VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2022 Václav Pastušek

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV RADIOELEKTRONIKY

DEPARTMENT OF RADIO ELECTRONICS

PROSTŘEDÍ PRO NÁVRH DIGITÁLNÍCH OBVODŮ S VYUŽITÍM VLASTNÍHO JAZYKA TYPU HLS

HLS DEVELOPMENT TOOL FOR DSP WITH CUSTOM PROGRAMMING LANGUAGE

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — Václav Pastušek

AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. Lukáš Fujcik, Ph.D.

SUPERVISOR

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Elektronika a komunikační technologie**

Ústav radioelektroniky

Student: Václav Pastušek ***ID:*** 204437

Ročník: 3 ***Akademický rok:*** 2021/22

NÁZEV TÉMATU:

Prostředí pro návrh digitálních obvodů s využitím vlastního jazyka typu HLS

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s vysokoúrovňovým popisem digitálních obvodů pro zpracování signálů. V rámci semestrálního projektu bude vytvořen programovací jazyk, systém knihoven a překladač. Dále bude zpracována podrobná dokumentace jazyka a průvodní zpráva.

Navrhněte prostředí, které bude obsahovat vlastní programovací jazyk, jehož syntaxe bude podobná např. skriptu pro simulace v prostředí PSpice. V rámci práce bude vytvořena sada knihoven pro typické obvodové prvky využívané v DSP, např. násobení, sčítání, zpoždění, převod číselných soustav, butterfly blok apod. Dále bude možné toto prostředí rozšiřovat o vlastní knihovny.

Výstupem z vytvořeného vývojového prostředí budou kódy v jazyce VHDL a soubory pro nastavení parametrů cílového FPGA (constraints) pro platformu Xilinx ISE a Xilinx VIVADO. Uživatelské rozhraní bude textové nebo GUI. Ověřte funkci na reálném hardwarovém řešení, tj. ADC, FPGA a DAC. Pro demonstraci funkce bude vytvořena minimálně jedna kompletní aplikace DSP pomocí navrženého vývojového prostředí, např. FFT, FIR filter, nebo jiné.

DOPORUČENÁ LITERATURA:

[1] Alexander Meduna, Automata and Languages: Theory and Applications, 2000, Springer

[2] Peter Ashenden, Designer's Guide to VHDL 3rd edition, 2008, Elsevier Science & Technology

Termín zadání: 11.2.2022 ***Termín odevzdání:*** 1.6.2022

Vedoucí práce: doc. Ing. Lukáš Fujcik, Ph.D.

doc. Ing. Lucie Hudcová, Ph.D.
předseda rady studijního programu

ABSTRAKT

V dnešní době existuje spoustu různých vysokoúrovňových syntéz pro popis digitálních obvodů. Ty nejznámější pak generují VHDL kód z programovacích jazyků jako jsou např.: ANSI C, C++, SystemC, SystemVerilog a MATLAB. Ale ne každý se ztotožní s programováním toho typu, proto je občas dobré přejít na vyšší úroveň abstrakce, kdy se schová vnitřní část komponentů, a pak se dané komponenty volají se vstupy a výstupy. Tato práce se zabývá problematikou návrhu HLS, návrhem vstupního pseudokódu, pseudoknihoven, překladače vytvořeném v jazyce Python, jeho moduly a praktickým použitím.

KLÍČOVÁ SLOVA

VHDL, HLS, MyHDL, FPGA, vysokoúrovňová syntéza, překladač, pseudokód, Python

ABSTRACT

Nowadays, there are many different high-level syntheses for describing digital circuits. The best known ones generate VHDL code from programming languages such as ANSI C, C++, SystemC, SystemVerilog and MATLAB. But not everyone will identify with that type of programming, so sometimes it's good to go to a higher level of abstraction, where the internals of the components are hidden, and then the components are called with inputs and outputs. This thesis deals with the design of HLS, the design of input pseudocode, pseudo-libraries, compiler created in Python, its modules and practical application.

KEYWORDS

VHDL, HLS, MyHDL, FPGA, high-level synthesis, compiler, pseudocode, Python

PASTUŠEK, Václav. *Prostředí pro návrh digitálních obvodů s využitím vlastního jazyka typu HLS*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav radioelektroniky, 2022, 44 s. Bakalářská práce. Vedoucí práce: doc. Ing. Lukáš Fujcik, Ph.D.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Václav Pastušek
VUT ID autora:	204437
Typ práce:	Bakalářská práce
Akademický rok:	2021/22
Téma závěrečné práce:	Prostředí pro návrh digitálních obvodů s využitím vlastního jazyka typu HLS

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu doc. Ing. Lukáši Fujcikovi Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Obsah

Seznam obrázků

Seznam výpisů

Úvod

V dnešní době jsou kladeny nároky na rychlost a jednoduchost návrhu logických integrovaných obvodů. Tyto integrované obvody můžeme dělit na IC (Monolithic Integrated Circuit - monolitický integrovaný obvod) a PLD (Programmable Logic Device - programovatelný logický obvod). Monolitické IC se skládají z logických hradel a mají pevně stanovenou funkci, oproti tomu PLD jsou elektronické součástky, které se používají k vytváření rekonfigurovatelných digitálních obvodů a musí být naprogramovány pomocí specifického programu k tomu určenému. IC můžeme dělit na analogové, digitální a smíšené podle typů obvodů, které pak pracují s analogovým nebo digitálním signálem. S IC se nejčastěji setkáváme v podobě ASIC (Application Specific Integrated Circuit - zákaznický integrovaný obvod) nebo jako digitální paměťové čipy. PLD dělíme na PLA (Programmable logic array - programovatelné logické pole), PAL (Programmable Array Logic - programovatelné logické pole), GAL (Generic array logic - obecná logická pole), CPLD (Complex Programmable Logic Device - komplexní programovatelné logické zařízení), FPGA (Field Programmable Gate Array - programovatelná hradlová pole) a EPLD (Electronically programmable logic devices - elektronicky programovatelná logická zařízení) [1][2].

Prvně se PLD programovalo jazyky nižší úrovně HDL (Hardware Description Language - programovací jazyk pro popis hardwaru) jako jsou ABEL, CUPL a PALASM. S postupem času a s důrazem na rychlost vývoje se přešlo na trochu vyšší abstrakci s jazyky RTL (Register Transfer Language - jazyk pro přenos registrů) jako jsou VHDL (VHSIC Hardware Description Language - programovací jazyk pro popis hardwaru s velmi rychlými integrovanými obvody) a Verilog. Bohužel i tyto jazyky narážejí na hranu svých možností z hlediska abstrakce a proto se vyvíjí překladače, které překládají jazyky vyšší úrovně na úroveň RTL, tomuto překladu se říka HLS (High-level Synthesis - vysokoúrovňová syntéza). Z průzkumu provedeného v prosinci roku 2015 se nejvíce používaly vstupní jazyky pro HLS: C, C++ a jeho varianty, BSV, MaxJ, BDL a upuštěno bylo u jazyků Java a MATLAB [3]. Jejich použití dokáže značně zrychlit návrh a verifikaci logických integrovaných obvodů, avšak je potřeba znát docela podrobně daný vstupní programovací jazyk.

Hlavní myšlenkou této práce je vyvinout jednodušší prostředí HLS, než je pro zmiňované jazyky, kdy není potřeba vědět C, C++ ani VHDL, ale stačí se naučit jednoduchý pseudokód, který je inspirovaný jazykem SPICE (Simulation Program with Integrated Circuit Emphasis - simulační program s důrazem na integrované obvody) a bude podrobně popsán níže viz str. 23. HLS bude pro přehlednost napsána v jazyce Pythonu, která bude brát vstupní pseudokód, pseudoknihovny (pro začátečníky nepodstatné) a vstupní parametry, které budou popsány níže viz str. 22. Výstupem bude funkční podmnožina VHDL kódu.

