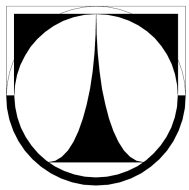
TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií



DIPLOMOVÁ PRÁCE

Liberec 2006 Michal Calta

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: N 2612 – Elektrotechnika a informatika

Studijní obor: 1802T007 – Informační technologie

Tvorba „stimuli“ souborů pomocí grafického rozhraní

The generation of testbench files by graphical user interface

Diplomová práce

Autor: **Michal Calta**

Vedoucí DP: Doc. Ing. Zdeněk Plíva, PhD.

Konzultant:

V Liberci 19.10. 2005

Originální zadání práce

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé DP a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Diplomovou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum 19.5.2006

Podpis

Poděkování

Na tomto místě bych chtěl poděkovat panu Doc. Ing. Zdeňku Plívovi, PhD., vedoucímu diplomové práce, za poskytnuté informace a rady při vypracovávání diplomové práce.

Abstrakt

Diplomová práce je věnována vytváření „stimuli“ souborů. „Stimuli“ soubory obsahují informace, které jsou potřeba při testování správné funkčnosti číslicových obvodů navržených za pomoci *HDL* (*hardware description language* – jazyk pro popis číslicových obvodů). Pozornost je věnována jazyku VHDL, pro jehož „stimuli“ soubory se vžilo anglické označení *testbench files.* Cílem diplomové práce je návrh a realizace programu, který uživateli umožní tvorbu „stimuli“ souborů za pomoci grafického rozhraní.

První kapitola je věnována obecně jazykům pro popis číslicových elektronických obvodů. Podrobněji je popsán jazyk VHDL. Následuje kapitola ověřování funkčnosti navržených modelů, speciálně jsou probírány *testbench* soubory jazyka VHDL. Aby navrhovaný program dodržel zaběhnuté standardy v ovládání a zobrazování, jsou ve třetí kapitole recenzovány programy, které se již v této oblasti používají. Další části jsou věnovány navrženému programu StimuliEdit. Zabývají se reprezentací, možnostmi grafického nastavování a editování číslicových signálů. Jsou zde uvedeny nejvhodnější grafické nástroje, které navržený program používá. Další kapitoly jsou věnovány problematikám načítání popisu signálů ze zdrojových souborů VHDL a ukládání nastavených průběhů do *testbench* souborů. Závěrečná kapitola je věnována ovládání a nastavení programu.

Klíčová slova:

HDL, VHDL, stimuli soubor, grafické editování signálů.

Abstract

This work deals with testbench files. These files contain informations, which are useful for testing right functionality of digital circuits designed by hardware description language. The attention is paid to language VHDL (Very high speed integrated circuit Hardware Description Language). The main point of this work is to design and construct the program, which will we able to create testbench files by graphical user interface.

First section deals with common hardware description languages. Second section is about checking right functionality of created models. Next chapter reviews the programs used for creating testbench files now. Final chapters describe construction of program StimuliEdit. Here are mentioned used principles, data structures and the control.

Key words:

HDL, VHDL, testbench files, editing signal by graphical user interface

Obsah

Úvod

Jazyk pro popis číslicových obvodů označovaný jako VHDL (z anglického *Very high speed integrated circuit Hardware Description Language*) je dnes velice používaným nástrojem. Máme-li vytvořený číslicový obvod pomocí VHDL, je třeba otestovat jeho správnou funkčnost. Pro testování funkčnosti se používají simulátory, které potřebují informace o průběhu vstupních signálů. Tyto informace nesou tzv. „stimuli“ soubory. Pro „stimuli“ soubory jazyka VHDL se vžilo anglické označení *testbench files.*

Diplomová práce se věnuje návrhu grafického rozhraní, které umožňuje jednoduché vytváření „stimuli“ souborů. V první kapitole lze nalézt obecný popis jazyků pro návrh číslicových obvodů, druhá kapitola je věnována ověřování funkčnosti navržených modelů. Třetí kapitola je věnována analýze programů, které se dnes používají pro práci se „stimuli“. Další kapitoly jsou věnovány navrženému programu StimuliEdit. Kapitoly popisují, jakým způsobem je možné reprezentovat číslicové signály za pomoci datových struktur, grafické nástroje používané při editování průběhů číslicových signálů. Závěrečné kapitoly jsou věnovány ovládání a nastavení programu StimuliEdit.

1. Jazyky pro popis číslicových elektronických obvodů

Slovo hardware se v dnešní době používá v kontextu se širokou škálou termínů. Na jedné straně se jedná o kompletní systémy jako jsou osobní počítače PC (*Personal Computer*), na straně druhé jde o jednotlivé součástky, jako například integrované číslicové obvody. Tyto obvody se popisují jazyky pro popis číslicových elektronických obvodů, které se označují jako HDL (*Hardware Description Language*). Využití HDL jazyků neustále roste s využitím programovatelných obvodů (*programmable designing circuit*) jako jsou PLD (*Programmable Logic Devices*), CPLD (*Complex PLD*), FPGA (*Field Programmable Gate Array*) aj.

Popis číslicových obvodů pomocí jazyků HDL lze zpravidla provádět na různých úrovních abstrakce. Pro komplexní systémy je většinou využíván behaviorální popis. Jedná se o popis obvodu „zvenku“ při vysoké úrovni abstrakce. Při této technice popisu jsou skryty veškeré

implementační detaily (neuvažují se šířky sběrnic, hodnoty hodinových signálů atd.). Na druhé straně je popis na úrovni základních logických členů. Pro rozsáhlejší návrhy je tato metoda nepoužitelná pro svojí rozsáhlost a složitost [26].

Takto by se daly jednoduše představit jazyky HDL. Následující kapitoly jsou věnovány převážně jazyku VHDL, ostatní jazyky jsou zde uvedeny pouze pro ilustraci.

1.1 Jazyk VHDL

VHDL (*Very High Speed Integrated Circuits Hardware Description Language*) je jazyk pro popis digitálních elektronických obvodů. Vzešel z amerického vládního programu VHSIC (*Very High Speed Integrated Circuits*), který byl zahájen v roce 1980. Po svém dokončení byl přijat americkým institutem IEEE (*Institute of Electrical and Electronic Engineers*) jako standard IEEE Std. 1076-1987 [1]. Následně byl jazyk VHDL revidován normou IEEE Std. 1076-1993 [2]. Tímto je jazyk VHDL definován v BNF (*Backus-Naur Form*) s podrobným anglickým popisem všech konstrukcí a signálů. V roce 1999 vyšla revize VHDL-AMS 1076.1 (*Analogue & Mixed Signals*). (Informace o standardech jazyka VHDL viz [13].)

Základní vlastnosti jazyka VHDL:

- Je to **otevřený standard** *(open standard)*. K jeho použití pro sestavení návrhových systémů není třeba licence jeho vlastníka, jako je tomu u jiných jazyků HDL (například u jazyka ABEL-HDL). To je jeden z důvodů, proč je tento jazyk v návrhových systémech často používán.

- Umožňuje pracovat na návrhu, aniž je předtím zvolen cílový obvod. Ten může být zvolen až v okamžiku, kdy jsou známy definitivní požadavky na prostředí, v němž má navrhovaný systém pracovat, a je možno cílový obvod měnit podle potřeby při zachování textu popisujícího systém *(Device-independent design)*.

- Je možné provést simulaci navrženého obvodu na základě téhož zdrojového textu, který pak bude použit pro syntézu a implementaci v cílovém obvodu. Zdrojový text je možné zpracovávat v různých simulátorech a v syntetizérech různých výrobců. Odsimulovaný text může být použit v dalších projektech s různými cílovými obvody, což je podporováno

hierarchickou strukturou jazyka. Této vlastnosti jazyka se říká přenositelnost kódu *(code portability)*.

- V případě úspěšného zavedení výrobku na trh lze popis modelu systému v jazyku VHDL použít jako podklad pro jeho implementaci do obvodů ASIC vhodných pro velké série.

Jazyk VHDL je navržen tak, aby pokryl široké spektrum požadavků při návrhu digitálních elektronických obvodů. Byl původně určen především pro simulaci velkých číslicových systémů. Jeho použití pro syntézu číslicových systémů se začalo rozpracovávat později a využívá se při něm jen část jeho prostředků. V počátečních fázích tohoto směru využití jazyka VHDL byly vytvářeny jednoduché syntetizéry se značně omezeným rozsahem prostředků jazyka. Postupně, jak byly syntetizéry vylepšovány, se tento rozsah rozšiřoval. Proto je možné očekávat, že základní konstrukty jazyka budou různými syntetizéry zpracovány stejně, ale neobvyklé způsoby popisu a některé vyšší jazykové struktury mohou dávat různé výsledky při zpracování různými syntetizéry. Je také nutné počítat s tím, že různé syntetizéry podporují odlišný rozsah prostředků jazyka. Postupem doby, jak se budou syntetizéry vylepšovat, se patrně bude rozšiřovat oblast jazykových konstruktů využitelných v syntéze, které budou dávat výsledky syntézy skutečně nezávislé na použitých návrhových systémech.

Způsoby sestavení modelu:

- Postup **shora dolů** *(top-down)*: definuje se funkce navrhovaného systému jako celku, pak se v něm vyčlení bloky, jejich funkce se specifikuje spolu s vzájemnou návazností jednotlivých bloků, a tyto bloky pak mohou detailně zpracovávat různí konstruktéři.

- Postup **zdola nahoru** *(bottom-up)*: nejprve se vytvoří dílčí bloky modelu a ty se pak skládají do větších celků.

- Model **plochého typu** *(flat)* neobsahuje členění odpovídající výše uvedeným typům. Představuje systém jako jeden monolitický blok.

První dva způsoby sestavení konstrukce se označují jako hierarchické. Jazyk VHDL hierarchičnost konstrukcí podporuje. Rozumně zvolená hierarchie usnadňuje orientaci v popisu modelu navrhovaného systému [8].

1.2 Struktura souboru VHDL

Pro názvy souborů VHDL se používá přípona `*.vhd`. Soubor má pevně definovanou strukturu. V této kapitole budou popsány pouze ty části, které nesou informace potřebné pro vytváření „stimuli" souborů, které budou probírány v následujících kapitolách. Detailnější popis jazyka není náplní této práce, existuje spousta publikací, ze kterých lze jazyk snadno nastudovat. Proto pouze odkazuji na [5], [6], [7], [26].

Pro ilustraci následuje krátký příklad kódu VHDL. Popisuje jednoduchou osmibitovou sčítačku.

```
LIBRARY ieee ;                --knihovna obsahuje definice
USE ieee.std_logic_1164.all ; --STD_LOGIC a STD_LOGIC_VECTOR aj.

entity SCITAC is        --hlavička, obsahuje popis vstupů a výstupů obvodu
  generic(Delay: time:= 10 ns);     --generická složka zpoždění
  port (a, b: in std_logic_vector(7 downto 0);
        cin: in std_logic;
        cout: out std_logic;
        sum: out std_logic_vector (7 downto 0));
end scitac;

architecture behav of scitac is  --architektura popisující chování obvodu
begin
 process (cin, a, b)
 variable temp : std_logic_vector (8 downto 0);
 begin
  temp := ('0' & a) + ('0' & b) + ("00000000"& cin);
  cout <= temp(8) after Delay;
  sum <= temp (7 downto 0) after Delay;
 end process;
end behav;
```

Příklad 1.2.1 Popis osmibitové sčítačky v kódu VHDL

1.2.1 Deklarace entity

Číslicové systémy se většinou navrhují jako kolekce modulů s určitou hierarchií. Každý modul má definovanou množinu portů, které přestavují vstupně-výstupní rozhraní mezi modulem a okolním světem. V jazyku VHDL přestavuje *entita* modul, který může být použit buď jako komponenta sytému, nebo může představovat výsledný systém [5].

Následuje syntaxe deklarování *entity* (`entity_declaration`) zapsaná v BNF (Backus-Naur Form) (tučně jsou tištěna klíčová slova, nepovinné části jsou uzavřeny v hranatých závorkách [ ]).

```
entity_declaration ::=
      entity identifier is
            entity_header
            entity_declarative_part
      [ begin
            entity_statement_part ]
      end [ entity_simple_name ] ;
entity_declarative_part ::= { entity_declarative_item }
entity_header ::=
      [ formal_generic_clause ]
      [ formal_port_clause ]
entity_header ::=
      [ formal_generic_clause ]
      [ formal_port_clause ]
generic_clause ::= generic ( generic_list ) ;
generic_list ::= generic_interface_list
generic_interface_list ::=
[ constant ] identifier_list : [ in ] subtype_indication
                                    [ := static_expression ]
port_clause ::= port ( port_list ) ;
port_list ::= port_interface_list
interface_signal_declaration ::=
      [ signal ] identifier_list : [ mode ] subtype_indication [ bus ]
            [ := static_expression ]
```

Definice 1.2.1 Syntaxe deklarace entity

Identifikátor (`identifier`) se musí řídit následujícími pravidly:

- nerozlišují se velká a malá písmena
- musí začínat písmenem
- může obsahovat písmena, číslice a podtržítka
- jméno nesmí obsahovat mezeru
- nelze použít dvě podtržítka za sebou
- podtržítko nesmí být posledním znakem
- nesmí být totožný s klíčovými slovy
- musí být unikátní

Deklarativní část entity (`entity_declarative_part`) může být použita k deklarování datových objektů, které jsou implementovány v *entitě*. Zde je uvedena pouze pro úplnost, obvykle se totiž deklarace datových objektů provádí uvnitř architektury příslušející k entitě, viz kapitola 1.2.2.

Hlavička entity (`entity_header`) je nejdůležitější část deklarace entity. Hlavička může obsahovat specifikaci generických konstant (*`formal`*`_generic_clause`).

Seznam generických konstant (*`generic`*`_interface_list`) obsahuje položky, které jsou třídy konstanta (*class constants* viz [5] kapitola 2.2.8) a vyhovují uvedenému popisu. Hodnota každé položky ze seznamu identifikátorů (`identifier_list`) je konstanta, která je poskytnuta vždy, když je entita použita jako komponenta v jiném návrhu. Konstantě může být přiřazena hodnota (*`static`*`_expression`). Identifikátory splňují podmínky stejné jako identifikátor entity, jednotlivé položky jsou odděleny čárkou.

Typy generických konstant (`subtype_indication`) jsou podrobně vysvětleny v [5] kapitola 2.2.7. Kromě seznamu generických konstant obsahuje hlavička entity také seznam portů (*`formal`*`_port_clause`).

Seznam portů (*`port`*`_interface_list`) má podobnou strukturu jako seznam generických konstant, ale každá jeho položka je třídy signál. Klíčové slovo **`signal`** bývá většinou vypouštěno, zato klíčové slovo **`bus`** by mělo být použito tehdy, jestliže se předpokládá, že port bude připojen přes sběrnici k více než jednomu dalšímu portu (kapitola 6.1 a 6.2 v [5]). Porty mohou být v jednom z následujících módů:

- **`IN`** – data lze z portu pouze číst
- **`OUT`** – data vycházejí z portu (výstupní signál nemůže být použit jako vstup uvnitř entity)
- **`BUFFER`** – výstup se zpětnou vazbou, kterou je možné číst uvnitř entity
- **`INOUT`** – obousměrný tok
- **`LINKAGE`** – neznámý směr datového toku – obousměrný (návaznost na jiné než VHDL modely, např. Verilog)

Definice podtypu (`subtype_indication`) zde nebude probírána, pouze pro ilustraci následují příklady nejpoužívanějších podtypů (přesnou definici je možné nalézt v [5] kapitola 6.1):

- `bit` – nabývá hodnot '0' a '1'
- `bit_vector (7 downto 0)` – představuje osmibitový vektor
- atd.
- podtypy definované v knihovně `ieee`:
  - `std_ulogic` – položka, která může nabývat devíti různých stavů (U, X, 0, 1, Z, W, L, H, -) (viz [6] kapitola 2.16)
  - atd.

1.2.2 Deklarace architektury

Jakmile je entita se všemi jejími náležitostmi deklarována, následuje deklarace architektury. Architektura popisuje chování systému. Ke každé entitě přísluší alespoň jedna architektura. To znamená, že k jedné entitě může příslušet i více architektur, kde každá architektura může poskytovat různý pohled na entitu. Například jedna architektura může popisovat entitu behaviorálně (viz [5] kapitola 2 a 4), zatímco další může popisovat entitu jako hierarchicky komponovanou kolekci komponent. Syntaxe zápisu deklarace architektury v BNF je zachycena v definici 1.2 [5], [6].

```
architecture_body ::=
      architecture identifier of entity_name is
            architecture_declarative_part
      begin
            architecture_statement_part
      end [ architecture_simple_name ] ;
architecture_declarative_part ::= { block_declarative_item }
block_declarative_item ::=
      subprogram_declaration
      | subprogram_body
      | type_declaration
      | subtype_declaration
      | constant_declaration
      | signal_declaration
      | alias_declaration
      | component_declaration
      | configuration_specification
      | use_clause
architecture_statement_part ::= { concurrent_statement }
concurrent_statement ::=
      block_statement
      | component_instantiation_statement
```

Definice 1.2.2 Syntaxe deklarace architektury

Pro identifikátor architektury platí stejná pravidla jako pro identifikátor entity (viz předchozí kapitola 1.2.1). Následuje jméno entity (`entity_name`), ke které přísluší architektura. Deklarativní část architektury (`architecture_declarative_part`) definuje položky, které budou použity při konstrukci popisu architektury.

