Chapter 1

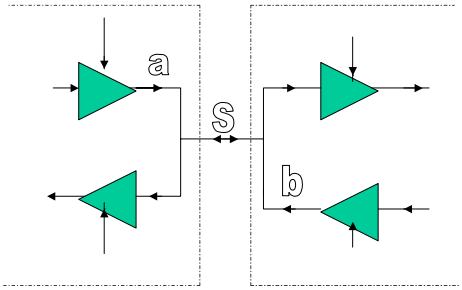
Simulace číslicových obvodů

1.1 Základní principy simulace

V doporučeních firmy Xilinx, kde píší jakým způsobem navrhovat: "pozor, vyhněte se asynchronnímu návrhu, pokud se tomu nevyhnete, zlikviduje to váš projekt, zdraví a možná i život" :).

Obor simulačních hodnot

– dvouhodnotový systém: 0,1
  • nejstarší, nevystihuje všechny okolnosti
– tříhodnotový systém: 0, 1, X
  • X….neznámá hodnota (0 nebo 1)
– čtyřhodnotový systém: 0,1, X, Z
  • Z….vysoká impedance

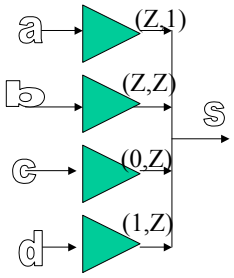


Rozhodovací funkce

• pro dva budiče:

a \ b	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

• pro více budičů



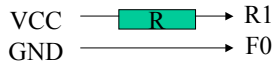

```
begin
S:= Z;
S:= RF(S,a);  (Z,1)
S:= RF(S,b);  (Z,1)
S:= RF(S,c);   (0,1)
S:= RF(S,d);   (X,1)
end
```


Obor simulačních hodnot

– devítistavový systém: Z0, Z1, ZX, R0, R1, RX, F0, F1, FX

hodnoty: 0 síly: Z…..(week)
1 R….resistive
X F ….forcing

př: MOS invertor, otevřený kolektor



– dvanáctistavový systém: + neznámá síla U

př.: Z0 nebo F0 → U0

síly \ hodnoty			
Z	Z0	Z1	ZX
R	R0	R1	RX
F	F0	F1	FX
U	U0	U1	UX

pozn.: exist. i jednostodvacetihodnotový systém

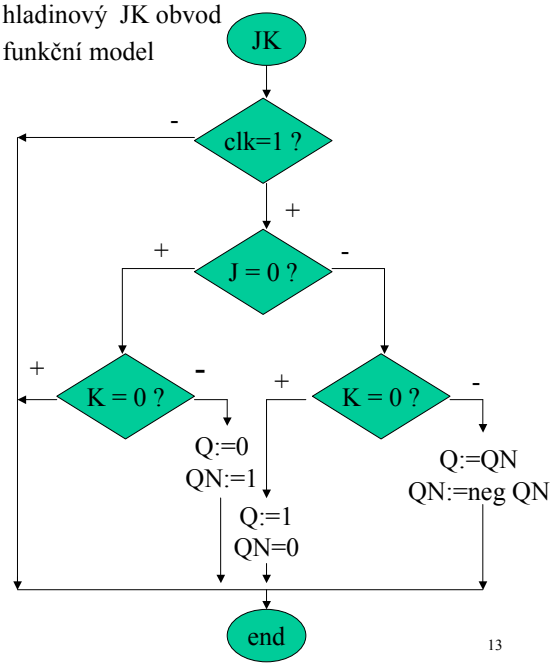
Zpracování zpoždění signálu

• **synchronní simulace**
  – zpoždění se neuvažuje
  – metoda pevného časového kroku

• **asynchronní simulace**
  – zpoždění se uvažuje:
    • jednotkové - stejné pro všechny obvody
    • násobné - násobek jednotkového zpoždění
    • libovolné
  – podrobnější přístup
    umožňuje:
    • zjistit statické hazardy a dynamické hazardy
    • ověřit dodržení předstihů a přesahů
    • ověřit správnost frekvence hodinových pulsů
  – metoda proměnného časového kroku