1 Teorie překladačů

Překladač (kompilátor) je počítačový program, který převádí kód z jednoho programovacího jazyka na kód v jiném programovacím jazyku. Většinou se tak jedná o překlad z jazyka vyšší úrovně na úroveň nižší jako je HLS za účelem zjednodušení práce s jazykem na nižší úrovni.

Překladače můžeme dělit na klasické a konverzační. Klasický překladač se používá nejčastěji. Na vstup se pošle už celý kód a většinou není možné jej v průběhu činnosti měnit, kromě terminace programu. Oproti tomu konverzační překladač dokáže částečně komunikovat s uživatelem během překladu, tzv. přijímat další a další řádky vstupního programu a za pochodu ho zpracovávat. Tento překladač můžeme dále dělit na interpretační (provádí hned i interpretaci neboli vykonává kód) a kompilační (generuje cílový kód nebo intermediální kód).

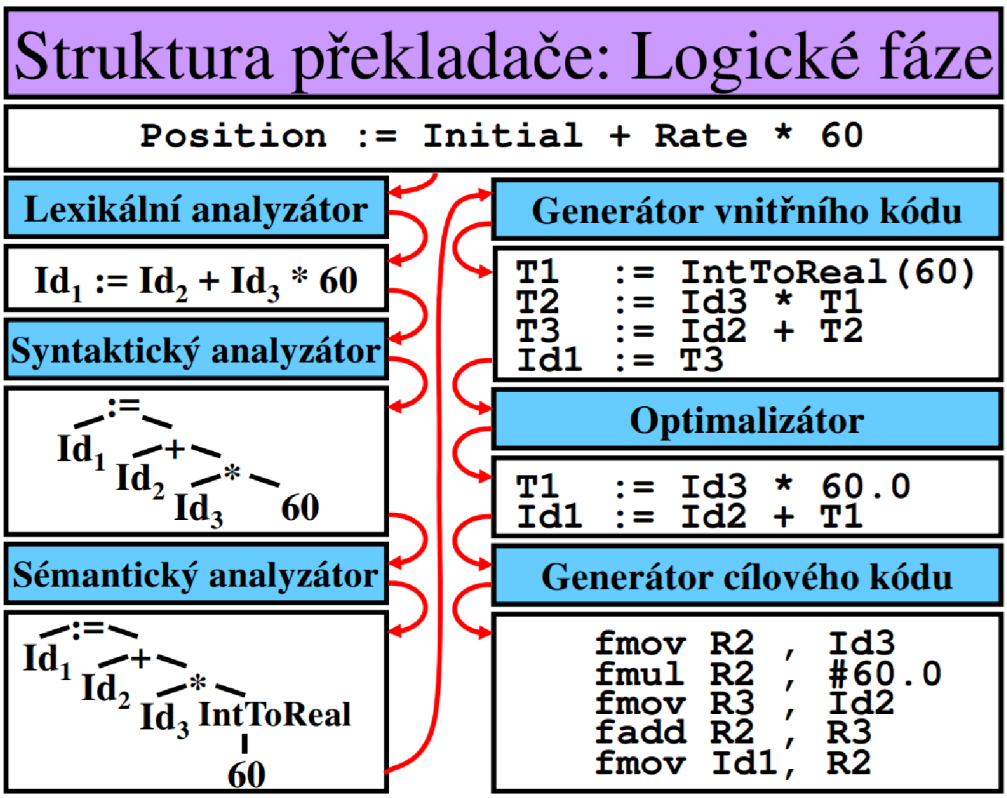
Lehce odlišným typem překladače je dekompilátor, který dělá pravý opak překladače, kdy se provádí překlad z nízkoúrovňového jazyka na jazyk vyšší úrovně.

Následující informace vychází primárně z knihy [4] a výukových materiálů [5] od Prof. RNDr. Alexandra Meduny CSc, dostupné na http://www.fit.vutbr.cz/~meduna/work/doku.php?id=lectures:books:eocd_teach#lectures_pdf.

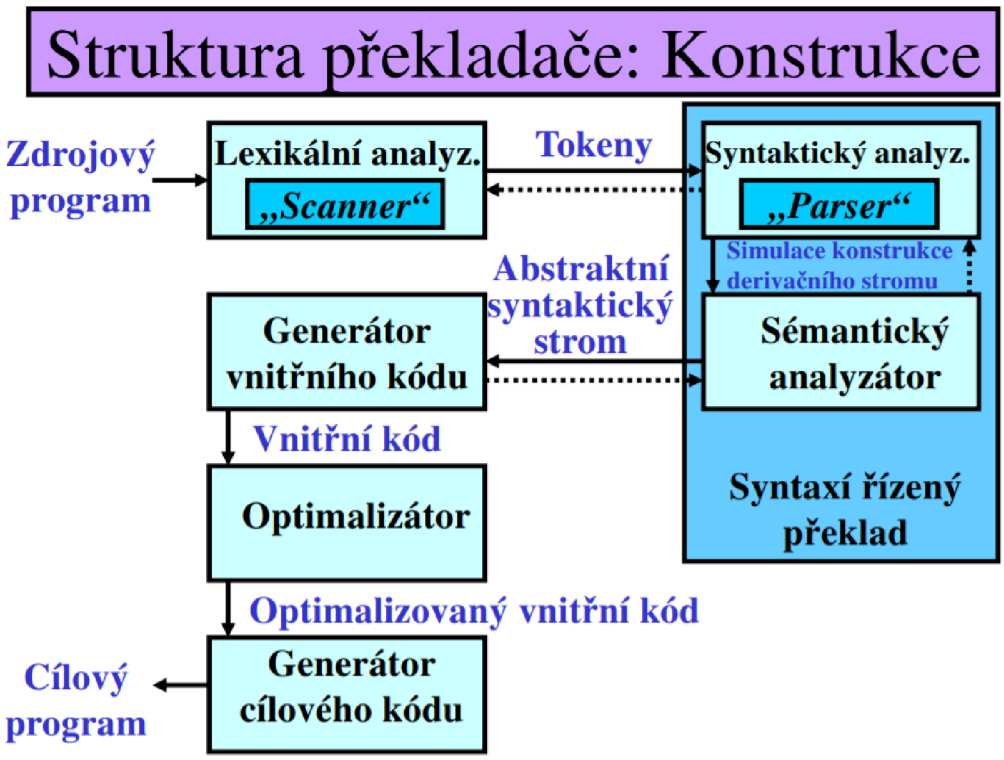
1.1 Části překladače

Překladač se dá dělit na fron-tend, middle-end a back-end. Front-end obsahuje lexikální, syntaktický a sémantický analyzátor, do kterých nemá uživatel většinou přístup. Middle-end obsahuje generátor vnitřního kódu a optimalizátor, kdy je možné se podívat na výstupní intermediální kód a poslední částí je back-end, ve kterém je generátor cílového kódu a uživatel se zde dostává k požadovanému cílovému kódu.

Standardní struktura logické fáze překladače viz obr. 1.1 a konstrukce viz obr. 1.2.



Obr. 1.1: Struktura překladače - Logická fáze



Obr. 1.2: Struktura překladače - Konstrukce

1.1.1 Lexikální analyzátor

Lexikální analýza je činnost, při které lexikální analyzátor načte zdrojový program po znacích a spojí jej do lexémů. Tyto lexémy jsou pak reprezentovány tokeny, které mohou mít další atributy a ty pak putují do syntaktického analyzátoru. Lexikální analýza ve většině případů odstraňuje bílé znaky a komentáře (výjimkou může být třeba ezoterický programovací jazyk Whitespace, kdy se jeho syntaxe sestává jen z bílých znaků) a komunikuje s tabulkou symbolů, která je často řešena zásobníkem, aby nedošlo k záměně stejného názvu proměnné při volání různých procedur na různých úrovních.

Lexikální analyzátor je většinou realizován konečným automatem a pro kontrolu lexému se často používají regulární výrazy.

