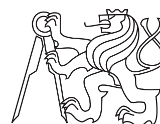
Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra číslicového návrhu



Bakalářská práce

Jednoduchý překladač z jazyka C do mikroprogramu

Jiří Brom

Vedoucí práce: Ing. Pavel Kubalík, Ph.D.

16. května 2012

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce, panu Ing. Pavlu Kubalíkovi, Ph.D., za trpělivost při spolupráci na tvorbě překladače a za ochotu řešit veškeré mé dotazy a problémy okamžitě.

Dále bych rád poděkoval své rodině, která mne podporovala po celou dobu mého studia a pomohla mi s korekturou mé práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 16. května 2012 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2012 Jiří Brom. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Jiří Brom. *Jednoduchý překladač z jazyka C do mikroprogramu: Bakalářská práce*. Praha: ČVUT v Praze, Fakulta informačních technologií, 2012.

Abstract

This bachelor thesis contains design and description of simple translator from C to VHDL language. It also evaluates current available solutions of translation from high-level languages to hardware description languages. Implementation is made in Java language.

The translator can be useful in the design of hardware project algorithms, which are easily described in high-level language.

Keywords translator, C, VHDL, mikrocode, micro-controller, Java

Abstrakt

Práce se zabývá návrhem a implementací jednoduchého překladače z jazyka C do mikroprogramu, zapsaného v jazyce VHDL. Práce zároveň vyhodnocuje stávající řešení problematiky překladu kódu z vyšších programovacích jazyků do jazyků popisujících hardware. Implementace je provedena v jazyce Java.

Překladač může být užitečný při návrhu hardwarových projektů, kde je schopen značně ušetřit čas vygenerováním algoritmů, které jsou snadněji popsatelné v jazyce C.

Klíčová slova překladač, C, VHDL, mikroprogram, mikrořadič, Java

Obsah

Úvod **17**

1 Rešerše **19**
1.1 Alternativy jazyků C a VHDL . . . 19
1.2 Alternativní produkty . . . 20

2 Analýza a návrh **21**
2.1 Metody implementace překladače . . . 21
2.2 Jazyk C . . . 21
2.3 Jazyk VHDL . . . 22
2.4 Porovnání C a VHDL . . . 24
2.5 Návrh úpravy syntaxe vstupního jazyka . . . 24
2.6 Návrh struktury výstupního kódu . . . 27
2.7 Charakteristika jazyka Java . . . 30
2.8 Vývojové prostředí . . . 31
2.9 Návrh struktury překladače . . . 31

3 Realizace **33**
3.1 Architektura hlavních částí . . . 33
3.2 Lexikální analyzátor . . . 34
3.3 Syntaktický analyzátor . . . 36
3.4 Ukládání dat . . . 37
3.5 Generátor mezikódu . . . 41
3.6 Generátor kódu . . . 45

4 Testování **47**
4.1 Použitý software . . . 47
4.2 Příklad 1: Násobička . . . 47
4.3 Příklad 2: Transmitter . . . 48

Závěr **51**

Literatura **53**

A Seznam použitých zkratek **55**

B Obsah přiloženého CD **57**

Seznam obrázků

2.1 Blokové schéma mikrořadiče 29
2.2 Návrh struktury překladače 31

3.1 Architektura hlavních částí programu 33
3.2 Konečný automat pro sestavování tokenů 35
3.3 Komunikační diagram pro zařazení přijatého tokenu 36
3.4 Komunikační diagram pro načtení funkce If 37
3.5 Konečný automat načítání tokenů funkce If 38
3.6 Class diagram datových tříd 38
3.7 Class diagram programových tříd 39
3.8 Ukázka syntaktického stromu 40
3.9 Realizace cyklu 42
3.10 Realizace funkce if-else 43
3.11 Realizace funkce goto 44

Seznam tabulek

2.1 Úprava vstupní syntaxe 25
2.2 Příklad kolize při zápisu do registru 25
2.3 Příklad použití příkazu *clk* 26
2.4 Zkratky použité v blokovém schématu 28
2.5 Rozvětvení programu na dvě části 29

3.1 Význam symbolů použitých v automatu 35
3.2 Ukázka rozskoku podmíněné funkce 41
3.3 Ukázka vyplnění paměti pro cyklus 42
3.4 Ukázka vyplnění paměti pro funkci if-else 43

Úvod

Hlavním cílem této bakalářské práce je navrhnout a zrealizovat překladač z programu popsaného podmnožinou příkazů jazyka C do mikroprogramu popsaného v jazyce VHDL. Realizovaný překladač může být využitelný při návrhu hardware, kde dokáže značně ušetřit čas vygenerováním kódu popsatelného v jazyce C, oproti zdlouhavému návrhu VHDL. Problematikou překladu vyšších programovacích jazyků na popis hardware se na trhu zabývá již několik produktů, povětšinou však pracují s jinými jazyky než C a VHDL.

