České vysoké učení technické v Praze
Fakulta elektrotechnická

Bakalářská práce

Vstupní a výstupní konverze VHDL

Martin Halamíček

Vedoucí práce: Ing. Petr Fišer

Studijní program: Elektrotechnika a informatika strukturovaný bakalářský

Obor: Informatika a výpočetní technika

Červen 2006

Poděkování

Tímto bych chtěl poděkovat mému vedoucímu Ing. Petrovi Fišerovi za vstřícný přístup a rady k této práci. Také bych chtěl poděkovat Ing. Janu Schmidtovi, Ph.D za poskytnutí dokumentace k EDA systému.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 16.6. 2006 .....

Abstract

The objective of this project is to implement the input and output conversion of a subset of the VHDL language into the current EDA (Electronic design automation) system. This project consists of an analysis part, describing the selected subset of VHDL language to be implemented, and the description of the very implementation into the structure of the EDA system. Further the project contains a part describing the EDA system, implementation and testing of implemented modules.

Anotace

Cílem této práce bylo doprogramovat do stávajícího EDA (Electronic design automation) systému vstupní a výstupní konverzi podmnožiny jazyka VHDL. Tato práce obsahuje část analýzy ve které je stanovena podmonožina jazyka která bude implementována a navržen způsob jakým budou jednotlivé konstrukce jazyka VHDL převedeny do struktury EDA systému. Dále práce obsahuje popis EDA systému a popis implementace a testování naimplementovaných modulů.

Obsah

Seznam obrázků **xiii**

Seznam tabulek **xv**

1 Úvod **1**

2 Popis HUBu **2**
2.1 Úvod . . . 2
2.2 Popis obecných tříd . . . 2
2.2.1 Pojmenované jednotky . . . 2
2.2.2 Jmenný prostor . . . 2
2.2.3 Atributy . . . 3
2.2.4 Výjimky . . . 3
2.3 Vrchní vrtva . . . 4
2.3.1 Knihovna . . . 4
2.3.2 Návrh . . . 4
2.3.3 Entita . . . 4
2.3.4 Prototyp rozhraní portů . . . 6
2.3.5 Prototyp portu . . . 6
2.3.6 Architektura . . . 6
2.3.7 Určení místa v hubu . . . 6
2.4 Strukturní popis . . . 6
2.4.1 Spoj . . . 7
2.4.2 Instance a rozhraní instance . . . 7
2.4.3 Port instance a port strukturní architektury . . . 7
2.5 Popis chováním (Behaviorální) . . . 9
2.5.1 Plně Symetrická funkce . . . 9
2.5.2 Klopný obvod RTL . . . 9
2.5.3 PLA popis . . . 9
2.5.4 Konečný automat FSM . . . 9
2.6 Importní subsystém . . . 10
2.7 Exportní subsystém . . . 10
2.7.1 DBG výstup . . . 10

3 Popis jazyka VHDL **12**
3.1 Úvod . . . 12
3.2 Objekty . . . 12
3.3 Atributy . . . 13
3.4 Rozhraní . . . 13
3.5 Program . . . 13
3.6 Deklarace Entity . . . 13
3.7 Deklarace architektury . . . 14
3.7.1 Paralelní prostředí . . . 14
3.7.2 Sekvenční prostředí (popis chování) . . . 15
3.8 Deklarace a tělo Knihovní jednotky (package) . . . 16

4 Implementovaná podmnožina jazyka VHDL **17**
4.1 Tělo programu . . . 17

4.2 Deklarace Entity . . . 17
4.3 Deklarace Architektury . . . 18
4.4 Deklarace rozhraní (signálu) . . . 18
4.5 Deklarace komponenty . . . 19
4.6 Paralelní příkaz přiřazení signálu . . . 19
4.7 Příkaz instalace komponenty . . . 19
4.8 Specifikace jména . . . 21
4.9 Výrazy . . . 21

5 Generátor syntaktických analyzátorů Bison 22
5.1 Formát vstupního souboru analyzátoru . . . 22
5.2 Syntaxe pravidel gramatiky . . . 22
5.3 Datové typy sémantických hodnot symbolů . . . 23
5.4 Přístup k sémantickým hodnotám symbolů . . . 23
5.5 Rozhraní parseru . . . 23
5.5.1 Rozhraní v jazyce C . . . 23
5.5.2 Rozhraní v jazyce C++ . . . 24
5.6 Spuštění Bisonu . . . 24

6 Analýza 25
6.1 Úvod . . . 25
6.2 Import . . . 25
6.2.1 Syntaktická analýza . . . 25
6.2.2 Převod konstrukcí VHDL do HUBu . . . 25
6.2.3 Porty entity . . . 25
6.2.4 Architektury . . . 26
6.2.5 Komponenty . . . 26
6.2.6 Deklarace signálu . . . 26
6.2.7 Příkazová část architektury . . . 26
6.3 Export . . . 27
6.3.1 Výpis knihoven . . . 27
6.3.2 Popis architektury strukturou . . . 27

7 Popis Implementace Importního Systému 29
7.1 Lexikální analyzátor . . . 29
7.2 Parser . . . 29
7.2.1 Vstupní soubor programu Bison . . . 30
7.3 Přidané třídy výjimek . . . 30
7.4 Importní modul . . . 31
7.5 Import jednotlivých konstrukcí VHDL . . . 32
7.5.1 Entita . . . 32
7.5.2 Architektura . . . 33
7.5.3 Deklarace Signálu . . . 33
7.5.4 Deklarace Komponenty . . . 34
7.5.5 Příkaz přiřazení signálu . . . 34
7.5.6 Příkaz instalace komponenty . . . 36

8 Popis Implementace Exportního Systému 37
8.1 Úvod . . . 37
8.2 Traverzery . . . 37
8.2.1 Traverzer vrchní vrstvy HUBu . . . 37

8.2.2 Traverzer strukturní architektury 38
8.3 Export sémanticých konstrukcí jazyka VHDL 39
8.3.1 Popis architektury strukturou 39
8.3.2 Popis architektury chováním 40

9 Testování **41**
9.1 Testovací aplikace 41
9.2 Import 41
9.3 Export 42

10 Závěr **43**

11 Seznam literatury **45**

A Import VHDL **47**
A.1 Vstupní soubor 47
A.1.1 Poloviční sčítačka - halfadder.vhdl 47
A.1.2 1-bitová úplná sčítačka - adder.vhdl 47
A.1.3 4-bitová sčítačka - 4bitAdder.vhdl 48
A.2 Výstupní soubor 49
A.2.1 Dbg výstup 49

B Export do VHDL **53**
B.1 Struktura uložená v HUBu 53
B.2 Výstup ve formátu VHDL 55
B.2.1 Knihovna work 55
B.2.2 Knihovna bench_primitives 56

C Obsah přiloženého CD **59**

Seznam obrázků

2.1 HUB - Vrchní vrstva 5
2.2 HUB - Strukturní popis 8

C.1 Obsah přiloženého CD 59

Seznam tabulek

4.1 Soubor VHDL 17
4.2 Deklarace Entity 17
4.3 Deklarace Architektury 18
4.4 Deklarace rozhraní 18
4.5 Deklarace rozhraní 19
4.6 Specifikace jména 21
4.7 Primární výraz 21

1 Úvod

Cílem této práce bylo doprogramovat do stávajícího EDA (Electronic design automation) systému vstupní a výstupní konverzi podmnožiny jazyka VHDL. Tento EDA systém je napsán v jazyce C++ a je tvořen třemi základními částmi jádrem HUB, importním a exportním podsystémem. Importní a exportní podsystémy obsahují moduly pro import a export různých formátů pro specifikaci číslicových obvodů a definují obecné rozhraní které bude mít každý z těchto modulů aby mohl pracovat s jádrem systému. Tato práce se tedy skládá ze dvou nezávislých částí a to importního a exportního modulu.
Před započetím implemantace je třeba stanovit jakým způsobem budou jednotlivé konstrukce v jazyce VHDL konvertovány do struktury HUBu. Tímto se zabývá část analýzy, která zároveň stanovuje které konstrukce jazyka VHDL budou podporovány. Požadavkem je aby bylo možné importovat strukturně navržené obvody v jazyce VHDL. Bude třeba vybrat takovou podmnožinu jazyka aby tohoto požadavku bylo docíleno.
Pro exportní část bude naopak potřeba stanovit jakým způsobem exportovat strukturu uloženou v HUBu do jazyka VHDL. Výstup musí být nezávislý na tom jakým způsobem byla struktura do HUBu naimportována a výstupní kód musí být VHDL validní a funkčně musí návrh obvodu odpovídat původnímu návrhu.
Po dohodě s vedoucím byl k této práci přidán popis HUBu. K tomuto popisu jsem vyžíval kromě zdrojových kódů HUBu také jeho dokumentaci [5].

2 Popis HUBu

2.1 Úvod

Hub představuje základní součást výukového **EDA**[1] systému , jedná se o datovou strukturu sloužící k popisu číslicových obvodů chováním a strukturou na úrovni hradel.
Tato struktura byla inspirována jazykem **VHDL**. Centrálním pojmem popisu je tedy Entita, ta představuje abstrakci více stejných obvodů. Entita představuje blok jehož "vnitřek" je zvenčí skrytý a poskytuje pouze rozhraní vstupních, výstupních popř. vstupně výstupních portů tedy svoje rozhraní pomocí kterého komunikuje s ostatními obvody. Entita může být popsána bud chováním (behaviorální) nebo strukturou. Entita která nemá žádný popis se nazývá Externí a má popsáno tedy pouze svoje rozhraní. Každá entita může mít několik Architektur ty představují alternativní popisy "vnitřku" konkrétní entity.
Strukturní popis entit je realizován pomocí instancí entit a spojů. Tento popis je blízký klasickému propojování jednotlivých hradel a integrovaných obvodů na tištěném spoji. Rozhraní entity si lze představit jako konektory na desce, vodivé cesty jsou spoje a jednotlivé součástky představují instance. Instance je vytvořena z entity a představuje konkrétní obvod ve struktuře. Z jedné entity může být v jedné architektuře vytvořen libovolný počet instancí. Entity z kterých byli vytvořeny instance v architektuře můžou být samozřejmě také popsány strukturou, návrh je tedy hierarchický. Spoj představuje množinu elektricky spojených vývodů (portů instancí nebo portů rozhraní).
Hub je napsán v jazyce C++ a je tvořen třemi základními částmi. Jádrem které představuje organizované úložiště dat, importním systém obsahujícím komponenty pro import různých formátů a exportním systémem sloužícím k exportu do různých formátů popisu.
Jádro se dělí na tři části. Části které obsahují třídy pro strukturní a behaviorální popis Entit a část která se stará a organizaci entit tzv. vrchní vrstva.
Každou jednotku jako je (knihovna,entita,prototyp portu, architektura , . . .) tvoří jedna třída. Tyto jednotky mají různé společné vlastnosti např. jsou pojmenované a můžou vlastnit nějaké další jednotky nebo definovat nějaké vlastnosti.
Pro tyto potřeby obsahuje HUB třídy které definují tyto a další vlastnosti, které jednotky HUBu využívají:

2.2 Popis obecných tříd

2.2.1 Pojmenované jednotky

Každá třída která představuje pojmenovanou jednotku je potomkem třídy `NameHaving`.
Tato třída v sobě obsahuje objekt třídy `Name` ve kterém je uloženo jméno. Je potomkem abstraktní třídy `Named` která definuje rozhraní pro vše co má jméno, tedy metodu:

```
virtual const Name& name (void) const=0;
```

Tato metoda vrací jméno každé pojmenované jednotky.
Jméno je realizováno třídou Name , která je potomkem knihovní třídy string. Poskytuje metody cons pro jmenné konvence a skládání jmen. Také poskytuje metodu `Augment` sloužící k přidání číselného indexu na konec jména a metodu nextAugment zvětšující tento index o jedna.

2.2.2 Jmenný prostor

Slouží jako úložiště objektů, umožnuje je vytvářet a vyhledávat. Např. knihovna vlastní jmenný prostor entit. Objekty jsou identifikovány svým jménem (objekt třídy `Name`), jmenný prostor

[1] EDA = Electronic Design Automation

není citlivý na velikost písma.
Je realizován šablonou `Namespace` jejímž parametrem je typ objektů jež bude jmenný prostor obsahovat. Je potomkem standardní knihovní třídy `map< Name, Thing*, CaseComp>`. Jméno `Name` představuje index v mapě a `CaseComp` je funktor sloužící k porovnávání při vyhledávání v mapě, `Thing` je parametr šablony. Namespace poskytuje metodu `Find` pro vyhledávání objektů podle jejich jména. Také poskytuje metody `FindOrCreate` a `Create` pro vytváření objektů. Metoda `Create` před vytvořením objektu kontroluje zda již stejně pojmenovaný objekt ve jmenném prostoru neexistuje, pokud existuje vrátí `NULL`. Pro vkládání a vyhledávání objektů jež jsou potomky objektů třídy ve jmenném prostoru obsažených , poskytuje šablony metod pro vytváření a vyhledávání, které vracejí přímo ukazatel na typ potomka a není tedy potřeba používat `dynamic_cast`. Poskytuje také metodu `CreateUnique`, která slouží k vytvoření objektu s unikátním jménem tzn. pokud objekt se zadaným jménem ve jmenném prostoru již existuje přidá ke jménu číselný index.

2.2.3 Atributy

Množina atributů Tato množina se využívá pro nastavení vlastností (flagů) u některých jednotek struktury HUBu. Např. u prototypů portů se pomocí množiny atributů nastavuje mód portu.
Množina obsahuje binární atributy, každý z nich může být buď známý nebo neznámý. Pokud je známý má hodnotu true nebo false.
Je realizována šablonou `AttrSet`.

```
template <typename Key, int N> class AttrSet
```

Parametr šablony `Key` odlišuje proměnné různých typů, které pak nelze vzájemně přiřadit. Je možno použít libovolný typ, ale zpravidla se používá třída, jejímž členem je indexovaná množina. Parametr N udává počet bitů množiny. Vektor může být indexován objekty třídy `AttrIndex<Key>`. Množiny atributů je možno spojovat pomocí přetížených operítorů +, += . Poskytuje také metodu `isSubset` která zjišťuje zda se jedná o podmnožinu.

Index množiny atributů Typ proměnné je realizován šablonou `AttrIndex<Key>`. Objekty této třídy nesou celočíselnou hodnotu a jsou vybaveny základními metodami:
srovnání: `operator == ()` a `operator < ()` , přiřazení: `operator = ()` a kopírovací konstruktor.
Parametr šablony **Key** odlišuje proměnné různých typů, které pak nelze vzájemně přiřadit. Je možno použít libovolný typ, ale zpravidla se používá třída, jejímž členem je indexovaný vektor.
Třída `AttrUnique` dědí `AttrIndex` a realizuje proměnnou, která při inicializaci nabyde hodnoty, unikátní mezi všemi objekty této třídy. P o zániku objektu je hodnota volná a může být (nikoli nutně musí) být přidělena jinému objektu. Jestliže přiřazení takto inicializované proměnné je nežádoucí (má to být konstanta), stačí ji deklarovat jako const.
Třída `StatAttrUnique` je shodná s `AttrUnique` až na to, že je určena k inicializaci v čase inicializace modulu, to jest jako statická proměnná modulu. Objekty této třídy nemohou být automatickými nebo dynamickými proměnnými.

2.2.4 Výjimky

Využívají se pro ohlášení neočekávané události při práci s HUBem, např. syntaktická chyba v importovaném souboru nebo nenalezená architektura entity.
Každá výjimka má důvod (I/O, nekonzistence struktur, syntax,. . .), místo (v souboru, v bloku, . . .), kontext, systémovou kategorii . Třídy reprezentující tyto čtyři komponenty jsou parametry šablony `CException`. Místo a kategorie mají implicitní prázdné třídy. Každá třída výjimky z

této šablony odvozená je potomkem tříd systémové kategorie a důvodu. Podobjekt výjimky odpovídající systémové kategorii je inicializovány tak, aby metoda `what()` dala kompletní popis výjimky.
Jednotlivé komponenty výjimky se musí inicializovat pomocnými objekty, tedy např.

```
throw IOException ( TokenReason (currentToken, expected), FilePosition (filename, lineno))
```

Každá nově definovaná třída důvodu, pozice nebo kontextu musí mít:

- konstruktor, který bere potřebné hodnoty; jestliže tento konstruktor bere (void), musí brát i nevýznamný parametr např. typu int (jinak překladač nerozpozná, že se konstruuje objekt)
- kopírovací konstruktor
- funkci std::string tell (void), která dodá čitelný popis obsahu objektu.