1.1.2 Syntaktický analyzátor

Syntaktická analýza je činnost, při které syntaktický analyzátor kontroluje syntaktickou správnost řetězce přijatých tokenů. Nad řetězci se provádí derivační strom, který kontroluje jejich správnost, samotná konstrukce derivačního stromu je založena na gramatických pravidlech. Výstupem pak je simulace konstrukce derivačního stromu, která putuje do sémantického analyzátoru. Derivační strom může být konstruován, buď shora dolů (z S[1] směrem ke vstupnímu řetězci) nebo zdola nahoru (ze vstupního řetězce směrem k S[1]).

Na obr. 1.2 můžeme vidět, že syntaktický analyzátor může být naprogramován jako hlavní funkce, která si volá lexikální analyzátor, jenž čte zdrojový kód podobně jako Turingův stroj pásku do doby, než načte platný lexém, ten upraví na token a pošle zpět do syntaktického analyzátoru. Tento proces se opakuje dokud se nedojde na konec zdrojového programu, který je reprezentován příkazem EOF (End-of-file - konec souboru).

1.1.3 Sémantický analyzátor

Sémantická analýza je činnost, při které sémantický analyzátor kontroluje typy (při ní může provádět implicitní konverze, jako je integer na float apod.) nebo deklarace proměnných. Některé analyzátory jsou schopné implicitního přetypování, kdy u operandů různého typu se operandy přetypují na typ operandu s vyšší prioritou. Modernější jazyky pak povolují přetěžování funkcí, kdy je u překladačů potřeba správně určit, které funkce se volají, jejich správnost a správný počet a typ parametrů.

Sémantickou analýzu většinou řídí syntaktický analyzátor a provádí sémantické akce a generování abstraktního syntaktického stromu viz obr. 1.2. Také často komunikuje s tabulkou symbolů kvůli kontrolám. V případě realiace pomocí dynamického seznamu řádku je vhodné převést tabulku na stromové či hašovací struktury.

[1] S označuje počáteční neterminál v gramatice

1.1.4 Generátor vnitřního kódu

Tato část vytváří svoji vlastní reprezentaci vstupního programu na intermediální kód z abstraktního syntaktického stromu. Tento kód je většinou 3-adresný podobný assembleru, důvody mohou být následující: jednotnost, tento kód lze snadno optimalizovat a přímý překlad je někdy příliž složitý a neprůhledný.

1.1.5 Optimalizátor

Tato část optimalizuje intermediální kód, pro zvýšení efektivity překladu. Tato část nemusí být zastoupena v překladači spolu s generátorem vnitřního kódu.

Optimalizace nejčastěji obsahuje: šíření konstanty ($a = 1,\ b = 3,\ c = a + b => c = 4$), šíření kopírováním ($b = a,\ c = b => c = a$), eliminace mrtvého kódu atd.

1.1.6 Generátor kódu

Poslední částí překladače je generátor kódu. Převádí optimalizovaný kód na kód cílový v daném jazyce.

V praxi se většinou jedná o assembler, zdrojový kód nebo v poslední době JavaScript. Tento výstupní kód je připraven na další překladač nebo interpret, který jej po spuštění přečte a začne jej vykonávat.

2 Návrh HLS

Ze začátku byl navržen vstupní pseudokód a pseudoknihovny, které byly v průběhu práce vylepšovány, aby výstupní kód zahrnul, pokud možno, co největší podmnožinu z jazyka VHDL. HLS se na venek tváří jako klasický překladač, protože je časově výhodnější načíst celý soubor a pak s ním pracovat, než neustále přistupovat do paměti a číst po řádku, avšak má to jednu nevýhodu, když se program snaží otevřít větší soubor, tak Python dokáže načíst maximálně do 2 GB, než dojde na MemoryError. Lze to obejít postupným načítáním větších částí souboru, avšak pro načtení pseudokódu a pseudoknihoven toto není potřeba, protože maximální velikost načtení se vztahuje pouze na 1 soubor, nikoliv na všechny a zároveň žádný soubor nepřesáhl 10 kB. Kvůli časové optimalizaci se načítají pseudoknihovny rekurzivně a jejich komponenty se ukládají na zásobník.

Překladač mám rozdělen pouze na 2 části a to na front-end a back-end. První část obsahuje lexikální, syntaktický a sémantický analyzátor a druhá jen generátor cílového kódu. Chybí zde middle-end, ve kterém bývá generátor vnitřního kódu a optimalizátor viz. obr. 1.1 a 1.2.

Dodatečné informace o VHDL byly zjištěny ze stránky:
https://www.ics.uci.edu/~jmoorkan/vhdlref/

2.1 Popis vývojového prostředí

HLS je vyvíjena v jazyce Python ve verzi 3.8.5. Minimální požadavky na spuštění je Python s verzí 3.8, avšak po úpravě 2 walrus operátorů v modulu *pymod/arguments/arguments.py*, se dá teoreticky snížit minimální verze na 3.6 ve které se, oproti předchozím verzím, změnil datový typ slovníku na uspořádaný slovník, ale koncem roku 2021 skončila jeho podpora.

2.1.1 Výhody Pythonu

Výhody zvoleného jazyka jsou pro mě jednoduchost, relativně dobrá čitelnost a debugování. Také je zde speciální datový typ slovník, který se v jiných jazycích moc nevyskytuje, v ojedinělých případech jako mapy (C++) nebo hashmapy (Java). Avšak u nich se předem definuje datový typ klíčů i itemů, u Pythonu klíče mohou obsahovat čísla i slova zároveň a itemy mohou mít všechny kombinace datových typů. Slovník obsahuje klíče a k nim vždy 1 item. Při přidání stejného klíče se jen aktualizuje hodnota itemu u původního klíče. Ukázka vytvoření slovníku a získání itemu pomocí klíče:

Výpis 2.1: Ukázka slovníku v jazyce Python

```
# dictionary = {key:item}
oblibena_cisla = {"Pavel":1,
                  "Karel":7,
                  "Petr":1000}

oblibena_cisla["Petr"] # = 1000
```

2.1.2 Nevýhody Pythonu

Nevýhody jsou v rychlosti překladu, protože se jedná o skriptovací objektově orientovaný jazyk s dynamickou kontrolou datových typů, avšak oproti jiným podobným jazykům se dá z nich optimalizovat nejlépe s pomocí knihovny Cython. Taková úroveň optimalizace zatím není potřeba, protože samotný překlad je v desítkách milisekund.

2.1.3 Použité IDE

Pro programování v Pythonu byl použit software Spyder stáhnutý přes Anacondu, který je open-source IDE a umožňuje přidání Linuxového terminálu Bash. Díky tomuto rozšíření bylo možné testovat překladač pro Microsoft Windows i Linux bez přepínání bez přepínání na dual-bootu nebo virtuální mašiny. (Bash terminál na Windows je možný pouze u verze Professional, po dodatečném zapnutí v nastavení.)

2.1.4 Záloha

Záloha byla prováděna pravidelně na disk, ale i na privátní GitLab repozitář.
K přístupu privátnímu repozitáři používám aplikaci GitAhead, která je zcela zdarma oproti původně používané aplikaci GitKraken. Nevýhodou je trochu horší GUI aplikace.

2.1.5 Optimální systémové parametry počítače

Systémové parametry mého ultrabooku:

- Operační systém: *Microsoft Windows 10 Pro*
- Typ systému: *64bitový*
- Model: *Dell Vostro 5568*
- Procesor: *Intel Core i5-7200U*
- CPU: *2.5 GHz*
- RAM: *8 GB*

Minimální požadavky Pythonu: RAM $\geq$ *512 MB*.

2.2 Volání programu

2.2.1 IPython konzole

Pokud si IDE pamatuje cestu k souboru, tak stačí zadat: *runfile('main.py')*, jinak je potřeba dodat cestu: *runfile('main.py', wdir='PATH')*, ukázka *PATH*: *D:/.../lastfile*. Argumenty se přidávají za sebe do *args=´´* jako jeden dlouhý řetězec a oddělují se mezery. Ukázka: *runfile('main.py', wdir='D:/lastfile', args = '-help -example')*.

