného procesu je uvedena na další straně.

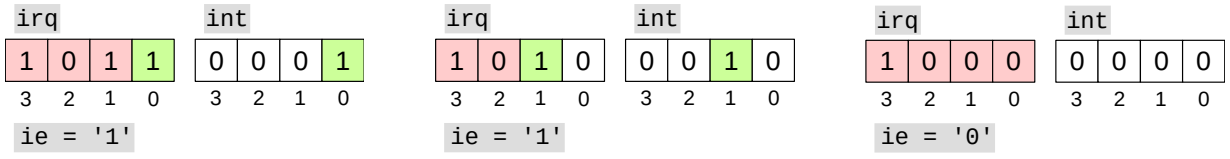
```
binary_to_udecimal: process(en, binary)
begin

  -- zapis vychozi hodnoty unsigned_decimal
  unsigned_decimal <= -1;

  if (en = '1') then
    -- hodnota unsigned_decimal na konci procesu jiz nebude -1
    case binary is
      when "00" =>
        unsigned_decimal <= 0;
      when "01" =>
        unsigned_decimal <= 1;
      when "10" =>
        unsigned_decimal <= 2;
      when others =>
        unsigned_decimal <= 3;
    end case;
    -- uplatnil se totiz posledni platny zapis
  end if;

end process binary_to_udecimal;
```

Výše v uvedených VHDL kódech při popisu větvení (`if` blok a `case` blok) v procesech bylo v komentářích uvedeno, že na rozdíl od konstrukcí v části paralelních příkazů architektury, nemusíme vždy pokrýt, příp. vykonat, při větvení větve všech kombinací hodnot proměnných, podle kterých větvíme. Nyní by bylo velmi vhodné poznamenat, že pokud chceme zachovat vlastnosti kombinačního procesu (který momentálně popisujeme), tak existují fundamentální podmínky, které musíme vždy splnit a jejich výklad nám napomůže k pochopení toho, jak to vlastně s větvením uvnitř procesu je (zejména druhá podmínka). Jsou to tyto podmínky:

- všechny signály a porty, které se uvnitř procesu čtou, se musí uvést do citlivostního seznamu
- libovolná kombinace hodnot všech signálů a vstupních portů, které se uvnitř procesu čtou, musí vždy aktualizovat všechny signály a výstupní porty, do kterých se zapisuje
    - *jinými slovy: neexistuje korektně definovaný digitální kombinační obvod, který má pro nějakou kombinaci vstupních hodnot nedefinovanou výstupní hodnotu*

Úloha č. 7 - řadič přerušení

V této úloze budete navrhovat kombinační obvod pomocí kombinačního procesu. Jedná se o řadič přerušení pro velmi jednoduchý procesor. Na **vstupu** architektury bude mít 1bitový signál `ie` (Interrupt Enable) a 4bitový vektorový signál `irq` (Interrupt ReQuest). Jeho **výstupem** pak bude 4bitový vektor `int` (INTerrupt), který by vedl přímo do procesoru. Tyto vstupy a výstupy budou datového typu standardní logiky a se všemi bude pracovat váš proces. **Číst** tak bude z `ie` a `irq` a **zapisovat** pouze do `int`. Hodnota `'1'` na n-tém bitu `int` by pak vyvolala v procesoru n-tou rutinu obsluhy přerušení (jedná se o obdobu funkce). Jednotlivé bity `irq` na vstupu vaší `entity` by vystupovaly ze čtyř zařízení, která by tímto signálem skrze váš řadič mohla požádat o přerušení procesoru. Samotný proces bude mít za úkol aplikovat na vektor `irq` statickou prioritu podle n-tého bitu, kde nižší hodnota n znamená vyšší prioritu. Tuto žádost o přerušení bude propagovat na n-tý bit `int`, pokud budou žádosti o přerušení povoleny signálem `ie` v hodnotě `'1'`. To znamená, že do `int` se vždy zapíše maximálně jedna hodnota `'1'` na nejvíce pravou pozici bitu, kde `irq` na stejné pozici bitu má hodnotu `'1'`. Demonstrace způsobu práce vašeho modulu:



Základy sekvenční logiky

Klopné obvody, které jsou základem sekvenční logiky, nám VHDL umožňuje implementovat právě pomocí bloku `process`. Proces tedy v popisu VHDL našeho designu neslouží jenom k tomu, aby nám umožnil strukturovaně popsat kombinační obvod, případně jeho část. Někteří jej dokonce k tomu účelu vůbec nepoužívají. Jeho skutečná síla je právě v sekvenční logice.

Pro proces sekvenční logiky platí všechna tvrzení z předchozí kapitoly Kombinační proces kromě těch, která byla směřována výhradně na kombinační proces. Zejména se jednalo o určitá omezení, která by nám mohla vytvořit sekvenční obvod namísto kombinačního obvodu (což nebylo žádoucí). Proto se v této kapitole zaměříme spíše na postupy, které se v procesu sekvenční logiky liší.