Program bude implementován v jazyce Java a dokáže přeložit základní prvky jazyka C, jako jsou aritmetické a logické operace, podmíněné výrazy a cykly. Jazyk C bude podporovat dva typy proměnných: registr a vodič. Hlavní součástí návrhu bude mikroprogramovatelný řadič, který bude řídit průběh výsledného VHDL kódu.

Práce je rozdělena do čtyř částí. První částí je rešerše, která podává informaci o stávajících dostupných produktech, řešících podobnou problematiku. Druhou částí je návrh a analýza, kde jsou podrobně rozebrány základní otázky týkající se způsobu použití překladače a prvotního návrhu architektury. Třetí částí je řešení, které detailně popisuje problémové části implementace. Čtvrtou částí je testování, které obsahuje dva ukázkové příklady funkce překladače.

Kapitola 1

Rešerše

Při prohledávání různých informačních zdrojů na internetu nebyl nalezen žádný produkt, který by se přímo shodoval s předmětem mé bakalářské práce. Bylo však nalezeno několik variant produktů, které se svou funkcí problematice této práce přibližují, ačkoliv se nejedná přímo o překladače z jazyka C do VHDL.

Většinou se jedná o produkty využívající některé z následujících alternativ jazyka C nebo jazyka VHDL.

1.1 Alternativy jazyků C a VHDL

1.1.1 System C

Knihovna C++ vytvořená skupinou společností zvanou Open SystemC Initiative (OSCI). Byla navržena pro manipulaci s nízkoúrovňovými systémy, podporuje metody a funkce pro práci s porty, signály a dalšími hardwarovými prvky. Umožňuje simulaci paralelních procesů s použitím C++ syntaxe. Sémanticky se podobá HDL jazykům jako VHDL a Verilog.(3)

1.1.2 Verilog

HDL jazyk vyvinutý roku 1985 firmou Automated Integrated Design Systems pro modelování elektronických systémů. Jedná se pravděpodobně o nejbližší alternativu VHDL.

Cílem návrhu byl jazyk se syntaxí podobnou jazyku C, což je jeho největší rozdíl oproti VHDL. Verilog podporuje klíčová slova if/else, while apod. Dalšími rozdíly oproti VHDL jsou rozlišování velikosti písmen (case-sensitive), nižší striktnost a jednodušší datové typy.(1)

1.2 Alternativní produkty

Dostupné produkty, které využívají výše uvedených alternativ jazyka C a VHDL k překladu jazyka vyšší úrovně na některý z HDL jazyků.

1.2.1 C to Verilog

Open source aplikace online dostupná na webu http://www.c-to-verilog.com/. Jedná se o studijní projekt Haifa University v Izraeli. Jako vstupní jazyk využívá mírně omezenou podmnožinu jazyka C. Výstup je generován v jazyce Verilog. Na vstupu aplikace C to Verilog lze použít většinu vlastností jazyka C kromě těch, které nelze reprezentovat pomocí hardware, jako například ukazatele, rekurzivní funkce, struktury, knihovní funkce printf apod.

Podle vyjádření autora na výše zmíněné adrese je do budoucna v plánu vytvořit také překladač C to VHDL. (10)

1.2.2 Catapult C

Syntetizační nástroj vyvinutý firmou Calypto Design Systems. Na vstupu využívá jazyk C++ obohacený o knihovnu System C. Výstup generuje v RTL kódu, což je nadmnožina HDL jazyků, jako jsou VHDL a Verilog. Dokáže tedy na výstup vygenerovat i VHDL kód. Obsahuje grafické prostředí, kde je možno při návrhu vidět schéma výsledného hardware.(2)

Kapitola 2

Analýza a návrh

2.1 Metody implementace překladače

Podle způsobu implementace vstupního programovacího jazyka do cílového se překladače rozdělují na kompilační a interpretační.(8)

Kompilační překladač (tzv. kompilátor)

Transformuje zdrojový program ve vyšším jazyku na ekvivalentní cílový program v jazyku počítače. Cílový program pak může být počítačem přímo proveden.

Interpretační překladač

Zdrojový program se přeloží do vnitřní formy, která není strojovým jazykem fyzického procesoru, ale jazykem virtuálního počítače. Provedení programu ve vnitřní formě na konkrétním počítači pak zajistí program zvaný *interpret*. Ve srovnání s kompilační metodou je interpretační metoda méně náročná na překlad, avšak náročnější z hlediska doby výpočtu.