Varování

Při spuštění bez parametru *-input='file'* se vyvolá hláška [WinError 10038], je to způsobeno tím, že IPython konzole nefunguje jako terminál a nelze u ní číst ze STDIN, avšak program pokračuje dál a snaží se číst předdefinovaný soubor *prog/prog1*. Vypisuji u toho varování, které popisuje, jak se program chová a jak se dá volání s parametry upravit.

2.2.2 Windows terminál

Program se vykoná příkazem: *python main.py*, za kterým může být posloupnost argumentů, ukázka volání: *python main.py -e -h*.

Varování

Podobně jako u IPython konzole se může vyvolat hláška [WinError 10093], avšak ta je způsobena čtením STDIN přes select, který ale není umožněn v systémech Windows a program pokračuje dál.

2.2.3 Linuxový (Bash) terminál

Program se spustí příkazem podobným pro Windows: *python3 main.py*, za kterým může být posloupnost argumentů, ukázka: *python3 main.py --h --e*. Oproti předchozím možnostem se nevyvolá varování při spuštění bez parametru *-input='file'*, ale je čekáno na vstup ze STDIN (z příkazové řádky), avšak jen do doby, než se nezavře. Toto je vyřešeno pomocí časovače s předdefinovaným nastavitelným časem 5 sekund. Který se resetuje po každém zmáčknutí enteru. Čas se dá přenastavit parametrem *-t='time'* v sekundách. Ukázka volání: *python3 main.py -t=60*.

2.2.4 Základní příkazy pro Windows a Linux terminál

Tab. 2.1: Základní příkazy pro Windows a Linux terminál

Linux	Windows	Význam
ls	*dir*	výpis složek a souborů na daném adresáři
cd ..	*cd ..*	posunutí se blíže ke kořenové složce o 1 složku
cd PATH	*cd PATH*	posunutí se o PATH (Linux má / a Windows \)
cd ../../d/PATH	*D:*	změna na disk D:
time python3 SCRIPT	-	výpis časů Real, User a Sys daného skriptu

U posledního příkazu se vykoná skript a následně se vypíšou 3 různé časy. Ukázka volání: *time python3 main.py*.

- Real: *skutečný uplynulý čas*
- User: *doba CPU v uživatelském režimu (mimo jádro)*
- Sys: *doba CPU v jádře*

2.2.5 Vstupní parametry

Vstupní parametry mohou být volány v libovolném pořadí za sebou, avšak pouze jen jedenkrát. Vstupní parametry:

-h/--h/-help/--help ..... vypíše nápovědu

-d/--d/-debug/--debug ..... zapne debugovací mód

-e/--e/example/--example ..... ukáže příklady volání programu s parametry

-doc/--doc/-documentation/
--documentation ..... vypíše nápovědu k dokumentaci pydoc

-doc-b/--doc-b/-documentation-b/
--documentation-b ..... vytvoří interaktivní dokumentaci na HTTP serveru s volným portem

-doc-k='keyword'/--doc-k='keyword'/-documentation-k='keyword'/
--documentation-k='keyword' ..... prohledá všechny moduly podle klíče

-doc-n='hostname'/--doc-n='hostname'/ -documentation-n='hostname'/
--documentation-n='hostname' ..... vytvoří HTTP server s požadovaným hostname (základní je localhost)

-doc-p='port'/--doc-p='port'/-documentation-p='port'/
--documentation-p='port' ..... vytvoří HTTP server s požadovaným portem (při napsaní 0 se vybere nepoužitý port)

-doc-w='name'/--doc-w='name'/-documentation-w='name'/
--documentation-w='name' ..... vytvoří dokumentaci kódu podle name do html souborů (doporučené: *-doc-w=.\/*, kdy pak překladač přesune všechny vygenerované html soubory do složky *html*)

-i='file'/--i='file'/-input='file'/--input='file'/-stdin='file'/
--stdin='file' ..... vstupní soubor (bez něj se čte ze STDIN terminálu)

-v='file'/--v='file'/-vhdl='file'/--vhdl='file' ..... výstupní soubor (ve VHDL)

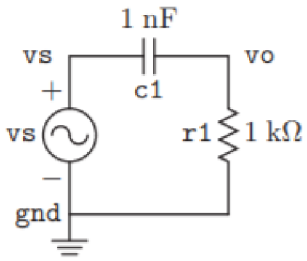
-t='time'/--t='time'/-timeout='time'/
--timeout='time' ..... čas pro časovač zavírající STDIN

2.3 Pseudokód vstupního programu

Program nerozlišuje malá a velká písmena u pseudokódu vstupního souboru ani pseudokódu knihoven. Kód je inspirovaný jazykem SPICE, který ale má velké množství různých variant. Ukázka HSPICE kódu a jeho zobrazení viz obr. 2.1, použita z návodu [6]:

Výpis 2.2: Střídavá analýza vysokopropustného filtru

```
EE105 SPICE Tutorial Example 4 - Simple RC High-Pass Filter
vs vs gnd ac 1V
c1 vs vo 1nF
r1 vo gnd 1k
.ac dec 500 100 1G
.option post=2
.end
```



Obr. 2.1: Střídavá analýza vysokopropustného filtru

Z kódu lze vidět, že z každého prvního lexému na řádku lze vybrat finální pravidla pro všechny zbylé lexémy na stejném řádku. Jinak řečeno, stačí otevřít soubor, načíst obsah jako pole řetězců, ty pak jdou dále rozdělit na lexémy a nad nimi provést lexikální, syntaktickou a sémantickou analýzu, které vygenerují vnitřní data pro generátor cílového kódu, který pak generuje použité VHDL knihovny a balíčky, název entity a architektury, konstanty, vstupy, výstupy, globální proměnné, přiřazení a komponenty. Pro toto generování se jednotlivé části vkládají do podkoster a následně do hlavní kostry. Hlavní kostra a podkostry obahují části VHDL a místa pro vložení různých proměnných, často ošetřeny regulárními výrazy.

Pro optimalizaci se odstraňují jednořádkové komenty, které mohou začínat na prázdném řádku nebo hned za pseudokódem bez mezery. Pro začátek komentáře byl použit znak #. Ukázka komentáře v pseudokódu:

Výpis 2.3: Ukázka komentářů v pseudokódu

```
#komentar
in a#a:in std_logic;
in b # b : std_logic;
# =>
in a
in b
```

Také se kontroluje znak zlomu kódu \, který se používá v případě delších řádků. Vyskytuje se jen mezi lexémy. Tento znak se stejně používá v Pythonu a nebo v Matlabu, ale zde má tvar 3 teček. Před prvním zlomem musí být na stejném řádku aspoň 1 lexém, jinak nastane chyba s číslem 30. Komentáře i zlom se dají jednoduše změnit v souboru *pymod/globals.py*. Ukázka zlomu ve vstupním programu:

Výpis 2.4: Ukázka zlomu ve vstupním programu

```
#\ zlom se v komentari eliminuje
in A#\
# =>
in A
#=====================================================#
in A B C:vec 5
# <=> # ekvivalentni kod
in\#in
\
\
\
A\#a
B\#b
C\#c
\
:\#:
vec\#std_logic_vector
5#4 downto 0
# => VHDL
A:in std_logic_vector(4 downto 0);
B:in std_logic_vector(4 downto 0);
C:in std_logic_vector(4 downto 0);
```

Počáteční lexémy jsou *.vhdllib*, *.use*, *.lib*, *.entity*, *.architecture*, *const*, *in*, *inout*, *out*, *sig*, *mov* a názvy komponent načtené z knihoven. Ukázka všech lexémů ve vstupním programu:

Výpis 2.5: Ukázka lexémů ve vstupním programu

```
.vhdllib my_vhdl_library # => library my_vhdl_library;
.use my_vhdl_library.name # => use my_vhdl_library.name.all;
.lib .lib/logic.lib # nacteni komponent z pseudoknihoven
.ent entity_name # nazev entity
.arch architecture_name # nazev architektury
const B <= 1 # => constant B:std_logic:='1'; v casti generic();
in A # => A:in std_logic; v casti port();
inout B_IO  # => B_IO:inout std_logic; v casti port()
out Y # => Y:out std_logic; v casti port();
sig s_a <= A # => signal s_a:std_logic; v casti architecture
mov B_IO <= B after 10 ms # => B_IO <= B after 10 ms;
# prirazeni konstanty B do B_IO po 10 ms
AND Y s_a B # doplnena konstrukce z pseudoknihovny => Y<=s_a and B;
```