Ačkoli nám VHDL umožňuje explicitně popsat všechny základní klopné obvody a pak je používat jako komponenty, velmi často se v tomto směru využívá spíše implicitní implementace registrů, která je přímou součástí jazyka. Syntetizér pak může efektivně využít zabudovaných registrů v FPGA namísto jejich implementace do propojení univerzálních LUT. Programátor si pak jen pohodlně zvolí jejich citlivost na hladinovou úroveň signálu nebo jeho hranu.

Je třeba ještě upozornit, že pokud se použijí v projektu VHDL registry, které jsou citlivé na hranu nějakého signálu (zejména hodinového), tato hrana by měl být napříč celým VHDL projektem identická - používají se jen náběžné hrany nebo jen sestupné hrany, nikoli však kombinace obou.

Proměnné datových objektů `variable` i `signal` mohou být použity k implementaci obou výše uvedených registrů - hladinová citlivost a citlivost na hranu signálu. Následuje příklad demonstrující zápis do registru s **hladinovou citlivostí** datového objektu `signal`, který je definován v patřičné deklarační části architektury následovně:

```
signal d_reg: std_logic;
```

Můžeme si povšimnout, že se jedná o syntakticky stejnou deklaraci jako v případě kapitoly Signál jako identifikátor vodiče. Signál může být totiž chápán dvěma způsoby. První způsob byl ukázán ve zmíněné kapitole, druhý způsob vytvoří ze signálu registr (případně více registrů - vektor).

Tento příklad vystupuje s následujícím procesem:

```
level_sig_reg: process(en)
begin
  -- kontrola hodnoty pro hladinove prirazeni
  if (en = '1') then
    -- d_reg_in muze byt jiny signal nebo vstupni port entity
    d_reg <= d_reg_in;
  end if;
end process level_sig_reg;
```

Další příklad demonstruje inkrementaci hodnoty registru s **citlivostí na náběžnou hranu** hodinového signálu. V tomto případě ovšem bude použit speciální příklad s datovým objektem `variable`. Níže si vysvětlíme, proč je příklad speciální.

```
edge_var_reg: process(clk)
  variable count: integer := 0; -- pocatecni inicializace
begin
  if (rising_edge(clk)) then -- funkce pro kontrolu nabezne hrany
    -- prvni se cte predchozi hodnota, pak se zapisuje nova
    count := count + 1;
  end if;
end process edge_var_reg;
```

Příklad je speciální svým použitím `variable` jako registru. Proměnná datového objektu `variable` má totiž pro vznik registru speciální požadavek na rozdíl od proměnné objektu `signal`. Vychází to z faktu, že `variable` proměnné mají lokální charakter uvnitř procesu a z vnějšku k nim nelze standardním způsobem přistupovat. Tento požadavek je následující: proměnná objektu `variable` se pro vytvoření registru musí uvnitř procesu alespoň při jedné kombinaci řídicích hodnot dříve číst, než proběhne zápis do ní. Tento požadavek jsme splnili. Kdybychom však implementovali pomocí `variable` obdobu prvního procesu `level_sig_reg` (třeba i s hladinovou citlivostí), proces vypadal následovně:

```
level_var_reg: process(en)
  variable d_reg: std_logic;
begin
  -- kontrola hodnoty pro hladinove prirazeni
  if (en = '1') then
    -- d_reg_in muze byt signal nebo vstupni port entity
    d_reg := d_reg_in;
  end if;
end process level_var_reg;
```

Je asi zřejmé, že v tomto případě jsme zmíněný požadavek nesplnili. Hodnotu proměnné dokonce nečteme ani po zápisu. Není tedy důvod, aby syntetizér vytvořil registr (pokud to rozpozná).

"Proč to tak je?"

- vysvětlení je jednoduché: pokud nečteme `variable` dříve, než do něj zapisujeme, syntetizér nemá důvod implementovat k vůli tomu registr, raději tento `variable` použije jako identifikátor na vodič, jenž je obrazem přiřazované funkce pro aktuální průběh procesu

Sluší se ještě vzpomenout, že tento proces `level_var_reg` stejně jako předchozí `edge_var_reg` nepopisují žádnou funkci nějakého výstupního portu `entity` v závislosti na vstupních portech téže `entity`. Může se tedy stát, že syntetizér takový proces zcela odstraní.

Na tomto místě je taky velmi vhodné upozornit, že pokud výstupní porty top-level `entity` našeho VHDL projektu nebudou definovány funkcí definovanou i vstupními porty této `entity`, nebo pokud bude chování našeho VHDL popisu vykazovat velmi jednoduché funkce na výstupních portech této `entity`, může se stát, že syntetizér celý náš VHDL popis zjednoduší na nejjednodušší možnou implementaci (příp. i prázdnou implementaci). Tato výsledná implementace bude mít totiž stejné chování ke vstupům a výstupům top-level `entity`.