2.3 Vrchní vrtva

Kořen celé datové struktury představuje objekt třídy `Hub` , je pravděpodobné, že objekt této třídy bude v každé aplikaci vždy jeden, ale není to nutné. `Hub` vlastní ve svých jmenných prostorech knihovny a návrhy. Poskytuje funkce pro vytváření a vyhledávání knihoven a návrhů. Poskytuje iterátory pro procházení prostotu knihoven a návrhů. Třída `Hub` je potomkem třídy `NameHaving`. Objekt třídy `Hub` má tedy jméno které může být měněno metodou rodičovské třídy. Struktura vrchnní vrsty HUBu je znázorněna na obrázku 2.1

2.3.1 Knihovna

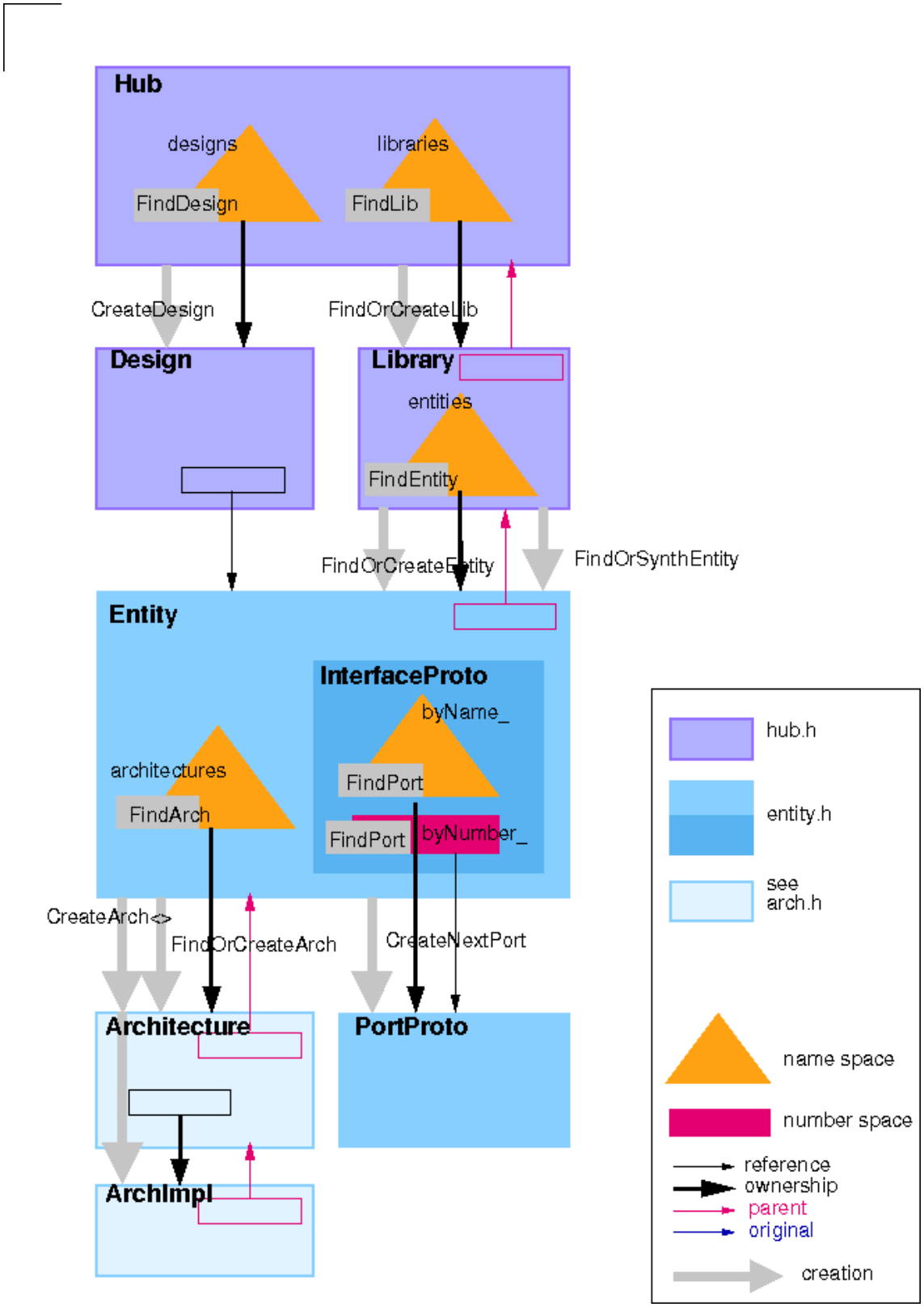
Je pojmenována , je reprezentována třídou `Library`. Obsahuje jmenný prostor `Entit`. Umožňuje vytvářet entity a vyhledávat je podle jména nebo podle funkce. Poskytuje také iterátory pro procházení jmenného prostoru entit. Umožňuje také vytvoření unikátní entity.

2.3.2 Návrh

Je to pojmenovaný odkaz na entitu. Je reprezentován třídou `Design`.

2.3.3 Entita

Je cenrálním pojmem popisu, pojmenovaná, reprezentována třídou `Entity`. Reprezentuje typ bloku. Entita obsahuje informace o rozhraní portů InterfaceProto a jmenný prostor architektur. Architektury jsou alternativní popisy "vnitřku" entity které mohou být popsány strukturou nebo chováním. Umožňuje vytvářet a vyhledávat objekty třídy Architecture i s příslušnými objekty implementace architektury (potomci třídy `ArchImpl`). Obsahuje také metody pro manipulaci s rozhraním, vyhledávání a vytváření prototypů portů. Poskytuje iterátory pro procházení architektur a portů rozhraní.
Obsahuje také metodu pro instanciaci entity, která slouží k vytvoření instance této entity v nějaké strukturně popsané architektuře. Jejími parametry jsou jméno instance, ukazatel na strukturní architekturu ve které je instance vytvářena a architekturu se kterou bude instance vytvořena tzn. některá z architektur jež má entita ve svém jmenném prostoru.



Obrázek 2.1: HUB - Vrchní vrstva
Autorem tohoto diagramu je Ing. Jan Schmidt, Ph.D.

2.3.4 Prototyp rozhraní portů

Je reprezentován třídou `InterfaceProto` která obsahuje jmenný a číselný prostor prototypů portů. Poskytuje metody pro vyhledávání prototypů portů dle různých kritérií.

2.3.5 Prototyp portu

Prototyp portu je reprezentován třídou `PortProto`. Má jméno i číslo Jeho další hlavní hlavní funkcí je nést otevřenou množinu binárních atributů prototypu portu. Třída PortProto parametrizuje šablony `AttrIndex` a `AttrSet` a exportuje jejich typy pod jednoduššími identifikátory. Třída sama definuje dvě vlastnosti - vstup (input) a výstup (output). Pro snazší nastavení a srovnání definuje množiny, které odpovídají všem kombinacím těchto binárních atributů:

```
AttrUnique <PortProto> PortProto :: input;
AttrUnique <PortProto> PortProto :: output;
  //predefined attribute sets
AttrSet <PortProto, 32> PortProto :: inDir (input, 1, output, 0);
AttrSet <PortProto, 32> PortProto :: outDir (input, 0, output, 1);
AttrSet <PortProto, 32> PortProto :: inOutDir (input, 1, output, 1);
AttrSet <PortProto, 32> PortProto :: noDir; // direction unknown
```

2.3.6 Architektura

Architektura má jméno a je reprezentována třídou `Architecture`. Informace, která závisí na typu popisu, je obsažena v implementačních objektech. Rozhraní těchto objektů tvoří abstraktní třída `ArchImpl`. Obsahuje ukazatel na objekt třídy `ArchImpl` a poskytuje šablonu metody `Create` která slouží k vytvoření implementačního objektu architektury. Parametrem šablony je typ objektu (potomek třídy `ArchImpl`) který bude vytvořen. Potomky třídy `ArchImpl` jsou třídy `StructArch` která představuje strukturní popis a třída `BehavArch` která představuje popis chováním.

2.3.7 Určení místa v hubu

Importovaný popis obvodu popisuje různé objekty, od strukturní architektury entity (např. Bench) po kompletní strukturu knihoven a návrhů (EDIF). Pro operace importu, exportu a rušení je zapotřebí ukázat na knihovnu, entitu, architekturu podle okolností. Takovým sjednoceným odkazem je objekt třídy Location, který se dá zkonstruovat z posloupnosti jmen knihovny, entity, architektury. Tato jména jsou shrnuta v objektu třídy `LocationNames`.

2.4 Strukturní popis

Strukturně popsanou architekturu reprezentuje třída `StructArch` která je potomkem třídy `ArchImpl`, představuje implementační objekt architektury.
Strukturní popis architektury se skládá z instancí a spojů, které propojují porty instancí mezi sebou a s prototypy portů rozhraní popisované entity. Implementace hubu realizuje poslední vlastnost tak, že architektura má vlastní rozhraní s porty (nikoli prototypy portů), které slouží jako zástupná místa rozhraní entity. Třídu `StructArch` si lze představit jako desku na které jsou připojeny vstupní a výstupní porty a pomocí integrovaných obvodů, hradel a vodičů na ní propojíme obvod který bude realizovat funkci kterou po entitě požadujeme.
Třída `StructArch` vlastní jmenný prostot instancí a spojů v této architektuře obsažených, také obsahuje Interface portů, které představuje rozhraní portů popisované entity. Rozhraní

je realizováno šablonou Interface která je číselným prostorem portů. Parametrem této šablony typ vlastnického objektu v tomto případě je to třída StructArch porty v ní obsažené jsou tedy typu `Port<StructArch>`.
Poskytuje iterátory pro průchod instancemi, spoji a porty rozhraní. Poskytuje metody pro vytváření a vyhledávání portů, instancí a spojů. Strukturní popis architektury je znázorněn na obrázku 2.2

2.4.1 Spoj

Reprezentuje skupinu vzájemně propojených portů. Spoj je pojmenovaná množina odkazů na porty, reprezentovaná třídou `Net`. Množina je implementována seznamem a třída exportuje jeho iterátor. Umožňuje k sobě připojovat porty (objekty typu `Joinable`) metodou `Join`.

2.4.2 Instance a rozhraní instance

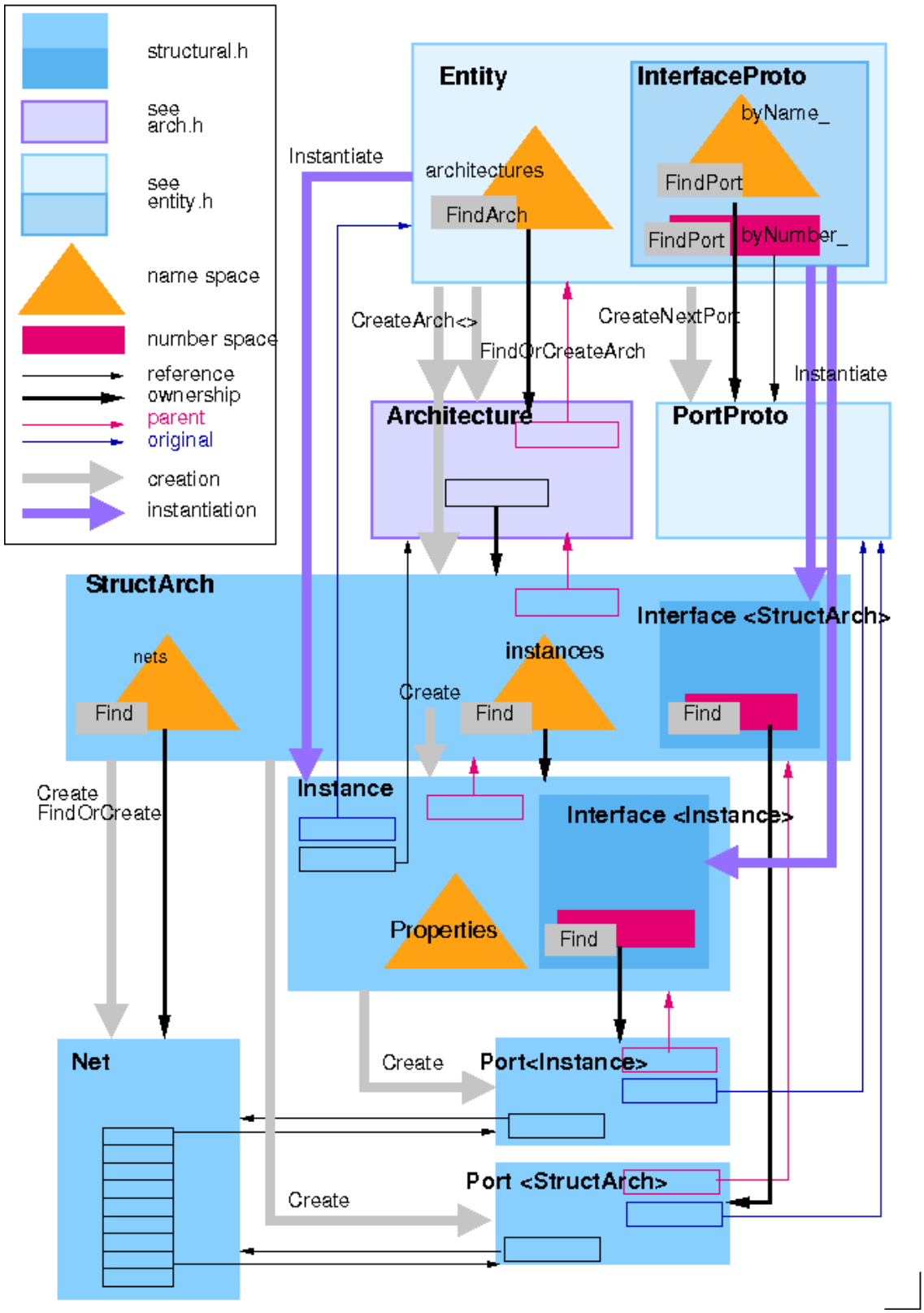
Instance má jméno, je reprezentována třidou `Instance`. Nese následující hlavní informace:

- číselný prostor portů instance (třída `Interface`)
- odkaz na aktuálně nakonfigurovanou architekturu
- jmenný prostor dvojic jméno-hodnota, které zobrazují vlastnosti z EDIFu (třída Property) - provizorium

Obsahuje metody pro hledání portu podle čísla a jména. Obsahuje odkaz na entitu z které byla tato instance vytvořena. Pokud je instance vytvořena přímo z nějaké entity její metodou Instantiate má již vytvořené rozhraní portů instance ke kterému lze připojovat spoje. Může mít přiřazenu vždy pouze jednu architekturu.

2.4.3 Port instance a port strukturní architektury

Oba tyto druhy portů jsou v podstatě jen odkazy na svůj prototyp a veškeré metody na něj delegují, včetně operací nad množinou binárních atributů. Liší se pouze v typu odkazu na vlastnický objekt a proto jsou implementovány šablonou `Port`. Protože spoj odkazuje na objekty různých tříd, dědí všechny instance šablony třídu `Joinable`. Navíc poskytuje metodu `connect(Net* net)` která slouží k připojení spoje k portu. Také poskytuje metodu `connected()` která vrací odkaz na spoj ke kterému je port připojený.



Obrázek 2.2: HUB - Strukturní popis
Autorem tohoto diagramu je Ing. Jan Schmidt, Ph.D.