Pro tvorbu „stimuli“ souborů (viz další kapitoly) nese důležité informace položka deklarace signálu (`signal_declaration`), ta je námětem kapitoly 1.2.3, deklarace komponent (`component_declaration`) je probírána v kapitole 1.2.5, ostatní položky zde nebudou diskutovány (lze nalézt v [5] kapitola 3).

Příkazová část architektury (`architecture_statement_part`) obsahuje vlastní popis architektury. Blok příkazů (`block_statement`) je probírán v kapitole 1.2.4.

1.2.3 Deklarace signálů

Signály se používají pro připojování submodulů do návrhu.

```
signal_declaration ::=
      signal identifier_list : subtype_indication [ signal_kind ]
                                                    [ := expression ] ;
signal_kind ::= register | bus
```

Definice 1.2.3 Syntaxe deklarace signálu

Popis specifikace druhu signálu (`signal_kind`) je možné nalézt v [5] kapitola 6.2. Při vynechání této položky je zaveden signál, který je specifikován podtypem (`subtype_indication`) (viz kapitola 1.2.1). Výraz (`expression`) se používá k nastavení signálu na inicializační hodnotu. Přesnou syntaxi zápisu výrazu lze nalézt v [5] kapitola 2.3. Položka je nepovinná, při vynechání je signál inicializován standardní přednastavenou hodnotou.

1.2.4 Bloky

Jednotlivé submoduly v architektuře mohou být popsány jako bloky. Blok je jednotka, která má vlastní rozhraní připojené pomocí signálů k ostatním blokům nebo portům. Blok má v BNF následující syntaxi:

```
block_statement ::=
      block_label :
            block [ ( guard_expression ) ]
                  block_header
                  block_declarative_part
            begin
                  block_statement_part
            end block [ block_label ] ;
block_header ::=
      [ generic_clause
      [ generic_map_aspect ; ] ]
      [ port_clause
      [ port_map_aspect ; ] ]
generic_map_aspect ::= generic map ( generic_association_list )
port_map_aspect ::= port map ( port_association_list )
block_declarative_part ::= { block_declarative_item }
block_statement_part ::= { concurrent_statement }
```

Definice 1.2.4 Syntaxe deklarace bloku

Položka `guard_expression` zde nebude probírána, zpravidla se položka vypouští, lze ji nalézt v [2] vysvětlivky B.112. Hlavička bloku (`block_header`), stejně jako hlavička entity (viz kapitola 1.2.1), definuje vstupně-výstupní rozhraní.

Specifikace generických konstant (`generic_clause`) má stejnou syntaxi jako v hlavičce entity (viz kapitola 1.2.1). Tyto konstanty mají omezenou platnost pouze na aktuální blok. Seznam generických konstant (`generic_association_list`) obsahuje jména generických konstant, jejichž hodnoty budou postupně asociovány s generickými konstantami přiloženého bloku nebo architektury (viz deklarace komponenty kapitola 1.2.5). Asociativní seznam portů (`port_association_list`) specifikuje, který signál nebo port přiloženého bloku nebo architektury má být připojen ke kterému portu aktuálního bloku.

Další částí bloku je deklarativní část bloku (`block_declarative_part`), ta má stejný význam jako `entity_declarative_part` (viz kapitola 1.2.1).

Následuje příkazová část bloku. Ta také může obsahovat deklaraci bloku, takže návrh může být komponován jako hierarchie bloků s behaviorálním popisem struktury na nejnižším stupni

hierarchie [5]. V tomto dokumentu bude dále příkazová část probírána v kapitole 2.2, ale pouze v souvislosti se zápisem průběhu signálu, vše ostatní je popsáno např. v [6].

1.2.5 Deklarace komponenty

Jako tělo architektury může být také použita již vytvořená architektura, kterou je možné získat z uložené knihovny. V tomto případě musí být v architektuře deklarována komponenta (podrobnější informace o vytváření knihoven můžeme najít v [5] kapitole 3.3). Syntaxe deklarování komponenty zapsaná v BNF:

```
component_declaration ::=
      component identifier
            [ local_generic_clause ]
            [ local_port_clause ]
      end component ;
```

Definice 1.2.5 Syntaxe deklarace komponenty

Pro identifikátor platí stejná pravidla jako pro předchozí identifikátory. `local_generic_clause` a `local_port_clause` jsou lokální paralely ke generickým konstantám a portům, jaké se definují v hlavičce entity viz kapitola 1.2.1.

Pro ilustraci následují dva krátké příklady ukazující deklarování komponent.

```
component nand3
      generic (Tpd : Time := 1 ns);
      port (a, b, c : in logic_level;
            y : out logic_level);
end component;

component read_only_memory
      generic (data_bits, addr_bits : positive);
      port (en : in bit;
            addr : in bit_vector(depth-1 downto 0);
            data : out bit_vector(width-1 downto 0) );
end component;
```

Příklad 1.2.2.1 Deklarace komponent

První deklarace v příkladu 1.2.2.1 je brána se třemi vstupy a jednou generickou konstantou specifikující zpoždění přenosu. Další deklaruje paměť pouze pro čtení s proměnnou šířkou adresové a datové sběrnice. Šířka sběrnic závisí na dříve definovaných generických konstantách.

1.2.6 Konkretizace komponent

Komponenty definované v architektuře je možné konkretizovat za použití následující syntaxe:

```
component_instantiation_statement ::=
      instantiation_label :
            component_name
                  [ generic_map_aspect ]
                  [ port_map_aspect ] ;
```

Definice 1.2.6 Syntaxe konkretizace komponenty

Tento zápis říká, že architektura obsahuje komponentu jména `component_name`, jejíž generické konstanty jsou inicializovány hodnotami z architektury (`generic_map_aspect`) a rozhraní komponenty je připojeno místními signály (`port_map_aspect`). Syntaxe zápisu obou položek bude upřesněna v kapitole 1.2.8. Následuje příklad konkretizace komponent definovaných v předchozí kapitole v příkladu 1.2.2.1.

```
enable_gate: nand3
      port map (a => en1, b => en2, c => int_req, y => interrupt);

parameter_rom: read_only_memory
      generic map (data_bits => 16, addr_bits => 8);
      port map (en => rom_sel, data => param, addr => a(7 downto 0);
```

Příklad 1.2.2.2 Konkretizace komponent

Generická konstanta `Tpd` první instance `nand3` není konkretizována, proto bude inicializována defaultní hodnotou. Druhá instance má specifikované obě generické konstanty, které následně určují šířku datové a adresové sběrnice. Napojení jednotlivých signálů rozhraní je evidentní. V druhém příkladu je vektor `addr` připojen pouze na část rozměrově většího vektoru `a`. Toto ilustruje jakým způsobem připojit signály na širší sběrnici a podobně.

1.2.7 Knihovny a jednotky

Jestliže vytváříte návrh ve VHDL jazyce, zapisujete jej do návrhového souboru (*design file*). Při spuštění kompilátoru je kód nejdříve analyzován, a poté vložen do návrhové knihovny (*design library*). To umožňuje separátně analyzovat a vkládat do knihovny větší množství konstrukcí. Konstrukce se nazývá knihovní jednotka (`library_unit`). Primární knihovní jednotky (`primary_unit`) jsou: deklarace entit, deklarace balíků (*package declarations*) (viz [5] kapitola 2.5.3) a deklarace konfigurací (*configuration declarations*) (viz kapitola 1.2.8). Sekundární knihovní jednotky (`secondary_unit`) jsou těla architektur a balíků. Sekundární knihovní jednotky závisí na specifikaci jejich rozhraní v jejich odpovídající primární jednotce, proto musejí být primární jednotky kompilovány dříve než sekundární. Návrhový soubor se může skládat z několika knihovních jednotek. Strukturu takového souboru je možné popsat následující syntaxí:

```
design_file ::= design_unit { design_unit }
design_unit ::= context_clause library_unit
context_clause ::= { context_item }
context_item ::= library_clause | use_clause
library_clause ::= library logical_name_list ;
use_clause ::= use selected_name { , selected_name } ;
selected_name ::= prefix . suffix
logical_name_list ::= logical_name { , logical_name }
library_unit ::= primary_unit | secondary_unit
primary_unit ::=entity_declaration
                | configuration_declaration
                | package_declaration
secondary_unit ::= architecture_body | package_body
```

Definice 1.2.7 Syntaxe deklarace knihoven a jednotek

Knihovny používají identifikátory, které se nazývají logická jména (`logical_name`). Logická jména musí být přeložena do odpovídajícího tvaru pro hostitelský operační systém. Například návrhové knihovny mohou být implementovány jako databázové soubory. Jména knihoven představují i jména databázových souborů. Jména knihovních jednotek pak mohou mít jako prefix logické jméno knihovny. Například logické jméno `ieee.std_logic_arith` představuje knihovní jednotku *std_logic_arith* (`suffix`), která je umístěna v knihovně *ieee* (`prefix`).

Kontextová klauzule (`context_clause`) předcházející každé knihovní jednotce specifikuje, jaké jiné knihovny a balíky jednotka používá.

Existují dvě speciální knihovny, které jsou implicitně dostupné všem navrženým jednotkám a není třeba je uvádět v knihovní klauzuli (`library_clause`). První se nazývá pracovní (*work*) a odkazuje na knihovnu, do které analyzátor umístil jednotky překládaného návrhu. Druhá speciální knihovna se nazývá standardní (*std*). Ta obsahuje všechny definované typy datových objektů a definovaných funkcí. Všechny položky této knihovny jsou implicitně viditelné, proto přístup do této knihovny není třeba nikde uvádět.

1.2.8 Konfigurace

V předchozích kapitolách 1.2.5 a 1.2.6 je probíráno, jakým způsobem můžeme deklarovat a specifikovat komponentu. Měli bychom se zmínit, že deklarované komponenty mohou být brány jako šablony pro návrh entity. Vazby entity na tuto šablonu je dosaženo za pomoci deklarace konfigurace. Této deklarace se také dá použít ke specifikaci aktuálních generických konstant pro komponenty nebo bloky. Takže deklarace konfigurace hraje rozhodující roli v organizování popisu návrhu v přípravě pro simulaci nebo pro jiné zpracování.

```
configuration_declaration ::=
      configuration identifier of entity_name is
            configuration_declarative_part
            block_configuration
      end [ configuration_simple_name ] ;
configuration_declarative_part ::= { configuration_declarative_item }
configuration_declarative_item ::= use_clause
block_configuration ::=
      for block_specification
            { use_clause }
            { configuration_item }
      end for ;
block_specification ::= architecture_name | block_statement_label
configuration_item ::= block_configuration | component_configuration
component_configuration ::=
      for component_specification
            [ use binding_indication ; ]
            [ block_configuration ]
      end for ;
component_specification ::= instantiation_list : component_name
instantiation_list ::=
      instantiation_label { , instantiation_label )
      | others
      | all
binding_indication ::=
      entity_aspect
            [ generic_map_aspect ]
            [ port_map_aspect ]
entity_aspect ::=
      entity entity_name [ ( architecture_identifier ) ]
      | configuration configuration_name
      | open
generic_map_aspect ::= generic map ( generic_association_list )
port_map_aspect ::= port map ( port_association_list )
```

Definice 1.2.8 Syntaxe deklarace konfigurace

Deklarativní část konfigurace (`configuration_declarative_part`) dovoluje konfiguraci používat jednotky z knihoven a balíků. Nejvzdálenější blok (`block_configuration`) v deklaraci konfigurace definuje konfiguraci pro architekturu jmenované entity (*`entity`*`_name`).

Následující příklad obsahuje obecnou deklaraci entity a architektury fiktivního procesoru.

```
entity processor is
      generic (max_clock_speed : frequency := 30 MHz);
      port ( port list );
end processor;

architecture block_structure of processor is
      declarations
begin
      control_unit : block
            port ( port list );
            port map ( association list );
            declarations for control_unit
      begin
            statements for control_unit
      end block control_unit;

      data_path : block
            port ( port list );
            port map ( association list );
            declarations for data_path
      begin
            statements for data_path
            end block data_path;
      end block_structure;
```

Příklad 1.2.3 Deklarace entity a architektury fiktivního procesoru

Struktura deklarace konfigurace pro tuto architekturu by mohla vypadat následovně:

```
configuration test_config of processor is
      use work.processor_types.all
      for block_structure
            configuration items
      end for;
end test_config;
```

Příklad 1.2.4 Deklarace konfigurace architektury z příkladu 1.2.3

V příkladu 1.2.4 druhý řádek zviditelňuje obsah balíku `processor_types`, který je umístěný v aktuální pracovní knihovně. Blok konfigurace odkazuje na architekturu `block_structure` entity `processor`. Submoduly architektury mohou být konfigurovány uvnitř bloku konfigurace. Tyto submoduly obsahují instance bloků a komponent. Blok je konfigurován vloženým blokem konfigurace. Architektura z příkladu 1.2.3 může být konfigurována i takto:

```
configuration test_config of processor is
      use work.processor_types.all
      for block_structure
            for control_unit
            configuration items
            end for;
            for data_path
                  configuration items
            end for;
      end for;
end test_config;
```

Příklad 1.2.5 Deklarace konfigurace architektury z příkladu 1.2.3

```
data_path : block
      port ( port list );
      port map ( association list );
      component alu
            port (function : in alu_function;
                  op1, op2 : in bit_vector_32;
                  result : out bit_vector_32);
      end component;
      other declarations for data_path
begin
      data_alu : alu
      port map (function => alu_fn, op1 => b1, op2 => b2, result => alu_r);
      other statements for data_path
end block data_path;
```

Příklad 1.2.6 Struktura bloku `data_path`

Tam, kde jsou submoduly instancemi komponent, představuje konfigurace komponent vazbu entity na instanci komponenty. Pro ilustraci předpokládejme, že blok `data_path` obsahuje komponentu `alu`, deklarovanou jak ukazuje příklad 1.2.6. Předpokládejme také, že knihovna *project_cells* obsahuje entitu nazvanou *`alu_cell`*, definovanou v následujícím příkladu. Blok konfigurace `data_path` by mohl být specifikován jako je tomu v příkladu 1.2.8.

```
entity alu_cell is
      generic (width : positive);
      port (function_code : in alu_function;
            operand1, operand2 : in bit_vector(width-1 downto 0);
            result : out bit_vector(width-1 downto 0);
            flags : out alu_flags);
end alu_cell;
```

Příklad 1.2.7 Deklarace entity `alu_cell`

```
for data_path
  for data_alu : alu
      use entity project_cells.alu_cell(behaviour)
      generic map (width => 32)
      port map (function_code => function,
                operand1 => op1,
                operand2 =>op2,
                result => result,
                flags => open);
      end for;
      other configuration items
end for;
```

Příklad 1.2.8 Blok konfigurace používající entit z knihovny

Alternativně, jestliže knihovna také obsahuje konfiguraci nazvanou `alu_struct` pro architekturu entity `alu_cell` , pak by blok konfigurace mohl vypadat následovně:

```
for data_path
      for data_alu : alu
            use configuration project_cells.alu_struct
            generic map (width => 32)
            port map (function_code => function,
                      operand1 => op1,
                      operand2 => op2,
                      result => result,
                      flags => open);
      end for;
      other configuration items
end for;
```

Příklad 1.2.9 Blok konfigurace používající jinou konfiguraci

1.3 Další jazyky HDL

1.3.1 Jazyk Verilog

Verilog je, stejně jako předchozí, jazyk pro popis digitálních elektronických obvodů. Vývoj jazyka odstartovala v osmdesátých letech firma Gateway Design Automation tím, že vyvinula jazyk s logickým simulátorem Verilog-XL. Jazyk byl normalizován v roce 1995 standardem IEEE Std. 1364-1995. Revize jazyka vyšla v roce 2001 jako IEEE Std. 1364-2001. V současné době se očekává standardizace jazyka SystemVerilog, který by měl rozšířit možnosti jazyka.

Verilog byl vyvinut jako jazyk pro simulaci. Použití jazyka pro syntézu obvodů byl až dodatečný nápad. Syntaxi jazyka popisuje dokument [15]. K získání bližších informací o tomto jazyku odkazuji na servery: www.verilog.com, www.verilog.net, www.doulos.com. Servery nabízejí bohaté informace a nástroje pro tento jazyk.

1.3.2 Jazyk ABEL-HDL

ABEL (*Advanced Boolean Equation Language*) HDL je, stejně jako VHDL, jazyk pro popis číslicových elektronických obvodů. Byl vyvinut firmou Data I/O Corporation pro programovatelné logické obvody PLD (*Programmable Logic Devices*). Oproti jazyku VHDL je ABEL jednodušší, není schopen popisovat obvody s takovou komplexností. Výhodám a nevýhodám, které má jazyk

Abel HDL oproti VHDL, je věnován dokument [12].

ABEL může být použit pro popis systému v různých formách, které zahrnují logické formule, pravdivostní tabulky a stavové diagramy podobné výrazům jazyka C. ABEL kompiler umožňuje simulaci a implementaci systémů do PLD obvodů, jako jsou PAL, CPLD, FPGA. Syntaxi a použití jazyka ABEL jsou věnovány například dokumenty [10] a [11].