Inicializace sekvenčního obvodu

Asi není pochyb o tom, že by bylo vhodné, kdybychom uměli digitální systémy inicializovat do nějakého známého (příp. počátečního, výchozího, iniciálního) stavu. Pokud pracujeme s nějakou proměnnou, kterou jsme žádným způsobem neinicializovali, nemůžeme o její hodnotě předpokládat vůbec nic. Proto si vysvětlíme několik způsobů, kterými se dá dosáhnout inicializace sekvenčních obvodů.

Ačkoli požadované množství hardwarových prostředků obvodu pro reset zpravidla nebývá veliké, je vhodné tento obvod omezit jen na nutné množství proměnných. Každá proměnná tedy nemusí mít svůj stav známý neprodleně po resetu. Musíme však dát pozor, abychom absencí inicializace takové proměnné nedostali později obvod do nedeterministického stavu.

První způsob již známe. Jedná se o **inicializaci pomocí operátoru** "`:=`" za deklaraci proměnné libovolného nám známého datového objektu. Na úvod by bylo vhodné ihned konstatovat, že tato konstrukce je pro proměnnou téměř neproveditelné implementovat v reálném hardwaru (ASIC). Reálné využití tak nalezne především při definici konstant.

Nicméně i tak inicializace proměnných ve VHDL tímto operátorem není syntakticky špatně a dokonce ji při práci s pokročilejšími typy FPGA lze implementovat. Pokud je tedy VHDL projekt zaměřen pouze na takové FPGA, lze této inicializaci připsat i využití zde.

V předchozí kapitole Základy sekvenční logiky jsme si definovali dva způsoby řízení sekvenčního obvodu. Zde i dále budeme používat pouze procesy citlivé na hranu signálu (konkrétně na náběžnou hranu hodinového signálu `clk`).

Předchozí způsob neumožňuje opakovaný návrat do známého stavu sekvenčního obvodu. Jedná se tedy pouze o jednou vykonanou inicializaci (v některých případech to může stačit). Následující dva další způsoby však budou tento nedostatek eliminovat a jsou standardním řešením inicializace obvodu (známý stav obvodu). Jsou to metody **synchronního resetu** a **asynchronního resetu**.

Na příkladu procesu implementovaného těmito dvěma metodami si ukážeme rozdíly. Do tohoto procesu bude vstupovat signál `clk` vstupního portu entity a bude mít za úkol na signálu `clk_div` propagovat signál o poloviční frekvenci `clk` se střídou 1:1. Signál `clk_div` je datového typu `std_logic` datového objektu `signal` a je definován v deklarační části architektury následovně:

```
signal clk_div: std_logic; -- zadna pocatecni inicializace
```

Vstupní port `rst` v `'1'` nastaví proměnné `clk_div` hodnotu `'0'`. Procesy tak mohou vypadat následovně (vlevo synchronní reset, vpravo asynchronní reset):

```
sync_reset: process(clk)
begin
  if (rising_edge(clk)) then
    if (rst = '1') then
      clk_div <= '0';
    else
      clk_div <= not clk_div;
    end if;
  end if;
end process sync_reset;
```

```
async_reset: process(clk, rst)
begin
  if (rst = '1') then
    clk_div <= '0';
  elsif (rising_edge(clk)) then
    clk_div <= not clk_div;
  end if;
end process async_reset;
```

Asi je zřejmé, že celý sekvenční obvod se musí nejprve resetovat a teprve poté jsme schopni definovat hodnotu na výstupu vzhledem k hodnotám vstupu. V uvedeném příkladu není neprovedení resetu tak kritické, ovšem u větších obvodů může způsobit nemalé problémy.

Můžeme vidět, že procesy se liší hned při jejich deklaraci. Asynchronní implementace totiž musí do svého citlivostního seznamu obsáhnout i `rst` signál a tím tak zruší vztah signálu `clk` a `rst`. Důsledkem toho je, že `rst` může být sepnut do hodnoty `'1'` i na menší dobu, než je perioda hodinového signálu `clk` a obvod tak uvést do iniciálního stavu, což v synchronní variantě vždy nelze. Pro představu (vlevo synchronní reset, vpravo asynchronní reset):

V asynchronní variantě jsme tedy schopni dosáhnout nižší odezvy resetu sekvenčního obvodu, nemusíme totiž čekat na náběžnou hranu hodinového signálu. Nicméně se dopouštíme toho, že chování takového obvodu nemusí být vždy definováno. Právě v případě, kdy se sestupná hrana signálu `rst` potká s náběžnou hranou signálu `clk` ve stejný okamžik, není chování takového obvodu definováno. Pokud bychom chtěli takový problém vyřešit, vztah mezi `clk` a `rst` už nebude úplně zrušen. Výše uvedená tvrzení platí i v obecnějších případech.