2.5 Popis chováním (Behaviorální)

Třída `BehavArch` je potomkem třídy `ArchImpl`, říká o architektuře že je popsána nějakým behaviorálním popisem. Na rozdíl od implementace neobsahuje tato třída již přímo informace o realizaci architektury. Protože popisů chování může být více druhů (např. sym funkce, PLA, FSM,. . .) je použit idiom pimpl (pointer to implementation) a třída `BehavArch` obsahuje pouze odkaz na konkrétní popis jehož rozhraní tvoří abstraktní třída `BehavDescr`. `BehavArch` poskytuje šablonu metody pro vytváření popisů jež jsou potomky `BehavDescr`.
Třida `BehavDescr` je abstraktní a definuje pouze metodu která vrací jméno konkrétního popisu. Pro vytváření a porovnávání behaviorálních popisů se používá agent realizovaný třídou `BehavAgent`. Rovněž se jedná o abstraktní definující metodu `Eq` která slouží k porovnávání ekvivalence popisů a metodu CreateDescr která vytvoří konkrétní popis. Pomocí agenta lze např. v knihovnách vyhledávat architektury podle funkce. Pomocí toho lze zabránit tomu aby se v knihovně vyskytovala entita se shodným popisem vícekrát.
Každý nově přidaný popis chování musí tvořit tedy potomek třídy `BehavDescr` který bude obsahovat konkrétní popis a svého agenta pro porovnávání a vytváření popisů.
Zde je popis v současné době implementovaných popisů chování v HUBu:

2.5.1 Plně Symetrická funkce

SymFunc
Reprezentuje kombinační plně symetrické funkce s jedním výstupem. Obsahuje informaci o typu fukce a počtu vstupů.
Typ funkce je definovaný výčtovým typem obsahujícím:

- Funkce bez vstupů: 0, 1
- Funkce s jednim vstupem: Negace, Buffer
- Funkce se dvěma a více vstupy: And, Xor, Or, Nor , Xnor, Nand

SymFuncAgent
Agent pro vytváření logických popisů funkcá a porovnávání ekvivalence symetrických funkcí. Obsahuje v sobě objekt třídy SymFunc reprezentující symetrickou funkci s kterým porovnává a umožňuje vytvořit jeho kopii. Konstruktor má 2 parametry, prvním je typ funkce, druhým počet vstupů.

2.5.2 Klopný obvod RTL

RTL[2] popis klopného obvodu, neobsahuje hodiny a reset. Realizováno třídou RTLReg s příslušným agentem RTLRegAgent.

2.5.3 PLA popis

Není implementováno (implementováno jen zčásti)

2.5.4 Konečný automat FSM

Není implementováno.

[2]RTL = Register Transfer Level

2.6 Importní subsystém

Slouží k importu různých vstupních formátu pro specifikaci popisu číslicových obvodů, v současné verzi je implementován import z formátů: EDIF, BENCH , VHDL (součást této práce).
Rozhraní importního systému tvoří třída Import která je potomkem třídy `Location`, obsahuje obecné parametry pro import.
Obsahuje jméno souboru z kterého bude probíhat import a objekt třídy `LocationNames`, který nese jména aktuálně nastaveného prostředí v `Location`. Parametrem konstruktoru je jméno importovaného souboru a specifikace místa v hubu tzn. Určení zda bude specifikována instance, entita, nebo architektura. Obsahuje také přetížený operátor volání funkce, kterým se import spouští.
Pro každý importovaný formát je třeba vytvořit třídu která se stará o import daného formátu a dědí třídu `Import`. Tato třída zpravidla obsahuje objekt realizující parser (syntaktický analyzátor) a definice sémantických metod které jsou volány parserem. Zároveň je také potomkem třídy která představuje klienta parseru a jsou v ní deklarované sémantické metody.

2.7 Exportní subsystém

Slouží k exportu struktury nebo jejich částí do specifikovaného výstupního formátu. Umožňuje export specifikované architektury, entity, knihovny nebo obsahu všech knihoven. Rozhraní exportního systému tvoří třída `Export`. Obsahuje v sobě výstupní stream souboru do které ho se bude výstup generovat. Má přetížen operátor volání funkce kterým se import spouští. Jako parametr může být specifikováno zda se exportuje obsah celého hubu, obsah knihovny, pouze entita se všemi svými architekturami nebo pouze entita se specifikovanou architekturou. Každá třída která realizuje export do nějakého výstupního formátu musí být potomkem této třídy. Export se realizuje pomocí traverzeru který je zavolán exportní třídou a spustí se v místě které je dané nastavením v `Location`. Traverzer při průchodu strukturou volá sémantické metody které obsahuje exportní třída daného formátu. Tyto metody generují do souboru jednotlivé konstrukce daného jazyka.
Pro export strukturně popsaných architektur se využívá traveu strukturní architektury, který prochází jednotlivé instance,spoje a porty.
Pro export behaviorálně popsaných architektur se využívá exportní mapy kterou definuje třída `Export`. Tuto mapu tvoří dispatcher, do kterého se při inicializaci přidávají ukazatele na reprezentační funkce generující kód konkrétních popisů. Příklad: přidání funkce reprezentující popis symetrické funkce pro kontrolní dbg výstup:

```
int DbgSymFunc::wantInit =
  Export::exportMap.Enlist<DbgExport&,SymFunc&,&DbgSymFunc::Represent>();
```

Při exportu je poté volána metoda exportní mapy `perform` se dvěma parametry představujícími exportní objekt a popis chování (potomek BehavDescr). Na základě typů těchto parametrů se dispatcher pokusí vyhledat v mapě, kterou obsahuje, reprezentační funkci odpovídající těmto typům. Pokud je funkce nalezena je spuštěna s danými parametry.
V současné verzi je implementován export do formátu VHDL (součást této práce) a kontrolní výstup dbg.

2.7.1 DBG výstup

Tento výstup slouží ke kontrole struktur uložených v HUBu. Pro procházení vrchní vrstvy HUBu využívá traverseru který prochází jednotlivé knihovny, v každé knihovně projde všechny entity a pro entitu projde všechny její architektury. Pokud je architektura popsána strukturou

prochází tuto strukturu traverzerem strukturní architektury, který postupně projde všechny instance a spoje. Pokud je architektura popsána chováním využije se k popisu exportní mapy. Výstupní soubor má následující formát:

```
+unit
    +library primitives
+entity and2
    =port I1 attr in
    =port I2 attr in
    =port O attr out
    +arch vhdl attr behav
symmetric func AND, 2 inputs
    -arch vhdl
-entity and2
    -library primitives
    +library work
+entity example
    =port A attr in
    =port B
    =port C attr out
    +arch example_arch attr struct
=instance and2_1 and2.primitives
+net a A, I1.and2_1
-net a
+net b B, I2.and2_1
-net b
+net c C, O.and2_1
-net c
    -arch example_arch
-entity example
    -library work
-unit
```

3 Popis jazyka VHDL

3.1 Úvod

Zde je uveden stručný popis tohoto jazyka, úplný popis je k dipsozici viz. [4] nebo [1]. Jazyk **VHDL**[1] je určen k použití ve všech fázích návrhu elektronických systému, podporuje vývoj, syntézu a testování hardwarových návrhů. Umožňuje popisovat číslicové systémy na různé úrovni abstrakce. Primární jednotkou abstrakce je entita. Každý spustitelný program ve **VHDL** je tvořen alespoň ze dvou samostatných jednotek, entity a její architektury a má podobu modelu. Entita specifikuje rozhraní modelu tj. vstupní a výstupní porty kterými může komunikovat s okolím. Architektura definuje vnitřní popis modelu který může být popsán buď chováním nebo strukturou s použitím dalších komponent z nichž každá je definovaná opět dvojicí entita a architektura. Samotný program může být tedy použit jako komponenta v jiném složitějším modelu tzn. návrh je hierarchický. Každá entita může mít více alternativních popisů (architektur).
Architektura nabízí dvě různá prostředí ve kterých ji můžeme popsat jedná se buď o prostředí sekvenční nebo prostředí paralelní. Sekvenční prostředí slouží k popisu chování (algoritmicky) a to lze popsat procesem, podprogramem nebo funkcí. Paralelní prostředí popisuje vazby mezi bloky strukturním popisem tj. hierarchie dílčích komponent, nebo pomocí data-flow popisu (paralelní příkazy).
VHDL podporuje také knihovny s knihovními jednotkami. Knihovna fyzicky existuje jako adresář který obsahuje soubory s knihovními jednotkami. Každá knihovní jednotka je samostatně analyzovatelná část. Knihovní jednotku tvoří primární a sekundární jednotky (popis viz. níže). V každém programu jsou dvě předdefinované knihovny. Pracovní knihovna `work` a standardní knihovna `std`. Pokud se v programu používají jednotky z jiných knihoven musí být zviditelněny. VHDL je typový jazyk obsahuje 4 základní skupiny typů:

- Skalární (diskrétní , real, výčtový)
- Složené typy (pole , záznamy)
- Přístupový typ (ukazatel)
- Soubor (file)

Typy existují jako předdefinované v knihovnách nebo lze deklarovat i vlastní typy. Také je možné definuvat různé podtypy dat: vymezení podmnožiny hodnot určitého předem deklarovanécho typu. Deklarace typů neobsahují pouze rozsah hodnot ale definují i funkce operátorů. VHDL je silně typový jazyk, konverze mezi typy musí být uvedeny explicitně. Základní typy mají své předdefinované atributy (vlastnosti).

3.2 Objekty

Nejedná se o objekty jako v klasickém objektově orientovaném jazyce, jejich účelem zde je reprezentovat atributy simulovaného systému. **VHDL** obsahuje 3 třídy obejktů: **constant** (nemění hodnoty, sekv. i par. prostředí), **variable** (mění hodnoty, sekv. prostředí), **signal** (mění hodnoty, sekv. i par. prostředí). Každý objekt ve **VHDL** musí být nějakého datového typu. Objekty mají své předdefinované atributy.

[1] VHDL = Very high speed integrated circuit Hardware Description Language

3.3 Atributy

Slouží k přidání informace (vlasnosti) pojmenovaným jednotkám: entita, architektura, konfigurace, procedura, funkce, package, type, podtyp, constant, signal, variable, komponenta, label, literal, group nebo soubor. Atribut je tvořen dvojicí jméno + hodnota. Než bude nějakému objektu přiřazen atribut musí být nejdříve deklarován. Deklarace se skládá se jména atributu a datového typu jeho hodnoty.

3.4 Rozhraní

Popis rozhraní se vyskytuje u entity, komponenty, podprogramu (funkce ,procedura) nebo bloku. Je tvořeno seznamem jednotlivých objektů rozhraní. U rozhraní portů (entita,komponenta,blok) tvoří objekty rozhraní deklarace signálů:

```
[signal] identifier_list:[mode] subtype_indication [bus] [:= static_expression]
```

U rozhraní generických parametrů (entita,komponenta,blok) tvoří objekty rozhraní deklarace konstanty:

```
[constant]identifier_list:[in]subtype_indication[:=static_expression]
```

Objekty rozhraní u rozhraní parametrů podprogramu tvoří deklarace signálů, konstant, proměnných nebo souborů.

3.5 Program

Soubor s programem může obsahovat příkazy pro zviditelnění knihoven a knihovních jednotek, a deklarace primárních a sekundárních jednotek. Sekundární jednotka musí být definována až po deklaraci jí příslušné primární jednotky.

```
Library IEEE;  -- Zviditelnění knihovny
Use ieee.std_logic_1164.all;  -- Zviditelnění package
```

Primární jednotky:

- Deklarace entity (entity)
- Deklarace konfigurace (configuration)
- Deklarace balíčku (package)

Sekundární jednotky:

- Tělo architektury (architecture)
- Tělo balíčku (package body)

3.6 Deklarace Entity

```
entity jmeno_entity is
-- generické parametry
port ( a,b : in std_logic; sum,carry: out std_logic  ) ;
-- deklarační část
[ begin
 -- příkazová část entity
end [ entity ] [jmeno_entity] ;
```

Generické parametry:
Slouží k předaní statické (konstantní) informace entitě zvenčí. Generické parametry jsou nepovinné.
Porty:
Každý port rozhraní má definované své jméno, svůj mód [in, out, inout, buffer, linkage] a datový typ. Deklarace portů je nepovinná.
Deklarační část:
V této části mohou být deklarace podprogramů, typů, podtypů, konstant a atributů.
Příkazová část:
V této části mohou být pouze pasivní příkazy tzn. ty které nemění hodnoty žádných signálů. Jedná se o příkaz assert (kontroluje zadanou podmínku a popř. zaloguje report o chybě), pasivní proces a pasivní volání procedury.

3.7 Deklarace architektury

Architektura popisuje vnitřní strukturu nebo funkci entity, může být popsána strukturou, datovými toky nebo chováním. Tyto popisy mohou být kombinovány i v rámci jedné architektury. Skládá se z deklarační části a části příkazů.

```
architecture jmeno_arch of jmeno_entity is
-- deklarační část architektury
    begin
-- příkazová část architektury
 end [ architecture ] [ architecture_simple_name ] ;
```

Deklarační část může obsahovat deklarace typů, signálů, konstant, subprogramů (funkce a procedury), komponent a skupin (groups).

3.7.1 Paralelní prostředí

Pořadí výpočtu je zde dáno aktivitou signálů. Paralelní příkazy v architektuře definují vztah mezi vstupy a výstupy. Tento vztah může být vyjádřen několika typy příkazů: Paralelní přiřazení signálu, vytvoření instance komponenty, volání procedury, příkaz generate, příkaz assert a příkaz bloku.
Data Flow popis architektury:
Popis architektury datovými toky je realizován paralelním příkazem přiřazení signálu. Každý z těchto příkazů je proveden když nastane nějaká změna na vstupním signálu obsaženým v tomto příkazu. Tyto příkazy mohou být podmíněné nebo nepodmíněné.

```
Nepodmíněný příkaz: output <= input1 or input2;
Podmíněný příkaz: output <= input1 when input2 ='0' else '1';
```

Strukturní popis architektury:
Tento popis vyjadřuje konkrétní vazby mezi dílčími jednotkami, zahrnuje jejich modely. Umožňuje vytvořit hierarchický popis tzn. že zapojená jednotka může mít opět strukturní architekturu složenou s dalších dílčích jednotek.

```
Architecture struct_arch of adder is
signal sum_int,carry1,carry2 : std_logic;
component half_adder
port( a,b : in std_logic;
         sum,carry:out std_logic);
```

```
end component;
begin
--příkazová část
end;
```

Strukturní popis je realizován pomocí komponent a lokálních signálů (propojovací vodiče). Komponenta slouží k rezervaci místa pro vnoření entit, lze si ji představit jako sokl na desce plošných spojů do kterého se zapojí konkrétní obvod (entita). Do jedné komponenty lze tedy zapojit různé entity (musí mít shodné rozhraní). Přiřazení entit a architektur dílčím komponentám se říká konfigurace a lze ji provést více způsoby. Instalace (zapojení) komponent do architektury se provádí v příkazové časti.

Osazení (konfigurace) komponenty:
Pokud komponenta a entita mají stejné jméno a rozhraní lze explicitní konfiguraci vynechat komponenta bude automaticky nakonfigurována s entitou se shodným jménem. Jinak pomocí příkazu konfigurace komponenty v dekl. části architektury:

```
for ha_1 : half_adder use entity work.half_adder2(arch_1);
```

Mapování portů entity a komponenty:
Následuje po konfiguraci komponenty. Pokud se shodují jména odpovídajících si portů entity a komponenty lze mapování vynechat, jinak nutno mapovat explicitně pomocí port map.

Instalace komponent:
Provádí se v příkazové části architektury. Každá komponenta může být v architektuře nainstalována vícekrát, v zapojení jsou z nich vytvořeny instance které jsou označeny jménem. Mapování portů komponenty se realizuje příkazem `port map`, který připojuje porty komponenty k portům entity nebo k lokálním signálům.

```
Jmenné mapování:
ha_1 : half_adder port map (  a => x, b => y, sum => z, carry => w);
Poziční mapování na signály x,y,z,w:
ha_1 : half_adder port map (  x, y, z, w);
```

Přímá instalace komponent:
Osazení entitou je provedeno přímo v příkazové části při instalaci, díky tomu odpadá deklarace komponenty.

```
ha_1: entity work.half_adder2(arch_1) port map(x,y,z,w);
```

3.7.2 Sekvenční prostředí (popis chování)

Příkazy jsou vykonávány sekvenčně v pořadí jakém jsou napsány. Sekvenční prostředí existuje pouze v příkazové části procesu, procedury nebo funkce. Přímo v příkazové části architektury je prostředí paralelní.