1.3.3 Jazyk STIL

STIL (*Standard Test Interface Language*) je jazyk, který poskytuje rozhraní mezi testerem a nástrojem pro vytváření testovacích úloh. Je reprezentací informací potřebných k vytvoření číslicové testovací úlohy. Zatímco účel STILu je poskytnout data k testování, celkově je jazyk mnohem flexibilnější než pouhý tester. Kód jazyka je nezávislý na použitém testeru. Jazyk podporuje hierarchické definice průběhu vstupních signálů, proto je vhodný pro popis signálů na sběrnicích mikroprocesorových struktur [16].

Jazyk byl standardizován jako IEEE Std. 1450-1999 [17]. Jeho zařazení mezi jazyky HDL není zcela jednoznačné, protože se používá pro simulaci a testování.

2. Simulace, ověření funkčnosti modelu

Po sestavení modelu navrhovaného systému v jazyku HDL je třeba zjistit, je-li model navržen správně, tj. splňuje-li požadavky zadání jak po stránce funkční, tj. jsou-li na výstupech signály správně odpovídající signálům vstupním, tak i po stránce časové, tj. jsou-li dodrženy požadavky na rychlost reakce (zpoždění), na potřebný kmitočet hodinového signálu a podobně. První z těchto dvou skupin testů představuje simulace funkční *(functional simulation)*, druhou představuje simulace časová *(timing simulation)*.

Funkční simulace může být provedena před syntézou *(pre-synthesis simulation*) nebo po syntéze. O simulátorech pro simulaci před syntézou se mluví jako o simulátorech kódu HDL, simulátory pro simulaci po syntéze se označují jako simulátory kódu JEDEC nebo dalších typů kódů vytvářených při syntéze. Časová simulace se před syntézou neprovádí, protože v této fázi ještě nejsou známy časové parametry částí cílového obvodu, kterými bude signál procházet. Provádět

funkční simulaci před syntézou má smysl tehdy, je-li navrhovaný systém natolik složitý, že syntéza trvá delší dobu, která je v případě chyby odhalené v návrhu při simulaci po syntéze zbytečně vyplýtvána. Funkční simulace provedená před syntézou tedy takové chyby odhalí bez nutnosti čekání na ukončení syntézy. Simulace prováděná po syntéze však může vycházet z mnohem podrobnějších údajů o cílovém obvodu a některé chyby, které simulace provedená před syntézou neodhalí, je v této fázi možné zachytit. Je tedy zpravidla účelné odstranit případné zásadní chyby složitých modelů simulací před syntézou a detailní chování těchto modelů odladit simulací po syntéze. U jednoduchých modelů bývá vhodnější přejít rovnou k simulaci po syntéze.

Simulátory bývají obvykle samostatné programové bloky. Někdy jsou dodávány společně s nástroji pro syntézu, tyto nástroje však zpravidla umožňují připojení různých typů simulátorů. Pro simulaci po syntéze se při syntéze vytvářejí soubory dat popisující model v příslušných formátech, které simulátory ke své činnosti vyžadují. Typ generovaných dat se volí při spuštění syntézy. Syntetizéry často vytvářejí také modely v příslušném kódu charakterizující detailní propojení cílových obvodů PLD a FPGA *(post-synthesis, post-fit, post-layout VHDL model)*, které pak mohou být zpracovány simulátory kódu. Tyto modely mají strukturu odlišnou od modelů, které se požívají pro syntézu. Jsou typicky sestaveny z vložených komponent odpovídajících architektuře cílového obvodu a z popisu jejich propojení, takže bývají pro člověka obtížně srozumitelné, i když jsou vytvořeny v souladu se syntaktickými pravidly např. jazyka VHDL.

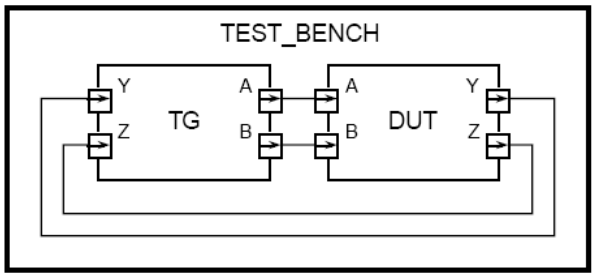
Vedle zdrojového kódu nebo popisu vnitřního propojení cílového obvodu potřebuje simulátor ke své činnosti také data představující zkušební vektory. Ta se mohou vytvářet různými způsoby. Nejjednodušší a nejsnadněji zvládnutelný je způsob vytváření vstupních dat pomocí interaktivního grafického rozhraní, které nabízí různé pomůcky pro vytváření často používaných druhů stimulů, jako jsou konstantní hodnoty vstupních signálů, pulsní průběhy a podobně, a pro jejich následnou editaci. Poněkud obtížnější je vytváření zkušebních vektorů v textové formě, pro kterou se používá označení „stimuli“ soubory nebo *test bench* (zkušební zařízení, zkušební stav, zkušební stolice). Uživatelé jazyku ABEL tento způsob vyjádření vektorů znají, v jazyku VHDL je však syntaxe méně přátelská k uživateli, zejména se zde projevuje rozvleklost jazykových konstruktů.

Textový zápis zkušebních vektorů je však mnohem vhodnější k dokumentaci než jejich grafický záznam a je lépe přenositelný na další typy simulátorů, popřípadě může sloužit k simulaci prováděné na různých úrovních, od simulace zdrojového textu po simulaci po syntéze. Proto bývá

častým doplňkem grafických rozhraní určených pro interaktivní simulaci programový blok vytvářející z dat zadaných v grafické formě textovou formu zkušebních vektorů. Ty pak může simulátor opět načíst a vytvořit odpovídající grafické vyjádření, dostatečně zběhlý konstruktér je však může přímo v textovém tvaru prostudovat a případně podle potřeby editovat. Obvykle však bývá text vytvářený přímo konstruktérem srozumitelnější než „testbench“ vygenerovaný z grafické podoby vektorů [8].

2.1 „Testbench“ jazyka VHDL

Jak již bylo řečeno, „testbench“ slouží k ověření funkčnosti navrženého obvodu. Testbench je na nejvyšší úrovni hierarchie návrhu a jedná se o textový soubor jazyka VHDL.



Obrázek 2.1.1 „Testbench“ obvod

Entita nejvyšší úrovně testovaného návrhu DUT (*design under test*) je použita jako komponenta v testbench obvodu s jinou entitou TG (*test generator*), jehož účelem je nastavovat vstupní signál pro DUT a zároveň vyhodnocovat jeho výstup. Hodnoty signálu mohou být sledovány za pomoci simulačního monitoru nebo přímo test generátorem [5].

Existuje mnoho způsobů, jakými se dají psát testbenche. Nyní budou uvedeny ty nejpoužívanější.

- Pouze stimuly (*Stimulus only*) – obsahuje pouze vstupní vektory a DUT; neprovádí žádnou kontrolu výsledků.

- Úplný testbench (*Full testbench*) – obsahuje vstupní vektory, známé správné výstupní vektory a realizuje porovnání výsledků.

- Charakteristika simulátoru (*Simulator specific*) – testbench je napsán ve formátu specifikujícím simulátor.

- Rychlý testbench (*Fast testbench*) – napsaný k dosažení největší rychlosti simulace.

- Hybridní testbench (*Hybrid testbench*) – kombinuje techniky z více než jednoho stylu.

Příklady ke všem výše uvedeným způsobům jsou řešeny v [7]. Zde bude pro ilustraci uveden jednoduchý příklad testbenche. Jedná se o zkušební úlohu pro obvod číslicového multiplexeru se čtyřmi vstupy, který je jazykem VHDL popsán následovně:

```
LIBRARY ieee ;                --knihovna obsahuje definice
USE ieee.std_logic_1164.all ; --STD_LOGIC a STD_LOGIC_VECTOR aj.

ENTITY multiplexer4 IS        --hlavička
      PORT( A, B, C, D: IN STD_LOGIC;          --vstupy multiplexeru
                  SEL: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
                                    -- 2bit selektor
                    Q: OUT STD_LOGIC);         -- vystup multiplexeru
END multiplexer4;

ARCHITECTURE main OF multiplexer4 IS
BEGIN                         --tělo popisující obvod
      WITH SEL SELECT
            Q <=  A WHEN "00",  --do Q přenes A je-li SEL="00"
                  B WHEN "01",  --do Q přenes B je-li SEL="01"
                  C WHEN "10",  --do Q přenes C je-li SEL="10"
                  D WHEN others;--do Q přenes D je-li všechno ostatní

END main;
```

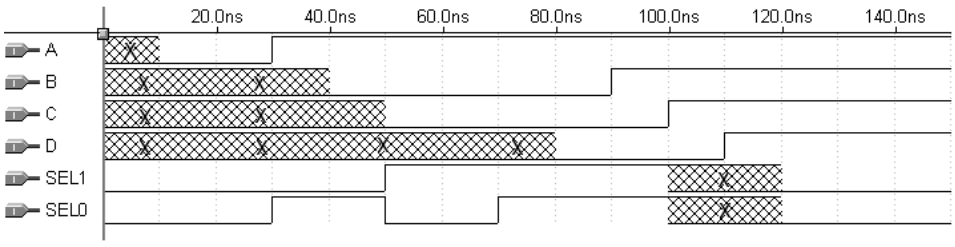
Příklad 2.1.1 Číslicový multiplexer se čtyřmi vstupy

Klíčová slova jazyku VHDL jsou v textu vytištěna silně, komentáře kurzívou. Příkazy, které jsou použity uvnitř této architektury nejsou z hlediska „stimuli“ souborů důležité, proto v tomto dokumentu nebudou probírány (lze nastudovat viz [26] atd.) Testbench výše uvedeného multiplexeru, který obsahuje pouze stimuly vstupních signálů, by mohl vypadat například takto:

```
LIBRARY ieee ;                  --knihovna obsahuje definice
USE ieee.std_logic_1164.all ; --STD_LOGIC a STD_LOGIC_VECTOR aj.

ENTITY testbench IS END ;      --entita je prázdná

ARCHITECTURE tg OF testbench IS
COMPONENT multiplexer4  --požití dříve definované entity
      PORT( A, B, C, D  : IN STD_LOGIC;
                          SEL: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
                          Q  : OUT STD_LOGIC);
END COMPONENT ;

SIGNAL sel : STD_LOGIC_VECTOR (1 DOWNTO 0) ;
SIGNAL a, b, c, d, q : STD_LOGIC ;

BEGIN
sel <="00", "01" AFTER 30 ns, "10" AFTER 20 ns,
      "11" AFTER 20 ns, "XX" AFTER 30 ns, "00" AFTER 20 ns ;
                  --průběh signálu selektoru
a <= 'X', '0' AFTER 10 ns, '1' AFTER 20 ns ;
b <= 'X', '0' AFTER 40 ns, '1' AFTER 50 ns ;
c <= 'X', '0' AFTER 50 ns, '1' AFTER 50 ns ;
d <= 'X', '0' AFTER 80 ns, '1' AFTER 110 ns ;
                  --průběhy na jednotlivých vstupech
M : multiplexer4 PORT MAP ( a, b, c, d, sel, q ) ;
                  --asociace signálů se vstupy a výstupy
                        --entity multiplexer4
END tg ;
```

Příklad 2.1.2 „Testbench“ multiplexeru z příkladu 2.1.1

Pro pochopení uvedených průběhů je ještě třeba vysvětlit, jakým způsobem VHDL umožňuje nastavovat průběh signálu. Toto je námětem kapitoly 2.1.1. V příkladu 2.1.2 entita *multiplexer4* představuje testovaný návrh (blok DUT), entita *testbench* vytváří vstupní signály obvodu (představuje blok TG). Průběh vytvářených signálů je zachycen na obrázku 2.1.2.



Obrázek 2.1.2 Průběhy vstupních signálů multilexeru4

2.1.1 Průběh signálu

Jazyk VHDL umožňuje nastavovat průběh číslicového signálu (*signal_assignment_statement*) za použití syntaxe v definici 2.1. Předpokládá se alespoň jedna změna hodnoty signálu.

```
signal_assignment_statement ::= target <= [ transport ] waveform ;
target ::= name | aggregate
waveform ::= waveform_element { , waveform_element }
waveform_element ::=
      value_expression [ after time_expression ]
      | null [ after time_expression ]
```

Definice 2.1 Syntaxe nastavení průběhu signálu

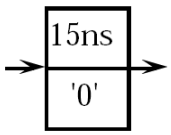
Cíl (`target`) musí reprezentovat signál nebo vektor signálu (viz [5] kapitola 2.4.1). Jestliže je časový výraz (*time*`_expression`) vynechán, pak je zavedena hodnota 0 fs.

Každý signál má k sobě přiřazen plánovaný průběh (*projected output waveform*), což je posloupnost časových údajů s hodnotami signálu. Každý nalezený příkaz nastavení hodnoty je přidán do této posloupnosti. Příkaz v příkladu 2.1.3 způsobí 15 ns po svém provedení změnu hodnoty signálu `s` na `'0'`.

```
s <= '0' after 15 ns;
```

Příklad 2.1.3 Nastavení plánovaného průběhu

Plánovaný průběh z příkladu 2.1.3 můžeme graficky znázornit například takto:

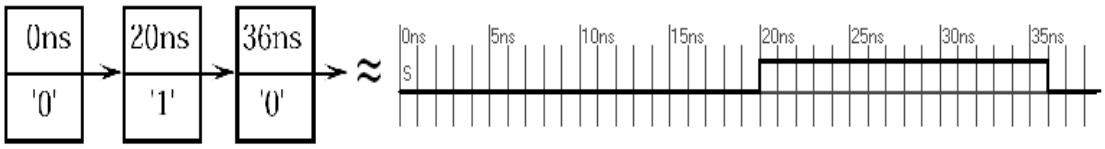


Obrázek 2.1.3 Grafická reprezentace plánovaného průběhu z příkladu 2.1.3

Další příklad představuje inicializaci signálu `s` hodnotou `'0'`a od času 20 ns do 36 ns má signál hodnotu `'1'`.

```
s <= '0', '1' after 20 ns, '0' after 36 ns;
```

Příklad 2.1.4 Nastavení plánovaného průběhu



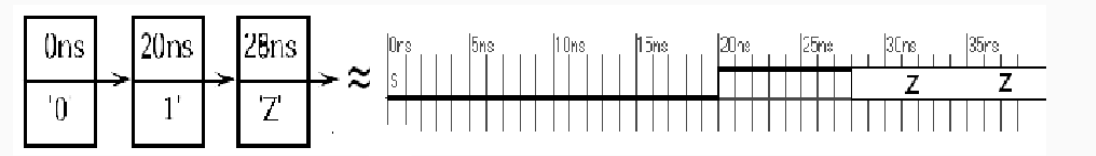
Obrázek 2.1.4 Grafická reprezentace plánovaného průběhu z příkladu 2.1.4

Při zadávání průběhů s více změnami musí být položky seřazeny vzestupně podle času. Jestliže simulátor při překladu narazí na příkaz nastavení hodnoty signálu, pro který již má nastavený průběh, některé staré položky průběhu signálu mohou být smazány. O tom, jestli bude položka smazána, rozhoduje klíčové slovo **`transport`** u nové položky. Jestliže se vyskytuje u nové položky, jako tomu je v příkladu 2.1.5, pak tento příkaz nastavuje transportní zpoždění (*transport delay*). V tom případě budou všechny položky plánovaného výstupního průběhu následující po specifikovaném čase nahrazeny jedinou nastavovanou hodnotou.

```
s <= transport 'Z' after 10 ns;
```

Příklad 2.1.5 Nastavení plánovaného průběhu

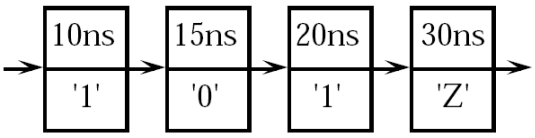
Předpokládejme, že plánovaný průběh má podobu zachycenou na obrázku 2.1.4. Jestliže bude simulátor provádět v čase 18 ns příkaz v příkladu 2.1.5, pak výsledný plánovaný průběh bude vypadat následovně:



Obrázek 2.1.5 Grafická reprezentace plánovaného průběhu z příkladu 2.1.5

Druhým případem zpoždění je inertní zpoždění (*inertial delay*). Toto zpoždění se používá při modelování systémů, které nereagují na kratší změny signálu než je jejich výstupní zpoždění. Inertní zpoždění se specifikuje vypuštěním klíčového slova **`transport`**. Jestliže je přidána položka

inertního zpoždění do již existujícího plánovaného výstupního průběhu, nejdříve jsou smazány všechny položky, které jsou naplánované na časové ose až za přidávanou položkou. Pak je položka inertního zpoždění přidána jako v případě transportního zpoždění. Následně jsou prohledány všechny zbylé staré položky. Jestliže se mezi nimi vyskytuje položka s jinou hodnotou signálu než je hodnota přidávané položky, jsou všechny položky až k poslední položce s rozdílnou hodnotou smazány. Zbylé položky se stejnou hodnotou jsou ponechány [5].