I z toho důvodu se častěji používá synchronní reset a my jej budeme používat taky. Musíme však dohlédnout na to, aby signály, které řídí tento sekvenční obvod (v našem případě `rst`) měli dobu platnosti svých hodnot větší nebo alespoň rovnu hodnotě periody hodinového signálu. Syntetizér nás pak odmění plně synchronizovaným sekvenčním obvodem a občas i skromnějšími požadavky na hardwarové prostředky.

Simulace sekvenčních obvodů

Jelikož se simulace sekvenčních obvodů mírně liší od simulace kombinačních obvodů rozebírané v kapitole Základy simulace, bude i tomuto tématu věnována velmi krátká kapitola, která popíše jak navrhnout testbench soubor pro sekvenční obvody. Hlavní a téměř jediná problematika, které nyní čelíme, je nalezení způsobu implementace hodinového signálu. Proto v případě, když pracujeme na testbench souboru sekvenčních obvodů, postupujeme stejně, jako tomu bylo u kombinačních obvodů, s výjimkou práce s hodinovým signálem `clk`. Pro hodinový signál totiž vytvoříme samostatný proces, který bude `clk` signál řídit. Takový proces bude vypadat téměř identicky pro každý testbench soubor. Následuje jeho demonstrace:

```
sim_clk: process
begin

  -- cyklus se znamy poctem opakovani
  for i in 0 to 63 loop
    -- perioda hodinoveho signalu bude 10 ns
    clk <= '0';
    wait for 5 ns;
    clk <= '1';
    wait for 5 ns;
  end loop;

  wait;

end process sim_clk;
```

Tento proces tedy zajistí střídání hodnoty na `clk` signálu 64 krát s periodou 10 ns. To by mohlo pro naše sekvenční designy stačit. Nezapomeňte pak, že perioda `clk` by měla být rovna jednotlivým zpožděním mezi přiřazení hodnot na vstupy simulované komponenty v procesu pro simulaci (`sim`).

Vzhledem k tomu, že většina simulátorů VHDL nevykonává simulaci při spuštění do té doby, než její všechny procesy nenarazí na `wait;`, ale mají definovanou nějakou výchozí dobu jako zarážku (např. 1 ms), výše uvedený cyklus `for` by se v takových případech mohl úplně vypustit. Proces by se totiž spouštěl pořád znovu, protože nemá citlivostní seznam. Ovšem v některých simulátorech by to naopak mohlo vést ke snaze vykonat simulaci po nekonečně dlouhou dobu a program simulace by pak mohl nekonečně cyklit.

Obecně se cykly ve VHLD používají především pro simulace a zpracování preprocesorem syntetizéru (třeba pro generování instancí komponent), na implementaci se používají však zřídkakdy a jejich syntéza není stejně tak přímočará, jako jejich syntaxe.

Soubor **d_reg_tb.vhd** s kompletní demonstračním kódem je zveřejněn a můžete z něj vycházet při návrhu testbench souborů pro sekvenční obvody. Jedná se o testbench D klopného obvodu citlivého na náběžnou hranu hodinového signálu. Jelikož se v mnohých sekvenčních obvodech používá i signál pro synchronní reset, je i zde implementován (ačkoli D registr jej nepotřebuje), aby vám tato šablona usnadnila práci.

Úloha č. 8 - pulzně šířková modulace

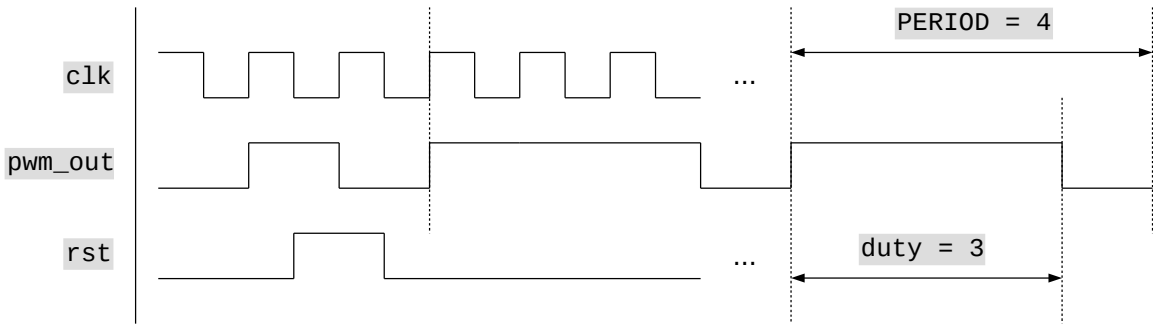
V tomto úkolu se podíváte na pulzně šířkovou modulaci (dále už jen PWM). Implementovat ji však budete trochu netradičně (budete používat `integer` namísto `std_logic_vector`), a tak je možné, že nebude implementace nejoptimálnější. Nicméně právě díky tomu se můžete na VHDL podívat i z jiného pohledu, kde je elegantní a velmi abstraktní (na poměry jazyka popisující hardware). A právě to je cílem této úlohy.