Proces:
Příkaz procesu je jeden z paralelních příkazů jehož funkce je popsána sekvenčním algoritmem. Deklarace procesu může obsahovat citlivostní seznam signálů při jejichž změně bude proces spuštěn.

```
[process_label:] process [ ( sensitivity_list ) ] [ is ]
  process_declarations
     begin
      sequential_statements
  end process [ process_label ] ;
```

3.8 Deklarace a tělo Knihovní jednotky (package)

Package je jednotka která slouží k seskupování různých deklarací, které mohou poté sdíleny a použity v různých návrzích. Knihovní jednotky (packages) jsou ukládány do knihoven. Package se skládá s deklarace (povinná) a těla které je nepovinné. Smyslem package je deklarovat sdílené typy,podtypy,konstanty, signály, soubory, aliasy, komponenty a atributy, použitelné ve větším počtu návrhů. Položky deklarované v jednotce se v návrhu zpřístupní pomocí klauzule use.

4 Implementovaná podmnožina jazyka VHDL

V následující části uvedu výpis všech konstrukcí v jazyce VHDL které importní systém do HUBu v současné době podporuje. Pokud syntaktický analyzátor rozpozná konstrukci která není implementována vyvolá výjimku typu not yet implemented. Tato podmnožina jazyka byla stanovena vedoucím práce.

4.1 Tělo programu

Konstrukce	**poznámka**	**Implementováno**
Zviditelnění knihovny	Přeskakuje se	Ne
Zviditelnění knih. Jednotky	Přeskakuje se	Ne
Primární jednotky		
Deklarace entity		Ano
Deklarace konfigurace		Ne
Deklarace package		Ne
Sekundární jednotky		
Architektura		Ano
Tělo knih. Jednotky (package)		Ne

Tabulka 4.1: Soubor VHDL

4.2 Deklarace Entity

Konstrukce	**poznámka**	**Implementováno**
Hlavička Entity		
Generické parametry		Ne
Deklarace portů		Ano
Deklarační část Entity	Nejsou povoleny žádne deklarace	Ne
Příkazová část Entity	Nejsou povoleny žádné příkazy	Ne

Tabulka 4.2: Deklarace Entity

4.3 Deklarace Architektury

Konstrukce	**poznámka**	**Implementováno**
Deklarační část architektury		
Deklarace a tělo Subprogramu		Ne
Deklarace typu a podtypu		Ne
Deklarace konstanty		Ne
Deklarace signálu		Ano
Deklarace proměnné (variable)		Ne
Deklarace souboru (file)		Ne
Deklarace aliasu		Ne
Deklarace komponenty		Ano
Deklarace a specifikace atributu		Ne
Specifikace konfigurace		Ne
Specifikace odpojení (disconnection)		Ne
Klauzule use		Ne
Příkazová část architektury		
Příkaz bloku		Ne
Příkaz procesu		Ne
Paralelní volání procedury		Ne
Paralelní příkaz assert		Ne
Paralelní příkaz přiřazení signálu		Ano
Příkaz instalace komponenty		Ano
Příkaz generate		Ne

Tabulka 4.3: Deklarace Architektury

4.4 Deklarace rozhraní (signálu)

```
[signal] identifier_list : [mode] subtype_indication [bus] [:=static_expression]
```

Konstrukce	**poznámka**	**Implementováno**
seznam identifikátorů		Ano
mód		Ano
typ signálu	pouze typ std_logic a std_logic_vector	Ano
Inicializace		Ne

Tabulka 4.4: Deklarace rozhraní

Pokud je signál typu std_logic_vector musí mít zadán rozsah, zadání rozsahu je povoleno pouze číslem typu integer. Př: std_logic_vector(7 downto 0)

4.5 Deklarace komponenty

```
component jméno [ is ]
[ generické_parametry ]
[ rozhraní_portů ]
end component [ jméno ] ;
```

Konstrukce	poznámka	Implementováno
Generické parametry		Ne
Rozhraní portů		Ano

Tabulka 4.5: Deklarace rozhraní

Je podporováno pouze automatické osazení komponenty, tzn. není možné specifikovat explicitní konfiguraci komponenty. Jméno komponenty tedy musí odpovídat jménu konfigurované entity a rozhraní entity a komponenty musí být také shodné. Pokud je třeba specifikovat s jakou architekturou má být entita nakonfigurována je možné využít automatické instalace komponenty v příkazové části.

4.6 Paralelní příkaz přiřazení signálu

Je podporován pouze nepodmíněný příkaz. Jako operandy se mohou vyskytovat jména signálů (u více bitových s indexací), porty rozhraní entity, a konstanty '1' nebo '0'. Při práci se signály nebo porty které jsou typu pole `std_logic_vector` je povoleno pouze jednoduché indexování. Následující operace pro porty typu `std_logic_vector` nejsou povoleny:

```
x(1 downto 0) <= A(1 downto 0);
x <= A;
```

Je povoleno jednoduché indexování:

```
x(0) <= A(0);
x(1) <= A(1);
```

4.7 Příkaz instalace komponenty

```
jméno_instance :
[ component ] jméno_komponenty
[ mapování_generických_parametrů ]
[ mapování_portů ] ;
```

Automatická instalace komponenty:

```
jméno_instance :
entity jméno_entity [ ( identifikátor_architektury ) ]
[ mapování_generických_parametrů ]
[ mapování_portů ] ;
```

Mapování generických parametrů není povoleno. Mapování portů je řešeno seznamem asociativních elementů:

```
[ formal_part => ] actual_part
```

Je podporováno jmenné i asociativní mapování. Jako formální a aktuální část může být dosazeno jméno. Aktuální část podporuje i parametr **open** tzn. že port komponenty nebude zapojen.

4.8 Specifikace jména

Konstrukce	**poznámka**	**Implementováno**
Identifikátor		Ano
Vybrané jméno (prefix.sufix)		Ne
Indexace		Ano
Jméno atributu		Ne

Tabulka 4.6: Specifikace jména

4.9 Výrazy

Struktura výrazů je zpracována tak jak je popsána v gramatice, části které nejsou implementovány jsou až u primárního výrazu.

Konstrukce	**poznámka**	**Implementováno**
Jméno		Ano
Literál		Ano
Aggregate		Ne
Volání funkce		Ne
Qualified expression		Ne
Alokátor		Ne
(výraz)		Ano

Tabulka 4.7: Primární výraz

5 Generátor syntaktických analyzátorů Bison

Jedná se o program spustitlený v prostředí Linuxu a je distribuován jako svobodný sw pod licencí GNU GPL. Vstupem programu Bison je bezkontextová gramatika zapsaná v BNF formě, generátor na základě této gramatiky vygeneruje zdrojový kód parseru zpracovávajícího uvedenou gramatiku.
Gramatika se skládá z terminálních a neterminálních symbolů, terminální symboly generuje lexikální analyzátor. Gramatika je popsána jednotlivými pravidly kdy na levé straně je vždy neterminální symbol a na pravé straně je rozklad na jednodušší konstrukce složené opět z neterminálních a terminálních symbolů. Každý terminální i neterminální symbol může mít přiřazenu sémantickou hodnotu. Každé pravidlo může mít přiřazenu sémantickou akci která je zavolána poté co parser rozpozná ve vstupním souboru dané pravidlo. Každý lex. element a pravidlo gramatiky má kromě sémantické hodnoty také přiřazenu informaci o umístění v souboru (Location). Informace o umístění obsahuje jméno souboru, řádek a sloupec na kterém symbol začíná a končí. Tyto Informace jsou automaticky vypočítávány při zpracování vstupního souboru parserem vygeneroavným Bisonem. Parser pracuje na principu zásobníkového automatu. Kompletní popis programu Bison je k dispozici viz. [3]

5.1 Formát vstupního souboru analyzátoru

Vstupní soubor obsahující gramatiku jazyka má následující strukturu:

```
%{
  Úvod
%}
  deklarace
%%
  pravidla_gramatiky
%%
  Závěr
```

Úvod může obsahovat deklarace typů a proměnných používaných v sémantických akcích, můžou zde být příkazy preprocesoru. Musí zde být deklarovaný lexikální analyzátor **yylex** a funkce na hlášení chyb **yyerror**.
Deklarační část obsahuje seznam jmen neterminálních a terminálních symbolů a můžou zde být určeny sémantické typy jednotlivých symbolů a definována priorita operátorů.
V části gramatických pravidel jsou popsána jednotlivá pravidla která říkají jakým způsobem zkonstruovat jednotlivé neterminální symboly z jejich součástí.
Závěr může obsahovat libovolný kod jazyce C(C++) typicky definice funkcí deklarovaných v prologu.

5.2 Syntaxe pravidel gramatiky

```
vysledek: casti_pravidla
        ;
```

Výsledek je neterminální symbol který popisuje dané pravidlo a **casti_pravidla** jsou různé terminální a neterminální symboly z nichž se skládá dané pravidlo. Mezi každým symbolem může být zavolána sémantická akce, je ohraničena složenými závorkami a je popsána v jazyce C(C++). Různá pravidla která mají stejný výsledek mohou být vypsána samostatně nebo oddělena znakem "—".
Příklad:

```
vyraz : vyraz '+' vyraz { cout << "toto je semanticka akce";}
  | vyraz '-' vyraz { cout << "odecteni dvou vyrazu"; }
  | faktor
  ;
```

5.3 Datové typy sémantických hodnot symbolů

Pokud mají všechny symboly pouze jeden typ sémantických hodnot definuje se tento typ pomoci makra YYSTYPE :

```
#define YYSTYPE double
```

Pokud je typů více lze je definovat pomocí kolekce všech typů příkazem %union nebo přiřadit zvlášt každému symbolu jeho sémantický typ. To se provede v části prologu u neterminálního symbolu deklarací:

```
\%token <jmeno_typu> jmeno_neterminalniho_symbolu
```

U terminálního symbolu:

```
\%type <jmeno_typu> jmeno_terminalniho_symbolu
```

5.4 Přístup k sémantickým hodnotám symbolů

V sémantické akci každého pravidla je možné přistupovat k sémantickým hodnotám symbolů z nichž je pravidlo sestaveno, přistupuje se k ní konstrukcí $n která vrací hodnotu n-tého symbolu. K sémantické hodnotě právě konstruovaného pravidla se přistupuje pomocí konstrukce $$.
Příklad:

```
vyraz :
    ...
    | vyraz '+' vyraz { $$ = $1 + $2; }
    ;
```

5.5 Rozhraní parseru

Bison umožnuje vygenerovat zdrojový kód pouze v jazyce C nebo vytvořit třídu parseru v jazyce C++.

5.5.1 Rozhraní v jazyce C

Je vygenerována funkce `yyparse` která po zavolání spustí syntaktickou analýzu. Tato funkce čte neterminální symboly které dostává od lexikálního analyzátoru a spouští sémantické akce. Lze definovat přidání parametrů funkci které potom mohou být využity v sémantických akcích:

```
%parse-param {argument-declaration}
```

Lexikální analyzátor je volán funkcí `yylex` která nemá žádný parametr a vrací identifikátor neterminálního symbolu. Sémantické hodnoty jsou předávány pomocí globální proměnné `yylval`. Informace o umístění symbolu jsou také předávány pomocí globální proměnné `yyloc`.

5.5.2 Rozhraní v jazyce C++

Pokud chceme vygenerovat parser v jazyce C++ je třeba mu zvolit kostru podle které jej vygeneruje, to se provede deklarací **%skeleton "lalr1.cc"**. Po spuštění Bizon vytvoří následující soubory:

- Position.hh , location.hh : Obsahují třídy pro určení pozice symbolů
- Stach.hh : Přídavná třída Stach využívaná parterem
- file.hh , file.hh : Deklarace a implementace třídy realizující parser zpracovávající zadanou gramatiku

Třída realizující parser obsahuje:

- Konstruktor implicitně bez parametru, parametry lze přidat deklarací:
- Metoda parse: spustí syntaktickou analýzu
- Metoda set_debug_level(debug_level 1): umožňuje nastavit výpis hlášení o zpracování gramatiky
- Metoda error (const location type& l, const std::string& m): Hlášení syntaktických chyb, volá jí parser pokud narazí na chybu. Definice musí být dopsána uživatelem.

Rozhraní lexikálního analyzátoru: Parser ovládá lexikální analyzátor voláním funkce `yylex`. Funkce `yylex` musí mít následující rozhraní:

```
int yylex (semantic_value_type& yylval, location_type& yyloc);
```

5.6 Spuštění Bisonu

Vygenerování zdrojového kódu se provede spuštěním programu Bison:

```
bison -r all -d jmeno_souboru_s_gramatikou
```

Výstupem jsou zdrojové kódy parseru popř. soubory s přídavnými třídami a také soubor s koncovkou .output ve kterém je výpis případných konfliktů a popis celého zásobníkového automatu s kterým parser pracuje.

6 Analýza

6.1 Úvod

Před započetím implementace modulu importu **VHDL** do **HUBu** a modulu exportujícího struktury uložené v **HUBu** zpět do **VHDL** bylo potřeba se důkladně seznámit se strukturou HUBu a poté zvolit způsob jakým budou jednotlivé prvky jazyka VHDL převedeny.
Jelikož **VHDL** je velice komplexní jazyk bylo by velmi náročné ho implementovat kompletně celý. Požadavkem bylo se zaměřit zejména na podporu strukturně navržených obvodů ve **VHDL**, bylo tedy potřeba vybrat části jazyka v těchto návrzích používané.

6.2 Import

6.2.1 Syntaktická analýza

Prvním krokem bylo vytvoření syntaktického analyzátoru který bude zpracovávat konstrukce jazyka **VHDL** . Nejprve se uvažovalo že by se napsal analyzátor zpracovávající pouze implementovanou podmnožinu jazyku, toto řešení by bylo ale velmi nepraktické pokud by bylo potřeba rozšířit implementovanou podmnožinu. Bylo tedy zvoleno řešení kdy bude implementována kompletní gramatika jazyka **VHDL** a pokud se v importovaném návrhu vyskytnou konstrukce které nejsou implementovány ohlásí se chyba. Jelikož gramatika **VHDL** je poměrně rozsáhlá zvolil jsem před tím než psát analyzátor ručně využití generátoru syntaktických analyzátorů Bison. Výhodou bylo zejména to že dostupná gramatika jazyka **VHDL** byla psána ve velmi podobné syntaxi jako je ta kterou zpracovává Bison.

6.2.2 Převod konstrukcí VHDL do HUBu

Základní jednotkou popisu ve VHDL je pojmenovaná entita, která představuje stejně jako ve HUBu abstrakci nějakého obvodu . V **HUBu** jsou základní jednotkou popisu knihovny a návrhy. Knihovny obsahují jednotlivé entity. Před importem je tedy potřeba specifikovat knihovnu do které se budou entity ukládat. Knihovna může být buď specifikována před spustěním importu, nebo pokud není specifikována musí být knihovna vytvořena. Zde byla zvolena možnost vytoření pracovní knihovny **work** do které se importované entity budou ukládat, byla zvažována také možnost pojmenovat knihovnu podle názvu importovaného **VHDL** souboru.
Entita v **HUBu** obsahuje prototyp rozhraní portů a jmenný prostor architektur. Architektury tvoří alternativní popisy vnitřku bloku a to buď chováním nebo strukturou. Deklarace entity ve vhdl obsahuje hlavičku ve které může být deklarace portů entity a generických parametrů.