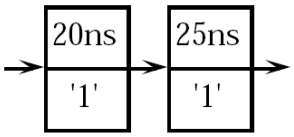


Obrázek 2.1.6 Grafická reprezentace plánovaného průběhu

Pro ilustraci předpokládejme, že se v čase 0 ns provede příkaz nastavení plánovaného výstupního průběhu, který je graficky zachycen na obrázku 2.1.6. Poté, také v čase 0 ns, bude proveden příkaz příkladu 2.1.6. Výsledný plánovaný průběh je zachycen na obrázku 2.1.7.

```
s <= '1' after 25 ns;
```

Příklad 2.1.6 Nastavení plánovaného průběhu



Obrázek 2.1.7 Grafická reprezentace plánovaného průběhu

Jestliže nastavujeme inertní zpoždění pro předpokládaný průběh s více změnami najednou, pouze první položka bude zavedena s inertním zpožděním, ostatní položky jsou zavedeny se zpožděním transportním.

2.2 Struktura „stimuli“ souboru jazyka VHDL

„Stimuli“ soubor jazyka VHDL má definovanou strukturu pouze v několika základních bodech, proto se „stimuli“ soubory od sebe navzájem mohou velice lišit. Následuje několik bodů, které jsou pro všechny společné:

1) Stejně jako na začátku každého VHDL souboru i „stimuli“ soubory začínají deklarováním použitých globálních knihoven a balíků. To se děje za pomoci klíčových slov **`library`** a **`use`**.

2) Následuje deklarace entity (viz kapitola 1.2.1). Tělo entity je většinou prázdné, jak ukazuje příklad 2.1.2, nebo může obsahovat deklaraci generických konstant (viz příklad 2.2.2).

3) Další částí je deklarace architektury (viz kapitola 1.2.2). V případě „stimuli“ souborů zde bývají deklarovány komponenty, generické konstanty, signály, proměnné aj. Tělo architektury pak většinou tvoří pouze nastavení předpokládaných výstupních průběhů (jak je tomu i u příkladů 2.1.2 a 2.2.2) a konkretizace nastavení komponent. V případě složitějších „stimuli“ souborů může tělo architektury obsahovat kromě uvedeného také očekávané průběhy výstupních signálů, popřípadě realizovat porovnání očekávaných a skutečných výstupních průběhů a hlášení vypisovat na terminál. Z důvodu rozsáhlosti takovýchto kódů zde nebude uveden příklad. Příklad takového *úplného „testbenche“* je možné najít v [26].

4) Na závěr většinou bývá deklarována konfigurace (viz kapitola 1.2.8).

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- addN Entity Description
entity addN is
   generic(N: INTEGER := 4);
   port(
      A: in std_ulogic_vector(N-1 downto 0);
      B: in std_ulogic_vector(N-1 downto 0);
      D: out std_ulogic_vector(N-1 downto 0);
      CIN: in std_ulogic;
      COUT: out std_ulogic
   );
end addN;

-- addN Architecture Description
architecture rtl of addN is
   signal pre_D : std_ulogic_vector(N downto 0);
   signal pre_OV : std_ulogic;
begin
   ARITHMETIC_Process: process(A,B,CIN)
      variable fct_out : std_ulogic_vector(N downto 0);
      variable a_ext,b_ext : std_ulogic_vector(N downto 0);
      variable carry_ext : std_ulogic_vector(1 downto 0);
      variable msb : integer;
   begin
      -- zero extend inputs to include carry bit
      a_ext := '0' & A;
      b_ext := '0' & B;
      carry_ext := '0' & CIN;

      -- ADD
      fct_out := a_ext + b_ext + carry_ext;

      -- Assign to signal for use outside process
      pre_D <= fct_out;

      -- Calculate overflow bit
      if (a_ext(N-1) = b_ext(N-1) and fct_out(N-1) = not a_ext(N-1)) then
         pre_OV <= '1';
      else
         pre_OV <= '0';
      end if;
   end process ARITHMETIC_Process;

   -- Assign the outputs
   D <= pre_D(N-1 downto 0);

   -- Assign flags
   COUT <= pre_D(N);
end rtl;
```

Příklad 2.2.1 N-bitová sčítačka v kódu VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity test_alus is
   generic(N: INTEGER := 4);
end test_alus;

architecture rtl of test_alus is

component addN
   port(
      A: in std_ulogic_vector(N-1 downto 0);
      B: in std_ulogic_vector(N-1 downto 0);
      D: out std_ulogic_vector(N-1 downto 0);
      CIN: in std_ulogic;
      COUT: out std_ulogic
   );
end component;

 for ALL : addN use entity WORK.addN(rtl);

 signal A,B,DOUT : std_ulogic_vector(N-1 downto 0);
 signal CIN : std_ulogic;

 BEGIN

Adder : addN
     generic map (N=>N);
     port map (A=>A,
               B=>B,
               CIN=>CIN,
               D=>DOUT,
               COUT=>OPEN);

--stimuli

SUB <= '0','1' after 200 ns;
CIN <= '0';

A <= "10101000",
     "01011001" after 100 ns,
     "11011011" after 200 ns,
     "01101001" after 300 ns;
B <= "00111000",
     "01001011" after 100 ns,
     "01010001" after 200 ns,
     "11001001" after 300 ns;
END rtl;
```

Příklad 2.2.2 „Testbench“ sčítačky z příkladu 2.2.1

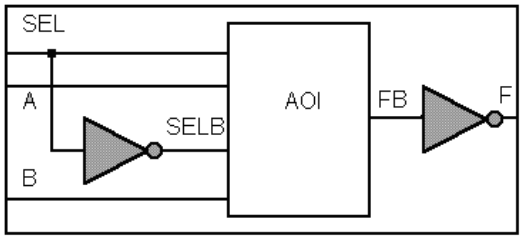
2.3 „Stimuli“ soubory dalších jazyků

2.3.1 „Stimuli“ soubory jazyka Verilog

Jazyk verilog je bohatě vybaven pro testování funkční i časové. Pro „stimuli“ soubory tohoto jazyka se používá anglického označení *test fixture* (testovací vybavení). Následuje příklad kódu jazyka Verilog, ke kterému bude uveden „test fixture“ soubor [21]. Kód reprezentuje multiplexer se dvěma vstupy, který je na obrázku 2.3.1.

```
module INV (input A, output F);   // invertor
  assign F = ~A;
endmodule

module AOI (input A, B, C, D, output F);// blok AOI
  assign F = ~((A & B) | (C & D));
endmodule

module MUX2 (input SEL, A, B, output F);   // 2:1 multiplexer
  // proměné SELB a FB jsou implicitní
  INV G1 (SEL, SELB);
  AOI G2 (SELB, A, SEL, B, FB);
  INV G3 (.A(FB), .F(F));            // mapování V/V
endmodule
```

Příklad 2.3.1 Multiplexer s dvěma vstupy v kódu VerilogHDL

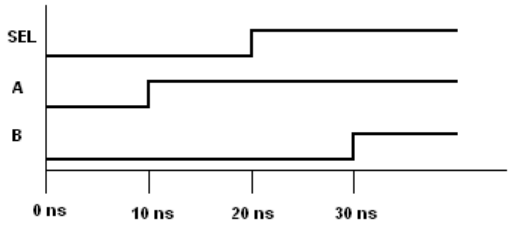


Obrázek 2.3.1 Multiplexer se dvěma vstupy

Analogie mezi obrázkem a uvedeným kódem je zřejmá. „Test fixture“ soubor pro tento obvod, jenž reprezentuje průběhy zachycené na obrázku 2.3.2, by vypadal následovně:

```
module MUX2TEST;                    // bez portů
initial                             // Stimulus
begin
  SEL = 0; A = 0; B = 0;
  #10 A = 1;                        // po 10ns A=1
  #10 SEL = 1;
  #10 B = 1;
end
  MUX2 M (SEL, A, B, F);
initial                             // Odezva
  $monitor($time, , SEL, A, B, F);
endmodule
```

Příklad 2.3.2 „Test fixture“ souboru k příkladu 2.3.1



Obrázek 2.3.2 Průběh signálu na vstupech multiplexeru

2.3.2 Test vektory jazyka ABEL-HDL

Zdrojový soubor jazyka ABEL-HDL obsahuje tzv. moduly, které jsou nezávislé, každý obsahuje kompletní popis příslušné části obvodu. Každý modul lze rozdělit do následujících částí: hlavička, deklarace, popis logiky, testovací vektory. Uvedeme si použití poslední části.

Testovací vektory (*TEST_VECTORS*) jsou použity k ověření funkčnosti navrženého obvodu. Kontrola může probíhat ve dvou úrovních. Vektory mohou být použity pro simulace softwarem nebo programátorem pro PLD k testování reálných operací obvodu. Je třeba mít na paměti, že v obou úrovních jde pouze o funkční simulaci, časovou simulaci s tímto prostředkem provádět nelze. Pro ilustraci následuje syntaxe výrazu a krátký příklad použití [10], [20].

```
TEST_VECTORS [ note ]
( input [, input ]... -> output[, output ]...)
[ invalues -> outvalues ; ]
      :
```

Definice 2.2 Syntaxe nastavení test vektoru jazyka ABEL-HDL

- note - nepovinný popisek testvektoru
- input - identifikátor(y) vstupního(ch) signálu(ů)
- output - identifikátor(y) výstupního(ch) signálu(ů)
- invalids - hodnota(ty) vstupního(ch) signálu(ů)
- outvalues - hodnota(ty) výstupního(ch) signálu(ů)

Následující jednoduchý příklad popisuje hradlo *and* se dvěma vstupy:

```
MODULE and1
TITLE 'and1 – dvojvstupe hradlo AND'
IN1, IN2, OUT1 pin;
EQUATIONS
OUT1 = IN1 & IN2;

TEST_VECTORS
([ IN1, IN2] -> [OUT1])
[ 0, 0] -> [ 0];
[ 0, 1] -> [ 0];
[ 1, 0] -> [ 0];
[ 1, 1] -> [ 1];
END
```

Příklad 2.3.3 „Test fixture“ souboru

3. Dostupné programy pro práci se „stimuli“ soubory

Využití jazyků HDL se neustále rozšiřuje. Proto také roste počet programů určených k designování číslicových obvodů. V této kapitole bude uvedeno několik programů, které umožňují navrhování a simulaci na programovatelných logických obvodech. Pozornost bude věnována tomu, jakým způsobem tyto programy realizují časovou simulaci.

3.1 MAX+PLUS II Waveform Editor

Prvním z představených programů je MAX+PLUS II, který uvedla na trh americká firma Altera Corporation. Firma začala s vývojem softwaru v roce 1993. Nyní je k dispozici verze 10.2, která je volně ke stažení na stránkách www.altera.com. Před tím je třeba provést bezplatnou registraci, stažená verze nepodporuje některé obvody a nejsou k dispozici všechny funkce, které nabízí verze plná.

Tento software je komplexní, na architektuře nezávislý prostředek, pro návrh programovatelných číslicových obvodů Altera, jako jsou: Classic™, ACEX 1K, MAX 3000, MAX® 5000, MAX 7000, MAX 9000, FLEX® 6000, FLEX 8000, FLEX 10K (MAX+PLUS II také podporuje FLASHlogic™ a APEX obvody). MAX+PLUS II je schopen pokrýt všechny úrovně návrhu logických obvodů. Umožňuje použití různých metod pro hierarchické návrhy, časovou a funkční simulaci, detailní časovou analýzu, automatické vyhledávání chyb, programování a verifikaci logických programovatelných obvodů. Následující kapitola se věnuje nástroji *waveform editor*, pomocí něhož lze vytvářet vstupní vektory pro časovou simulaci a také zpracovává výstupní informaci simulace. Část 3.1.2 nazvaná „MAX+PLUS II Timing Analyzer“ je věnována analyzátoru zpoždění. Ten sice přímo se „stimuli“ soubory nesouvisí, ale jak název napovídá, interpretuje výsledky časové analýzy.

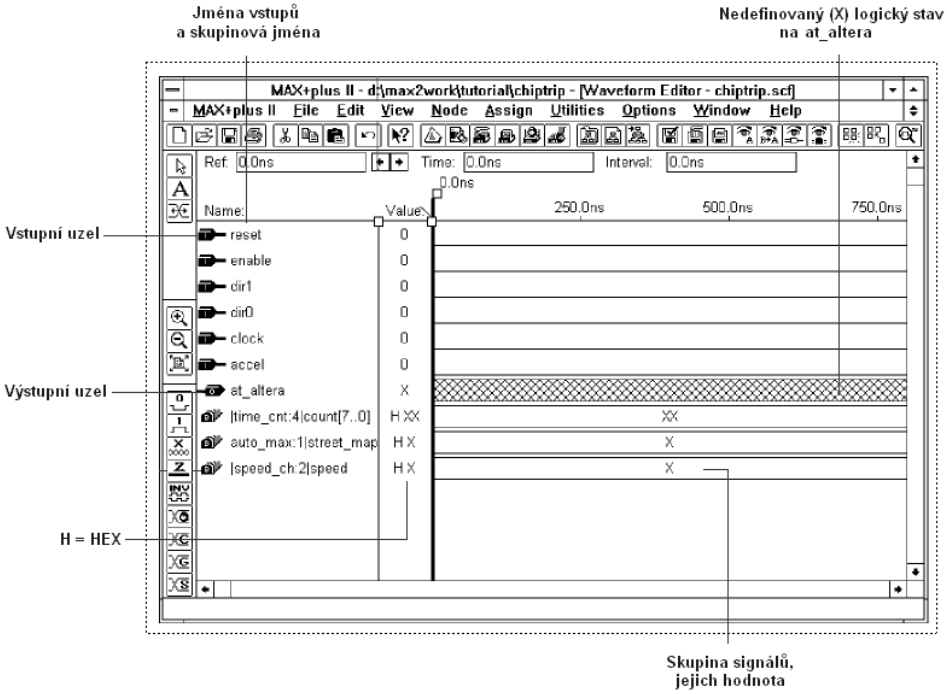
Max +PLUS II podporuje následující standardy: EDIF soubory (*Electronic Design Interchange format*, *.edf, *.edo), VHDL soubory (*.vhd), Verilog HDL soubory (*.v) a OrCAD schematicé soubory (*.gdf, *.sch). Software také umí importovat návrhy ze svých předchůdců, jako jsou A+PLUS™, SAM+PLUS™ a MAX+PLUS (DOS) firmy Altera Corp. [22].

V současné době firma Altera Corp. propaguje nástupce, software s názvem Quartus II. Ten by měl být kompatibilní s MAX+PLUS II, také by měl nabízet nová ulehčení při návrhu a měl by podporovat více obvodů typů CPLD a FPGA. Plná verze na 12 měsíců by v současné době měla stát $2000.

3.1.1 MAX+PLUS II Waveform Editor

Tento nástroj je použit pro vytváření vstupních vektorů a zobrazování výsledků simulace. Editor

umí načítat a ukládat vytvořené průběhy ve dvou různých formátech. První je označován jako *waveform design file* (*.wdf), obsahuje také návrh logiky pro projekt. Druhý, *simulator channel file* (*.scf), obsahuje pouze vstupní vektory pro simulaci. Jednotlivé vstupy a výstupy můžeme zadávat a odebírat ručně, nebo editor může použít soubor označený *simulator netlist file* (*.snf), který je vytvořen při kompilování zdrojového textu k obvodu, a použít všechny nebo vybrané vstupy obvodu.



Obrázek 3.1 MAX+PLUS II Waveform Editor

Na obrázku 3.1 je vidět, jak vypadá okno waveform editoru. Na levé straně je umístěna lišta s nástroji pro editaci průběhu signálů. Jestliže vybereme vstupní signál, můžeme ho nastavit celý do jistého stavu, měnit hodnotu konkrétního úseku nebo zadat periodickou změnu signálu. Také je možné sdružovat vstupy do skupin a zadávat jim hromadné hodnoty ve formě celých čísel. Signál lze měnit pouze v mřížce (*grid*). Velikost mřížky (*grid size*) lze měnit v libovolném intervalu.

3.1.2 MAX+PLUS II Timing Analyzer

Timing anlyzer (časový analyzátor) je nástroj pro analýzu návrhu obvodu po tom, co byl optimalizován kompilátorem. Pomocí něho lze sledovat zpoždění jednotlivých cest a determinovat cestu, která limituje výkonnost obvodu. Analyzátor používá informace ze souboru *simulator netlist*

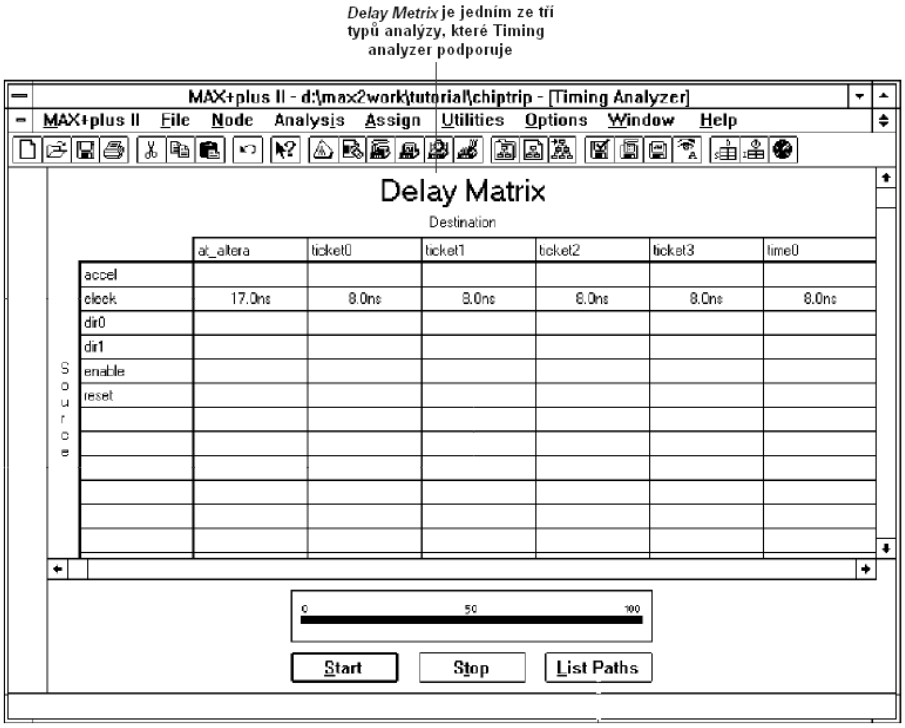
file (*.snf), který je vytvořen při kompilování zdrojového textu k obvodu. Umí generovat tři druhy analýzy:

- Matici zpoždění (*The Delay Matrix*), která ukazuje nejkratší a nejdelší zpoždění jednotlivých cest mezi vstupními a výstupními uzly.