Vaším úkolem tedy bude vytvořit VHDL modul, jehož **vstupy** budou `clk` a `rst` datového typu `std_logic` a `duty` datového typu `integer`. **Výstupem** pak bude pouze `pwm_out`, který je datového typu `std_logic`.

Vstup `clk` slouží standardně jako hodinový signál, vstup `rst` hodnotou `'1'` vyvolá synchronní reset vašeho obvodu. Tento reset si navrhněte sami, musíte však splnit, že jeho vyvolání přiřadí hodnotu `'0'` na výstup `pwm_out`. Inicializace jakýchkoliv proměnných není povolena (musíte použít váš synchronní reset), případná deklarace dalších proměnných (registrů) je omezena pouze na datový typ `integer`.

Váš modul bude mít i na rozhraní v `generic` bloku parametr `PERIOD` datového typu `integer`, který bude udávat, kolik náběžných hran hodinového signálu tvoří periodu celého PWM cyklu. Tento parametr inicializujte na nějakou přijatelnou hodnotu (např. `256`).

Vstup `duty` je datového typu `integer` a určuje, kolik náběžných hran od začátku periody PWM (po resetu) bude mít výstup `pwm_out` hodnotu `'1'`, ve zbylých případech bude mít hodnotu `'0'`. Nezapomeňte si ji proto na začátku PWM cyklu uložit do registru, aby se hodnota `duty` na vstupu mohla bez problému měnit i v průběhu PWM cyklu. Tato hodnota (stejně jako hodnota zmíněného registru) by neměla být větší než `PERIOD`. Naznačte to pomocí omezení rozsahu (`range`).



***Nápověda**: Z výstupního portu (např. `pwm_out`) lze vytvořit registr obdobně, jako je tomu u datového objektu `signal` (přiřazením v procesu sekvenční logiky). Ve starších verzích VHDL jej však nesmíte číst nikde v architektuře (což by zde stejně nemělo moc vadit).*

Bloková paměť

Doposud jsme pracovali s kombinační logikou, která se do FPGA implementuje pomocí univerzálních LUT. Taky jsme pracovali se sekvenční logikou, která se implementuje do FPGA pomocí zabudovaných registrů. Tato sekvenční logika se skládala nejvíce z několika registrů. Pro implementaci velkého pole registrů by nám však dozajista tyto prostředky FPGA (registry) nemusely stačit. Z toho důvodu jeho výrobci do FPGA velmi často implementují i blokovou paměť. Předtím, než si popíšeme, jak univerzálně a mnohdy i přenositelně pracovat s blokovou pamětí ve VHDL, musíme si popsat jednu vlastnost pole registrů, na kterém je bloková paměť založena.

Tou vlastností je způsob čtení takového pole registrů. Dělí se na **synchronní** způsob čtení a **asynchronní** způsob čtení. Následuje příklad demonstrující práci s polem čtyř 16bitových registrů, které je definováno v deklarační části architektury takto:

```
-- zavadime vlastni datovy typ pole o 4 polozkach 16bitovych promennych
type array_t is array (3 downto 0) of std_logic_vector(15 downto 0);
-- deklarujeme reg_array jako zavedeny typ s pripadnou inicializaci
signal reg_array: array_t := (
  1 => "0000111100001111",
  3 => "0011001100110011",
  others => "0000000000000000" -- inicializace všech nedefinovanych adres
);
```

Čtení tohoto pole adresujeme signálem `a_rd`, zápis do něj signálem `a_wr`. Oba jsou datového typu `integer`. Podle tohoto způsobu práce s adresami má pole na vstupu **dual-port**. O **single-port** by se jednalo, kdyby signál adresy byl identický pro zápis i čtení. Zápis do pole se povoluje signálem `wr_en` a čtení se provádí vždy nepodmíněně. Proces popisující takové chování deklarovaného pole by mohl vypadat následovně (nalevo synchronní čtení, napravo asynchronní čtení):

```
sync_read: process(clk)
begin
  if (rising_edge(clk)) then

    -- zapis, reg_array(a_rd) se
    -- necte nikde v architekture
    if (wr_en = '1') then
      reg_array(a_wr) <= data_in;
    end if;

    -- synchronni cteni, tento
    -- signal se cte v architekture
    data_out <= reg_array(a_rd);

  end if;
end process sync_read;
```