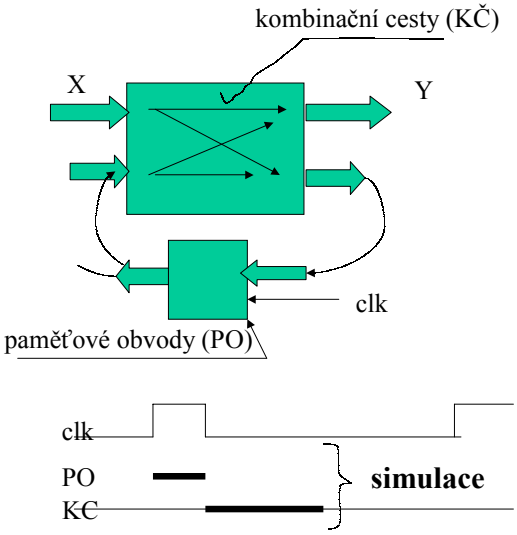
Synchronní model JK klopného obvodu

- hladinový JK obvod
- funkční model



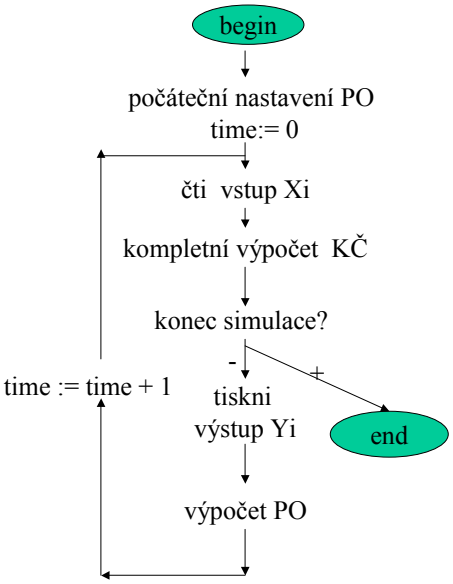
Principy simulace strukturních obvodů

- synchronní simulace



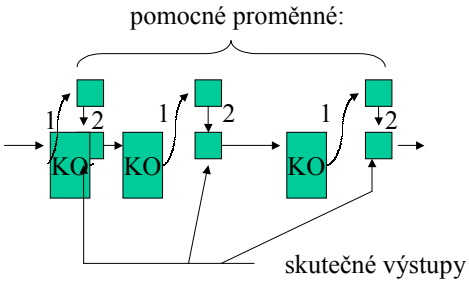
Řídící program synchronní simulace

Princip: lze oddělit výpočet kombinační a paměťové části
Algoritmus výpočtu:



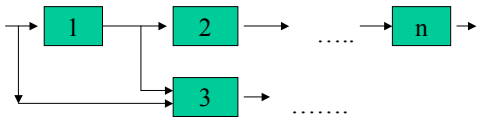
Organizace výpočtu při simulaci struktur

- paměťové obvody PO
  - často se PO vzájemně neovlivňují (jsou odděleny kombinační částí) => libovolná posloupnost výpočtu
  - obecný algoritmus: - dvoufázový průchod
    1) výpočet a zapamatování nových hodnot
    2) přenos nových hodnot do skutečných výstupů



Organizace výpočtu při simulaci struktur

- kombinační cesty KČ

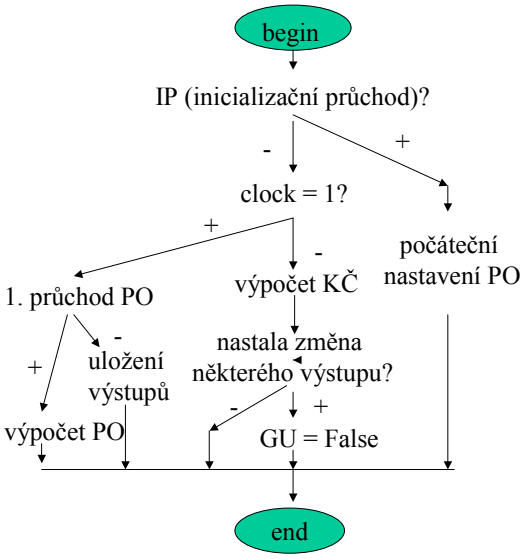


možné strategie výpočtu struktur:

- repetiční vyhodnocení neuspořádaných prvků:
  - například: n,….,3,2,1,
  - správná odezva: maximálně n průchodů (acyklický graf),
  - po n + 1 průchodech - ustálení všech výstupů ,
  - zpomalení výpočtu,
- vyhodnocení uspořádaných prvků ve směru toku signálu:
  - například: 1, 2, 3,……..…., n
  - správná odezva: 1 průchod,
  - problém uspořádání (obecně nelze jednorázově při překladu: viz obousměrná spojení),
  - nutno provádět dynamicky a zjednodušeně v průběhu simulace => problém jednoznačného chování,
  - obecný algoritmus: oddělení výpočtu odezvy a uložení hodnot => zpomalení

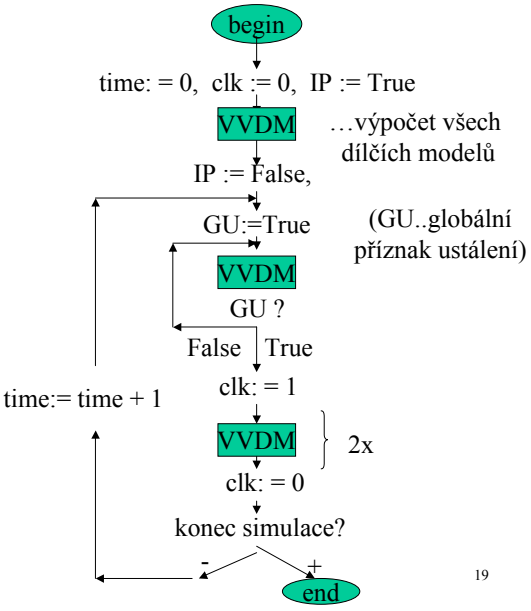
Model obecného prvku při použití repetičního výpočtu struktur

poznámka: GU…příznak globálního ustálení



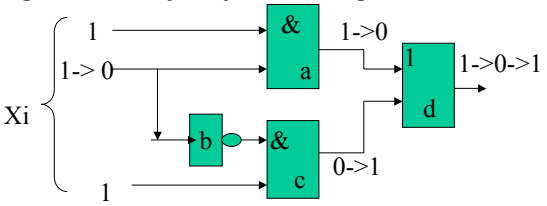
Řídící algoritmus repetičního výpočtu pro synchronní simulaci struktur