2.3.1 .vhdllib a .use

Oba tyto lexémy jsou nepovinné. V případě použití musí následovat 1 až n lexémů, které se nesmí opakovat. Tyto lexémy u *.vhdllib* jsou názvy knihoven pro VHDL. Předdefinovaná je knihovna IEEE. U *.use* se jedná o balíčky pro volané knihovny, které musí být až po volání knihovny. Předdefinované jsou IEEE.STD_LOGIC_1164.ALL a IEEE.NUMERIC_STD.ALL. Ukázka přidání VHDL knihoven a položek:

Výpis 2.6: Ukázka přidání VHDL knihoven a položek

```
# Pseudocode:
# VHDL libraries:
.vhdllib mylib mylib2
.use mylib.name mylib2.name
# <=>
.vhdllib mylib
.vhdllib mylib2
.use mylib.name
.use mylib2.name
# => ukazka ve vygenerovanem VHDL:
library ieee; -- preddefinovano
library mylib; -- pridano
library mylib2; -- pridano
use ieee.std_logic_1164.all; -- preddefinovano
use ieee.numeric_std.all; -- preddefinovano
use mylib.name; -- pridano
use mylib2.name; -- pridano
```

2.3.2 .lib

Tento lexém je potřebný, pokud chceme pracovat s komponenty. Za ním musí následovat pouze 1 lexém, který reprezentuje cestu ke pseudoknihovně z adresářové pozice, kde je main.py. Knihovna se načte se všemy komponenty a lze volat pouze jednou. Všechny knihovny jsou v adresáři *.lib*. Ukázka volání pseudoknihovny:

Výpis 2.7: Ukázka volání pseudoknihovny

```
# Pseudocode:
.lib .lib/logic.lib # knihovna, ktera si vola vsechny ostatni
# knihovny a tim se nactou vsechny komponenty, avsak sama
# neobsahuje zadnou komponentu
.lib .lib/shift_reg.lib #knihovna, ktera nevola jine knihovny,
# ale ma vlastni komponenty
```

2.3.3 .entity a .architecture

Tyto lexémy jsou potřebné, pokud chceme mít jiný název u entity a architektury. V kódu se mohou vyskytnout maximálně jednou a za ním nasleduje jediný lexém, který reprezentuje jejich název. Pokud chybí, tak se pracuje s předdefinovanými názvy entity_name a architecture_name. Ukázka přidání jmen entitě a architektuře:

Výpis 2.8: Ukázka přidání jmen entitě a architektuře

```
# Pseudocode:
.entity ENT1
.architecture ARCH1
# => VHDL
entity ENT1 is
end ENT1;

architecture ARCH1 of ENT1 is
begin
end ARCH1;
```

2.3.4 const, in, inout, out a sig

Všechny tyto lexémy spojuje stejná definice pro datový typ a přiřazení proměnné nebo konstanty. Datové typy mohou být bit, byte, vektor bitů, unsigned, signed, integer, natural a positive. Ukázka všech datových typu:

Výpis 2.9: Ukázka datových typů

```
# Pseudocode:
in A1 A2 # <=>
in A1 A2 : b # <=>
in A1 A2 : bit
# ve VHDL => A1, A2:in std_logic;
out B:byte
#ve VHDL => B:out std_logic_vector(7 downto 0);
const C:v 3<=101# <=>
const C : vec 3 <= 101 # <=>
const C : vector 3 <= 101
# ve VHDL => constant C : std_logic_vector(2 downto 0) := "101";
sig D:u 5<=10001# <=>
sig D : uns 5 <= 10001# <=>
sig D : unsigned 5 <= 10001
# ve VHDL => signal D : unsigned(4 downto 0) := "10001";
sig E:s 2# <=>
sig E : sig 2# <=>
sig E : signed 2
# ve VHDL => signal E : signed(1 downto 0);
# datove typy vector, signed, unsigned mohou nabyvat
# jen hodnot od 1 po 65536
in F:i<=-3# <=>
in F : int <= -3# <=>
in F : integer <= -3
# ve VHDL => F : in integer := -3;
# rozsah od -2147483649 po 2147483648
in G:n# <=>
in G : nat # <=>
in G : natural
# ve VHDL => G : in natural;
# rozsah od 0 po 2147483648
out H:p <=>
out H:pos <=>
out H:positive
# ve VHDL => H : out positive;
# rozsah od 1 po 2147483648
const I : nat 5 to 20 <= 15
# ve VHDL => constant I : natural range 5 to 20 := 15;
# pro integer, natural a positive se da ruzne nastavit rozsah
```

const

Tento lexém reprezentuje konstantu. Za tímto lexémem musí být posloupnost lexémů, reprezentující datový typ a přiřazení konstanty. Ve VHDL se vygeneruje v části *generic();* v entitě. Ukázka konstanty:

Výpis 2.10: Ukázka konstanty

```
# Pseudocode:
const A : int -5 to 5 <= 1
# => VHDL
entity ENT1 is
    generic(
        constant A : integer range -5 to 5 := 1
    );
end ENT1;

architecture ARCH1 of ENT1 is
begin
end ARCH1;
```

in, inout a out

Tyto lexémy reprezentují vstup a výstup. Za těmito lexémy musí být datový typ, avšak nemusí být přiřazení konstanty nebo čísla. Ve VHDL se vygeneruje v části *port();* v entitě. Ukázka vstupu a výstupu:

Výpis 2.11: Ukázka vstupu a výstupu

```
# Pseudocode:
in A : vec 5 <= 10 1 10
out B : vec 5
# => VHDL
entity ENT1 is
    port(
        A : in std_logic_vector(4 downto 0) := "10110";
        B : out std_logic_vector(4 downto 0);
    );
end ENT1;

architecture ARCH1 of ENT1 is
begin
end ARCH1;
```

sig

Tento lexém reprezentuje signál, který funguje jako pomocná proměnná. Platí u ní stejné definice datového typu a nepovinné přiřazení konstanty jako u in, inout a out. Ve VHDL se vygeneruje v architektuře před lexémem begin. Ukázka signálu:

Výpis 2.12: Ukázka vstupu a výstupu

```
# Pseudocode:
sig A
# => VHDL
entity ENT1 is
end ENT1;

architecture ARCH1 of ENT1 is
    signal A : std_logic;
begin
end ARCH1;
```

přiřazení hodnoty

Přiřazení mohou být různá. Může se jednat o bity, oktalové, dekadická nebo hexadecimální čísla. Ukázka přiřazení hodnoty:

Výpis 2.13: Ukázka přiřazení hodnoty

```
# Pseudocode:
const/in/inout/out/sig promenna : datovy typ <= hodnota
# cast za :
bit <= 0
byte <= 10101010 # <=> 1010 1010 <=> 1 0 1 0 1 0 1 0
#=============================================================#
vec 8 <= hex FF # hexadecimalni cislo zacina lexemem hex nebo x
# a plati pouze na jeden nasledujici lexem
vec 9 <= 1 hex FF # <=> hex FF 1 <=> hex F 1 hex F <=> 111 111 111
vec 3 <= oct 5 # oktalove cislo zacina lexemem oct nebo o
#a opet plati pouze na jeden nasledujici lexem
vec 8 <= oct 77 11 # <=> oct 7 1 oct 7 1 <=> 1111 1111
vec 4 <= all 0 # <=> 0000 # vyplneni bitovou hodnotou,
# maximalne 1 all
vec 4 <= all 0 11 # <=> 0011
vec 8 <= 101 all 0 # <=> 1010 0000
vec 12 <= hex F0 all 1 oct 0 # 1111 0000 1000 # vnitrni vyplneni
#=============================================================#
int range -333 to 666 <= 121 #pouze dekadicka hodnota
```