- Matici nastavení/držení (*The Setup/Hold Matrix*), která ukazuje minimální požadované nastavené časy a časy nutné pro reprezentaci signálu ze vstupního pinu do formy dat.

- Monitor výkonnosti (*The Registered Performance Display*), který ukazuje výsledky výkonnostní analýzy, zahrnující uživatelem definovaný počet zpoždění, která limitují výkonnost, minimální hodinovou periodu a maximální frekvenci obvodu.

Na obrázku 3.2 je zachyceno okno časového analyzátoru.



Obrázek 3.2 Timing Analyzer

3.2 Synapti CAD TestBencher Pro

Americká firma SynaptiCAD se od roku 1992 zabývá vývojem softwarových nástrojů

k designování číslicových obvodů. Na serveru www.synapticad.com si lze po bezplatné registraci stáhnout „trialovou“ (15ti denní) verzi několika jejich produktů. Standardní balík (*allproducts.exe*) (dnes je nabízena verze 10.02) obsahuje následující aplikace:

- **TestBencher Pro** je generátor **„**testbench**“** souborů, který podporuje jazyky Verilog a VHDL. V další kapitole bude probrán detailněji.

- **VeriLogger Pro** – plnohodnotný verilog simulátor umožňující zobrazování vstupních a výstupních průběhů signálu, editaci verilog zdrojového textu, grafické a konzolové spouštění a řízení simulace.

- **DataSheed Pro** poskytuje široké možnosti při vytváření profesionální technické dokumentace k číslicovým obvodům (*datasheets*).

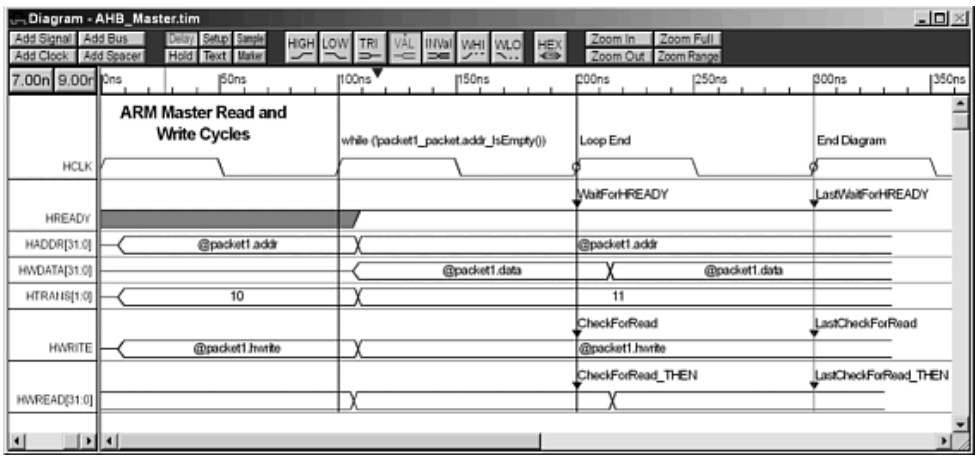
- **WaveFormer Pro** je interaktivní HDL simulátor, který podporuje jazyky jako: VHDL, Verilog, SPICE, Viewlogic, Mentor, OrCAD, aj. Také obsahuje editor časových diagramů obvodu.

- **Timing Diagrammer Pro** – editor časových diagramů obvodu.

- **GigaWave Viewer** – umožňuje zobrazovat vstupní a výstupní vektory obvodu. Prohlížeč podporuje mnoho požívaných formátů.

3.2.1 TestBencher Pro

TestBencher Pro je nástroj pro generování a ukládání „stimuli“ souborů. Podporuje jazyky VHDL a Verilog. Program byl představen ve své první verzi v roce 1996 jako návrhový systém pro obvody ASIC/FPGA. Dnes je použití zaměřeno hlavně na „testbench“ úlohy. Uživatel kreslí průběhy vstupních vektorů podobně, jako je tomu u předchozího zmiňovaného programu MAX+PLUS II.



Obrázek 3.3 TestBencher Pro

Jedním z mála rozdílů oproti předchozímu je, že program umožňuje zapnutí/vypnutí umisťování hran do mřížky (*grid*). Toto je určitě přínosem, protože uživatel může hranu přechodu umístit kam chce a nemusí složitě měnit velikost používané mřížky [23].

Ostatní programy z balíku zde nebudou probrány. Opticky působí programy velice podobně, jsou v nich používány stejné nástroje pro vykreslování průběhů. Distribuované bezplatné verze mají četná omezení, která brání důkladnější analýze.

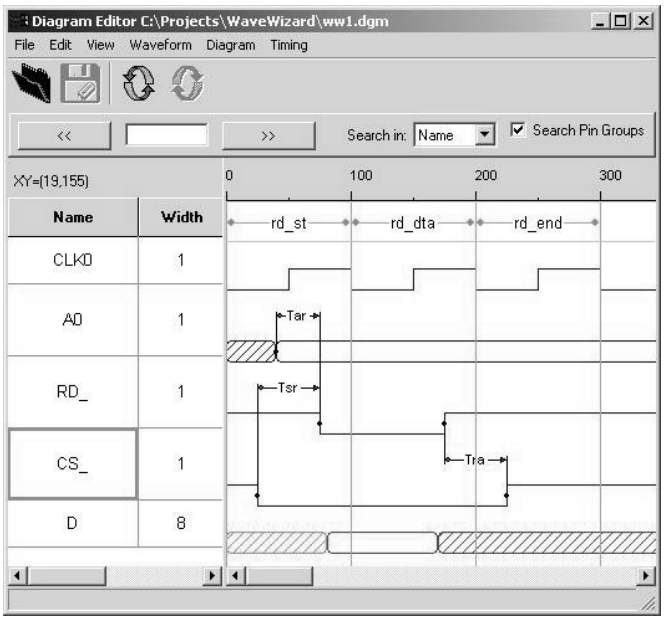
3.3 WaveWizard

Izraelská firma TestInsight Ltd. se zabývá vývojem softwarových prostředků pro automatické zkušební (testovací) zařízení ATE (*Automatic Test Equipment*). Jedním z jejich produktů je program WaveWizard. Bohužel firma nenabízí žádné zkušební ani ukázkové verze programu, dokumentaci k programům může stahovat pouze registrovaný uživatel softwaru, proto následující odstavec čerpá pouze z omezených informací uveřejněných na adrese www.testinsight.com.

Vstupem pro program mohou být „stimuli“ soubory ve formátech STIL, Verilog VCD (*Verilog Change Dump* – výpis změn verilog), EVCD (*Extended VCD* – rozšířený výpis změn verilog) a WGL (*Warwick Graphics Language* – jednoduchý jazyk pro popis průběhů signálů vyvinutý na anglické univerzitě ve Warwicku), nebo lze časové průběhy nakreslit v grafickém editoru a specifikaci obvodu provést pomocí integrovaného pomocníka. Výstupní soubory mohou být ve standardních formátech WGL a STIL, nebo ve speciálních formátech pro následující testery: Agilent HP93000, Agilent HP83000, LTX Vision, Teradyne Catalyst, Teradyne Tiger, Teradyne

J750, Teradyne Flex+, Teradyne J973, Teradyne J971.

Grafický editor signálů prý umožňuje snadné vytváření průběhů, ale podrobnější informace o použitelných nástrojích stránky nepodávají. Na obrázku 3.4 je vidět okno editoru.



Obrázek 3.4 WaweWizard – editor

3.4 Xilinx HDL Bencher

Firma Xilinx Inc. je na trhu s programy pro designování číslicových obvodů velice rozšířena a její produkty jsou špičkou ve své třídě. Na webový stránkách www.xilinx.com nabízí firma bezplatný balík programů s názvem *ISE WebPACK 7.1i.* Jedná se o zkušební verze programů, které nedisponují všemi funkcemi jako verze plné. Plná verze balíku ISE na 12 měsíců dnes stojí $2495.

Xilinx software podporuje designy obvodů v jazycích VHDL, Verilog, ABEL HDL, Schematic a EDIF. Pro vytváření a editaci „stimuli“ souborů Xilinx používá nástroj se jménem HDL Bencher [24].

3.4.1 HDL Bencher

Obrázek 3.5 dokumentuje, že tento nástroj se výrazně neliší od nástrojů dříve zmiňovaných programů. Umožňuje změnu signálu za pomoci myši a tedy ručním nastavením intervalů, zadávání

periodických změn signálu, dále také umožňuje sdružování signálů do skupin a nastavování hodnot v celých číslech. Velikost mřížky lze měnit a při zadávání periodických signálů lze umístit do jednoho okénka mřížky více period signálu.

Obrázek 3.5 HDL Bencher

4. Návrh grafického rozhraní pro tvorbu „stimuli“ souborů

Hlavním cílem diplomové práce je vytvořit grafické rozhraní, které umožní uživateli vytvořit „stimuli“ soubor. Požadavky na program jsou následující:

- Načíst definice signálů z návrhového souboru VHDL
  - ze zdrojového souboru obsahujícího popis obvodu,
  - z již vytvořeného „stimuli“ souboru, v tomto případě načíst také průběhy vstupních signálů.

- Zobrazit přečtené signály na obrazovku počítače,
  - u vektorových signálů umožnit volbu vektorového zobrazení (vektor je reprezentován jedním průběhem na obrazovce počítače) nebo bitového zobrazení (každý bit vektoru je představován jedním průběhem na obrazovce).

- Umožnit pomocí myši a standardních nástrojů měnit průběh signálů.

- Upravené průběhy signálů uložit do „stimuli“ souboru jazyka VHDL.

Motivace k vytvoření tohoto programu vzešla z Katedry elektroniky a zpracování signálů, kde se k navrhování číslicových obvodů používá například program, jehož název je „FPGA Advantage for HDL Design v 2004.1“. Lépe řečeno, je to balík programů od firmy Mentor Graphic, který obsahuje kompilátory a simulátory různých jazyků včetně jazyka VHDL. Pro simulaci balík používá program „ModelSim SE v 5.8c“. Tento program podporuje „stimuli“ soubory jazyka VHDL v určitém speciálním tvaru, který bude probírán v kapitole 5.6.

Bohužel balík neobsahuje žádný program, který by umožňoval vytváření „stimuli“ souborů. V případě testování funkčnosti navrženého obvodu si musel uživatel „stimuli“ soubor vytvořit za pomoci textového editoru. Proto bylo třeba vytvořit program, který uspoří uživateli čas a „stimuli“ soubor vytvoří za něj.

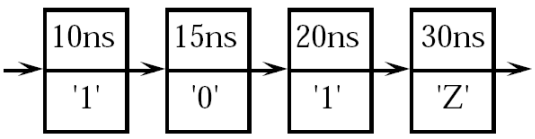
Následující kapitola popisuje program, který jsem nazval StimuliEdit_v1. Jednotlivé podkapitoly jsou seřazeny chronologicky, tak jak jsem postupoval při návrhu programu.

5. Program StimuliEdit

Pro realizaci programu jsem zvolil vývojové prostředí Borland Builder C++ v.6.0, které pracuje pod operačními systémy Microsoft Windows 98, Windows 2000 SP2 a Windows XP. Jak název napovídá, jedná se o programovací jazyk C++. Jazyk C++ vychází ze standardizovaného jazyka ANSI C, oproti kterému je rozšířen o objektové programování. Pro studium tohoto programovacího prostředí doporučuji [27] nebo manuál na stránkách firmy Borland (www.borland.com). Tento nástroj disponuje velkým množstvím knihoven, které bohatě pokryjí požadavky tohoto programu.

5.1 Logický signál reprezentovaný zřetězeným seznamem hodnot

První zásadní otázkou při realizaci programu bylo, jakým způsobem bude reprezentován průběh signálu. V analogii s reprezentací plánovaného průběhu signálu (viz kapitola 2.1.1) jsem se rozhodl signál reprezentovat jako jednosměrný zřetězený seznam hodnot.



Obrázek 5.1 Reprezentace plánovaného průběhu

5.1.1 Položky signálu

Jak ukazuje obrázek 5.1, každá položka ve VHDL obsahuje následující informace:

- časovou hodnotu, která udává vzdálenost na časové ose od nuly,
- hodnotu signálu, na kterou se signál mění v daném čase,
- ukazatel na následující položku.

Takováto konstrukce předpokládá na počátku, v čase *t=0,* inicializaci signálu na určitou hodnotu. Z toho vyplývá, že časová hodnota první položky v řadě je vždy 0. Také se předpokládá, že poslední položka trvá do nekonečna a její ukazatel míří „do prázdna“ (má nulovou hodnotu).

Pro lepší představitelnost a snadnější práci s řadou jsem se rozhodl pro úpravu informací nesených jednotlivými položkami. Program používá položky nesoucí následující informace:

- časovou hodnotu, která udává délku aktuální položky na časové ose,
- hodnotu signálu aktuální položky,
- ukazatel na následující položku.

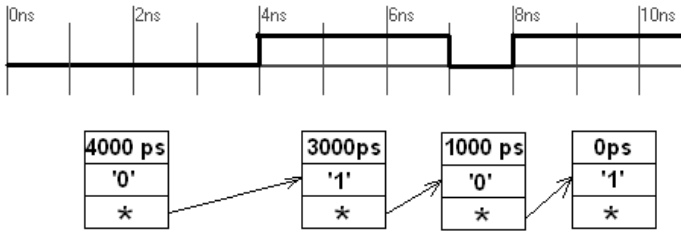
Při této konstrukci má poslední položka nulovou délku na časové ose a představuje nekonečný prvek. Každou z položek signálu jsem se rozhodl reprezentovat jako samostatný objekt. Třídu objektů jsem nazval *TItemOfSignal* a její definice je umístěna ve zdrojových souborech *SignalDef.cpp* a *SignalDef.h* (přiložené CD – adresář \sources\). Tato třída obsahuje tři chráněné proměnné:

- Proměnná *TItemOfSignal::Length* představuje délku položky na časové ose a je typu *unsigned __int64,* což je 64-bitová nezáporná celočíselná proměnná. Jako základní jednotky jsem zvolil pikosekundy, jelikož rozlišení časové osy na femtosekundy, což jsou základní jednotky jazyka VHDL, se používá velice ojediněle. Při této reprezentaci jedna položka může nabývat hodnoty 0 až asi 213 dnů (= 2^{64} ps). Proměnná je značně předimenzovaná, ale jelikož záporné hodnoty času v této aplikaci nemají žádný smysl a 32-bitová proměnná by představovala rozsah pouze 0 až 4 milisekundy, přistoupil jsem k této realizaci.

- Proměnná *TItemOfSignal::Value* představuje hodnotu aktuální položky signálu. Volba typu této proměnné představuje největší omezení programu. Jelikož VHDL umožňuje definovat uživatelské datové objekty od celočíselných hodnot, přes řetězce znaků až po strukturované proměnné typu Record (viz [5] kapitola 2.2), ideální by bylo definovat proměnou *TItemOfSignal::Value* pouze jako ukazatel na obecný datový objekt. Jelikož implementace takového modelu proměnné do programu by byla velice časově náročná, po konzultaci s vedoucím diplomové práce jsem se rozhodl přistoupit k následujícímu omezení. Proměnná *TItemOfSignal::Value* bude typu *unsigned int,* což představuje 32-bitové nezáporné číslo. Z tohoto omezení vyplývá, že program bude schopen pracovat pouze s bitovými vektory v rozsahu 1 až 32 bitů a jednotlivé bity vektoru lze nastavit pouze v dvoustavové logice (mohou tedy nabývat pouze 0 a 1). Důsledky tohoto omezení budou probírány i v dalších kapitolách.

- Proměnná *TItemOfSignal::Next* je typu *TitemOfSignal*,* představuje tedy ukazatel na následující položku signálu.

Následující obrázek ukazuje, jakým způsobem je průběh signálu zachycen v programu StimuliEdit.



Obrázek 5.2 Příklad reprezentace signálu v programu StimuliEdit

Kromě těchto chráněných proměnných obsahuje třída také řadu metod pro nastavování a čtení všech hodnot. Jelikož veškeré operace nad položkami signálu provádí objekt signálu, není třeba se těmito metodami více zabývat. Metody i s příslušným komentářem jsou definovány ve zdrojových souborech programu.