6.2.3 Porty entity

Jednotlivé porty v deklaraci portů odpovídají prototypům portů v rozhraní portů entity ve **HUBu**. Každá deklarace portu obsahuje jeho jméno, mód (in — out — inout — buffer — linkage) a identifikaci datového typu.
Jazyk **VHDL** podporuje deklarace typů a podtypů podobně jako je to u běžných programovacích jazyků. Tyto typy mohou být deklarovány v balíčcích (package), deklaračních částech entit a architektur. Základní typy obsahují knihovny `ieee.Std_Logic_1164` a `ieee.numeric_std`.
Jelikož implementace těchto typů by byla značně náročná, byly stanoveny 2 základní nejčastěji používané typy s kterými import bude pracovat. Jsou to `std_logic`, a `std_logic_vector`.
Port který bude typu `std_logic` je jednobitový port k němuž může být v HUBu vytvořen odpovídají prototyp portu stejného jména a směru.
Port který bude typu `std_logic_vector` (musí mít u sebe zadány meze) je vícebitový port označený jedním jménem, k jednotilvým bitům se ve VHDL přistupuje pomocí indexace. V

HUBu nejsou podporovány vícebitové porty, každému bitu bude tedy vytvořen jeden prototyp portu se jménem uvedeným v deklaraci s přidaným číslem bitu. Po importu budou tyto porty jako jednotlivé porty v rozhraní a již nebudou seskupeny, na funkčnost zapojení to nebude mít žádný vliv jde spíše o přehlednost a zkrácení zápisu, při exportu zpět do **VHDL** budou deklarovány každý bit jako samostatný port typu `std_logic`.
Deklarace dále může obsahovat deklarační a příkazovou část, tyto části nebyli implementovány.

6.2.4 Architektury

Vnitřky bloků(entit) jsou ve vhdl popsány rovněž jako u **HUBU** architekturami. Každá entita může mít několik různých architektur ,popsané buď chováním nebo strukturou. Požadavek byl se zaměřit hlavně na podporu strukturně navržených návrhů. Strukturní popis architektury je ve vhdl řešen pomocí komponent, které představují rozhraní entity která bude v architektuře propojena. Jejich deklarace se zapisuje v deklarační části architektury, mapování je realizováno v příkazové části architektury.

6.2.5 Komponenty

Existuje více způsobů deklarace komponent (viz. kapitola 3). Nejjednoduší způsob deklarace komponent je takový, že jméno komponenty se shoduje se jménem existující entity. U tohoto způsobu odpadá konfigurace komponent, nevýhoda je že nelze explicitně stanovit s jakou architekturou jsou instance entit do architektury zapojeny, je vybrána naposledy přeložená architektura. Konfigurace komponent se zapisuje následujícím způsobem:

```
for instantiation_list:component_name  use entity entity_name
[(architecture_identifier)]
end for;
```

Při tomto způsobu deklarace komponent by bylo třeba vytvářet pomocné pole entit bez architektur, které by představovalo komponenty. Při konfiguraci komponent by se konrolovalo rozhrani entit z pole komponent s rozhranim entity ktera by byla na komponentu konfigurovana. Také by bylo potřeba zaznamenávat jména instancí a jim příslušnou nakonfigurovanou entitu a její architekturu.
Výše zmíněná konfigurace zatím není naimplementována a je implementován první způsob deklarace komponent, kdy v deklaraci komponenty se kontroluje zda existuje entita se stejným jménem jako komponenta a pokud je nalezena kontroluje se zda se shodují rozhraní.
Pokud chceme explicitně stanovit z jakou architekturou má být entita namapována je možné při mapování komponenty použít přímou instalaci komponenty se specifikovanou entitou a její architekturou.

6.2.6 Deklarace signálu

Pro strukturní popis architektury je důležité aby byla naimplementována podpora deklarace signálů v deklarační části architektury. Signál ve **VHDL** v podstatě odpovídá spoji zde v **HUBu**, pro každý deklarovaný signál se tedy v implementaci architekrutry vytvoří stejně pojmenovaný spoj.

6.2.7 Příkazová část architektury

Může obsahovat proces, volání procedury, přiřazení signálu, příkaz bloku, příkaz assert, Příkaz generate, přiřazení vybraného signálu,příkaz generate a příkaz instantiace komponenty.

Aby návrh byl strukturní je zřejmé že musí být podporován příkaz instalace komponent. Aby mohly být používany symetrické funkce jako jsou and, nand, or ,... je potřeba povolit příkaz přiřazení signálu.
Příkaz přiřazení signálu se skládá z cílového spoje nebo signálu na levé straně a výrazu na pravé straně. Výraz může obsahovat logické operátory jejichž operandy tvoří jména spojů nebo portů rozhraní. Tento výraz představuje propojení entit primitivních funkcí. Jelikož výraz se v jazyce VHDL nevyskytuje pouze u příkazu přiřazení, není možné propojovat entity primitivních funkcí v něm obažených přímo při procházení syntaktickým analyzátorem ale je třeba vytvořit syntaktický strom jehož uzly budou mít metodu která bude sloužit k propojení prvků v HUBu. Tato metoda byla definována tak že vrací jméno spoje který byl vytvořen např. u binarního operátoru vrací jméno výstupního spoje instance primitivní entity představující funkci operátoru. Dále se zde vyskytl problém jak vyřešit propojení cílového spoje nebo portu se spojem který vrátí propojení výrazu na pravé straně. Ve VHDL je možné propojovat mezi sebou spoje ale HUB tuto vlastnost nepodporuje. Tento problém byl vyřešen tak, že z cíle bude vytvořen vždy spoj, tzn. pokud tam byl port je k němu připojen nový spoj. Výraz na pravé tedy musí vracet port. Kvůli tomu do metody propojující prvky stromu byl přidán parametr který říká aby nebyl vytvořen výstupní spoj, tím se docílí toho že operátor na vrcholu stromu vrátí port a bude moct být připojen k cílovému spoji. Toto řešení funguje když je na pravé straně výraz alespoň s jedním operátorem nebo pouze jméno portu. Nefunguje v případě že je na pravé straně pouze jménou spoje, tento případ je zatím řešen způsobem že je vyhozena vyjímka informující o tom že takto propojovat nelze. Pokud by následující propojení mělo být povoleno dalo by se řešit například vytvořením instance primitivní entity buffer a pomocí ní tyto dva spoje propojit.

6.3 Export

6.3.1 Výpis knihoven

Hub podporuje export architektury, entity (je vygenerován kód entity a všechny jeji architektury), knihovny (je vygenerován kód všech entit s architekturami v příslušné knihovně). Pokud není specifikována architektura, entita ani knihovna je exportován obsah všech knihoven. Zde může nastat problém, pokud by se postupně obsah všech knihoven postupně exportoval do jednoho souboru mohlo by se vyskytnout v tomto souboru více stejně pojmenovaných entit. Možnost jak tomuto předejít je buď zakázat export ze všech knihoven zároveň nebo obsah každé knihovny generovat do zvláštního souboru. Jako řešení byla zvolena druhá možnost, výstupní soubory jsou pojmenovány jako knihovna v HUBu. Vygenerovaný soubor tedy obsahuje deklarace entit a architektur které byly uloženy v dané knihovně.
Pro každou entitu je vygenerován kód s deklarací portů, po té se prochazejí všechny architekrury dané entity a generuje se jejich kód.

6.3.2 Popis architektury strukturou

Prvky strukturní architektury v HUBu tvoří porty rozhraní entity, instance a spoje. V deklarační části architektury je třeba vypsat deklarace signálů které jsou v architektuře zapojené tzn. projít všechny spoje architektury. Ve VHDL není povoleno deklarovat spoj který by se jmenoval stejně jako nějaký port rozhraní protože tyto porty mají vytvořeny stejně pojmenované spoje implicitně. Tento problém by se dal řešit tím že by se před jméno každého spoje přidal nějaký prefix ale mohlo by to vést k tomu že by se jméno spoje s prefixem mohlo shodovat se jménem nějakého portu rozhraní. Jako řešení byl zvolen způsob kdy se při deklaraci signálu kontroluje zda je spoj připojen ke stejně pojmenovannému portu , pokud ano jeho deklarace se nevypisuje. Pokud se jméno spoje shoduje se jménem nějakého jiného portu rozhraní ke

kterému není tento spoj připojen je k jeho jménu přidán index, se kterým se poté objevuje např. při mapování instancí nebo v příkazu přiřazení. Pro takto oindexovaný spoj se vypíše deklarace signálu.
Dále po deklaraci signálů nasledují deklarace komponent. První návrh řešení výpisu instancí obsažených v architektuře byl takový, že se nejprve projdou všechny instance a pro každou se vytvoří v deklaraci komponenta pojmenovaná jako entita z níž je instance vytvořena. Poté se v příkazové části architektury vygeneruje kód ve kterém budou tyto instance mapovány. Výše zmíněné řešení by bylo sice jednoduché a systematické ale jeho nevýhodu je že se vytvářejí komponenty i pro entity představující primitivní funkce (and, nand, nor, ...) a tím by se vygenerovaný kód stal delší a méně přehledný. Předchozí postup byl tedy ještě upraven o kontrolu zda se nejedná o instance primitivních entit. Pro instance primitivních entit se místo mapování komponenty vygeneruje příkaz přiřazení s funkcí odpovídající instance.

7 Popis Implementace Importního Systému

Importní systém se skládá ze tří základních objektů, lexikálního analyzátoru, parseru a importního objektu. Lexikální analyzátor prochází vstupní soubor a generuje lexikální elementy s jejich atributy. Parser zpracovává zadanou gramatiku jazyka VHDL, jeho součástí je lexikální analyzátor. Při průchodu konstrukcemi gramatiky jazyka VHDL volá sémantické metody, které jsou definovány v importním objektu. Tyto sémantické metody se starají o převod jednotlivých konstrukcí jazyka VHDL do struktury HUBu. Importní objekt nastaví místo v hubu do kterého bude probíhat import a poté spustí syntaktický analyzátor který začíná na startovním symbolu gramatiky.

7.1 Lexikální analyzátor

Je realizován třídou `LexanVhdl` která je potomkem třídy `Lexan`. Třída `Lexan` definuje obecné vlastnosti každého lexikálního analyzáru, obsahuje v sobě tedy souborový stream který představuje zpracovávaný vstupní soubor. Dále jsou v ní obsaženy seznamy klíčových slov a symbolů. Každou položku těchto seznamů tvoří dvojice číslo tokenu a jméno symbolu nebo klíčového slova typu string (Tyto dvojice do seznamu vkládá až analyzátor konkrétního jazyka). Také deklaruje proměnné do kterých se ukládají atributy lexikálních symbolů. Obsahuje abstraktní metodu `NextToken` která slouží ke generování lexikálního elementu procházením vstupního souboru. Konstruktor má jeden parametru typu `char*` který definuje jméno vstupního souboru.
Třída `LexanVhdl` by tedy měla definovat metodu `NextToken` tak aby byli generovány lexikální elementy jazyka VHDL. Jelikož je syntaktický analyzátor vygenerovaný programem **Bison** nebylo možno použít způsob zpracování takový že metoda `NextToken` vrátí pouze identifikátor lexikálního elementu a jeho případný atribut uloží do `public` proměnné zděděné od třídy `Lexan` ke které by poté měl přístup syntaktický analyzátor. Syntaktický analyzátor vygenerovaný programem Bison vyžaduje aby funkce kterou volá lexikální analyzátor měla jako parametr ukazatel na typ atributu, kam lexikální analyzátor přiřadí případný atribut lexikálního elementu. Metoda `NextToken` bez parametru byla tedy definována tak že vrací lex. element označující chybu a byla přidána nová metoda NextToken se 2 parametry. První parametr je ukazatel na atribut a druhý je ukazatel na proměnnou obsahující informace o umístění lex. elementu v souboru. Typ této proměnné rovněž definuje Bison. Funkce této metody je realizována konečným automatem.

7.2 Parser

Je realizován třídou `VhdlParser`, která v sobě obsahuje Lexikální analyzátor VHDL a ukazatel na klienta parseru. Klient parseru je realizován třídou `VhdlParserClient` a obsahuje deklarace sémantických metod které budou volány parserem. Definice těchto metod jsou prázdné, jedná se pouze o rozhraní, definice těchto metod předefinuje importní třída jež je potomkem tohoto klienta.
Konstruktor parseru má jeden parametr a to ukazatel na objekt typu klienta. Parser je konstruován importní třídou která jako paramter konstruktoru parseru zadá ukazatel sama na sebe a tím se zajistí že parser při průchodu gramatikou bude volat již správně definované sémantické metody.
Syntaktická analýza se spouští metodou `run` která má jako parametru název vstupního souboru. Tato metoda po zavolání inicializuje lexikální analyzátor a vytvoří objekt typu `VhdlBisonParser`. Jedná se o třídu vygenerovanou programem Bison která provádí syntaktickou analýzu vstupního souboru.

```
void VhdlParser::run (const std::string& f)
{
  l.init( f.c_str() );
  fn = f;
  yy::VhdlBisonParser parser( l , *this );
  parser.set_debug_level (trace_parsing);
  parser.parse ();
}
```

7.2.1 Vstupní soubor programu Bison

Gramatika jazyka VHDL je zapsána v souboru `vhdl_bison_parser.ypp`. Po přeložení tohoto souboru Bisonem vzniknou soubory `vhdl_bison_parser.hpp` a `vhdl_bison_parser.cpp` které obsahují deklaraci a implementace třídy `VhdlBisonParser` která provádí syntaktickou analýzu. Zároveň s nimi jsou vytvořeny soubory location.hh a postion.hh obsahující třídy `yy::location` a `yy::position` nesoucí informaci o poloze symbolů v souboru. Rozhraní těchto tříd je popsáno v manuálu programu Bison viz. [3].
Uvodní část vstupního souboru obsahuje informaci o tom že bude generován parser v jazyce C++ s použitou kostrou `lalr1.c` . Dále je zde definováno jméno třídy parseru která bude vytvořena a je zde definováno makro které určuje funkci kterou bude volán lexikální analyzátor:

```
#define yylex(yylval, location) lex.NextToken( (yylval) , (location) )
```

Datové struktury které představují sémantické typy symbolů jsou deklarovány v souborech `vhdl_parser_struct.h` a `vhdl_expressions.h` . Za úvodní částí následují deklarace neterminálních symbolů a určení sémantických typů jednotlivých symbolů.
Sémantické typy jsou následující:

- Seznam identifikátorů `IdentifierList`
- Asociativní element `GenAssocElem`
- Seznam asociativních elementů `GenAssocList`
- Jméno `SigName`
- Typ `Type`
- `StringToken`: identifikátor + informace o umístění tohoto identifikátoru v souboru
- Výraz `Expr`

Dále následuje část s pravidly gramatiky VHDL (IEEE Std 1076-2002), soubor s touto gramatikou je přiložen na CD. Tato gramatika po přeložení ve stejném tvaru jako byla dodána obsahovala konflikty, musela být mírně pozměněna ale jednalo se jen o drobné změny základ zůstal stejný a analyzátor je schopen rozpoznat kompletní syntaxi jazyka VHDL.

7.3 Přidané třídy výjimek

Každá výjimka v HUBu se skládá z důvodu, místa (v souboru, v bloku), kontextu a systémové kategorie. Určení místa je realizováno tak že se zjistí z lexikálního analyzátoru kde se právě nachází. Tento způsob ale nebylo možné použít při importu vhdl protože zde může dojít k tomu že chyba je identifikována až později a informace z lexikáního analyzátoru by neodpovídaly skutečnosti. K tomuto může dojít např. při zpracování příkazu přiřazení kdy je nejprve sestaven

syntakticý strom výrazu a až poté se kontroluje zda v něm obsažené identifikátory spojů nebo portů existují.
Je to tedy vyřešeno tak že pro každý identifikátor který je rozpoznán parserem je vytvořen nový objekt typu `StringToken` který v sobě obsahuje zároveň jméno identifikátoru i informaci o jeho umístění v souboru kterou definuje objekt třídy `yy::Location` který rovněž poskytne parser.
Při konstrukci výjimky se poté nepoužije místo které udává lexikální analyzátor ale použije se na to objekt třídy `yy::Locatation` který je součástí každého identifikátoru třídy `StringToken`. K tomu musely být definovány nové třídy vyjímek které jsou definovány v souboru `vhdl_err.h` .