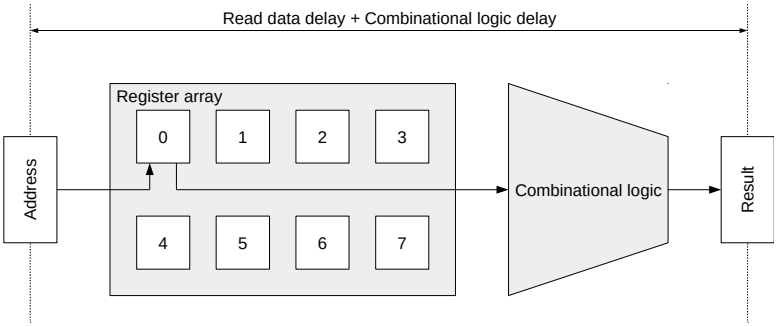
```
async_read: process(clk)
begin
  if (rising_edge(clk)) then

    -- zapis, reg_array(a_rd) se
    -- cte primo v architekture
    if (wr_en = '1') then
      reg_array(a_wr) <= data_in;
    end if;

  end if;
end process async_read;
```

Když tedy porovnáme výše uvedené, je zřejmé, že v synchronní variantě budeme mít přečtená data k dispozici vždy o jednu periodu hodinového signálu později, zatímco v asynchronní variantě budeme moci ihned používat aktuální výsledek.

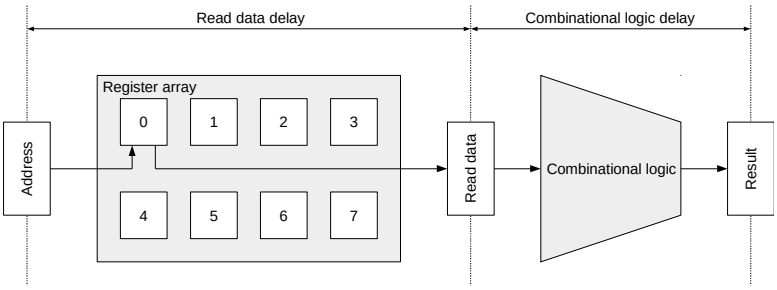
"Jaký je tedy důvod implementace synchronní čtení?"

- prvním důvodem je, že dedikovaná bloková paměť zvoleného FPGA nemusí vždy podporovat asynchronní čtení (pokud tedy cílíme na ni)
- druhý důvod je důležitější a pojednávají o něm dva následující odstavce na další straně

Doba čtení a zápisu pole registrů implementovaného jakýmkoliv způsobem trvá obecně déle, než operace čtení a zápisu nad jedním registrem (o to více to platí pro velká pole). Pokud využijeme asynchronní čtení pole, vždy tím snižujeme maximální frekvenci, pod kterou náš obvod může pracovat. Samotné čtení pole nějakou dobu trvá a k této době se ještě musí přičíst doba worst-casu (doba nejdelší možné cesty) čtených dat v kombinační části obvodu, kde se s výstupem pracuje. Následuje grafická podoba asynchronně implementovaného čtení:



Použijeme-li na druhou stranu při práci s polem čtení synchronní, téměř vždy tím neovlivníme maximální frekvenci, pod kterou náš obvod může pracovat. Samotné čtení z paměti si totiž můžeme představit jako kombinační obvod, na jehož konci je záchytný registr (z vyššího příkladu procesu nalevo to byl `data_out`). Hodnota tohoto registru je pak vstupem do kombinační části obvodu, kde se s ní pracuje jako se čtenou hodnotou pole se zpožděním jedné periody hodinového signálu. Maximální frekvence je tak ovlivněna pouze delším worst-casem z těchto dvou částí, na rozdíl od asynchronního čtení, kde je jejich součtem. Pro představu synchronní implementace čtení:



"Kdy tedy použít synchronní čtení a kdy asynchronní?"

- synchronní čtení používáme zejména při implementaci větších polí (prodleva přístupu je vyšší), kde nevadí zpoždění o jednu periodu hodinového signálu (např. operační paměť jednodušších procesorů)
- asynchronní čtení samozřejmě používáme u jednotlivých registrů a zejména u menších polí, které je potřeba číst ve stejné periodě a kde prodleva čtení takového pole navíc moc nevadí (např. sada registrů jednodušších procesorů)

"A kdy vůbec použít blokovou paměť a jak?"