5.1.2 Objekt signálu

Pro reprezentaci jednotlivých signálů používá program objekty z třídy *TSignalVector* (viz soubory

SignalDef.cpp a *SignalDef.h*, přiložené CD – adresář *\sources*). Metody, kterými disponuje objekt, vykonávají veškeré operace nad signálem (nastavování a čtení průběhu signálu), včetně nastavování *masky výběru.*

Jelikož uživatel bude potřebovat za pomoci myši označit část signálu, které následně bude chtít přiřadit nějakou hodnotu, rozhodl jsem se tuto *masku výběru* integrovat do objektu signálu. Proto jsem nejdříve definoval třídu objektů nazvanou *Tsignal* (definice viz soubory *SignalDef.cpp* a *SignalDef.h*, přiložené CD – adresář \sources\).

Objekt třídy *Tsignal* obsahuje tři chráněné proměnné:

- Proměnná *Tsignal::Name* je typu *AnsiString* a představuje jméno signálu. Na obsah proměnné typu *AnsiString* jsou kladeny podstatně menší nároky než na identifikátory jazyka VHDL, proto tato proměnná dokáže pojmout všechny možné varianty identifikátorů VHDL.
- Jelikož signál je reprezentován jednosměrným zřetězeným seznamem hodnot, je třeba neustále znát první položku signálu. Proměnná *Tsignal::FirstItem* je ukazatel typu *TitemOfSignal*,* který obsahuje adresu první položky signálu.
- Pro jednoduší práci se signálem jsem zavedl ještě jeden ukazatel na položku signálu a nazval jsem jej *Tsignal::CurrentItem*. Představuje pracovní, aktuální položku signálu, se kterou právě pracujeme.

Dále třída disponuje množstvím metod, které umožňují různé operace s objektem. Definice s příslušným komentářem samozřejmě lze nalézt ve zdrojových kódech. Zde bude uveden pouze velice hrubý popis.

- Metoda *Tsignal::setName(const AnsiString& inName)* nastavuje jméno signálu, čtení jména signálu se provádí za pomoci metody *Tsignal::giveName()*.
- Dalších devět metod slouží k pohybu po průběhu signálu. To znamená, že se jedná o metody, které nastavují ukazatel na aktuální položku (*Tsignal::CurrentItem*). Konkrétně jde o dvě skupiny metod. Jedny jsou určené k dotazování na možnost posunu na první (poslední) položku signálu nebo na položku předcházející (následující) za aktuální položkou. Ty vrací ukazatel na požadovanou položku; v případě, že přesun není možný, metody vrací *NULL*.
    - *Tsignal::giveFirstItem()* – vrací ukazatel na první položku signálu

- *Tsignal::giveCurrentItem()* – vrací ukazatel na aktuální položku
- *Tsignal::giveLastItem()* – vrací ukazatel na poslední položku signálu
- *Tsignal::giveNextItem()* – vrací ukazatel na následující položku
- *Tsignal::givePreviousItem()* – vrací ukazatel na předcházející položku

Druhá skupina funguje obdobně, ale metody zároveň realizují přesun na požadovanou položku (mění ukazatel *CurrentItem*). Jména metod jsou stejná jako u předchozích uvedených, ale místo *give* mají v názvu *goto* (vyjma metody *giveCurrentItem()*, ta v této skupině nemá svůj ekvivalent).

- Objekt třídy *Tsignal* také disponuje metodami, které souvisí s umístěním aktuálního prvku na časové ose a jeho hodnotou.
  - *Tsignal::giveCurrentLength()* – vrací velikost aktuální položky v pikosekundách.
  - *Tsignal::giveCurrentLengthFromStart()* – vrací počátek aktuální položky na časové ose v pikosekundách (tedy součet velikostí všech předešlých položek signálu).
  - *Tsignal::giveCurrentValue()* – vrací hodnotu aktuálního prvku signálu.

- Další skupinu představují metody, které nastavují hodnotu signálu v rozmezí časových intervalů.
  - *Tsignal::setValueFromTo (const unsigned int& inValue, const unsigned __int64& fromTime, const unsigned __int64& toTime)* – metoda nastavuje hodnotu signálu na *inValue* v intervalu od (*fromTime[ps])* do (*toTime[ps]*). Metoda je koncipována tak, že může nastat případ, kdy v řadě vzniknou položky nulové délky, popřípadě se vedle sebe mohou vyskytnout položky se stejnou hodnotou signálu. Z hlediska „stimuli“ souborů tyto případy nejsou žádoucí, proto je dobré po nastavení signálu spustit metodu *Tsignal::checkSignal()* (viz níže).
  - *Tsignal::bitAndValueFromTo (...)* – metoda provádí v zadaném časovém rozmezí logický bitový součin hodnoty průběhu signálu s hodnotou *inValue* (parametry jsou shodné s předchozí metodou.)
  - *Tsignal::bitOrValueFromTo (...)* – metoda podobná předcházející, realizuje bitový součet.

- Následují metody, které nepatří do předchozích skupin a ještě nebyly jmenovány:
  - *Tsignal::giveValueAt(unsigned __int64 atTime)* – metoda vrací hodnotu signálu v čase *inTime*[ps].

- *Tsignal::checkSignal()* – provádí kontrolu signálu, vypouští položky nulové délky a sdružuje sousední položky, které mají stejnou hodnotu signálu.

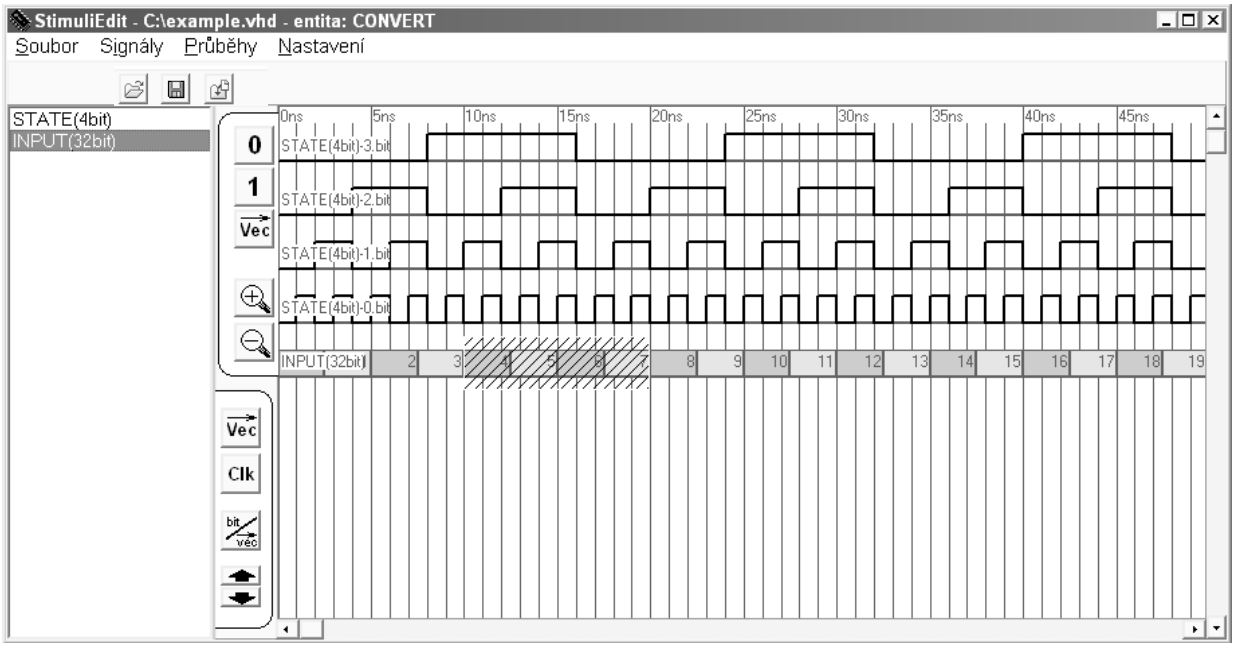
Objekty třídy *Tsignal* se přímo v programu nevyskytují. Signály v programu reprezentují objekty třídy *TSignalVector,* která je odvozená od třídy *Tsignal*. Odvozená třída znamená, že obsahuje všechny proměnné a metody svého předchůdce a může obsahovat navíc vlastní metody nebo proměnné. Třída *TSignalVector* tedy převzala všechny vlastnosti třídy *Tsignal* a navíc jsou definovány vlastní (definice viz soubory *SignalDef.cpp* a *SignalDef.h,* přiložené CD – adresář \sources\):

- Chráněná proměnná *TSignalVector::BitCount* je typu *int,* udává počet bitů vektoru. Jak již bylo řečeno, v programu StimuliEdit může proměnná nabývat hodnot 1 až 32.
- Veřejná proměnná *TSignalVector::Packed* je typu *bool.* V případě, že hodnota proměnné je *false,* signál je zobrazen jako průběh jednotlivých bitů signálu. V případě *true* je signál zobrazen jako vektor (viz kapitola 5.2).
- Veřejná proměnná *TSignalVector::Select* je typu *Tsignal**, tedy ukazatel na objekt třídy *Tsignal.* Jakmile je totiž vytvořen objekt třídy *TsignalVector*, konstruktor této třídy vytvoří ještě jeden objekt třídy *Tsignal* a na něj „namíří“ tento ukazatel. Nový objekt představuje již zmiňovanou masku výběru. Hodnota jednotlivých položek masky určuje bitové pokrytí maskou výběru (hodnota 0 ≈ žádný výběr, hodnota 2^{32} ≈ vybrány všechny bity 32-bitového signálu). Důvod, proč jsem zvolil tuto konstrukci, bude vysvětlen v kapitole 5.3.
- Metoda *TsignalVector::giveBitCount* vrací počet bitů signálu.
- Metoda *TsignalVector::clearSelect* nastavuje hodnotu masky výběru v celém rozsahu na 0.
- Metoda *TSignalVector::setGeneric(const unsigned __int64& inPeriod,const unsigned __int64& TimeTo)* nastavuje periodickou změnu hodnoty na nejnižším bitu signálu s periodou *inPeriod*[ps]. Používá se výhradně pro generování signálů, které reprezentují generické konstanty typu *time.* Důvod zavedení metody bude vysvětlen v kapitole 5.5.

5.2 Grafická reprezentace logického signálu

Formuláře, které nabízí programovací prostředí Builder C++, umožňují kreslit na celou jejich plochu za pomoci plátna (*canvas*). Na plátno (matice pixelů) se dají kreslit základní geometrické tvary v rozlišení, které udává formulář.

Program StimuliEdit používá pro vykreslování signálů plátno svého hlavního formuláře. Hlavní formulář (obrázek 5.3) je objekt třídy *TFormMain.* Definice třídy je umístěna v souborech *UnitMain.cpp, UnitMain.h* a vlastnosti formuláře jsou uloženy v souboru *UnitMain.dfm* (soubor „Builderu C++“ obsahující vlastnosti a umístění všech komponent na formuláři) (viz přiložené CD – adresář *\sources*).



Obrázek 5.3 Hlavní formulář programu StimuliEdit

Hlavní formulář by se dal rozdělit do čtyř hlavních částí. První tvoří komponenta hlavního menu, která je umístěna standardně ve vrchní části formuláře. Položky hlavního menu budou probrány později. Levou část formuláře tvoří komponenta seznamu signálů. Seznam je objekt typu *TListBox*, která je definovaná ve standardních knihovnách prostředí Borland Builder C++. Pravou část formuláře tvoří zmíněné plátno (ohraničené horizontálním a vertikálním posuvníkem), na které program vykresluje průběh signálů v takovém pořadí, v jakém jsou seřazeny v seznamu signálů. Vertikální panel, který odděluje seznam signálů a plátno, obsahuje tlačítka pro nastavování průběhu signálu (viz kapitola 5.4).

Při zobrazování průběhu číslicových vektorových signálů se používá dvou základních možností jak signál zobrazovat. První možnost je zobrazit celý signál jako vektor a změnu hodnoty vektoru zapisovat do grafu. Druhá možnost je zobrazovat signál po bitech. V takovém případě jsou zvlášť zobrazeny průběhy signálu na jednotlivých bitech vektoru. Způsob, jakým bude signál vykreslován, je dán proměnou *TsignalVector::Packed* (viz kapitola 5.1.2). Je-li hodnota proměnné

true, pak je signál zobrazen vektorově. StimuliEdit používá vykreslování signálů jaké je vidět na obrázku 5.3. První odshora je čtyřbitový vektor *STATE,* který je zobrazen po bitech, takže mu náleží první 4 vykreslované průběhy. Následuje 32-bitový vektor *INPUT* zobrazený vektorově. Na tomto vektoru je vidět i maska výběru (viz kapitola 5.3) v intervalu od 10 ns do 20 ns.

Vykreslování průběhů signálů na plátno je závislé na následujících veřejných proměnných hlavního formuláře:

- Proměnné *TFormMain::HeightOfSignal* a *TFormMain::DistanceBetweenSignals* jsou typu *int* a jejich jednotkami jsou pixely. Výška vykreslovaného vektorového průběhu v pixelech je dána proměnnou *TFormMain::HeightOfSignal*. Při bitovém zobrazení vektoru představuje proměnná výšku zobrazeného bitu, je-li ve stavu '1'.
  Rozestup mezi zobrazovanými vektorovými průběhy je dán proměnnou *TFormMain::DistanceBetweenSignals.*
- *TFormMain::GridFrom* a *TFormMain::GridTo* jsou nezáporné celočíselné 64-bitové proměnné (*unsigned __int64*). Udávají časový interval, ve kterém jsou signály a mřížka zobrazovány. Jednotky jsou pikosekundy. Nyní se dostáváme k dalšímu omezení programu, které není zdaleka tak limitující jako volba typu proměnné reprezentující hodnotu položky signálu (viz kapitola 5.1.1). Proměnné dokážou pojmout číslo 0 až 2^{64}, to by znamenalo rozsah asi 0 až 213 dnů. Jelikož pro vypisování časových údajů do výstupního „stimuli“ souboru jsou použity standardní převodní funkce, které nedokážou pracovat s nezápornými 64-bitovými hodnotami(*unsigned*), musel jsem rozsah, ve kterém je možné nastavovat hodnoty signálů, snížit na 0 ps až 2^{63} ps. Tento rozsah představuje asi 0 až 107 dnů, což je stále mnohokrát větší rozsah než bude pro aplikace potřeba. Proměnné menšího rozsahu jsem nemohl použít (viz kapitola 5.1.1).
- Proměnná *TFormMain::GridSize* udává rozměr jednoho okénka mřížky v pikosekundách. Je typu *__int64,* což znamená, že obor nezáporných hodnot této proměnné představuje časový rozsah, který podporuje aplikace StimuliEdit.

Vykreslování průběhů signálů na plátno většinou realizuje událost hlavního formuláře *OnPaint*. Událost nastane vždy, když operační systém potřebuje překreslit formulář (přetahování oken, minimalizace/maximalizace okna aplikace atd.). V takovém případě je spuštěna metoda hlavního formuláře *TFormMain::FormPaint(TObject *Sender).* Vykreslování průběhů signálů probíhá ve čtyřech navazujících krocích.

- Jelikož je funkce volána i v případech posunu po časové ose, změny pořadí signálů v seznamu atd., nejdříve je třeba smazat předchozí vykreslené průběhy. Metoda hlavního formuláře *TFormMain::RepaintBackground()* na celou plochu plátna vykreslí obdélník v barvě pozadí (barvu je možné nastavit viz kapitola 5.7). Plátno formuláře je připraveno pro kreslení.
- Na jednobarevnou plochu plátna je následně nakreslena mřížka časové osy. To realizuje metoda hlavního formuláře *TFormMain::RepaintGrid()*. Metoda vykreslí svislé čáry a jejich popisky v nastavené barvě (viz kapitola 5.7).
- Následně je volána metoda pro překreslení masky výběru *TFormMain::RepaintSelect()*. Ta v závislosti na pozici vertikálního posuvníku, který udává index do seznamu signálů prvního vykreslovaného průběhu, a na hodnotách proměnných uvedených výše postupně vykresluje masku výběru signálů tak, jak jsou seřazeny v seznamu.
- Jako poslední probíhá vykreslování vlastního průběhu signálů v závislosti na stejných parametrech, jako u vykreslování masky výběru. Metoda hlavního formuláře, která toto realizuje, se jmenuje *TFormMain::RepaintGrid()*.

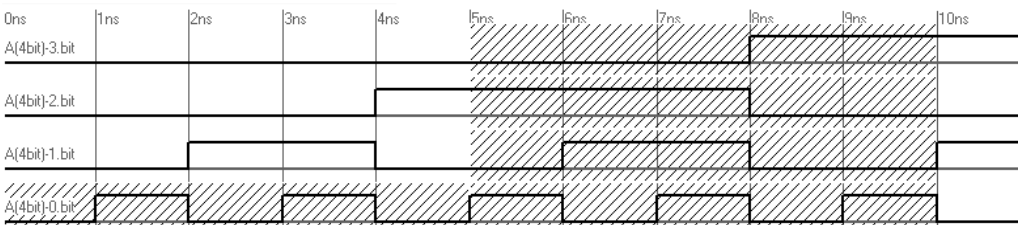
Uvedený princip vykreslování může způsobit problikávání průběhů signálů do barvy pozadí. To je způsobeno tím, že při překreslování dochází ke smazání celé plochy plátna. Tento neduh programu by se dal odstranit tak, že překreslování plátna by realizovala jedna metoda podle následujícího scénáře:

1. překresli přes plochu, kterou zaujímá první zobrazovaný průběh, obdélník barvy pozadí,
2. na stejnou plochu nakresli mřížku a průběh signálu,
3. pro všechny ostatní zobrazené průběhy zopakuj 1. a 2. bod.