7.4 Importní modul

Tento modul je tvořen třídou `VhdlImport` která dědí ze dvou tříd. První z nich je třída `VhdlParserClient` která obsahuje metody pro zpracování sémantických konstrukcí jazyka. Třída `VhdlImport` tyto metody předefinovává. Druhá třída jejímž je potomkem je třída `Import` která tvoří obecné rozhraní pro importní moduly HUBu.
Třída `VhdlImport` má přetížen operátor volání funkce s parametrem který určuje místo v hubu do kterého se bude importovan. Místo v hubu určuje třída `Location` která jejímž potomkem je třída `Import`. Místo v HUBu je tvořeno ukazatelem na knihovnu, entitu a architekturu.Jelikož jazyk VHDL zná pojem entita i architektura není povoleno aby před spuštěním byly specifikovány. Je povoleno pouze nastavit knihovnu do které se bude importovat, pokud není zvolena vytvoří se pracovní knihovna **work**.
Konstruktor třídy má dva parametry první specifikuje místo v HUBu a druhý obsahuje jméno souboru z kterého se bude importovat.
Kromě kontextu který je určen členskými proměnnými `theEntity_` , `theArch` a `theLib` které jsou zděděny ze třídy `Location`, jsou definovány ve třídě `VhdlImport` ještě další další členské proměnné které se využívají při importu jednotlivých konstrukcí:

```
hub::StructArch*    theArchImpl_; // current structure
hub::Instance*      theInstance_;   // current instance
hub::Name         synthLibName_; // library name for existing primitives
hub::Library*       theSynthLib_; // library for synthesized primitives
hub::Entity*       theComponent_; // current component (declaration)
hub::number_t       compPortsCount; // port count of current component

hub::Joinable* lastPort_;           // last port in expression
```

Tato třída v sobě také obsahuje objekt parseru jehož je tato třída klientem, tzn. že parser volá sémantické funkce které jsou definovány v této třídě. Operace importu se spustí zavoláním přetíženého operátoru třídy `VhdlImport`. Ta nejprve nastaví místo v HUBu a poté zavolá metodu parseru run kterou parser spustí jako parametr nastaví jméno souboru ze kterého bude import probíhat.
Zde je výpis deklarací všech sémantických metod starajících se o import implementovaných konstrukcí jazyka VHDL do hubu:

```
virtual void haveUnit(void);
virtual void endUnit(void);
    //primary & secondary units
virtual void haveEntity( const StringToken& entName );
virtual void endEntity(void);
virtual void haveArch(const StringToken& archName,const StringToken& entName );
```

```
virtual void endArch(const StringToken& archName ="" );
    //entity interface port declaration
virtual void havePort(InterfaceDecl& intDecl );
//signal declaration
virtual void haveSignalDeclar( InterfaceDecl intDecl );
//component declaration
virtual void haveComponent(const StringToken& compName);
virtual void endComponent(const StringToken& compName);
virtual void haveComponentPort(InterfaceDecl& intDecl );
// component instantion
virtual void haveComponentInst(const StringToken& label,
        const StringToken& compName,const StringToken& archName);
virtual void havePortAssoc(const StringToken& compPortName,
        const StringToken& NetName);
virtual void havePosPortAssoc(int portPosition, const StringToken& NetName);
virtual void endComponentInst(void);
//signal_assignment
virtual StringToken* haveSymFunc(const std::list<StringToken> listInputNames,
const std::string& func, bool createNet = true);
virtual void haveOperand(const StringToken& portName );
virtual void haveSignalAssignment(StringToken targetName, Expr* waveform );
```

7.5 Import jednotlivých konstrukcí VHDL

7.5.1 Entita

```
entity_declaration :
    LEX_V_ENTITY  LEX_V_IDENT
    {
      driver.client_->haveEntity(StringToken(*$2,@2));
      delete $2;
    }
    LEX_V_IS
          entity_header
          entity_declarative_part
          opt_entity_statement_part
    LEX_V_END opt_entity_end LEX_V_SEMICOL
    {
      driver.client_->endEntity();
    }
    ;
```

Metoda `haveEntity` vytvoří v aktuálně nastavené knihovně entitu se jménen které je obsaženo v proměnné typu `StringToken` v paramteru metody. Na tuto entitu bude ukazovat proměnná `theEnt_`. Pokud stejně pojmenovaná entita již v knihovně existuje je vyhozena výjimka s chybou. Metoda `endEntity` zruší nastavení aktuální entity `theEnt_`.
Hlavička entity může obsahovat deklaraci portů:

```
opt_formal_port_clause :
    /* nothing */
    | port_clause
    {
```

```
        for(InterfaceList::iterator i=$1->begin(); i != $1->end(); ++i)
          driver.client_->havePort(*i);
    }
    ;
```

Pravidlo `port_clause` vrátí seznam elementů rozhraní, pro každý element tohoto rozhraní je zavolána metoda `havePort`. Každý element rozhraní obsahuje jméno portu, mód a ukazatel na datový typ. Obsahuje také metodu `GeneratePortNames` která na základě typu portu vygeneruje vektor jmen portů. Existují 2 typy `SimpleType` a `ArrayType`, pokud je port typu `ArrayType` je vygenerováno tolik jmen jako je rozsah pole a jména jsou indexována. Metoda `HavePort` tedy nejprve vygeneruje seznam jmen portů a poté pro každé jméno vytvoří v rozhraní entity jeden prototyp portu.

7.5.2 Architektura

```
architecture_body :
    LEX_V_ARCHITECTURE LEX_V_IDENT LEX_V_OF entity_name LEX_V_IS
    {
      driver.client_->haveArch(StringToken(*$2,@2),*$4);
      delete $2;
      delete $4;
    }
          architecture_declarative_part
    LEX_V_BEGIN
          architecture_statement_part
    LEX_V_END opt_arch_end LEX_V_SEMICOL
     {
      driver.client_->endArch();
    }
    ;
```

Metoda `haveArch` vyhledá v aktuálně nastavené knihovně zda v ní existuje entita se jménem které je dodáno jako druhý parametr této metody. Pokud byla entita nalezena vytvoří se k ní architektura se strukturním popisem:

```
theArch_ = theEntity_->CreateArch <StructArch> (archName);
theArchImpl_ = theArch_->Impl<StructArch>();
```

Proměnná `theArchImpl_` je nastavena aby ukazovala na tuto nově vytvořenou architekturu. Poté se z prototypů portů nadřazené entity vytvoří instance portů v architektuře.

7.5.3 Deklarace Signálu

```
block_declarative_item :
 ...
| signal_declaration
     {
       for(InterfaceList::iterator i=$1->begin(); i != $1->end(); ++i)
           driver.client_->haveSignalDeclar(*i);
         delete $1;
     }
;
```

Symbol `signal_declaration` vrací seznam elementů rozhraní. Pro každý element tohoto rozhraní je zavolána metoda haveSignalDeclar. Ta vygeneruje vektor se jmény signálů a pro každé jméno vytvoří v aktuálně nastavené architektuře nový spoj. Kontroluje zda již stejně pojmenovanný signál nebo port rozhraní v architektuře neexistuje.

7.5.4 Deklarace Komponenty

Při rozpoznání konstrukce deklarace komponenty je zavolána metoda `haveComponent` s parametrem udávajícím jméno komponenty. Tato metoda vyhledá v aktuálně nastavené knihovně entitu se jménem odpovídajícím jménu komponenty. Pokud je nalezena je do proměnné `theComponent_` uložen ukazatel na tuto entitu.
Pokud komponenta obsahuje porty je vrácen seznam elementů rozhraní. Pro každý element tohoto rozhraní je zavolána metoda `haveComponentPort`. Tato metoda kontroluje zda entita na kterou ukazuje proměnná `theComponent_` obsahuje prototyp portu který je stejně pojmenován a má shodný mód.
Na konci deklarace komponenty je zavolána metoda `endComponent` která zruší ukazatel na aktuální komponentu a zkontroluje zda se počet portů komponenty které byly porovnávany shoduje s počtem portů nastavené entity.

7.5.5 Příkaz přiřazení signálu

```
conditional_signal_assignment :
    target  LEX_V_LOEQ opt_guarded delay_mechanism conditional_waveforms LEX_V_SEMICOL
    {
     driver.client_->haveSignalAssignment( $1->GenerateName() , $5 );
     delete $1;
     delete $5;
    }
;
```

Syntaktický strom výrazu Symbol `conditional_waveforms` vrací ukazatel na proměnnou typu `Expr` ta představuje kořen syntaktického stromu který byl vytvořen při zpracovávání výrazu syntaktickým analyzátorem. Třída `Expr` je abstraktní a tvoří rozhraní pro třídy které se mohou vyskytnout v syntaktickém stromu výrazu, deklararuje abstraktní metodu `Connect`:

```
virtual StringToken* Connect(VhdlParserClient* client,bool createNet=true)=0;
```

Strom výrazu se může skládat z objektů tříd představujících:

- binární operátor `BinOp`
- unární operátor `UnOp`
- Literál `LiteralExpr`
- Jméno signálu `NameExpr`

Třída `NameExpr` jejíž objekty představují jmenné operandy ve výrazu, má definovánu metodu connect tak že vrací jméno operandu které představuje spoj nebo port.
Třída `BinOp` obsahuje odkaz na levý a pravý podstrom a jméno operátoru. Metoda `Connect` u binárního operátoru slouží k vytvoření instance primitivní entity (odpovídající operátoru)v architektuře. Metoda funguje tak že nejprve zavolá `Connect` na levý a pravý podstrom tím získá jména spojů které představují vstupy do dané instance primitivní entity. Poté zavolá metodu klienta:

```
StringToken* output = client->haveSymFunc(inputs,operator_,createNet);
```

Tato metoda vytvoří v architektuře instanci primitivní entity, její návratovou hodnotu je jméno výstupního této instance. Toto jméno následně vrátí i metoda `Connect`.
Druhý parametr typu `boolean` metody `Connect` říká zda má být vytvořen port, využívá se toho u třídy `BinOp` a `UnOp`. Pokud je parametr nastaven na `false` nepřipojí se na výstup instance spoj.

Metoda haveSignalAssignment

```
driver.client_->haveSignalAssignment( $1->GenerateName() , $5 );
```

První parametr představuje jméno cílového spoje nebo portu. Druhý parametr je ukazatel na kořen syntaktického stromu výrazu. Metoda nejprve vyhledá zda existuje cílový spoj. Pokud spoj daného jména nebyl nalezen pokusí se vyhledat zda existuje port s takovým jménem, pokud byl nalezen je k němu vytvořen stejně pojmenovaný spoj. Poté je na kořen syntakticého stromu zavolána metoda `Connect s parametrem \verb`CreateNet— nastaveným na `false`. Tím se zajistí že dojde k propojení výrazu. Port který má být připojen k cílovému spoji bude nastaven v proměnné `lastPort_`, tato proměnná byla nastavena během zpracování syntaktického stromu metodou klienta `haveSymFunc` nebo `haveOperand`.
Pokud proměnná `lastPort_` nebyla nastavena znamená to že na pravé straně příkazu přiřazení byl pouze spoj nebo neexistující port a zapojení nemůže být provedeno. Pokud proměnná `lastPort_` byla nastavena je port na který ukazuje připojen k cílovému spoji příkazu přiřazení.

Vytvoření instance primitivní entity

```
StringToken* VhdlImport::haveSymFunc(const list<StringToken>  listInputNames,
            const std::string& func, bool createNet );
```

V tomto případě se budou vytvářet instance entity jejíž architektuta je popsána symetrickou funkcí kterou popisuje třída `SymFunc`. Nejprve je třeba zjistit jestli entita popisující danou symetrickou funkci v knihovně primitiv již neexistuje. Je tedy potřeba vytvořit knihovního agenta symetrické funkce pomocí kterého lze vyhledávat v knihovně:

```
BehavAgent* agent=new SymFuncAgent(decodeFunc(func),listInputNames.size());
```

V knihovně primitiv se pokusíme pomocí tohoto agenta nalézt opovídající popis pokud byl nalezen vráti metoda `Find` ukazatel na požadovanou architekturu. Tato architektura nám poskytne informaci o entitě jíž je podřízena. Z této entity již lze vytvořit instanci požadované symetrické funkce.
Pokud ale nebyla architektura nalezana znamená to že zatím neexistuje entita popisující danou symetrickou funkci a musí tedy výt vytvořena. Nejprve se v knihovně primitiv vytvoří unikátní entita která bude představovat tuto funkci. Poté je k ní vytvořena behaviorální architektura. Její popis vytvoří behaviorální agent který byl vytvořen pro vyhledáváni v knihovně:

```
Architecture* stdArch = stdEnt->CreateArch <BehavArch>("vhdl");
stdEnt->CurrentArch (stdArch);
BehavAech* behavArch = dynamic_cast <BehavArch*> (stdArch->Impl());
behavArch->description (agent->CreateDescr());
```

Nakonec jsou vytvořeny porty rozhraní této entity jeden výstupní port a tolik vstupních portů kolik obsahuje seznam vstupních portů této metody.
V tomto okamžiku již tedy máme entitu odpovídající požadované funkci která byla buď nalezena v knihovně nebo vytvořena a přidána do knihovny. Z této entity je v architektuře vytvořena

instance jejíž jméno je stejné jako jméno entity ale navíc je oindexováno aby byli rozlišeny jednotlivé instance téže entity. Dále je k instanci vytvořen a připojen výstupní spoj pokud byl parametr `createNet` nastaven jako `true`. Proměnná `lastPort_` je nastavena aby ukazovala na výstupní port této instance. Nakonec se prochází seznam vstupních signálů funkce, pro každý prvek tohoto seznamu se pokusí vyhledat zda existuje spoj odpovídající tomuto jménu. Pokud byl nalezen je připojen do vstupního portu této instance. Pokud nalezen nebyl vyhledává se zda existuje port architektury se stejným jménen. V případě že byl nalezen vytvoří se k němu spoj a ten ho propojí se vstupem instance.

7.5.6 Příkaz instalace komponenty

Pokud syntaktický analyzátor rozpozná konstrukci instalace komponenty zavolá nejprve metodu klienta:

```
void haveComponentInst(const StringToken& label, const StringToken& compName,
                       const StringToken& archName);
```

První parametr udává jméno instance, druhý jméno komponenty a třetí jméno architektury s kterou má být instance zapojena. Pokud není architektura specifikována vytvoří se instance s naposled přeloženou architekturou. Tato metoda tedy vyhledá v knihovně entitu odpovídající jménu komponenty a poté k ní vyhledá příslušnou architekturu. Nakonec z ní vytvoří pojmenovanou instanci a ukazatel na tuto instanci uloží do proměnné `theInstance_`.
Příkaz instalace komponenty dále obsahuje část ve které se mapují porty komponenty. Syntaktický analyzátor při zpracovávání pravidla `association_list` vytvoří seznam asociativních elemetů. Pro každý asociativní element je zavolána metoda metoda klienta `havePosPortAssoc` pokud jsou porty mapovány pozičně nebo `havePortAssoc` pokud jsou mapovány jmenným přiřazením.

Poziční mapování

```
virtual void havePosPortAssoc(int portPosition, const StringToken& NetName);
```

Nejprve je v architektuře vyhledán spoj který odpovídá jménu uloženém v parametru `NetName`, pokud není spoj nalezen pokusí se vyhledat port architektury se stejným jménem. Pokud je tento port nalezen vytvoří se k němu příslušný spoj. Dále je potřeba najít port komponenty který se mapuje. Odkaz na aktuálně mapovanou instanci je v proměnné `theInstance_`, zavolání metody find s parametrem udávajícim číslo portu se vyhledá požadovaný port. Pokud je port nalezen je propojen s dříve nalezeným spojem.

Jmenné mapování

```
virtual void havePortAssoc(const StringToken& compPortName,const StringToken& NetName);
```

Funguje stejným způsobem jako poziční mapování akorát porty instance jsou vyhledávány podle jména.

8 Popis Implementace Exportního Systému

8.1 Úvod

Hlavní část exportního systému tvoří třída `VhdlExport` která je potomkem třídy `Export` definující obecné rozhraní exportních tříd. Zároveň je také potomkem dvou tříd které tvoří klienty traverseru a definují virtuální metody pro export sémantických konstrukcí jazyka. Tyto metody jsou volány příslušným traverserem a třída `VhdlExport` je předefinovává. Třída vlastní objekt třídy `VhdlTraverse` která se stará o procházení struktury uložené v HUBu a volání sémantických metod. V konstruktoru exportní třídy může být specifikováno zda se bude exportovat obsah celého HUBu nebo pouze jedné knihovny, entity nebo architektury. Při konstrukci objektu této třídy se volá také konstruktor traverseru kterému se přidá informace o místě v hubu ze kterého bude export probíhat. Export se spustí zavoláním přetíženého operátotu volání funkce jehož parametrem je určení místa v HUBu a jméno souboru.