- jelikož si popisujeme univerzální způsob použití takové paměti, bylo by asi vhodné poznamenat, že se náš postup velmi opírá o syntetizér spíše než explicitní konstrukci VHDL
- my tedy pracovat přímo s VHDL komponentou blokové paměti nebudeme (ačkoli i to je možné), syntetizér by měl poznat sám, kdy je potřeba ji implementovat (dnes už to většina umí), právě proto by bylo vhodné spíše uvést za jakých podmínek lze očekávat, že ji syntetizér implementuje - tomuto tématu se budeme věnovat na následující straně

Bloková paměť je otázkou implementace do konkrétního FPGA, proto vždy musíme zkontrolovat podporu blokové paměti daného FPGA. Nicméně téměř se stoprocentní jistotou se stává, že dané FPGA podporuje blokovou paměť typu single-port i dual-port synchronně čtenou. Jiné možnosti už tedy nadále uvažovat nebudeme.

Pokud tedy chceme, aby syntetizér implementovat část našeho VHDL popisu jako blokovou paměť popsanou výše, musíme zajistit několik následujících podmínek:

- nesmíme číst z paměti asynchronně (tj. zejména v části paralelních příkazů architektury)

```
data_out <= reg_array(a_rd); -- pouze v sekvencnim procesu
```

- nesmíme provádět více operací čtení/zápisu různých adres než dvě z důvodu dual-portu (čtení a zápis stejné adresy však využije pouze jeden port paměti, musíme však myslet na to, že při čtení obdržíme hodnotu, kterou bychom právě při souběžném zápisu přepsali)

```
-- zakazano i v procesu
data_out_0 <= reg_array(a_rd_0);
data_out_1 <= reg_array(a_rd_1);
data_out_2 <= reg_array(a_rd_2);
```

"Co se stane, když syntetizér blokovou paměť neimplementuje?"

- namísto blokové paměti se pokusí použít stavební bloky FPGA pro sekvenční logiku

Pro představu je pro vás zveřejněn soubor **mem_sp.vhd**, který je kompletním řešením single-port paměti se synchronním čtením a syntetizér by jej měl při implementaci interpretovat jako blokovou paměť - to si můžete ověřit ve výpisu syntézy, kde by o tom mělo být hlášení. Tento soubor můžete také použít k vypracování poslední úlohy.

Příklad na sekvenční logiku

Jelikož příkladů na kombinační obvody bylo poměrně dost a sekvenční logika je obecně trochu obtížnější na pochopení, v této poslední kapitole se budeme věnovat pouze jednomu příkladu na sekvenční logiku, který si trochu detailněji rozebereme.

Bude se jednat o modul, kterým bude přijímač dat z jednoduché sériové sběrnice. Nebudeme řešit žádné synchronizační mechanismy a podobné skoro samozřejmé vlastnosti takové sběrnice. Na této sériové sběrnici bude zkrátka jednosměrný proud (stream) bitových dat, které náš modul rozdělí na 8bitové úseky a bude je propagovat na 8bitovou výstupní sběrnici. Na jednu periodu hodinového signálu od této propagace taky potvrdí platnost dat.

Vstupy jsou následující:

- `clk` - hodinový signál, sériová sběrnice je též časována tímto signálem
- `rst` - signál pro vyvolání synchronního resetu
- `ser_in` - hodnota aktuálního bitu přenášeného po sériové sběrnici

Výstupy jsou:

- `byte_rdy` - signál pro potvrzení platnosti dat na výstupní sběrnici
- `byte` - aktuální přečtený byte ze sériové sběrnice

Podoba rozhraní takového modulu je uvedena na další straně.

```
entity ser_rec is
  port(
    clk: in std_logic;
    rst: in std_logic;
    --
    ser_in: in std_logic;

    byte_rdy: out std_logic;
    byte: out std_logic_vector(7 downto 0)
  );
end ser_rec;
```

Architektura pak bude obsahovat pouze jediný proces, který je popsán následovně:

```
main: process(clk)
  -- posuvny registr pro prijem bitu
  variable shifter: std_logic_vector(7 downto 0);
  -- pocet prijatych bitu
  variable counter: integer range 0 to 8;
begin
  if (rising_edge(clk)) then

    -- vychozi hodnota byte_rdy pro kazdy pruchod procesu
    byte_rdy <= '0';

    if (rst = '1') then

      counter := 0;
      -- nulovani vsech bitu pomoci zkraceneho zapisu namisto "00000000"
      byte <= (others => '0');

    else

      -- prijeti dalsiho bitu
      counter := counter + 1;
      shifter := ser_in & shifter(7 downto 1);

      -- vsech 8 bitu je pripraveno
      if (counter = 8) then
        counter := 0;
        -- zapis do vystupu byte z nej udela vystupni registr
        byte <= shifter;
        -- vystup byte je nyni platny (posledni platne prirazeni)
        byte_rdy <= '1';
      end if;

    end if;
  end if;
end process main;
```