poznámka: synchronní simulace, pevný časový krok

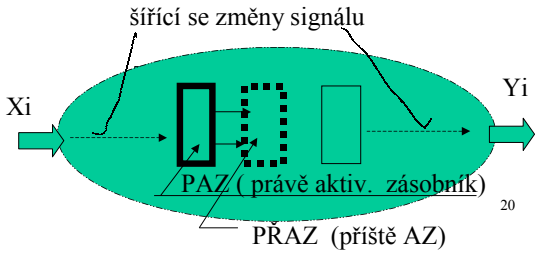


Asynchronní simulace

předpoklad: uvažujeme jednotkové zpoždění del



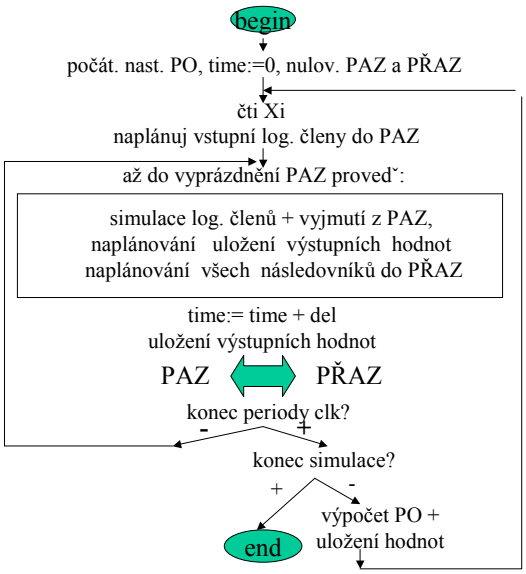
- synchronní simulace: vstup Xi, vyhodnocení a, b, c, d
  - nevystihuje paralelní šíření signálu
- asynchronní simulace: vstup Xi,
  - vyhodnocení a,b, time:= time+ del, uložení a,b,
  - vyhodnocení c,d, time:=time+ del, uložení c,d ,
  - vyhodnocení d , time:=time+ del, uložení d



Asynchronní simulace

Strategie výpočtu pro jednotkové zpoždění:

PAZ…právě aktivní zásobník, PŘAK..příště AZ



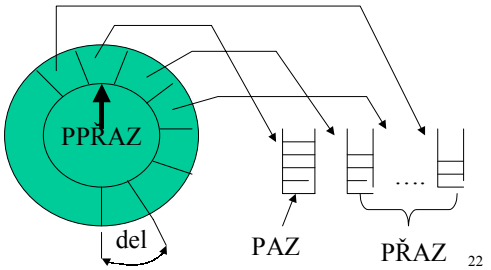
Asynchronní simulace

charakteristika předešlého přístupu:

- simulace standard. struktury (možno modifikovat pro obecný případ)
- dynamické uspořádání
- simulace všech logických členů
- metoda pevného časového kroku
- jednotkové (stejné) zpoždění

modifikace předešlého přístupu:

- připustíme násobné zpoždění: T= k * del



Asynchronní simulace

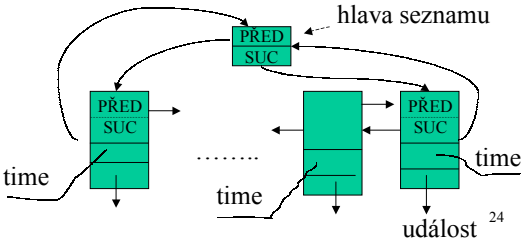
zobecnění předešlého přístupu:

- nerozlišujeme paměťové a kombinační členy
- připustíme libovolné zpoždění
- důsledně oddělíme výpočty dílčích členů a uložení vypočtených hodnot do výstupních signálů
- použijeme metodu proměnného časového kroku,
- provedení veškerých změn v modelu podmíníme existencí příslušného záznamu v seznamu událostí SU
  - PAZ + všechny PŘAZ nahradíme seznamem událostí SU,
  - každou událost doplníme časem výskytu
- použijeme metodu aktivní cesty
  - neplánujeme soustavně všechny logické členy do SU, ale pouze „kandidáty na změnu“ (t. j. členy u nichž došlo ke změně hodnoty některého ze vstupních signálů)

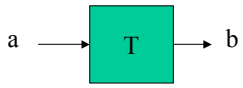
Asynchronní simulace

seznam událostí (event list):

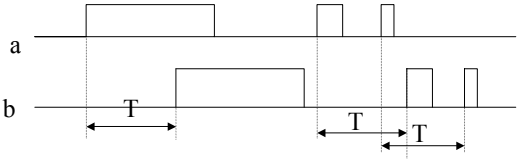
- **položky: záznamy událostí (event notices)**
  - čas události (KDY?)
  - specifikace události - odkaz na objekt (CO?)
- **struktura**
  - sekvenční, nesetříděný - nesnadné vkládání i výběr
  - sekvenční, setříděný (dle hodnot model. času) - výběr od začátku
  - jednocestně zřětězený, setříděný
  - dvoucestně zřetězený, setříděný
  - hierarchický apod.



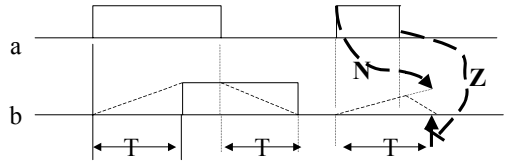
Typy zpoždění



- dopravní zpoždění: pro vodiče



- setrvačné zpoždění: pro logické členy



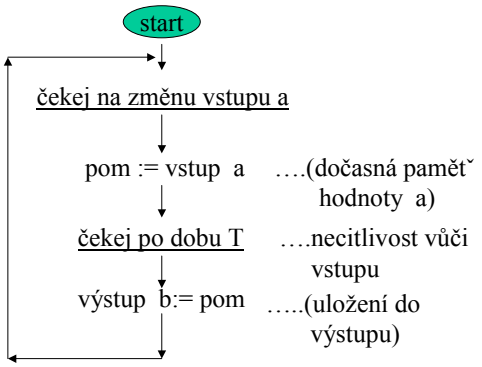
25

Model dopravního zpoždění

předpoklad: existuje jádro pro časovou synchronizaci dílčích elementů simulované struktury

nesprávný model dopravního zpoždění:

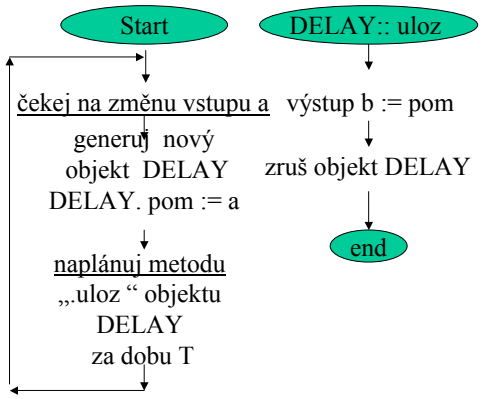
- použití procesu (koprogramu)
- omezení: modeluje správně pouze pulsy jejichž šířka i vzdálenost jsou větší než T



26

Model dopravního zpoždění

- chování objektu ZPOŽDĚNÍ: použití pomocných dynamických objektů DELAY a podprogramů



poznámky:

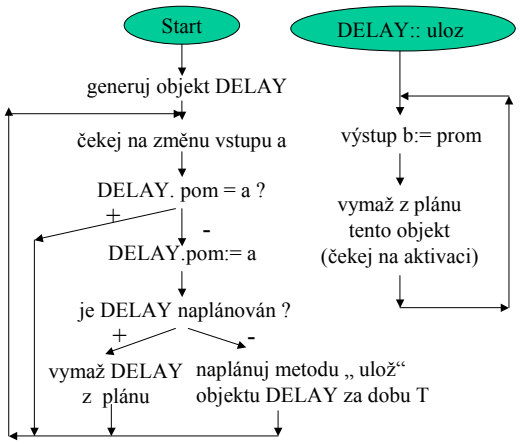
- přesun libovolného počtu pulsů
- pom… proměnná objektu DELAY sloužící jako dočasná paměťˇvýstupní hodnoty

27

Model setrvačného zpoždění

chování objektu zpoždění:

- použití pomocného objektu DELAY a podprogramů
- uvažujeme pouze dvouhodnotovou logiku

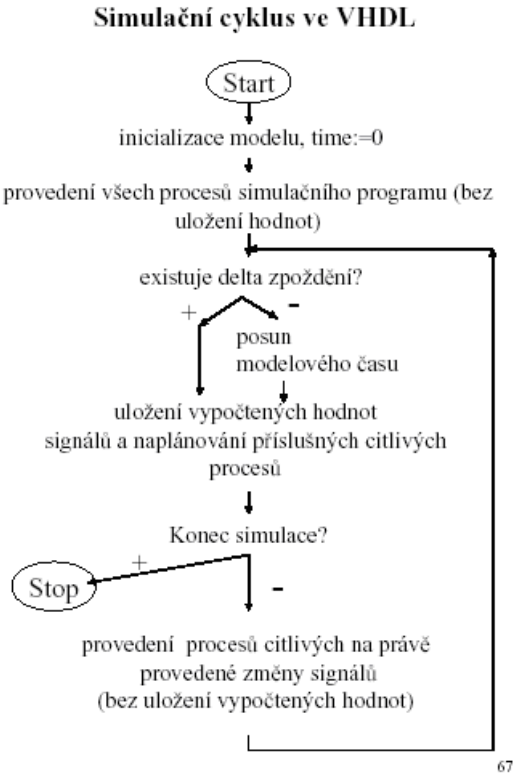


pom….proměnná objektu DELAY, která obsahuje poslední hodnotu výstupu nebo hodnotu ke které výstup právě klopí

28

1.2 Charakteristika VHDL

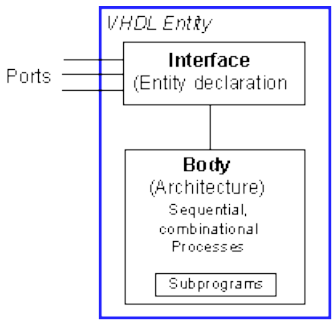
VHDL = VHSIC (*(Very High Speed Integrated Circuits) Hardware Description Language*). VHDL se vyvíjí již od 80. let minulého století. Nejznámějším konkurenčním projektem je Verilog (Cadence).



Jazyk VHDL je *case-INSENSITIVE*.

1.2.1 Základní struktura VHDL

Systém modelovaný VHDL jazykem je tvořen top-level entitou, která může obsahovat další podentity (komponenty). Každá entita je modelovaná pomocí *entity declaration* a *architecture body*.