8.2 Traverzery

Pro export do jazyka VHDL byli definovány dvě třídy které se starají o procházení struktury HUBu.

8.2.1 Traverzer vrchní vrstvy HUBu

Třída `VhdlExpTrav` se stará o procházení vrchní vrstvy HUBu. Jejího klienta tvoří objekt třídy `VhdlExpTravClient`. Tento klient definuje virtuální metody které budou volány traverserem. Odkaz na tohoto klienta je nastaven při konstrukci objektu, je konstruován právě třídou `VhdlExport` která nastaví sama sebe jako klienta traverzeru a tím se docílí toho že budou volány předefinované sémantické metody třídy VhdlExport.
Klient traverzeru procházejícího vrchní vrstvu HUBu je potomkem obecného klienta `TraverserClient`, takže dědí všechny jeho sémantické metody.
Navíc definuje tyto metody:

```
virtual void havePortClause(void);
virtual void endPortClause(void);
```

Kterými říká že začíná a končí část deklarace rozhraní portů entity.
Traverzer definuje tyto metody:

```
void travLib (hub::Library*);
void travEnt (hub::Entity*);
void travPorts(hub::Entity*);
```

Traverzer po spuštění zavolá metodu klienta `haveUnit` a poté pokud není specifikována knihovna ze které má export probíhat volá pro každou knihovnu v HUBu metodu `travLib` která prochází obsah vybrané knihovny. Pokud knihovna byla specifikována volá metodu `travLib` pouze pro tuto knihovnu. Nakonec zavolá metodu klienta `endUnit`.
Metoda `travLib` nejprve zavolá metodu klienta `haveLib` a poté pokud není specifikována entita která má být exportována volá pro každou entitu v této knihovně metodu `travEnt` která slouží k procházení entity. Pokud byla entita specifikována volá se tato metoda pouze pro vybranou entitu. Nakonec je zavolána metoda klienta `endLib` informující o konci procházení dané knihovny.
Metoda `travEnt` volá sémantické metody entity tj. nejprve zavolá metodu klienta `haveEnt` a poté informuje klienta o začátku deklarace portů, zavolá metodu na procházení portů a ukončí část deklarace portů a ukončí deklaraci entity metodou `endEnt`. Pokud nebyla specifikována

architektura která má být exportována projdou se všechny architektury patřící vybrané entitě. Pro každou se volá metoda `haveArch` a `endArch`.

8.2.2 Traverzer strukturní architektury

Třída `VhdlStructTraverser` se stará o procházení strukturní architektury. Jejího klienta tvoří objekt třídy `VhdlStructTravClient`. Klient se používá a konstruuje stejně jako u traverzeru vrchní vrstvy HUBu. Tento klient je potomkem obecného klienta traverzeru strukturní architektury `StructArchTravClient`. Dědí od něj tedy všechny sémantické metody a navíc definuje tyto metody:

```
virtual void haveNet (hub::StructArch* arch,hub::Net*);
virtual void haveComponent(hub::Entity*);
virtual void endComponent(void);
virtual void havePortClause(void); //declaration of component ports
virtual void endPortClause(void);
virtual void havePortProto(hub::PortProto*); // component port
virtual void havePrimInst(hub::Instance*);
```

Konstruktor traverzeru strukturní architektury má dva parametry, prvním je odkaz na strukturní implementaci architektury kterou bude procházet a druhým parametrem je odkaz na klienta který definuje sémantické metody.
Traverzer definuje tyto metody:

```
void travNets (void);
void travPorts (void);
void travInsts (void);
void travComponents(void);

bool isPrimitive(hub::Instance* inst);
```

Tyto metody jsou definovány jako public a jsou volány sémantickou metodou klienta exportující strukturní popis architektury. Jejich funkce je zřejmá z jejich pojmenování.
Zajímavá je akorát metoda `travComponents` která se stará o procházení komponent. V této metodě je vytvořen objekt typu `std::list` do kterého se ukládají odkazy na entity z nichž jsou vytvořeny instance každá entita je tam pouze jednou. Pomocí tohoto se docílí aby nebyla definována komponenta pro jednou entitu vícekrát. Metoda tedy prochází všechny instance architektury a kontroluje zda rodičovská entita existuje v seznamu entit a zda se jedná o instanci primitivní entity (ty se mapují jinak). Pokud není ani jedna z těchto podmínek splněna jsou zavolány metody klienta deklarující tuto komponentu. Poté je odkaz na rodičovskou entitu této instance přidán do seznamu entit.
Metoda `travInsts` prochází instance zapojené v dané architektuře a pro každou zavolá sémantickou metodu která vygeneruje kód zapojení instance. Pro instance primitivních entit je volána sémantická metoda `havePrimInst` která vygeneruje paralelní příkaz přiřazení signálu, pro ostatní instance je volána metoda která generuje příkaz mapování komponenty. Instance primitivních funkcí se poznají podle toho že jejich architektura je popsána objektem třídy `SymFunc`. K tomuto objektu se lze dostat přes aktuálně namapovanou architekturu instance. Metoda poskytující aktuální architekturu instance zde chyběla, po domluvě s vedoucím práce byla do třídy `Instance` přidána : `Architecture* CurrentArch(void)`.

8.3 Export sémanticých konstrukcí jazyka VHDL

O export těchto konstruckí se starají sémantické metody definované ve třídě `VhdlExport`, každá z těchto metod zapisuje do souboru kód v syntaxi jazyka vhdl. Zde je seznam těchto metod:

```
virtual void haveUnit (void);
virtual void endUnit  (void);
virtual void haveLibrary (hub::Library*);
virtual void endLibrary  (hub::Library*);
virtual void haveDesign (hub::Design*);
virtual void endDesign  (hub::Design*);
virtual void haveEntity (hub::Entity*);
virtual void endEntity  (hub::Entity*);
virtual void havePortProto (hub::PortProto*);
virtual void haveArch (hub::Architecture*);
virtual void endArch  (hub::Architecture*);
virtual void havePortClause(void);
virtual void endPortClause(void);

virtual void haveNet (hub::Net*);
virtual void haveNet (hub::StructArch* arch,hub::Net*);
virtual void endNet  (hub::Net*);
virtual void haveJoined (hub::Joinable*);
virtual void havePort (hub::Port <hub::StructArch>*);
virtual void haveInst (hub::Instance*);
virtual void havePrimInst(hub::Instance*);

virtual void haveComponent( hub::Entity* component );
virtual void endComponent(void);
```

Jejich význam je zřejmý z jejich pojemenování. Metoda `haveLibrary` navíc otevře nový soubor pojemnovanný podle jména knihovny a na jeho začátek vypíše VHDL příkaz pro zviditelnění standartních knihoven. Metoda `endLibrary` tento soubor zavře.
Metoda `haveArch` starající se o export architektury zjišťuje zda je daná architektura popsána strukturně nebo behaviorálně:

```
StructArch* sa = a->Impl <StructArch> ();
if (sa) {
   //projdi strukturni arch
}
BehavArch* ba = a->Impl <BehavArch> ();
if (ba) {
   //najdi funkci popisujici danou architekturu
}
```

8.3.1 Popis architektury strukturou

Pokud je popsána strukturně vytvoří se objekt třídy `VhdlStructTraverser` pomocí kterého se tato architektura bude procházet. Nejprve je zavolána metoda která projde jednotlivé spoje, pro každý spoj je vytvořena deklarace signálu. Při deklaraci signálu se používají pomocné metody třídy VHDL export:

```
hub::Name augmentNetName(hub::StructArch* arch,hub::Net* net);
```

```
hub::Name generateNetName(hub::StructArch* arch,hub::Net* net);
bool isPort(hub::Net* net);
```

Signál deklaruje metoda `haveNet (hub::StructArch* arch,hub::Net* net)` ve které je nejprve zavolána metoda která zjistí zda se nejedná o spoj který je propojen se stejně pojmenovaným portem, pokud je takto propojený deklarace se neprovede. Pokud není takto propojený vygeneruje se jeho deklarace a jeho jménu vypíše metoda `augmentNetName` která kontroluje zda rozhraní architektury neobsahuje stejně pojmenovanný port, pokud jej obsahuje přidá ke jménu spoje index.
Poté je zavolána metoda procházející komponenty, pro každou komponentu je vytvořena deklarace komponenty. Jméno komponenty odpovídá jménu entity která je na komponentu nakonfigurována.
V příkazové části architektury jsou zavolány metody které procházejí instance a porty. Pro generování kódu zapojení instance v architektře se používají dvě metody:

```
virtual void haveInst (hub::Instance*);
virtual void havePrimInst(hub::Instance*);
```

Metoda `haveInst` vygeneruje vhdl kód instalace komponenty s mapováním portů. Tato metoda funguje tak že prochází porty instance a vygeneruje jméno tohoto portu namapované na spoj ke kterému je port připojený. Tento spoj se získá metodou `Connected` zavolanou na daný port Instance. Tato metoda nebyla součástí HUBu, po domluvě s vedoucím byla do šablony `Port` přidána.
Metoda `havePrimInst` slouží k vygenerování kódu instancí které jsou instancemi primitivních entit (and, xor, nand, ...). Pro tyto instance je vygenerován VHDL příkaz přiřazení signálu.
Pro každý spoj ke kterému je připojen nějaký port instance je volána metoda `generateNetName` která vytvoří jeho jméno.
Nakonec se projdou porty rozhraní architektury a pro porty ke kterým nejsou připojeny stejně pojmenovanné spoje se vygeneruje příkaz přiřazení.

8.3.2 Popis architektury chováním

Pokud je architektura popsána behaviorálně, získá se ukazatel na její konkrétní popis (potomek třídy `BehavDescr`). Pokud tato architektura má nějaký popis je zavolána metoda perform na objektu exportní mapy, která vyhledá funkci reprezentující získaný typ behaviorálního popisu a spustí tuto funkci s jedním parametrem ukazujícím na exportní třídu (`VhdlExport`) a druhým parametrem ukazujícím na získaný popis. Tato funkce vygeneruje popis chování podle typu třídy popisu. Zatím je implementován pouze popis chování symetrických funkcí (`SymFunc`).

9 Testování

9.1 Testovací aplikace

Na přiloženém cd je zkompilovaný projekt pro **MSVC 2003** skládá se ze struktury hubu a aplikace sloužící pro práci s touto strukturou. Zdrojový kód této aplikace je v souboru `MSVC03_project/hub/apps/circtr.cpp`. Zdrojové kódy HUBu jsou ve složce `MSVC03_project/hub/`, zdrojové kódy importního a exportního VHDL modulu jsou uloženy ve složce `MSVC03_project/hub/vhdl`.
Přeložená konzolová aplikace pro **Win32** je rovněž přiložena na CD:
(`MSVC03_project/hub/apps/circtr/circtr.exe`).
Tato aplikace umožňuje importovat specifikaci návrhu, zapsanou v jednom z podporavných vstupních formátů, do struktury **HUBu** a poté exportovat tuto strukturu do jednoho z podporovaných výstupních formátů. Podporované vstupní formáty:

- EDIF
- BENCH
- VHDL

Podporované výstupní formáty:

- dbg (kontrolní výpis struktury uložené v HUBu)
- VHDL

Jedná se o konzolovou aplikaci, spustí se příkazem circctr s příslušnými parametry. Parametry mohou být následující:

```
circctr -I jmeno_souboru.xxx -l jm_knih -e jm_ent -a jm_arch
        -O jmeno_vyst_souboru.xxx -l jm_knih -e jm_ent -a jm_arch
```

Po parametru **I** následuje jméno vstupního souboru, o jaký se jedná formát se rozpozná podle koncovky tohoto souboru. Parametry l,e,a následující za parametrem pro import určují jméno knihovny,entitym a architektury importované struktury. U importu z **EDIFu** není na tyto specifikace brán zřetel protože **EDIF** tyto pojmy zná. U importu z **VHDL** je možné nastavit pouze jméno knihovny, další pojmy **VHDL** rovněž zná. U importu z **BENCHe** je možné nastavit všechny tyto specifikace.
Po parametru **O** následuje jméno výstupního souboru, o jaký se jedná formát se rozpozná podle koncovky tohoto souboru. Parametry l,e,a následující za parametrem pro export definují jaká knihovna, entita nebo architektura bude exportována.
U exportu do **VHDL** se export každé knihovny provádí do zvláštního souboru pojmnenovaného stejně jako daná knihovna.
V případě importu z jazyka VHDL kdy je návrh strukturovaný a každá entita je uložena v jiném souboru lze použít specifikaci importovaného souboru vícekrát za sebou. Nejprve se definují soubory obsahující entity vnořené nejhlouběji v návrhu a poté ty na vyšších urovní které ve svých strukturách obsahují dříve zmíněné entity.
Příklad:

```
circtr -I half_adder.vhdl -I adder.vhdl -I four_bit_adder.vhdl -o out.dbg
```

9.2 Import

Při testování jsem postupoval podle gramatiky jazyka **VHDL** a jednotlivé konstrukce jsem se pokoušel naimportovat do **HUBu**. Ty části které nejsou naimplementovány ohlásily chybu

včetně popisu chyby a místa v souboru kde se neimplementovaná konstrukce vyskytla. U částí které jsou naimplementované fungovalo vše tak jak bylo stanoveno v analýze, žádné neočekávané události nenastaly. Je akorát potřeba si dávat pozor na propojování portů, import nekontroluje v jakém jsou porty módu, lze tedy např. propojit dva vstupní porty mezi sebou.
V příloze A.1.1 na straně 47 je strukturní návrh 4-bitové sčítačky popsané v jazyce **VHDL**. Tato sčítačka je složena s komponent úplné sčítačky a půlsčítačky. Tato struktura byla naimportována do HUBu, na straně 49 je dbg výstup této struktury uložené v HUBu. V tomto výstupu jde přesně vidět jakým způsobem se jednotlivé konstrukce jazyka **VHDL** importují do struktury **HUBu**. Tento ukázkový kód zahrnuje většinu prvků jazyka VHDL které byly naimplementovány.

9.3 Export

Výstup do formátu **VHDL** jsem zkoušel se strukturami uloženými v **HUBu**, které byly předtím naimportovány ze souborů obsahujících návrhy obvodů v libovolných podporovaných vstupních formátech. Funkčnost vygenerovaného kódu ve formátu **VHDL** jsem ověřoval v prostředí **XILINX ISE** [2]. Ve všech případech se podařilo vygenerovaný kód vysyntetizovat. V příloze B.1 na straně 53 je ukázka návrhu uloženého v **HUBu** a za ní následuje vygenerovaný **VHDL** kód který odpovídá tomuto návrhu.
Další testovací vstupní soubory a jejich výstupy do dbg a VHDL jsou na přiloženém CD ve složce `Testy`.

10 Závěr

Tato bakalářská práce se skládala z implementace dvou na sobě nezávislých modůlu, importního a exportního modulu jazyka VHDL, do struktury HUB pro návrh číslicových obvodů která je součástí výukového EDA systému. Oba tyto moduly se podařilo navrhnout, implementovat a otestovat. Při práci na těchto modulech nebylo kromě dvou případů (str. 38, 40) nutno měnit stávající zdrojové kódy HUBu.
Importní modul dovede analyzovat kompletní gramatiku jazyka VHDL ale implementována je pouze podmnožina tohoto jazyka a to taková aby bylo možno do HUBu importovat strukturně navržené obvody. Nelze tedy importovat libovolný obvod navržený ve VHDL ale je třeba používat pouze konstrukce které jsou naimplementovány (viz. kapitola 4). Návrhy ve VHDL je tedy nutno psát s ohledem na podporovanou část jazyka VHDL.
Exportní modul umožňuje exportovat návrh uložený v HUBu do jazyka VHDL. Vygenerovaný návrh v jazyce VHDL obsahuje pouze takové konstrukce, které podporuje importní modul. Tím pádem je možno vygenerovaný kód opět importovat zpět do HUBu. Vygenerovaný kód se úspěšně podařilo vysyntetizovat v prostředí XILINX (viz. kapitola 9.3).
Přínosem této práce pro mně byla hlavně zkušenost s prací na cizím zdrojovém kódu. Díky přehlednému a logickému návrhu struktury HUBu jsem se brzy v kódu zorientoval a mohl jej využívat při implementaci přídavných modulů. Také jsem se díky této práci seznámil z jazykem VHDL který jsem přetím znal pouze zčásti.