Tento postup jsem nerealizoval z důvodu časové náročnosti.

5.3 Maska výběru logického signálu

Jak již bylo řečeno, maska výběru označuje části průběhu signálu, jejichž hodnota může být editována nástroji pro grafickou editaci signálu (viz kapitola 5.4). Každý signál v programu StimuliEdit má svojí masku výběru, která představuje objekt typu *Tsignal* (viz kapitola 5.1.2). Hodnoty položek masky výběru představují pokrytí signálu maskou.



Obrázek 5.4 Pokrytí 4-bitového signálu A maskou výběru

Příklad pokrytí signálu maskou je vidět na obrázku 5.4. Masku reprezentuje šrafovaná část. V tomto příkladě se maska skládá ze tří položek. První položka zaujímá interval ⟨0, 5) ns a její hodnota je $1 = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$ (pokrytí pouze nultého bitu). Druhá položka je v intervalu ⟨5, 10) ns a má hodnotu $15 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$ (pokrytí všech bitů). Třetí položka začíná na 10 ns včetně, končí v nekonečnu a její hodnota je 0 (signál není pokryt maskou).

Formuláře programovacího prostředí Borland Builder C++ nabízejí dvě události, které StimuliEdit používá pro nastavování masky výběru. První je událost hlavní formuláře *OnMouseDown,* která nastane vždy, když uživatel v prostoru formuláře zmáčkne tlačítko myši. Tato událost spustí metodu hlavního formuláře *TFormMain::FormMouseDown,* která nastaví zobrazované informace o signálu (*hint* položku a položku *Info* výsuvného menu plátna hlavního formuláře viz kapitola 5.7) a masku signálu, na kterém se nachází ukazatel myši, v závislosti na následujících parametrech:

- V případě, že bylo zmáčknuto pouze levé tlačítko myši, nejdříve se smažou masky výběrů všech signálů (v celém rozsahu jsou nastaveny na hodnotu 0). Pak je nastavena maska tak, aby pokrývala okénko mřížky, nad kterým se nachází kurzor. U vektorově zobrazeného signálu jsou maskou pokryty všechny bity signálu. V případě bitově zobrazeného vektoru je maskou pokryt pouze bit, nad jehož průběhem se nachází kurzor myši.
- V případě, že bylo zmáčknuto levé tlačítko myši a zároveň klávesa *Ctrl* nebo *Shift,* nejdříve proběhne kontrola, zda nebyl dříve označen maskou signál s jiným počtem bitů (jestliže maska pokrývá pouze některé bity vícebitového signálu, program bere masku jako by pokývala jednobitový signál). Má-li dříve označený signál jiný počet bitů, program zobrazí hlášení, že není možno tento signál označit. V opačném případě program přidá k dosavadní masce signálu pokrytí okénka, nad nímž se nachází kurzor myši.
- V případě, že bylo zmáčknuto prostřední tlačítko myši a zároveň klávesa *Ctrl* nebo *Shift,*

program se zachová podobně jako v předchozím případě, ale maska je z aktuálního okénka odstraněna.

- V případě stisku pouze prostředního tlačítka myši, program smaže pokrytí maskou u všech signálů.

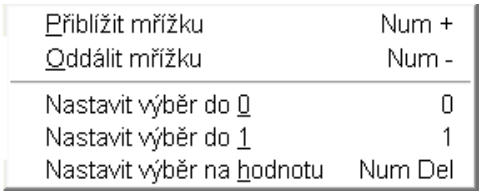
Jelikož předchozí metoda je volána pouze při stisku tlačítka myši, je třeba zajistit to, aby byla maska nastavena při pohybu myši, když je kontinuálně stisknuté tlačítko myši. Tento problém řeší událost hlavního formuláře *OnMouseMove,* která nastává vždy, když se kurzor myši pohybuje přes formulář. Tato událost spouští metodu hlavního formuláře *TFormMain::FormMouseMove.* Metoda obsahuje proceduru, která funguje analogicky k předchozí s tím rozdílem, že nenastavuje zobrazované informace o signálu.

5.4 Nástroje pro grafickou editaci logického signálu

Pro grafickou editaci průběhu logických signálů jsem zvolil standardní používané nástroje (viz kapitola 3).

5.4.1 Nástroje pracující s maskou výběru

První skupina nástrojů nastavuje hodnotu signálu v intervalech, které jsou pokryty maskou výběru. Tyto nástroje jsou dostupné v hlavním menu – skupina položek „Průběhy“. Tato položka obsahuje další menu zobrazené na obrázku 5.5. V levém sloupci je jméno nástroje, v pravém sloupci zkratková klávesa.



Obrázek 5.5 Skupina položek hlavního menu „Průběhy“

První dvě položky jsou od ostatních odděleny, protože nesouvisí s nastavováním hodnot signálů, ale s rozlišením mřížky na časové ose. Po kliknutí na „Přiblížit mřížku“ je zavolána metoda hlavního formuláře *TFormMain::ZoomGridInClick(TObject *Sender),* kliknutí na „Oddálit mřížku“ volá metodu *TFormMain::ZoomGridOutClick(TObject *Sender)*. Metody umožňují měnit velikost mřížky na časové ose, popřípadě měnit počet zobrazovaných okének. O tom, jestli bude změněn

počet okének nebo jestli se bude měnit velikost jednoho okénka, rozhodují dvě veřejné proměnné hlavního formuláře *TFormMain::MinGridCount* a *TFormMain::MaxGridCount*. Proměnné jsou typu *int* a udávají minimální a maximální počet zobrazovaných okének mřížky. Metoda přibližování se snaží snížit počet zobrazovaných okének na polovinu (to se děje změnou hodnoty *TFormMain::GridTo* viz kapitola 5.2). V případě, že polovina počtu zobrazovaných okének by byla menší než udává proměnná, pak se program snaží zmenšit rozměr okénka na polovinu (změna hodnoty *TFormMain::GridSize* viz kapitola 5.2). Není-li ani toto možné (velikost okénka mřížky = 1 ps), aplikace to oznámí hlášením. Analogicky, metoda oddalování se snaží zdvojnásobit počet okének mřížky. Není-li to možné, zdvojnásobuje se velikost okénka mřížky.

Následující dvě položky „Nastavit výběr do 0“ (metoda hlavního formuláře *TFormMain::SetSelectToFalseClick(TObject *Sender)*) a „Nastavit výběr do 1“ (metoda hlavního formuláře *TFormMain::SetSelectToTrueClick(TObject *Sender)*) nastavují hodnoty signálu v intervalech, které jsou pokryty maskou výběru. V případě, že je maskou výběru pokryt n-bitový signál (pro n > 1) zobrazený vektorově, všechny jeho bity jsou nastaveny na '0', respektive na '1'. To znamená, že jeho výsledná hodnota v nastaveném intervalu bude 0, respektive $2^n - 1$.

Poslední položka „Nastavit výběr na hodnotu“ (metoda hlavního formuláře *TFormMain::SetSelectToValueClick(TObject *Sender)*) přednastavuje hodnoty formuláře pro nastavení hodnoty vektoru (viz kapitola 5.4.2), který umožňuje nastavit signál na libovolnou hodnotu. Formulář je vyvolán pro každý interval, který je pokryt jednou položkou masky výběru. Metodu je možné použít pouze v případě, že maskou výběru je pokryt vícebitový signál zobrazený vektorově. Na jednobitové signály nebo na vícebitové signály zobrazené po jednotlivých bitech nemá metoda vliv.

5.4.2 Nástroje vztahující se k seznamu signálů

Nástroje pracující se seznamem signálů, jsou umístěny v hlavním menu, skupina položek „Signály“.

Obrázek 5.6 Skupina položek hlavního menu „Signály“

První dvě položky pouze mění hodnotu proměnné *TSignalVector::Packed* označeného signálu v seznamu. Signál je poté vykreslován vektorově nebo po jednotlivých bitech, v závislosti na zvolené položce. Zobrazení je možné měnit u všech signálů (i jednobitových) vyjma položek, které reprezentují generické konstanty typu *time* (viz kapitola 1.2.1). Při kliknutí na položku menu je volána metoda hlavního formuláře *TFormMain::ShowSignalsLikeVectorsClick(TObject *Sender),* respektive *TFormMain::ShowSignalsLikeBitsClick(TObject *Sender)*.

Druhé dvě položky umožňují měnit pořadí signálů v seznamu. Průběhy signálů jsou zobrazovány na plátno ve stejném pořadí, jako jsou seřazeny v seznamu signálů.

Kliknutím na položku „Nastavit hodnotu signálu“ se přednastaví hodnoty formuláře pro nastavení hodnoty vektoru a formulář se zobrazí. Definice formuláře je v souborech *UnitValue.cpp* a *UnitValue.h* (viz přiložené CD – adresář *\sources*).

Obrázek 5.7 Formulář pro nastavení hodnoty vektoru

Na obrázku 5.7 je zobrazen formulář pro nastavení hodnoty vektoru. Hodnoty políček „v čase od“ a „do“ jsou přednastaveny na meze právě zobrazovaného časového intervalu na plátnu hlavního formuláře. Uživatel může následně hodnoty časových mezí upravit. Nastavovanou hodnotu je možné zadávat ve třech různých číselných soustavách: hexadecimálně, dekadicky a binárně. Formulář umožňuje nastavit požadovanou hodnotu, popřípadě provést logický bitový součet, respektive součin, požadované hodnoty s hodnotu signálu v nastaveném časovém intervalu.

Kliknutí na položku „Nastavit periodický průběh“ spustí metodu hlavního formuláře *TFormMain::SetClockToSignalClick(TObject *Sender),* která přednastaví časové meze formuláře pro nastavení periodické změny signálu, zobrazeného na obrázku 5.8. Metoda nastaví stejné položky jako v předchozím příkladu, plus do obou políček periody doplní polovinu velikosti jednoho okénka mřížky. Jednotky periody nejsou poměrné, jak bývá zvykem, ale jsou stejné jako jednotky zvolené pro časový interval. Hodnoty zmiňovaných políček může uživatel měnit. Dále formulář umožňuje zadat počáteční hodnotu vektoru a inkrement, o který bude hodnota signálu při každém kroku navyšována. Definice tohoto formuláře je v souborech *UnitClock.cpp* a *UnitClock.h* (viz přiložené CD – adresář *\sources*).

Obrázek 5.8 Formulář pro nastavení periodické změny signálu

Poslední položka „Přidat signál typu GENERIC TIME“ spouští metodu hlavního formuláře *TFormMain::AddGenericSignalClick(TObject *Sender).* Tato rutina zobrazí formulář pro přidání generické konstanty typu *time*. Formulář je na obrázku 5.9 a jeho definice je umístěna v souborech *UnitGeneric.cpp* a *UnitGeneric.h* (viz přiložené CD – adresář *\sources*).

Obrázek 5.9 Formulář pro přidání generické konstanty

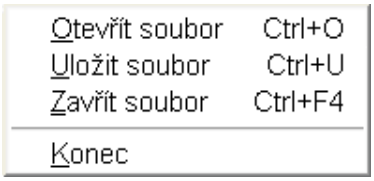
Formulář umožňuje nastavit jméno generické konstanty, s jakým bude zavedena do výsledného „stimuli“ souboru, periodu (velikost generické konstanty), a mez, do které se bude signál zobrazovat. Po kliknutí na tlačítko „OK“ formulář přidá do seznamu signál, jehož nastavené jméno bude doplněno o řetězec „(generic)“. To znamená, že v příkladu na obrázku 5.9 by v seznamu přibyl signál jména *CLK_PER(generic)*. Následně je vygenerován průběh s nastavenou periodou do zadané časové meze. Do políčka časové meze je při vyvolání formuláře doplněna hodnota proměnné *TFormMain::MaxCountForPeriodic* (*viz kapitola 5.7*) vynásobená velikostí periody . K tomuto omezení zobrazení průběhu jsem přistoupil z toho důvodu, že časová náročnost generování periodického průběhu signálu je přímo úměrná počtu generovaných změn signálu. Při nastavení příliš vysokého počtu změn může dojít k „zamrznutí“ aplikace.

Program StimuliEdit „generické“ signály reprezentuje stejně jako ostatní, ale se dvěma rozdíly. Jednak průběh takového signálu není možné editovat, a při zápisu do „stimuli“ souboru bude zaveden pouze do hlavičky entity jako generická konstanta typu *time.* StimuliEdit identifikuje signál typu „generic“ pouze podle toho, že jeho jméno obsahuje řetězec „(generic)“. Pro další verze programu by bylo lepší, kdyby se „generické“ signály zaváděly jako objekty jiného typu než *TSignalVector* (viz kapitola 5.1.2). Pak by se nemusel generovat průběh signálu a funkce zobrazování signálů (viz kapitola 5.2), která by musela dostát určitých změn, by mohla signál zobrazovat až „do nekonečna“.

5.5 Přečtení definic signálů z návrhového souboru

Skupina položek hlavního menu nazvaná „Soubor“, která je na obrázku 5.10, umožňuje následující operace: přečíst definice signálů z návrhového souboru VHDL, uložit nastavené průběhy do „stimuli“ souboru jazyka VHDL (viz kapitola 5.6) nebo zavřít otevřený návrhový soubor, to

znamená odebrat všechny signály ze seznamu.



Obrázek 5.10 Skupina položek hlavního menu nazvaná „Soubor“

Kliknutím na položku „Otevřít soubor“ se spustí metoda hlavního formuláře *TFormMain::OpenFileClick(TObject *Sender)*. Tato metoda nejdříve zobrazí standardní formulář operačního systému pro otevírání souboru.

Nynější verze programu StimuliEdit dovoluje otevírat pouze soubory jazyka VHDL. V dalších verzích programu, které by mohly podporovat i jiné jazyky HDL (viz kapitola 1.3), by se v tomto místě podle přípony souboru rozhodlo, jaká metoda bude použita pro otevření souboru. Tato verze programu spouští metodu hlavního formuláře *TFormMain::OpenFileVHDL(AnsiString FileName)* (viz kapitola 5.5.1), za jejíž parametr *FileName* je doplněno jméno otevíraného souboru. V případě, že otevírání souboru proběhlo úspěšně, jsou zobrazeny přečtené signály a uživatel může přistoupit k editování jejich průběhu. Jestliže otevírání souboru neproběhlo úspěšně, jsou ze seznamu odebrány případné signály a aplikace se ocitne ve stejném stavu, v jakém byla před otevíráním souboru.

5.5.1 Otevření návrhového souboru VHDL

Popis rutiny, kterou provádí metoda hlavního formuláře *TFormMain::OpenFileVHDL(AnsiString FileName),* je pro svou rozsáhlost umístěna do zvláštního souboru *OpenFileVHDL.cpp* (viz přiložené CD – adresář *\sources*). Soubor, jehož jméno je předáno metodě parametrem *FileName,* je nejprve načten do zápisníku hlavního formuláře *TFormMain::MemoIO,* což je objekt typu *TMemo,* který je definován ve standardních knihovnách prostředí Borland Builder C++. Tento objekt umožňuje pracovat s textovými soubory a provádět různé operace nad jejich textem.

Než uvedu popis rutiny pro otevírání souboru VHDL, je třeba se zmínit o tom, že tato rutina zároveň při načítání definic zapisuje některé informace do zápisníku ukládacího formuláře, což je objekt typu *TFormSave* (viz kapitola 5.6). Také je třeba zmínit, že jsem zvolil jednu rutinu pro otevírání jak „stimuli“ souborů, tak návrhových souborů obsahujících popis architektury obvodu. Rutina pro otevírání VHDL souboru postupuje podle následujícího scénáře:

1. Najdi platné klíčové slovo *entity.* Jestliže jsi ji našel, založ blok entity v zápisníku ukládacího formuláře a pokračuj, jinak ukonči proceduru.
2. Jestliže blok entity obsahuje definice generických konstant typu *time,* pro každou zobraz formulář pro přidání generické konstanty. (Rutina přečte i hodnotu generické konstanty a přednastaví všechny hodnoty formuláře. Formulář je zobrazen pro případnou potřebu změnit časovou mez, do které se generuje průběh „generického“ signálu.) Pokračuj.
3. Jestliže deklarace entity obsahuje platné klíčové slovo *port,* pak se pravděpodobně nejedná o „stimuli“ soubor, tudíž pokračuj, jinak pokračuj bodem 6.
4. Přečti definice všech deklarovaných signálů, následně založ blok architektury v zápisníku ukládacího formuláře a zapiš do bloku postupně jména všech signálů. Poté signály, které jsou v módu *IN, INOUT* nebo *LINKAGE* (viz kapitola 1.2.1) a jsou jedním z podporovaných typů (viz další odstavec), přidej je do seznamu signálů. Pokračuj.
5. Do zápisníku ukládacího formuláře přidej blok „port map“ (viz kapitola 1.2.4) a blok konfigurace (viz kapitola 1.2.8). Pokračuj bodem 8.
6. Pravděpodobně se jedná o „stimuli“ soubor, proto najdi blok architektury (viz kapitola 1.2.2), přečti definice všech signálů, založ blok architektury v zápisníku ukládacího formuláře a překopíruj do něj názvy signálů. Poté se pokus pro jednotlivé signály nalézt deklaraci plánovaného průběhu v těle architektury. Jestliže najdeš tuto deklaraci, přidej signál do seznamu hlavního formuláře a nastav jeho deklarovaný průběh. Nenajdeš-li deklarovaný průběh signálu, zřejmě se jedná o signál v módu *OUT* nebo *BUFFER*. Až projdeš všechny nalezené signály, pokračuj.
7. Do zápisníku ukládacího formuláře přidej blok „port map“ (viz kapitola 1.2.4) a blok konfigurace (viz kapitola 1.2.8). Pokračuj.
8. Poznamenej, že otevření souboru proběhlo v pořádku, a ukonči proceduru.