2.3.5 mov

Tento lexém spojuje 2 proměnné nebo proměnnou a konstantu. Proměnné a konstanty musí být před tímto lexémem definovány. Ve VHDL se vygeneruje v architektuře po lexémemu begin. Ukázka spojení:

Výpis 2.14: Ukázka spojení

```
# Pseudocode:
in A
out Y
mov Y <= A
# => VHDL
entity ENT1 is
    port(
        A : in std_logic;
        Y : out std_logic
    );
end ENT1;

architecture ARCH1 of ENT1 is
begin
    Y <= A;
end ARCH1;
```

přiřazení času

Za přiřazení se může napsat čas přiřazení, které udává, kdy se má daná proměnná nebo konstanta přiřadit. Toto časování jde u *inout, out, sig a mov*, ale nejde u *in a const*. Čas může být v fs, ps, ns, μs, ms, sec, min, hr a každý další na řádce musí být větší než předchozí. V simulaci ISim lze jít až na nanosekundy. Ukázka přiřazení času:

Výpis 2.15: Ukázka spojení

```
# Pseudocode:
in A B
out Y
mov Y <= A, B after 5 ns, A after 10 ns
# => VHDL
architecture ARCH1 of ENT1 is
begin
    Y <= A, B after 5 ns, A after 10 ns;
end ARCH1;
```

2.3.6 komponenty

Tento lexém představuje název komponenty, který musí být načten z *.lib* viz 2.3.2. Za lexémem jsou v přesném pořadí výstupy komponenty a pak vstupy komponenty, nicméně komponenta může postrádat vstupy i výstupy. Komponenta v pseudoknihovně kromě vstupů a výstupů obsahuje také část VHDL kódu, která se ale vkládá do architektury za první begin. Ukázka volání komponenty:

Výpis 2.16: Ukázka volání komponenty

```
# knihovna .lib/logic.lib
.lib .lib/bool_logic.lib NOT AND
# knihovna .lib/bool_logic.lib
.COMP NOT
  in a
  out y
.CODE
.VHDL
  $y$ <= not $a$;
.END
# Pseudocode:
.lib .lib/logic.lib
in in_A
out out_Y
NOT out_Y in_A
# => VHDL
entity entity_name is
    port(
        in_A : in std_logic;
        out_Y : out std_logic
    );
end ENT1;

architecture architecture_name of entity_name is
begin
    out_Y <= not in_A;
end architecture_name;
```

2.4 Pseudokód knihoven

Tento pseudokód se v mnoha ohledech liší od pseudokódu vstupního programu. Jediná podobnost je s komentáři a zalamováním textu viz strana 23 a se vstupy a výstupy. Komponenta musí obsahovat: lexém *.comp* s identickým názvem komponenty, za ním může být lokální vstupní proměnná pro vstupy a výstupy z komponenty (podobně jako u funkce u jiných programovacích jazyků), *.code*, *.vhdl*, VHDL kód a *.end*. V případě nedodržení požadované konstrukce nebo chyby v datovém typu u *in* nebo *out* se vyvolá chybová hláška 40 až 96. Ukázka viz předchozí strana 2.3.6.

2.5 Výstup programu

Výstup programu je v jazyce VHDL a ukládá se do stanoveného souboru. Pokud není stanoveno, tak se zvolí přednastavený soubor *vhdl/prog1.vhd*. Pro generování se používá navržená kostra a podkostry. Ukázka kostry a podkoster:

Výpis 2.17: Ukázka kostry a podkoster

```
use = "use {0};\n"
vhdl_lib = "library {0};\n"
bit = "{0} : {1} std_logic{2};\n"
vector = "{0} : {1} std_logic_vector({2} downto 0){3};\n"
...
positive = "{0} : {1} positive range {2} to {3}{4};\n"
mov = "{0} <= {1};\n"
generic = """generic(
{0}
    );"""
port = """port(
{0}
    );"""

body = """\
{0}
entity {1} is
    {2}
end {1};

architecture {3} of {1} is
{4}
begin
{5}
end {3};"""
```

2.6 Chybové hlášky

Chybové hlášky mají své unikátní číslo a mohou nabývat hodnot 1 až 325, avšak některá čísla pro chybové hlášky byla přeskočena na kulaté číslo, kvůli zpřehlednění daných chybových částí. Mimo to se v každém modulu kontrolují standardní knihovny a vlastní moduly. Chybové hlášky slouží k upozornění na chybu v pseudokódu a v pseudoknihovnách. Chybová hláška přesně popisuje daný problém, ve kterém souboru se tak stalo, na jakém řádku je chyba a pokud je potřeba i ve kterém slově (lexému). Všechny chybové hlášky se dají najít v modulu *pymod/data/data.py* ve třídě ErrorData.

Výpis 2.18: Ukázka chybové hlášky

```
Exit code: 190

You can't have multiple sizes with types: "vec","uns","sig"
problem on the line 17 word 6. in the file 'prog/prog1'.
```

V případě volání neexistující chybové hlášky se ukončí program s exitcode 420. Toto číslo bylo zvoleno kvůli unikátnosti a dostatečnému odstupu od 2. největšího čísla u chybové hlášky (exitcode 325).

2.7 Postup řešení

Pro návrh řešení bylo čerpáno hlavně z výukových materiálů viz strana 13.
Pro doplnění znalostí Pythonu byly použity následující stránky:

- https://docs.python.org/3/
- https://realpython.com/
- https://www.thecodingforums.com/forums/python.77/
- https://stackabuse.com/tag/python/
- https://stackoverflow.com/
- https://www.datacamp.com/community/tutorials
- https://docs.python-guide.org/

Pro pochopení práce s moduly a importy doporučuji tyto stránky:

- https://towardsdatascience.com/common-mistakes-when-dealing-with-multiple-python-files-b4f4dc4d5643
- https://www.blog.pythonlibrary.org/2016/03/01/python-101-all-about-imports/

U složitějších komponent jsem se inspiroval již vytvořenými VHDL kódy, a proto je u nich komentář s URL adresou.

2.7.1 Moduly

Všechny moduly pracují jen se standardními knihovny. Odkaz na názvy všech standardních knihoven: https://docs.python.org/3/library/

Primárním souborem je *main.py* v kořenové složce. Pro schování funkcí do modulů, funkce *main* volá pouze jedinou funkci *pymain*.

Aby každý modul měl přístup ke všem modulům, byl vytvořen soubor *imports.py*.

Dalším souborem je soubor *globals.py* obsahující konstaty: znak začátku komentáře, znak zalomení logického textu a přednastavený čas pro zavření STDIN.

V *debug_func.py* jsou 2 ladící funkce, které se zapnou při použití parametru *-debug* a jeho ekvivalencí.

V každé složce uvnitř složky *pymod* se nachází soubor *___init___.py*, který je potřeba pro starší verze Pythonu, avšak v mém případě jsou primárně pro dokumentační generátor pydoc, který je vyžaduje.

Funkce *pymain* si postupně volá funkce *parse_arguments*, *read_program*, *parser* a *gen*. První funkce zpracovává zvolené parametry. Druhá čte vstupní kód. Třetí parsuje vstupní kód, volá druhou funkci pro načtení pseudoknihoven, zpracovává pseudoknihovny, ukládá data o komponentách z knihoven. Poslední funkce slepuje tyto data s podkostrami do finální kostry VHDL kódu a ten uloží do požadovaného souboru. Všechny tyto funkce volají další funkce ve svých modulech a zároveň používají databázi a kontrolu pravidel v souboru *data/data.py*.