Entity declaration část obsahuje jméno entity a seznam vstupních a výstupních portů. Nepovinná část generic obsahuje zpravidla lokální konstanty ve tvaru: "`generic ( name: type [:=value] ; name: type [:=value] );`". Následuje vzor a jednoduchý příklad:

```
-- entity NAME_OF_ENTITY is [ generic generic_declarations);]
--    port (signal_names: mode type;
--          signal_names: mode type;
--            :
--          signal_names: mode type);
-- end [NAME_OF_ENTITY] ;

-- toto je jednoduchý příklad. Poznamenejme, že je lepší použít knihovní typ
-- std_logic než uvedený 'BIT', je však nutno importovat příslušnou knihovnu
entity OBVOD is
    port (A,B : in BIT;
```

```
          Y   : out BIT);
end OBVOD;
```

Zde je obdobný příklad entity pro popis MUXu 1 ze 4 využívající vektorový knihovní typ `std_logic_vector`:

```
entity mux4_to_1 is
    port (I0,I1,I2,I3: in std_logic_vector(7 downto 0);
       OUT1: out std_logic_vector(7 downto 0));
end mux4_to_1;
```

Architecture body část popisuje, jak obvod pracuje a jak je implementován. Opět následuje šablona včetně jednoduché ukázky:

```
-- architecture architecture_name of NAME_OF_ENTITY is
--  -- Declarations
--        -- components declarations
--        -- signal declarations
--        -- constant declarations
--        -- function declarations
--        -- procedure declarations
--        -- type declarations
--
--  begin
--        -- Statements
--
--  end architecture_name;

architecture UVNITR of OBVOD is
   const DELAY : TIME = 10ns;
   signal X : BIT;
   -- signal A : BIT; nesmý být deklarován, pokud je 'A' již v entitě
begin
   P1: process begin
      X <= A nor B after DELAY;
      wait on A,B;
   end process;
   P2: process begin
      Y <= X nand B after DELAY;
      wait on X,B;
   end process
end UVNITR;
```

1.2.2 Specifikace komponent

Stejným způsobem jako OBVOD můžeme vytvořit entitu NAND2, která má 2 vstupní porty A a B a jeden výstupní Y. Do naší architektury poté vložíme tuto entitu jako vlastní komponentu:

```
architecture UVNITR of OBVOD is
   component
       NAND2 port (in1, in2: in BIT; out1: out BIT);
   end component;
   signal X : BIT;
begin
   U1: NAND2 port map (A, B, Z);
     -- nebo port map (in1 => A, in2 => B, out1 => Y );
end UVNITR;
```

Externí knihovna se importuje příkazy `library` a `use`. První jmenovaný definuje o jakou knihovnu se jedná, druhý pak specifikuje balíček:

```
library ieee;
use ieee.std_logic_1164.all; -- knihovna obsahující mnohé datové typy
```

1.2.3 Proměnné a signály

Objekty ve VHDL mohou být: signály, proměnné, konstanty nebo soubory.
Konstanta může ale nemusí mít nastavenu implicitní hodnotu již v deklarační části. Pokud je konstanta deklarována uvnitř procesu, může být použita pouze v tomtéž procesu. Pokud je deklarována v deklarační části architektury může být použita kdekoliv v rámci architektury.
Proměnné mohou měnit hodnotu přiřazovacím příkazem (:=). Proměnné jsou na aktualizovány bez jakékoliv prodlevy v okamžiku, kdy je prováděn příslušný příkaz. Proměnné MUSÍ být deklarovány uvnitř procesu. Proměnné jsou pouze v sekvenčním prostředí (výjimka: shared variables v VHDL 93). Deklarace proměnné vypadá takto:

```
-- variable list_of_variable_names: type [ := initial value] ;

variable CNTR_BIT: bit :=0;                -- 0 nebo 1
variable VAR1: boolean :=FALSE;            -- mohou být logického typu
variable SUM: integer range 0 to 256 :=16; -- mohou mít specifikovaný rozsah
variable STS_BIT: bit_vector (7 downto 0); -- 8-mi bitová hodnota, např: 10001000
```

Signály se definují obdobně jako proměnné s tím, že použité klíčové slovo je `signal` místo `variable`. Hodnota se přiřazuje po provedení všech fází naplánovaných procesů (po uplynutí delta zpoždění), důvodem je zabránění nedeterministického chování v případě kvaziparalelního provádění více procesů (nesmí záležet na pořadí jejich provádění). I když nemusí být prodleva explicitně specifikována, přiřazení hodnoty není provedeno ihned:

```
architecture SIGN of EXAMPLE is
    signal TRIGGER, RESULT: integer := 0;
    signal signal1: integer :=40;
    signal signal2: integer :=50;
begin
    process begin
       wait on TRIGGER;
       signal2 <= signal2 + A;
       RESULT  <= signal2;                 -- RESULT se bude rovnat 50
    end process;
end SIGN;
```

Po skončení procesu bude v signálu `RESULT` hodnota 50 příkazy se provedou paralelně, jako u skutečného obvodu. Se signálem `TRIGGER` se provedou oba přiřazovací příkazy a nečeká se, až se změní `signal2` (vstupní signál pro signál RESULT). Signály mohou být také zaregistrovány přímo entitě jako vstupní (IN), výstupní (OUT), vstupně výstupní (INOUT) nebo jako registrovaný výstupní signál (BUFFER). Dělení signálů:

1. obyčejný
   - každý objekt třídy signal, který je nosičem datového typu nedisponujícího žádnou resoluční funkcí
   - může být buzen pouze jedním budičem
2. rozhodovaný (resolved)
   - objekt třídy signal vystupující jako nosič datového typu disponujícího resoluční funkcí
   - připouští neomezený počet budičů, které však nelze od daného signálu odpojovat
3. strážený
   - rozhodovaný signál, který je dále specifikován klíčovým slovem register nebo bus
   - připouští více budičů, které je však možné v průběhu simulace odpojovat (odpojení v sériovém prostředí přiřazením null)

- při odpojení všech budičů:
  - register si pamatuje svou poslední hodnotu (v takovém případě není ani volána příslušná resoluční funkce)
  - bus svou hodnotu po odpojení všech budičů ztrácí; resoluční funkce je proto volána vždy a musí být navržena tak, aby tuto hodnotu poskytla i v tomto případě