Ve třetím a šestém bodu je uvedeno slovo pravděpodobně, protože některé složitější konstrukce „stimuli souborů“ (zpravidla se jedná o úplné „stimuli“ soubory, viz kapitola 2.), mohou obsahovat i definice portů.

Čtvrtý bod představuje několik dalších omezení programu. První omezení je podpora proměnné pouze následujících typů: *SIGNED, UNSIGNED, BIT, STD_LOGIC, STD_ULOGIC, BIT_VECTOR, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR.* K tomuto se váže již dříve zmíněné omezení rozměru vektorů na 1-32 bitů. Další omezení oproti definici VHDL jazyka jsem se dopustil v tom, že rozsah vektoru musí být zadán pomocí numerických hodnot, nikoliv pomocí

obecného matematického výrazu, jak připouští jazyk VHDL.

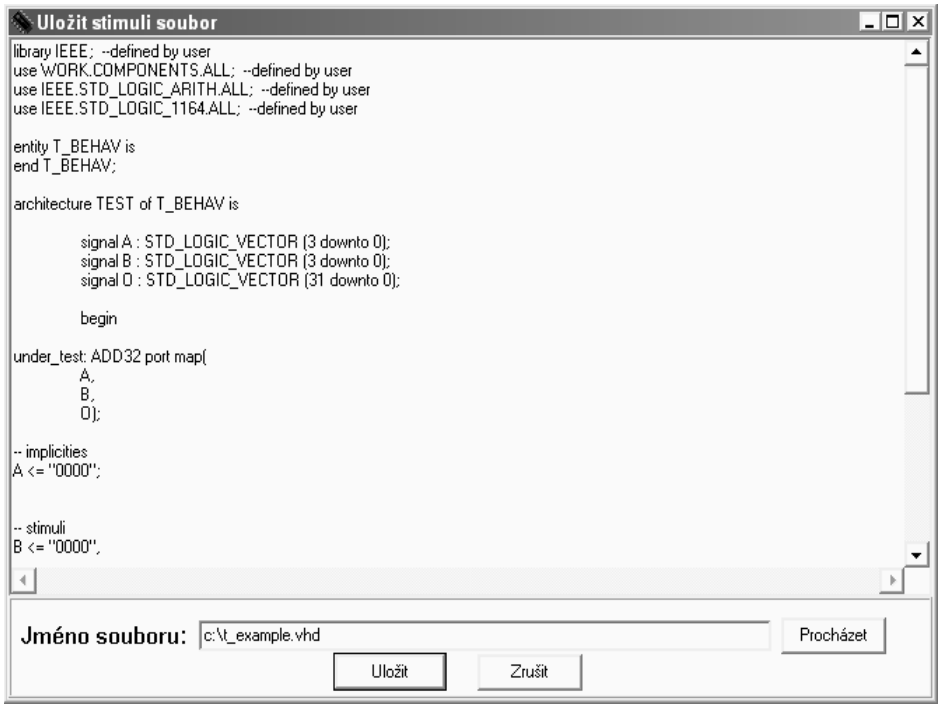
Další omezení vůči jazyku VHDL představuje šestý bod, kde může v definici signálu být uveden matematický výraz, na jehož hodnotu je signál inicializován (viz kapitola 1.2.3). Tuto možnost program zatím nepodporuje. Inicializace signálu musí být provedena v těle architektury numericky zadanou hodnotou. Deklarace plánovaného průběh signálu může obsahovat časové údaje zadané matematickým výrazem. Program podporuje pouze tři následující možnosti zadání časového údaje:

- numerická hodnota doprovázená jednotkou času (např. 100 ns) (program nepodporuje jednotku fs= 10^{-15} s),
- numerická hodnota následovaná znakem pro násobení ('*') a jménem generické konstanty typu *time* (například: *10*TCK_PER*),
- numerická hodnota následovaná znakem pro násobení ('*'), následuje jméno generické konstanty typu *time*, znaménko sčítání ('+') a jméno generické konstanty typu *time* následované dělením dvěma (například: *10*TCK_PER+TCK_PER/2*).

Rutina kontroluje syntaxi souboru jen na několika místech, což může vést k načtení souboru, ve kterém se vyskytují syntaktické chyby.

5.6 Uložení nastavených signálů do „stimuli“ souboru

Ukládací formulář, který je na obrázku 5.11, slouží k ukládání vytvořených průběhů. Definice tohoto formuláře je uložena v souborech *UnitSave.cpp* a *UnitSave.h* (viz přiložené CD – adresář *sources*\). Tento formulář obsahuje dva zápisníky – objekty typu *TMemo, TFormSave::Memo* a *TFormSave::MemoOut.* Zápisník *TFormSave::Memo* obsahuje po úspěšném otevření návrhového souboru připravený text „stimuli“ souboru (viz kapitola 5.5.1), vyjma deklarací plánovaných průběhů. Ukládací formulář je zobrazen metodou hlavního formuláře *TFormMain::SaveFileClick(TObject *Sender),* která je spuštěna v případě kliknutí na položku hlavního menu „Uložit soubor“. Tato metoda nejprve překopíruje připravený text „stimuli“ souboru ze zápisníku *TFormSave::Memo* do zápisníku *TFormSave::MemoOut.* Následně metoda doplní deklarace plánovaných průběhů signálů do zápisníku *TformSave::MemoOut* (v případě neměnného průběhu je deklarace přidána pod řádek s poznámku -- *implicities,* jinak pod řádek s poznámkou -- *stimuli*). Zápisník *TFormSave::Memo* s připraveným textem pro případné ukládání dalších „stimuli“ souborů je skrytý. Zobrazí se ukládací formulář (viz obrázek 5.11) se záznamníkem *TFormSave::MemoOut*, který uživateli umožňuje editovat výstupní text „stimuli“ souboru.



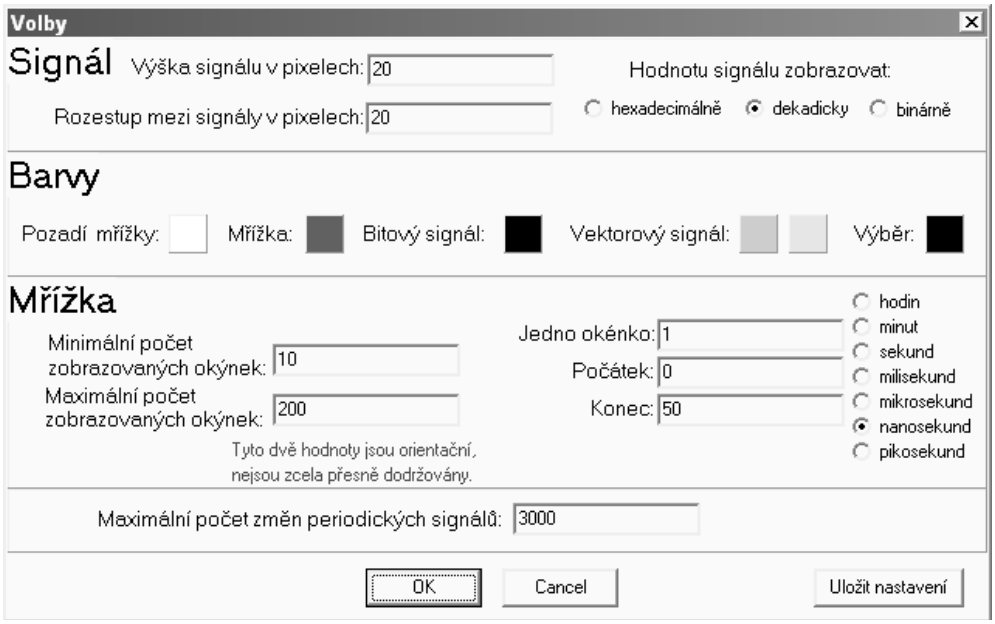
Obrázek 5.11 Ukládací formulář

Po kliknutí na tlačítko „Procházet“ se zobrazí standardní formulář operační systému pro výběr ukládaného souboru. Tlačítko „Uložit“ vytvoří „stimuli“ soubor požadovaného jména a uzavře ukládací formulář.

5.7 Nastavení a ovládání programu

Po spuštění se program StimuliEdit snaží najít soubor *StimuliEdit.conf s* konfigurací v adresáři, ze kterého je aplikace spuštěna. Nenajde-li program uvedený soubor, nastaví standardní hodnoty pro všechny položky formuláře nastavení programu. Následují nastavitelné parametry programu, které je možno uložit do souboru s konfigurací pomocí tlačítka „Uložit nastavení“ (viz obrázek 5.12):

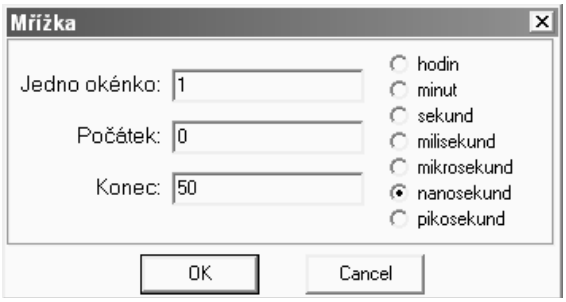
- výška signálu v pixelech, rozestup mezi signály v pixelech,
- číselná soustavy pro zobrazení hodnot signálu (binární, dekadická, hexadecimální),
- barva pozadí, mřížky a popisků signálů, bitově vykreslovaného signálu, dvě barvy vektorově zobrazeného signálu, masky výběru,
- minimální, maximální počet okének zobrazované mřížky,
- interval, ve kterém je mřížka zobrazena, a velikost jednoho okénka mřížky,
- maximální počet změn při nastavování periodických signálů (*TFormMain::MaxCountForPeriodic*).



Obrázek 5.12 Formulář pro nastavení programu

Definice formuláře pro nastavení je v souborech *UnitOptions.cpp* a *UnitOptions.h* (viz přiložené CD – adresář *\sources*).

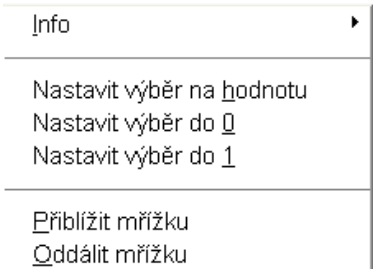
Parametry aktuálně zobrazované mřížky časové osy se dají přesně nastavit pomocí formuláře pro nastavení mřížky, jehož definice je v souborech *UnitGrid.cpp* a *UnitGrid.h* (viz přiložené CD – adresář *\sources*).



Obrázek 5.13 Formulář pro nastavení mřížky časové osy

Položka hlavního menu „Zobrazovat informace“ umožňuje zapnout/vypnout zobrazování informačních oken (tzv. *hints*). Je-li zapnuto zobrazování informací a klikne-li uživatel na průběh signálu, zobrazí se okno s informacemi o signálu. Hodnoty jsou odvozeny od pozice kurzoru myši na zobrazovaném průběhu signálu.

Podobné informace, jaké zobrazují informační okna „hint“, je možné zobrazit v menu, které se objeví po kliknutí pravého tlačítka myši na zobrazený průběh signálu (viz obrázek 5.14). Položky tohoto menu jsou stejné (vyjma *info* položky) jako položky hlavního menu ve skupině „Průběhy“.



Obrázek 5.14 „Vysunovací“ menu průběhu signálu

Seznam signálů má také „vysunovací“ menu, jehož položky jsou stejné jako položky hlavního menu ve skupině „Signály“. Položky se vztahují k označenému signálu v seznamu signálů.

Obrázek 5.15 „Vysunovací“ menu seznamu signálů

Závěr

Hlavním cílem této práce bylo navrhnout a realizovat program, který by umožňoval za pomoci grafických nástrojů sestavování „stimuli“ souborů jazyka VHDL. Program StimuliEdit jsem navrhnul na základě analýzy programů, které se dnes používají pro práci se „stimuli“ soubory (viz kapitola 3). Program umožňuje načíst definice signálů z návrhového souboru jazyka VHDL, a poté zobrazené průběhy signálů editovat za pomoci myši. Nastavené průběhy je možné uložit do „stimuli“ souborů, které se následně používají pro testování správné funkčnosti číslicových obvodů. Program umožňuje uložené průběhy znovu načíst a dále upravovat.

Pátá kapitola je věnována návrhu programu StimuliEdit. Podkapitoly jsou seřazeny chronologicky, jakým způsobem jsem postupoval při návrhu programu. Zde jsou také uvedena omezení programu a návrh, jakým způsobem by bylo množné tato omezení v dalších verzích řešit.

Program StimuliEdit je napsán v programovacím jazyce C++ a přeložen pro operační systémy Windows, které používají 32-bitovou verzi aplikačního programového rozhraní. Vygenerované „stimuli“ soubory mají správnou syntaxi, funkčnost jsem ověřil v programu *FPGA Advantage for HDL Design v 2004.1* od firmy Mentor.

Kompletní zdrojové kódy k programu StimuliEdit jsou na přiloženém CD, adresář *\sources*. Přeložený binární kód programu je v hlavním adresáři na přiloženém CD (jméno souboru *StimuliEdit.exe*)

Literatura

[1] IEEE std 1076-1987: *IEEE Standard VHDL Language Reference Manual*, 1987.

[2] IEEE Std 1076-1993: *IEEE Standard VHDL Language Reference Manual*, 1993.

[3] IEEE Std 1164-1993: *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164),* 1993.

[4] IEEE Std 1076.3-1997: *IEEE Standard VHDL Synthesis Packages,* 1997.

[5] Ashenden Peter J. : *The VHDL Cookbook.* First Edition, 1990.
http://tech-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf

[6] Synario Design Automation: *VHDL Reference Manual*, 1997.
http://soc.eurecom.fr/EDC/vhdlref.pdf

[7] Perry Douglas L. : *VHDL Programming by Example Fourth Edition*, 2002.
http://www.fpga.com.cn/hdl/training/McGraw.Hill.VHDL.Programming.by.Example.4th.Ed.zip

[8] Musil V., Kolouch J., Prokop R.:
Návrh digitálních integrovaných obvodů a jazyk VHDL, Brno, 2002.

[9] Kolář M.: *Přednášky k předmětu NHK,* Liberec, 2002.

[10] *ABEL-HDL Reference Manual,* Version 8.0, 1999.
http://www.ece.iit.edu/~jstine/ece446/handouts/abel_ref.pdf

[11] Spiegel J.V.: *ABEL-HDL Primer,* University of Pennsylvania, 1999.
http://www.seas.upenn.edu/ese/rca/software/abel/abel.primer.html

[12] Cypress Semiconductor Corporation: *Abel™-HDL vs. IEEE-1076 VHDL,*1997.
(Pro stažení následujícího dokumentu je potřeba se bezplatně na serveru zaregistrovat.)
http://www.eetasia.com/ART_8800080112_480100_27e2dfe5.HTM

[13] EDA Industry Working Groups web pages.
http://www.eda.org/

[14] Verilog Resources web pages.
http://www.verilog.com/
http://www.verilog.net/

[15] Doulos Ltd.: *The Verilog Golden Reference guide,* 1999.

[16] Working Group Areas web pages of STIL.
http://grouper.ieee.org/groups/1450/
http://grouper.ieee.org/groups/1450/Flyer_frame.html

[17] IEEE Std 1450-1999: *IEEE Standard Test InterfaceLanguage (STIL) for Digital Test Vector Data,* 1999.

[19] Lattice Semicon. Corp. : *ABEL Design Manual,* Version 8.0,1999.
http://www.leonardo.caltech.edu/~ee5x/eecs52/general/referenc/abeldes.pdf

[20] Atmel Corporation: *Tips on Using Test Vectors for Atmel PLDs,* 1999.
http://www.atmel.com/dyn/resources/prod_documents/DOC0479.PDF

[21] Doulos Ltd. Web pages.
http://www.doulos.com/knowhow/

[22] Altera Corporation: MAX+PLUS II manual, 1997.
http://www.altera.com/literature/manual/81_gs.pdf

[23] SynaptiCAD corp.: *SynaptiCAD on-line help.*
http://www.synapticad.com

[24] Lukasz Strozek: *Xilinx Tutorial – Software Guide,* 2004.
http://www.eecs.harvard.edu/cs141/resources/xilinx-software.pdf

[25] Lattice semiconductor corp.: ispDesignExpert User Manual, ver. 8.0.
http://t-des.web.cern.ch/IT-DES/ETS/electronical/dc/FPGA/Pld_doc/ispde80um.pdf

[26] Glauert, Wolfram H. : VHDL Tutorial.
www.vhdl-online.de

[27] Přívětivý, Pavel : Učebnice programování C++, 2001.
http://www.sfiles.host.sk/builder/index.html