Z dat jsou nejdůležitější *program_data* a *loaded_comp*. Data hlavního programu jsou pouze ve formě slovníku. Data o komponentách jsou uložena ve formě listu slovníků. Jejich kombinací s kostrami lze vygenerovat finální podmnožina VHDL kódu. Ukázka polonaplněných datových struktur:

Výpis 2.19: Ukázka dat

```
# program_data =
{'entity_name': 'entity_name',
'architecture_name': 'architecture_name',
'const': [same as 'in'],
'signal': [same as 'in'],
'in':   [{'name':'a',
        'type': 'pos',
        'size': ['1','to','%'],
        'value': '123'},...],
'out': [same as 'in'],
'inout': [same as 'in'],
'const_names': [same as 'inout_names'],
'signal_names': [same as 'inout_names'],
'inout_names': ['a', 'b', ...],
'mov':  [{'name': 'out2',
        'values': [['value', 'time', 'unit'],...]},...]
'comp': [{'name':'not', 'var':[y1', 'a1'], 'index':0},...],
'comp_names': ['not',...]
}

# loaded_comp =
[
{'comp': 'not_v1_%_%55_v1',
'var': ['%','%55'],
'in': [{'name':'a',
        'type': 'pos',
        'size': ['1','to','%']},
        {'name':'b',
        'type': 'pos',
        'size': ['1','to','%']}],
'out': [{'name':'y',
        'type': 'bit',
        'size': []}],
'inout': ['a','b','y'],
'vhdl': '   $y$ <= not $a$',
'lib': '.lib/bool_logic.lib'},
...]
```

Pro pěknější tisknutí dat při debugování byla použita standardní knihovna *pprint*. Data poté nejsou vypsána jen na jeden řádek, ale pěkně na více řádků.

3 Testovací část

Pro testování bylo vytvořena série pseudokódů ve složce *prog/* a pseudoknihoven ve složce *.lib/*. Výstupní VHDL se generuje do složky *vhdl/*. Vstup i výstup se dá změnit pomocí navolených parametrů u python skriptu.

Všechen vygenerovaný VHDL kód je funkční a byl zkontrolován aplikací ISE Project Navigator a ISim simulací. Ukázka volání skriptu: *python3 main.py -i=prog/prog2 -t=1 -v=vhdl/prog2.vhd*.

Moji HLS jsem dále porovnal s jinou vysokoúrovňovou syntézou MyHDL.

3.1 MyHDL

MyHDL je open-source knihovna napsaná v jazyce Python pro popis i verifikaci hardwaru. Dokáže modelovat i simulovat a navíc dokáže převést návrh do jazyka Verilog nebo VHDL. Podrobnější popis viz:
http://docs.myhdl.org/en/stable/manual/preface.html.

Historie MyHDL sahá až k lednu roku 2003, kdy byl proveden první commit viz: https://github.com/myhdl/myhdl/commits/master. Předchozí záznamy nebylo možné dohledat. Od 18. března roku 2015 začali podporovat Python 3 viz hlavní stránka: https://www.myhdl.org/. Do dneška na ní pracovalo 41 lidí viz statistika githubu: https://github.com/myhdl/myhdl a z PyPI (The Python Package Index - index Python balíčků) byla tato knihovna skoro 50 tisíckrát stažena viz statistika: https://pepy.tech/project/myhdl. Na GitHubu lze najít, že je knihovna neustále aktualizována a je zde možnost diskuze. Pro optimalizace používají knihovnu PyPy (Alternativní implementace Pythonu k jazyku CPython), s kterou dosahují vyšší optimalizace než s CPython viz statistika: https://www.myhdl.org/docs/performance.html

Ukázka modulu a jeho volání z hlavního skriptu v MyHDL:

Výpis 3.1: hlavní skript *main.py*

```
# A small sequential design
from myhdl import Signal, ResetSignal, modbv
from inc import inc

def convert_inc(hdl):
    """Convert inc block to Verilog or VHDL."""
    m = 8
    count = Signal(modbv(0)[m:])
    enable = Signal(bool(0))
    clock = Signal(bool(0))
    reset = ResetSignal(0, active=0, isasync=True)

    inc_1 = inc(count, enable, clock, reset)
    inc_1.convert(hdl=hdl)


convert_inc(hdl='Verilog')
convert_inc(hdl='VHDL')
```

Výpis 3.2: modul *inc.py*

```
from myhdl import block, always_seq

@block
def inc(count, enable, clock, reset):
    """ Incrementer with enable.

    count -- output
    enable -- control input, increment when 1
    clock -- clock input
    reset -- asynchronous reset input
    """
    @always_seq(clock.posedge, reset=reset)
    def seq():
        if enable:
            count.next = count + 1

    return seq
```

Výstupem jsou *inc.v* a *inc.vhd*. Ve složce *myhdl/* jsou ukázky sekvenčního, kombinačního a hierarchického návrhu, ROM a simulace D flip-flop.

Závěr

V rámci bakalářské práce byl navržen a podrobně popsán překladač typu HLS v jazyce Python. Celkem se jedná o 10 python skriptů s 1 hlavním skriptem: *main.py*. K překladači byl navržen vlastní pseudokód a pseudoknihovny. Program lze spouštět se spoustou různých parametrů zároveň, které jsou popsány výše na straně 22 a výstupem je soubor s kódem v jazyce VHDL. Dokumentace kódu je pro multifunkčnost psána v angličtině. Celkem bylo napsáno 4505 řádku v Pythonu. Příkazem *find . -name '*.*py' -type f | xargs wc -l* v bash terminálu lze zjistit přesný počet řádků v každém souboru končící příponou *.py* ve všech složkách a podsložkách.

Byla ošetřena většina možných chybových stavů s upozorněním na typ chyby a místo chyby. Jejich počet je nad 200 viz strana 34. Byla vytvořena sada vstupních pseudokódů a knihoven jako logické obvody, čítače, multiplexory, demultiplexory, flip-flopy, amplitudové modulace, konverze, násobičky, posuvné registry, radix 4 butterfly blok, ALU a ADC viz komponenty 2.3.6.

Porovnáním mojí HLS s MyHDL bylo zjištěno, že u většiny příkladů moje HLS překládá v rámci desítek milisekund a MyHDL v nižších stovkách milisekund. Toto může být způsobeno různými faktory, jako jsou generování menší modmnožiny jazyka VHDL a práce s pseudokódem v podobě 3-adresného kódu. Výhody konkurenční syntézy značně převyšují tuto malou drobnost, jako je generování VHDL i Verilog, simulování a verifikace.

Literatura

[1] ŠŤASTNÝ, Jakub. *FPGA prakticky: realizace číslicových systémů pro programovatelná hradlová pole.* Praha: BEN - technická literatura, 2010. ISBN 978-80-7300-261-9.

[2] HOROWITZ, Paul. *The art of electronics.* Third edition. New York, NY: Cambridge University Press, [2015]. ISBN 978-0-521-80926-9.

[3] NANE, Razvan, Vlad-Mihai SIMA, Christian PILATO, et al. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* [online]. 2016, 35(10), 1591-1604 [cit. 2021-12-22]. ISSN 0278-0070. Dostupné z: doi:10.1109/TCAD.2015.2513673

[4] MEDUNA, Alexander. *Automata and languages: theory and applications.* London: Springer, 2000. ISBN 978-1-85233-074-3.

[5] MEDUNA, Alexander. *Formal languages and computation: models and their applications.* Boca Raton: CRC Press, Taylor & Francis Group, [2014]. ISBN 978-1466513457.