1.2.4 Výstup programu

Příklad znamená také v tomto případě více než tisíce slov:

```
use std.textio.all;
architecture behavior of check is
begin
   process (x)                              -- totéž jako 'wait on x;'
      variable s : line;
      variable cnt : integer:=0;
   begin
      if (x='1' and x'last_value='0') then
        cnt:=cnt+1;
        if (cnt>MAX_COUNT) then
          write(s,"Pocet preteceni - ");
          write(s,cnt);
          writeline(output,s);              -- Tisk textu a proměnné CNT
        end if;
      end if;
   end process;
end behavior;
```

1.2.5 Některé další věci o VHDL z testu SIM

Tato kapitola obsahuje několik věcí ohledně VHDL, které nejsou přímo obsaženy v okruzích ke státnicím, nicméně Douša je má hodně rád. A pokud by byl v komisi, mohl by se ptát.

1.2.6 Vyjmenujte všechny typy paralelních příkazů VHDL

paralelní prostředí ve VHDL - 2 typy příkazů

1. funkční popisy typu data-flow (funkční popis)
   - vyjadřují jak funkční závislost výstupů na příslušných vstupech, tak i zároveň strukturu, do které nejsou vnořeny žádné další entity
   - vzájemné interakce: pouze pomocí signálů
   - pořadí vyhodnocení signálových příkazů: není dáno pořadím, ale nastane jako důsledek změny některého ze signálů pravé strany výrazu
   - vyhodnocení výrazů a uložení hodnot: není-li ve výrazu explicitně uvedeno nenulové zpoždění, pak skutečné přiřazení hodnoty danému signálu nastane v simulačním cyklu, který následuje za cyklem, v němž byl proveden přiřazovací příkaz
2. hierarchické strukturní popisy a jejich modifikace
   - pouze pro vyjádření vazeb mezi navzájem vnořenými entitami

`block`
`process`
`generate`
příkaz vytvoření instance komponenty:

```
HA1: HALF_ADDER generic map (DELAY => 10 NS) port map (X, Y, SUM);
```

podmíněné přiřazení do signálu

```
SUM <= A + B after 10 NS;
OUTPUT <=
      '1' after 10 NS when PRESET = '0' and CLEAR = '1' else
      '0' after 10 NS when PRESET = '1' and CLEAR = '0' else
      INPUT after 10 NS;
```

výběrové přiřazení do signálu (selected signal assignment)

```
with SELECT_LINES select
    OUTPUTS <= "0001" when "00",
               "0010" when "01",
               "0100" when "10",
               "1000" when "11";
```

paralelní příkaz `assert` paralelní volání procedury

1.2.7 Signály VHDL:

- důvod existence?
  reprezentují vodiče modelovaného obvodu a jsou jediným prostředkem pro komunikaci v paralelním prostředí
- možný výskyt jejich deklarací?
  pouze paralelní prostředí entity, architektury nebo knihovna
- prostředí, kde mohou být modifikovány jejich hodnoty?
  hodnoty lze měnit v paralelním i sériovém prostředí (dvojznak pro přiřazení "<=")

1.2.8 VHDL bloky: typy+rozdíly

Mechanismus pro vnitřní členění paralelního prostředí v architektuře (příkaz bloku = paralelní příkaz).

- bloky lze vzájemně vnořovat
- v bloku lze deklarovat vše co v architektuře
- deklarované objekty jsou v daném bloku lokální
- bloky mohou obsahovat porty, které umožňují mapovat signály z nadřazeného bloku do vnitřních signálů bloku

strážené bloky umožňují specifikovat podmínku pro synchronizaci paralelních příkazů strážený blok (guarded block)

- blok doplněný o tzv. strážený výraz (guarded expression)
- ve stráženém bloku je implicitně deklarován signál "guard", jehož hodnota automaticky sleduje v průběhu simulace hodnotu stráženého výrazu
- signál "guard" nelze explicitně budit žádným budičem uvnitř stráženého bloku a ani jej nelze připojovat k portům módu in, inout, buffer
- ve stráženém bloku lze použít signál guard k podmíněnému provedení tzv. stážených signálových příkazů (označených symbolem guarded); toto provedení nastane:

– při změně hodnoty stráženého výrazu z hodnoty false na hodnotu true
– v případě, že strážený výraz má hodnotu true a nastala událost na některém signálu vyskytujícím se na pravé straně stráženého příkazu

příklad řízení signálových paralelních (a vzájemně asynchronních) příkazů signálem guard:

```
B1: block ( control =´1´)
begin                                   --následuje operační část stráženého bloku
    X <= guarded A and B after 5 ns;    -- strážený příkaz
    Y <= A and B after 5 ns;  -- tento příkaz není strážený, proto ignoruje řízení
end block B1;
```

1.2.9 Rozdíly VHDL sekvenčních a paralelních příkazů

V sekvenčním prostředí jsou příkazy vykonávány stejným způsobem jako v jiných procedurálních jazycích, tj. sekvenčně. Sekvenční prostředí existuje pouze v operační části procesu, procedury nebo funkce. Většina nepodmíněných příkazů má obdobnou formu pro paralelní prostředí (kromě příkazu wait, cyklů apod.); syntax těchto forem se v případě podmíněných příkazů odlišuje.
Typy paralelních příkazů:

- nepodmíněné sign. přiřazovací příkazy (<= after), generate, podmíněné signálové přiřazovací příkazy (when, with), assert (paralelní), odložený příkaz assert, paralelní příkaz procedury, příkaz bloku (block), příkaz stráženého bloku (guarded)