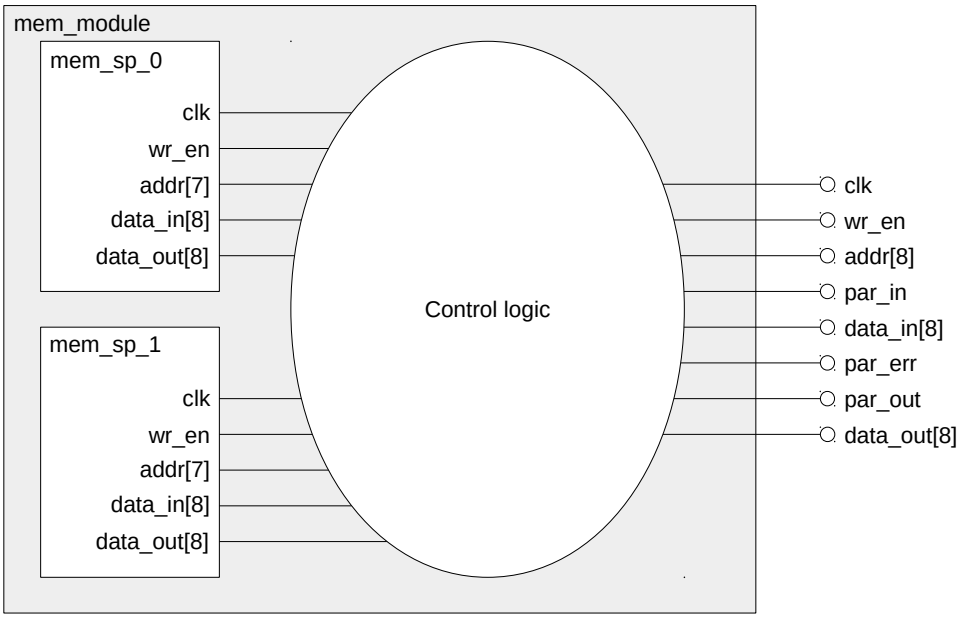
Klíčovou roli v našem procesu hraje posuvný registr `shifter`. Přijímáme jednotlivé bity ze sériové sběrnice tím, že posuneme celý registr doprava (ztrácí se LSB) a bity pak ukládáme vždy na místo MSB tohoto registru.

Za povšimnutí taky stojí fakt, že samotný registr `shifter` není synchronním resetem nijak ovlivněn, stačí pouze resetovat `counter`, který ovlivňuje jeho šíření dále (do `byte`).

Kompletní popis tohoto modulu je uveden v souboru **ser_rec.vhd**, jenž je zveřejněn.

Úloha č. 9 - paměťový modul

Vaší poslední úlohou bude navrhnout modul s pamětí zobrazený níže.



Číslo v hranatých závorkách na obrázku udává počet bitů signálu. Ovál 'Control logic' není samostatná komponenta, reprezentuje vnitřní řídicí logiku, kterou budete implementovat.

Výše uvedený obrázek mnohé popisuje, přesto se však budeme věnovat i detailnímu popisu chování. Tento modul se bude chovat jako single-port paměť se synchronním čtením obohacená o zabezpečení přenosu dat pomocí sudé parity. Paměťový modul bude pracovat s adresováním po osmicích bitů (čili po bytech) a celková kapacita, kterou zapouzdřuje, je 256 B. Jak si můžete povšimnout z obrázku výše, tuto kapacitu budou tvořit komponenty dvou 128B synchronně čtených pamětí (jejichž kód máte v souboru **mem_sp.vhd**).

Všechny použité signály (vstupy, výstupy a příp. vaše deklarované proměnné) budou datového typu standardní logiky.

Vstupy vašeho paměťového modulu budou:

- `clk` - jednotný hodinový signál v celém vašem systému
- `wr_en` - povolení zápisu vstupních dat (čtení probíhá vždy)
- `addr` - adresa pro čtení/zápis dat
- `par_in` - paritní bit vstupních dat
- `data_in` - vstupní data (zapisovaná data)

Výstupy pak budou:

- `par_err` - detekce chyby parity na vstupních datech
- `par_out` - paritní bit výstupních dat
- `data_out` - výstupní data (přečtená data)

Pokud je paritní bit vstupních dat nevalidní při zápisu do paměti (při čtení se kontrola paritního bitu ignoruje), zápis se neprovede a na výstupu `par_err` bude po tuto celou dobu hodnota `'1'`, která signalizuje tuto chybu.

Vysvětlivky

podtržený text - důležitá informace

modrý podtržený text - odkaz do WWW nebo tohoto dokumentu

text kurzívou - doplňující informace

tučný text - záchytný bod v textu

`text` - kód VHDL

`modrý text` - klíčová slova VHDL

`zelený text` - komentáře v kódu VHDL

`<růžový text v závorkách>` - zástupce větší části kódu VHDL

Autor

Dominik Salvet - github.com/dominiksalvet