[6] HSPICE Tutorial. *HSPICE Tutorial* [online]. UNIVERSITY OF CALIFORNIA AT BERKELEY: Department of Electrical Engineering and Computer Sciences, [2011] [cit. 2021-12-27]. Dostupné z: https://inst.eecs.berkeley.edu/∼ee105/sp11/tutorials/HSPICE_Tutorial.pdf

Seznam symbolů a zkratek

ASIC	Application Specific Integrated Circuit - zákaznický integrovaný obvod
CPLD	Complex Programmable Logic Device - komplexní programovatelné logické zařízení
CPU	Central Processing Unit - centrální procesorová jednotka
DSP	Digital Signal Processor - digitální signálový procesor
EOF	End-of-file - konec souboru
EPLD	Electronically programmable logic devices - elektronicky programovatelná logická zařízení
FPGA	Field Programmable Gate Array - programovatelná hradlová pole
GAL	Generic array logic - obecná logická pole
GUI	Graphic User Interface - grafické uživatelské rozhraní
HDL	Hardware Description Language - programovací jazyk pro popis hardwaru
HLS	High-level Synthesis - vysokoúrovňová syntéza
IC	Monolithic Integrated Circuit - monolitický integrovaný obvod
IDE	Integrated Development Environment - integrované vývojové prostředí
MCU	Microcontroller Unit - jednotka mikrokontroléru
MyHDL	Programovací jazyk založený na Pythonu popisující hardware
PAL	Programmable Array Logic - programovatelné logické pole
PLA	Programmable logic array - programovatelné logické pole
PLD	Programmable Logic Device - programovatelný logický obvod
PyPI	The Python Package Index - index Python balíčků
PyPy	Alternativní implementace Pythonu k jazyku CPython

RTL Register Transfer Language - jazyk pro přenos registrů

SPICE Simulation Program with Integrated Circuit Emphasis - simulační program s důrazem na integrované obvody

STDIN Standard input - standardní vstup

STDERR Standard error - standardní chybový výstup

STDOUT Standard output - standardní výstup

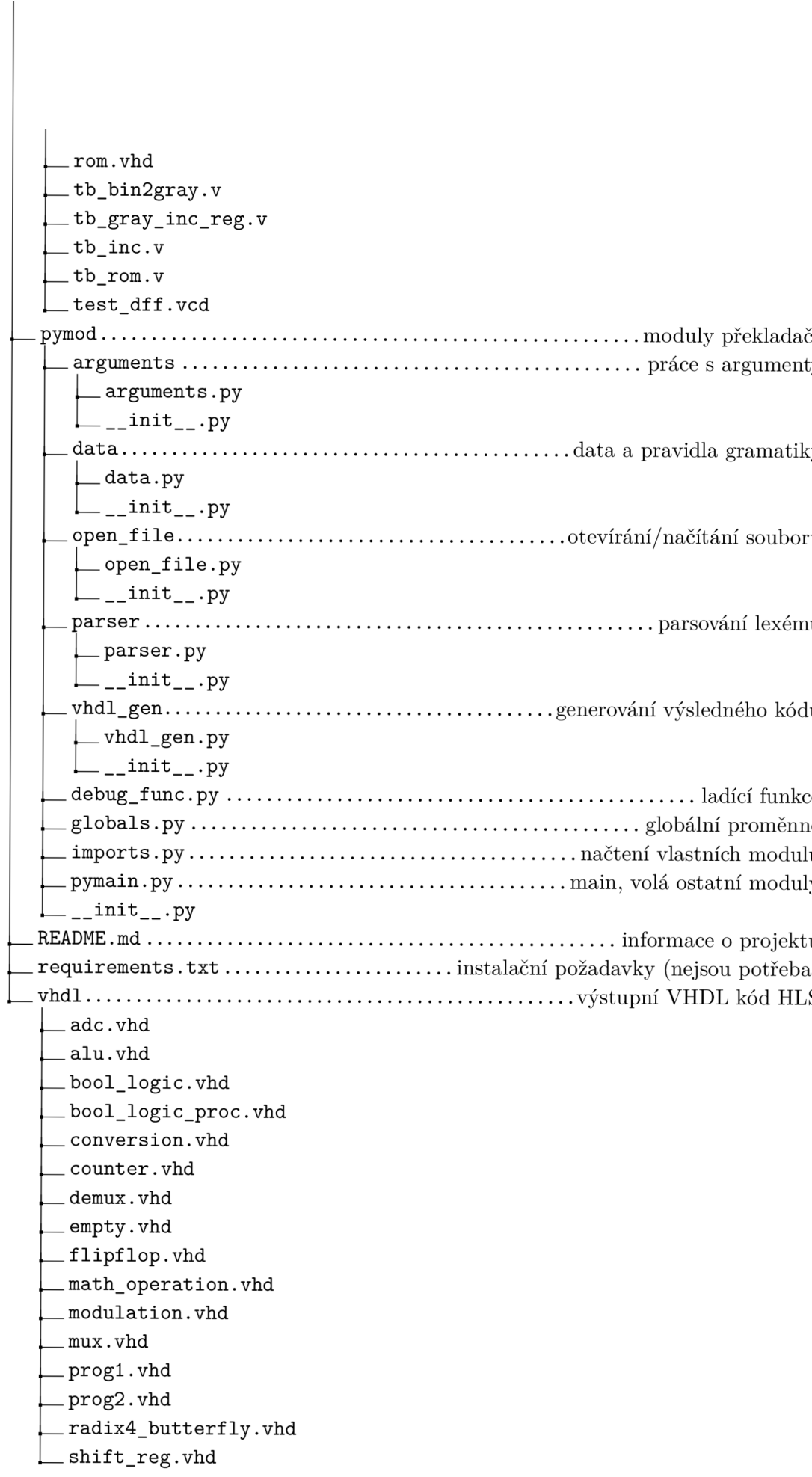
VHDL VHSIC Hardware Description Language - programovací jazyk pro popis hardwaru s velmi rychlými integrovanými obvody

VHSIC Very High Speed Integrated Circuit - Velmi rychlé integrované obvody

A Obsah elektronické přílohy

Program se spouští v terminálu příkazem: *python3 main.py* na Linuxu nebo *python main.py* na Windows. Program může přijímat různé parametry, pro jejich výpis použijte argument: *-help*, který se píše za příkaz v terminálu.

- / ..... kořenový adresář: bakalarka
  - .gitignore ..... odstranění nadbytečných souborů
  - .lib ..... pseudoknihovny
    - adc.lib
    - alu.lib
    - bool_logic.lib
    - bool_logic_proc.lib
    - bool_logic_vector.lib
    - bool_logic_vector_proc.lib
    - conversion.lib
    - counter.lib
    - demux.lib
    - flipflop.lib
    - logic.lib
    - math_operation.lib
    - modulation.lib
    - mux.lib
    - mylib1.lib
    - radix4_butterfly.lib
    - shift_reg.lib
  - doc ..... text semestrální práce
    - semestralni_prace.pdf
  - html ..... místo pro vygenerování dokumentace kódu
  - main.py ..... spouštěcí program, volá pymain.py
  - myhdl ..... hlavní program, komponenty a výstup v jazycích VHDL a Verilog
    - bin2gray.py
    - bin2gray.v
    - bin2gray.vhd
    - d_flip_flop.py
    - gray_inc.py
    - gray_inc_reg.py
    - gray_inc_reg.v
    - gray_inc_reg.vhd
    - inc.py
    - inc.v
    - inc.vhd
    - main.py
    - pck_myhdl_011.vhd
    - ROM.py
    - rom.v


rom.vhd
tb_bin2gray.v
tb_gray_inc_reg.v
tb_inc.v
tb_rom.v
test_dff.vcd
pymod ..... moduly překladače
arguments ..... práce s argumenty
arguments.py
__init__.py
data ..... data a pravidla gramatiky
data.py
__init__.py
open_file ..... otevírání/načítání souboru
open_file.py
__init__.py
parser ..... parsování lexémů
parser.py
__init__.py
vhdl_gen ..... generování výsledného kódu
vhdl_gen.py
__init__.py
debug_func.py ..... ladící funkce
globals.py ..... globální proměnné
imports.py ..... načtení vlastních modulů
pymain.py ..... main, volá ostatní moduly
__init__.py
README.md ..... informace o projektu
requirements.txt ..... instalační požadavky (nejsou potřeba)
vhdl ..... výstupní VHDL kód HLS
adc.vhd
alu.vhd
bool_logic.vhd
bool_logic_proc.vhd
conversion.vhd
counter.vhd
demux.vhd
empty.vhd
flipflop.vhd
math_operation.vhd
modulation.vhd
mux.vhd
prog1.vhd
prog2.vhd
radix4_butterfly.vhd
shift_reg.vhd