Typy sekvenčních příkazů ve VHDL:

- kterýkoliv příkaz v operační části procesu, procedury či fce, jeho provedení podléhá pravidlům procedurálních programovacích jazyků.
- příkaz assert (sekvenční), přiřazovací příkaz (sekvenční) pro signály, přiřazovací příkaz pro proměnné, volání podprogramů (sekvenční), sekvenční řídící příkazy (wait, if-then-end if, case, loop, exit, next, return, null)

1.2.10 Procesy - definice, komunikace a synchronizace

proces – posloupnost událostí v čase

popis procesu – zobrazení F z časové množiny do množiny atributů

- spojitý proces – F formou diferenciálních rovnic
- diskrétní proces – F formou posloupnosti událostí

vzájemná komunikace procesů

- sdílení dat (data sharing): př.: Simula 67, VHDL
- předávání zpráv (message passing): SDL - asynchronní komunikace pomocí signálů

časová synchronizace procesů

- jde o synchronizaci vzhledem k modelovému času: hodnoty reálného času mapujeme na hodnoty z množiny čísel (real nebo integer) a okamžitou hodnotu modelového času uchováváme v proměnné Time
- So: originál: atributy: a,b,..., reálný čas: t
- Sm: model: proměnné: A, B, ..., modelový čas: Time

- proměnná Time nabývá postupně neklesajících hodnot Tj,
- proměnné A, B, ... modelu Sm reprezentují hodnoty atributů a, b, ... originálu So v čase, tj. pokud platí Time = Tj

1.2.11 Výchozí předpoklady pro budování paralelního prostředí

- paralelní výpočet řízený více seznamy událostí
- jednotlivé části (procesy) navzájem komunikují pomocí zpráv
- jednotlivé procesy jsou vzájemně synchronizovány tak, aby výsledný efekt paralelně prováděného programu byl stejný jako efekt jeho provedení v kvaziparalelním prostředí; toho lze dosáhnout pokud výsledný efekt každého procesu respektuje příčinné závislosti a všechny události, které jsou současné z hlediska modelového času, jsou provedeny při paralelním i sériovém výpočtu ve stejném pořadí dvě základní skupiny algoritmů synchronizačních strategií paral. simulace:
- konservativní synchronizační algoritmy (metoda nulových zpráv (Chandy - Misra - Bryant), metoda uváznutí a zotavení procesů, synchronní metoda)
- přísně respektují příčinné závislosti (local causality constraint)
- jejich algoritmy jsou založeny na rozlišení tzv. bezpečných událostí (neexistuje možnost pozdějšího výskytu dalších událostí s menší hodnotou časové známky)
- optimistické synchronizační algoritmy (metody Time Warp)
- nerespektují pravidla příčinných závislostí, ale detekují jejich porušení
- pomocí zpětných běhů (tj. návratů k menším hodnotám modelového času) anulují efekty všech předčasně provedených událostí, pak pokračují novým dopředným během

1.2.12 Příkaz `generate` - proč se používá a kde

Paralelní příkaz, který umožňuje opakované provádění paralelních příkazů typu data-flow. Jeho použití je užitečné při popisech "pravidelně propojených elementů" se stejnou funkční závislostí. Tím míníme množinu paralelních příkazů, jejichž výrazy se liší pouze indexy výstupních či vstupních signálů.

"makro" pro generování "podobných" paralelních příkazů
dvě formy:

- iterační typ příkazu : `<for schema>`
  - analogie sekvenčního příkazu cyklu
  - pro generování pravidelných struktur
- podmíněný typ příkazu: `<if schema>`
  - analogie sekvenčního příkazu `if`
  - nelze použít alternativní větve (analogie s `elsif` nebo `else`)
  - použití: pro podmíněné generování odlišností

syntax:

- `<label> for <ident> in <rozsah> generate <paralelní příkaz> end generate;`

- `<label> if <podmínka> generate <paralelní příkaz> end generate;`

  řídicí proměnná cyklu je implicitně deklarována, nemůže být explicitně modifikována, je nedostupná vně příkazu generate

1.2.13 Význam následujících pojmů ve VHDL: `entity`, `architecture`, `package`, `package body` a `configuration`

entity – samostatně překládaná programová jednotka, která definuje rozhraní navrhované části. Pro specifikaci chování či struktury je nutná přidružená architektura. Specifikuje především vnější vstupní a výstupní porty (případně i parametry) modelu; tyto porty jsou jediným prostředkem pro komunikaci modelu s jeho okolím.

architecture
- Samostatně překládaná programová jednotka, která definuje interní strukturu či chování přidružené entity. Definuje hodnoty výstupů jako reakce na vstupy.
- Deklarace architektury A pro entitu E: architecture A of E is begin ... end A;
- `entity` + `architecture` = `model komponenty`

package
- exportní část uživatelské knihovny
- obsahuje veřejné deklarace, které lze zpřístupnit v jiných entitách, architekturách nebo knihovnách
- možné deklarace: datové typy a podtypy, signály, konstanty, hlavičky funkcí a procedur, aliases, komponenty, soubory, sdílené proměnné (pouze VHDL-93)
- nelze deklarovat: proměnné, entity, architektury
- `package <jméno knihovny> is ... end;`

package body
- privátní a implementační část uživatelské knihovny
- implementace a neveřejné deklarace
- možné deklarace: hlavičky privátních funkcí a procedur, privátní datové typy a podtypy, privátní konstanty
- implementace: operační části privátních funkcí a procedur, operační části exportovaných funkcí a procedur
- `package body <jméno knihovny> is ... end;`

configuration – přiřazení entity a architektury dílčí komponentě: lze v architektuře vynechat v případě, že: komponenta a entita mají stejná rozhraní (tj. jména & porty) – default configuration; osazení bude provedeno později (configuration)

konfigurace vně architektury (configuration declaration) – samostatná část zdrojového programu. Umožňuje odložit osazení soklů a toto soustředit do jednoho místa ⇒ možnost rychlé změny integrovaných obvodů bez nutnosti nového překladu osazované strukturní architektury