11 Seznam literatury

[1] Vhdl language reference guide
. https://service.felk.cvut.cz/doc/vhdl/Refguide.htm.

[2] Xilinx ise web pack - fpga design tool, 2006. http://www.xilinx.com.

[3] C. Donelly and R. Stallman. Manual for gnu bison, 2005. (na přiloženém CD).

[4] IEEE. *VHDL Language Reference Manual*, volume 1. 2000. (na přiloženém CD).

[5] J. Schmidt. Circuit hub - dokumentace. (na přiloženém CD).

A Import VHDL

A.1 Vstupní soubor

A.1.1 Poloviční sčítačka - halfadder.vhdl

```
library IEEE;
use ieee.Std_Logic_1164.all;
use ieee.numeric_std.all;

entity halfadder is
  port (
     A: in std_logic;
     B: in std_logic;
     S: out std_logic;
     Carry: out std_logic
    );
end halfadder;

ARCHITECTURE halfadder_arch OF halfadder IS

BEGIN
S <= A xor B;
Carry <= A and B;

END halfadder_arch;
```

A.1.2 1-bitová úplná sčítačka - adder.vhdl

```
library IEEE;
use ieee.Std_Logic_1164.all;
use ieee.numeric_std.all;

entity adder is
  port (
     A: in std_logic;
     B: in std_logic;
     C: in std_logic;
     SUM: out std_logic;
     Carry: out std_logic
    );
end adder;

ARCHITECTURE adder_arch OF adder IS

signal c1, c2, s: std_logic;

component halfadder
  port (
     A: in std_logic;
     B: in std_logic;
```

```
     S: out std_logic;
     Carry: out std_logic
    );
end component;

BEGIN
H1_map: halfadder
port map(A => A,
B => B,
Carry => c1,
S => s);
H2_map: halfadder
port map(A => C,
B => s,
Carry => c2,
S => SUM);

Carry <= c1 or c2;

END adder_arch;
```

A.1.3 4-bitová sčítačka - 4bitAdder.vhdl

```
library IEEE;
use ieee.Std_Logic_1164.all;
use ieee.numeric_std.all;

entity FourBitAdder is
  port (
     IN_1: in std_logic_vector(3 downto 0);
     IN_2: in std_logic_vector(3 downto 0);
     SUM: out std_logic_vector(3 downto 0);
     Carry: out std_logic
    );
end FourBitAdder;

ARCHITECTURE FourBitAdder_arch OF FourBitAdder IS

signal c: std_logic_vector(2 downto 0);

component halfadder
  port (
     A: in std_logic;
     B: in std_logic;
     S: out std_logic;
     Carry: out std_logic
    );
end component;

component adder
  port (
```

```
     A: in std_logic;
     B: in std_logic;
     C: in std_logic;
     SUM: out std_logic;
     Carry: out std_logic
    );
end component;

BEGIN
H_map: halfadder
port map(A => IN_1(0),
B => IN_2(0),
Carry => c(0),
S => SUM(0));
A1_map: adder
port map(A => IN_1(1),
B => IN_2(1),
C => c(0),
Carry => c(1),
SUM => SUM(1));
A2_map: adder
port map(A => IN_1(2),
B => IN_2(2),
C => c(1),
Carry => c(2),
SUM => SUM(2));
A3_map: adder
port map(A => IN_1(3),
B => IN_2(3),
C => c(2),
Carry => Carry,
SUM => SUM(3));

END FourBitAdder_arch;
```

A.2 Výstupní soubor

A.2.1 Dbg výstup

```
+unit
    +library primitives
+entity and2
    =port I1 attr in
    =port I2 attr in
    =port O attr out
    +arch vhdl attr behav
symmetric func AND, 2 inputs
    -arch vhdl
-entity and2
+entity or2
    =port I1 attr in
```

```
    =port I2 attr in
    =port O attr out
    +arch vhdl attr behav
symmetric func OR, 2 inputs
    -arch vhdl
-entity or2
+entity xor2
    =port I1 attr in
    =port I2 attr in
    =port O attr out
    +arch vhdl attr behav
symmetric func XOR, 2 inputs
    -arch vhdl
-entity xor2
    -library primitives
    +library work
+entity adder
    =port A attr in
    =port B attr in
    =port C attr in
    =port Carry attr out
    =port SUM attr out
    +arch adder_arch attr struct
=instance H1_map halfadder
=instance H2_map halfadder
=instance or2_1 or2.primitives
+net A A, A.H1_map
-net A
+net B B, B.H1_map
-net B
+net C C, A.H2_map
-net C
+net c1 Carry.H1_map, I1.or2_1
-net c1
+net c2 Carry.H2_map, I2.or2_1
-net c2
+net Carry Carry, O.or2_1
-net Carry
+net s S.H1_map, B.H2_map
-net s
+net SUM SUM, S.H2_map
-net SUM
    -arch adder_arch
-entity adder
+entity FourBitAdder
    =port Carry attr out
    =port IN_1_0 attr in
    =port IN_1_1 attr in
    =port IN_1_2 attr in
    =port IN_1_3 attr in
```

```
    =port IN_2_0 attr in
    =port IN_2_1 attr in
    =port IN_2_2 attr in
    =port IN_2_3 attr in
    =port SUM_0 attr out
    =port SUM_1 attr out
    =port SUM_2 attr out
    =port SUM_3 attr out
    +arch FourBitAdder_arch attr struct
=instance A1_map adder
=instance A2_map adder
=instance A3_map adder
=instance H_map halfadder
+net c_0 Carry.H_map, C.A1_map
-net c_0
+net c_1 Carry.A1_map, C.A2_map
-net c_1
+net c_2 Carry.A2_map, C.A3_map
-net c_2
+net Carry Carry, Carry.A3_map
-net Carry
+net IN_1_0 IN_1_0, A.H_map
-net IN_1_0
+net IN_1_1 IN_1_1, A.A1_map
-net IN_1_1
+net IN_1_2 IN_1_2, A.A2_map
-net IN_1_2
+net IN_1_3 IN_1_3, A.A3_map
-net IN_1_3
+net IN_2_0 IN_2_0, B.H_map
-net IN_2_0
+net IN_2_1 IN_2_1, B.A1_map
-net IN_2_1
+net IN_2_2 IN_2_2, B.A2_map
-net IN_2_2
+net IN_2_3 IN_2_3, B.A3_map
-net IN_2_3
+net SUM_0 SUM_0, S.H_map
-net SUM_0
+net SUM_1 SUM_1, SUM.A1_map
-net SUM_1
+net SUM_2 SUM_2, SUM.A2_map
-net SUM_2
+net SUM_3 SUM_3, SUM.A3_map
-net SUM_3
    -arch FourBitAdder_arch
-entity FourBitAdder
+entity halfadder
    =port A attr in
    =port B attr in
```

```
    =port Carry attr out
    =port S attr out
    +arch halfadder_arch attr struct
=instance and2_1 and2.primitives
=instance xor2_1 xor2.primitives
+net A A, I1.xor2_1, I1.and2_1
-net A
+net B B, I2.xor2_1, I2.and2_1
-net B
+net Carry Carry, O.and2_1
-net Carry
+net S S, O.xor2_1
-net S
    -arch halfadder_arch
-entity halfadder
    -library work
-unit
```

B Export do VHDL

B.1 Struktura uložená v HUBu

```
+unit
    +library bench_primitives
+entity and3
    =port I1 attr in
    =port I2 attr in
    =port I3 attr in
    =port O attr out
    +arch bench attr behav
symmetric func AND, 3 inputs
    -arch bench
-entity and3
+entity dff1
    =port D attr in
    =port O attr out
    =port Q attr out
    +arch bench attr behav
RTL register
    -arch bench
-entity dff1
+entity nand2
    =port I1 attr in
    =port I2 attr in
    =port O attr out
    +arch bench attr behav
symmetric func NAND, 2 inputs
    -arch bench
-entity nand2
+entity nor2
    =port I1 attr in
    =port I2 attr in
    =port O attr out
    +arch bench attr behav
symmetric func NOR, 2 inputs
    -arch bench
-entity nor2
+entity not1
    =port I1 attr in
    =port O attr out
    +arch bench attr behav
symmetric func INV, 1 inputs
    -arch bench
-entity not1
+entity or2
    =port I1 attr in
    =port I2 attr in
    =port O attr out
```

```
    +arch bench attr behav
symmetric func OR, 2 inputs
    -arch bench
-entity or2
+entity or3
    =port I1 attr in
    =port I2 attr in
    =port I3 attr in
    =port O attr out
    +arch bench attr behav
symmetric func OR, 3 inputs
    -arch bench
-entity or3
    -library bench_primitives
    +library work
+entity s27
    =port G0 attr in
    =port G1 attr in
    =port G17 attr out
    =port G2 attr in
    =port G3 attr in
    +arch bench attr struct
=instance G10 nor2.bench_primitives
=instance G11 nor2.bench_primitives
=instance G12 nor2.bench_primitives
=instance G13 nor2.bench_primitives
=instance G14 not1.bench_primitives
=instance G15 or3.bench_primitives
=instance G16 or2.bench_primitives
=instance G17 not1.bench_primitives
=instance G5 dff1.bench_primitives
=instance G6 dff1.bench_primitives
=instance G7 dff1.bench_primitives
=instance G8 and3.bench_primitives
=instance G9 nand2.bench_primitives
+net G0 G0, I1.G14, I3.G15
-net G0
+net G1 G1, I1.G12
-net G1
+net G10 D.G5, O.G10
-net G10
+net G11 D.G6, I1.G17, I2.G10, O.G11
-net G11
+net G12 I1.G15, O.G12, I2.G13
-net G12
+net G13 D.G7, O.G13
-net G13
+net G14 O.G14, I1.G8, I1.G10
-net G14
+net G15 O.G15, I2.G9
```

```
-net G15
+net G16 O.G16, I1.G9
-net G16
+net G17 G17, O.G17
-net G17
+net G2 G2, I1.G13
-net G2
+net G3 G3, I1.G16
-net G3
+net G5 O.G5, I1.G11
-net G5
+net G6 O.G6, I2.G8
-net G6
+net G7 O.G7, I3.G8, I2.G12
-net G7
+net G8 O.G8, I2.G15, I2.G16
-net G8
+net G9 O.G9, I2.G11
-net G9
    -arch bench
-entity s27
    -library work
-unit
```

B.2 Výstup ve formátu VHDL

B.2.1 Knihovna work

```
library IEEE;
use ieee.Std_Logic_1164.all;
use ieee.numeric_std.all;

entity s27 is
  port (
    G0 : in std_logic;
    G1 : in std_logic;
    G17 : out std_logic;
    G2 : in std_logic;
    G3 : in std_logic
  );
end entity s27;

architecture bench of s27 is
  signal G10 : std_logic;
  signal G11 : std_logic;
  signal G12 : std_logic;
  signal G13 : std_logic;
  signal G14 : std_logic;
  signal G15 : std_logic;
  signal G16 : std_logic;
  signal G5 : std_logic;
```

```
  signal G6 : std_logic;
  signal G7 : std_logic;
  signal G8 : std_logic;
  signal G9 : std_logic;

  component dff1
    port (
      D : in std_logic;
      O : out std_logic;
      Q : out std_logic
    );
  end component;

begin

  G10 <= G14 nor G11;
  G11 <= G5 nor G9;
  G12 <= G1 nor G7;
  G13 <= G2 nor G12;
  G14 <= not G0;
  G15 <= G12 or G8 or G0;
  G16 <= G3 or G8;
  G17 <= not G11;
  G5: dff1
   port map(
     O => G5,
     D => G10,
     Q => open
   );
  G6: dff1
   port map(
     O => G6,
     D => G11,
     Q => open
   );
  G7: dff1
   port map(
     O => G7,
     D => G13,
     Q => open
   );
  G8 <= G14 and G6 and G7;
  G9 <= G16 nand G15;
end architecture bench;
```

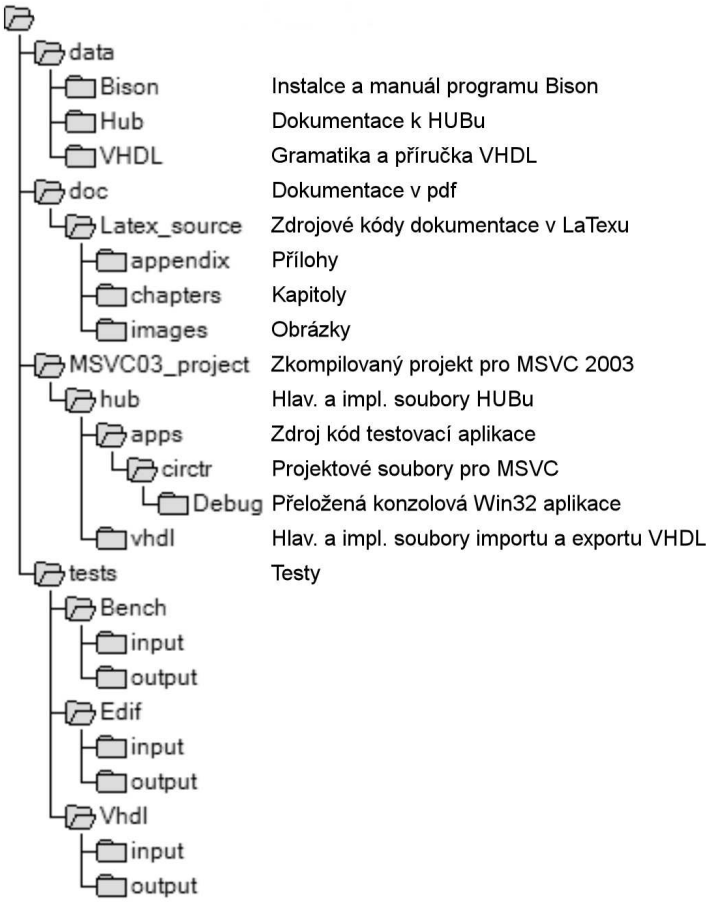
B.2.2 Knihovna bench_primitives

```
library IEEE;
use ieee.Std_Logic_1164.all;
use ieee.numeric_std.all;
```

```
entity and3 is
  port (
    I1 : in std_logic;
    I2 : in std_logic;
    I3 : in std_logic;
    O : out std_logic
  );
end entity and3;

architecture bench of and3 is
begin
  O <= I1 and I2 and I3;
end architecture bench;

entity dff1 is
  port (
    D : in std_logic;
    O : out std_logic;
    Q : out std_logic
  );
end entity dff1;

architecture bench of dff1 is
begin
  -- unknown architecture
end architecture bench;

entity nand2 is
  port (
    I1 : in std_logic;
    I2 : in std_logic;
    O : out std_logic
  );
end entity nand2;

architecture bench of nand2 is
begin
  O <= I1 nand I2;
end architecture bench;

entity nor2 is
  port (
    I1 : in std_logic;
    I2 : in std_logic;
    O : out std_logic
  );
end entity nor2;

architecture bench of nor2 is
```

```
begin
  O <= I1 nor I2;
end architecture bench;

entity not1 is
  port (
    I1 : in std_logic;
    O : out std_logic
  );
end entity not1;

architecture bench of not1 is
begin
  O <= not I1;
end architecture bench;

entity or2 is
  port (
    I1 : in std_logic;
    I2 : in std_logic;
    O : out std_logic
  );
end entity or2;

architecture bench of or2 is
begin
  O <= I1 or I2;
end architecture bench;

entity or3 is
  port (
    I1 : in std_logic;
    I2 : in std_logic;
    I3 : in std_logic;
    O : out std_logic
  );
end entity or3;

architecture bench of or3 is
begin
  O <= I1 or I2 or I3;
end architecture bench;
```

C Obsah přiloženého CD



Obrázek C.1: Obsah přiloženého CD