1.2.14 Nástroje VHDL pro strukturní simulaci

- deklarace komponent

```
component <jméno typu komponenty>
    generic...;     -- formální  parametry
    port (...);     -- formální  porty
end component;
```

- specifikace komponent:

```
-- for <identifikátor komponenty>:<jméno typu komponenty>
-- use entity<jméno knihovny>.<jméno entity> [(<jméno architektury>)]
-- [generic map (<přiřazení parametrů>)] [port map (<přiřazení portů>)];
```

- instalace a zapojení komponent:

```
-- <jméno komponenty> : <jméno typu komponenty>
-- [generic map (<seznam parametrů>)]  --aktuální parametry
-- port map (<mapování portů>) ;
```

- bloky – automatické generování struktur: generate; konfigurace strukturních schémat: configuration

1.2.15 Možnosti osazování a instalace komponent ve VHDL

osazování komponent:

- jde vždy o mapování portů na porty stejného módu (in, out, ...)
- toto mapování je možno vynechat, pokud se shodují jména parametrů a portů u entity i komponenty
-
```
for <identifikátor komponenty>:<jméno typu komponenty>
use entity<jméno knihovny>.<jméno entity> [(<jméno architektury>)]
[generic map (<přiřazení parametrů>)] [port map (<přiřazení portů>)];
```
- př.: osazení komponenty K1 typu S16 v architektuře A
- $\Rightarrow$ předpoklad: v knihovně work existuje entita EIO, která je spolu s architekturou AIO určena pro osazení komponenty S16 v architektuře A

```
entity E is ... end E;
architecture A of E is
    signal ...;
    component S16 port(...); end component;
    for K1:S16 use entity work.EIO(AIO) --osazení K1
        port map(...);
        begin ... end AIO;
```

instalace komponent:

- při instalaci komponent v operační části architektury jde o mapování formálních portů instalované komponenty na lokální signály architektury nebo na porty k ní přidružené entity
- při pozičním mapování lze vynechat formální porty (tj. porty komponenty)
- nezapojený výstup se mapuje na hodnotu open
-
```
<jméno komponenty> : <jméno typu komponenty>
    [generic map (<seznam parametrů>)]  --aktuální parametry
    port map (<mapování portů>) ;
```
- př.: instalace komponenty K1 typu S16 v architektuře A

```
entity E is port (...) end E;
architecture A of E is
    signal ...
    begin    K1:S16 port map (...); --příkaz komponenty K1 typu S16
end A;
```

1.2.16 Atributy signálu ve VHDL

dvě skupiny atributů:

funkce : poskytují informace o transakcích nebo událostech na zvoleném signálu

signály : druhotně odvozené od událostí či transakcí zvolených signálů; jsou použitelné opět jako signály

- funkce
  - s'active – true pokud je v daném simulačním cyklu signál s aktivní (nastala transakce)
  - s'last_active – vrací časový interval, který uplynul od poslední transakce na signálu s
  - s'event – true pokud nastala v daném simulačním cyklu změna hodnoty signálu s
  - s'last_event – vrací časový interval, který uplynul od poslední události na signálu s
  - s'last_value – vrací hodnotu signálu s před poslední událostí
- signály
  - s'delayed – vytvoří nový signál stejného typu, ale zpožděný o delta zpoždění
  - s'delayed(T) – vytvoří nový signál stejného typu, ale zpožděný o hodnotu T
  - s'stable – nový signál typu boolean, který má hodnotu true, pokud v daném simulačním cyklu nenastala na signálu s žádná událost, jinak false
  - s'stable(T) – nový signál typu boolean, který má hodnotu true, pokud na signálu s nenastala během intervalu T žádná událost, jinak false
  - s'quiet – nový signál typu boolean, který má hodnotu true, pokud na signálu s v daném simulačním cyklu nenastala žádná transakce
  - s'quiet(T) – nový signál typu boolean, který má hodnotu true, pokud na signálu s nenastala během intervalu T žádná transakce
  - s'transaction – nový signál typu bit, který mění svou hodnotu při každém výskytu transakce na signálu s; nová hodnota je negací hodnoty předchozí,

1.2.17 Příkaz wait ve VHDL. Kde ho lze použít?

- vytvoří reaktivační bod, přeruší proces a odevzdá řízení jádru
- lze použít pouze v sekvenčním prostředí (v procesu)

`wait on <seznam signálů>` proces je aktivován po změně některého ze signálů

`wait until <podmínka>` proces je aktivován po splnění podmínky

`wait for <časový údaj>` proces je aktivován za časový interval

příklady forem příkazu watt

```
wait for 10 ns; -- čeká po dobu 10 ns
wait on a; -- čeká na změnu signálu a
wait until a = '1'; -- čeká na náběžnou hranu signálu a
wait until a = '0'; -- čeká na náběžnou hranu signálu a
wait on x,y until u='1' or w='1' for 10 ns;
    -- aktivace pouze změnou na x nebo y, pokud splněna podmínka
    -- until; po 10 ns aktivován v každém případě
wait; -- čeká navždy
```