Jelikož VHDL není strojovým jazykem cílového počítače, jedná se v našem případě o překladač tohoto typu.

2.2 Jazyk C

2.2.1 Charakteristika

Jde o vyšší, obecně použitelný programovací jazyk, který byl původně navržen pro vývoj operačního systému Unix. Veškeré konstrukce jsou srozumitelné a snadno ilustrovatelné. Umožňuje operace blízké strojovému kódu, jako například operace s adresami, alokace paměti, ukazatele. Vytvořený kód je velice efektivní a výsledné programy jsou dobře přenositelné. Syntaxe výrazů a příkazů byla převzata řadou dalších jazyků.(4)

2.2.2 Základní datové typy(7)

- int - celočíselný typ
- double - číselný typ s pohyblivou řádovou čárkou
- char - reprezentace jednoho znaku ASCII tabulky
- void - bezhodnotový typ

2.2.3 Základní řídící příkazy

- **if-else**

```
if (podmínka){
    příkaz;
} else {
    příkaz;
}
```

- **for**

```
for (inicializace;podmínka;inkrement){
    příkaz;
}
```

- **while**

```
while (podmínka){
    příkaz;
}
```

2.3 Jazyk VHDL

2.3.1 Charakteristika

Z názvu VHDL je patrné, že jde o jazyk druhu HDL (hardware description language), tzn. nízkoúrovňový jazyk sloužící pro popis hardware. Běh programu umožňuje sekvenční, ale i souběžné provádění, tudíž může vykonávat více instrukcí najednou. Používá se pro návrh a simulaci integrovaných obvodů jako například FPGA či ASIC. Při návrhu pomocí VHDL je potřeba určitá znalost výsledného hardware.

Informace o charakteristice a základních prvcích syntaxe jsou čerpány z předmětu BI-PNO.(9)

2.3.2 Základní datové typy

- bit - jednobitový typ s hodnotami „0, 1“
- std_logic - jednobitový typ s hodnotami „0, 1, Z, X, L, H, W, U, -“
- std_logic_vector - vícebitový typ

2.3.3 Porty

- IN - vstupní port
- OUT - výstupní port

2.3.4 Základní konstrukce

- **Příklad entity**

```
entity MULTIPLEXER is
port(
    A,B,SEL : in std_logic; -- deklarace vstupních signálů
    Y : out std_logic -- deklarace výstupních signálů
);
end MULTIPLEXER;
```

- **Příklad architektury**

```
architecture MUX_BODY of MULTIPLEXER is
-- deklarační část:
signal SELNON, ASEL, BSEL : std_logic;
-- deklarace konstant, signálů, komponent, datových typů
begin
-- řídící část:

-- procesy, instance komponent
end MUX_BODY;
```

- **Příklad procesu**

```
MUXPR : process (SEL, A, B) -- proces reagující při
begin                       -- změně signálu SEL, A, B
    if SEL = ‘0’ then
        Y <= A;
    else
        Y <= B;
    end if;
end process MUXPR;
```

- **Příklad přiřazení**

```
D <= S(0) & D(n-1 downto 1); -- do D ulož nejnižší bit z S
                             -- a nejvyšších n-1 bitů z D
```

2.4 Porovnání C a VHDL

Při porovnání jazyka C a VHDL jsou vidět značné rozdíly již v základních vlastnostech. První rozdíl je patrný v rozdílných datových typech. Zatímco C pracuje minimálně s jedním bytem, ve VHDL je možné manipulovat přímo s jednotlivými bity. Pro funkční VHDL kód je navíc potřeba definovat vstupní a výstupní porty. Další rozdíl je v běhu programu. C je čistě sekvenční jazyk, VHDL zvládá i paralelní běh procesů.

Na základě těchto odlišností je technicky nemožné vytvořit překladač, který by generoval smysluplný a funkční VHDL kód bez jakýchkoliv dodatečných informací o datových typech, či informaci o počtu taktů apod. Z tohoto důvodu bude potřeba vstupní jazyk překladače doplnit o některé definice a vlastnosti.

2.5 Návrh úpravy syntaxe vstupního jazyka

2.5.1 Definice datových typů

Program bude podporovat dva základní datové typy: registr a vodič (signál). Kromě vnitřních registrů potřebuje VHDL kód také znát směr vstupních a výstupních portů. Proto bude vstupní kód doplněn o následující definice:

- IN - definice vstupních registrů
- OUT - definice výstupních registrů
- REG - definice vnitřních registrů
- WIRE - definice signálů
- CONST - definice konstant

2.5.2 Bitové operace

Při popisu hardwarových obvodů je běžné pracovat s daty na úrovni bitů, což nabízí možnost vybírat požadované bity vektorů a řetězit je s jinými bity. Pro definici vektorů je navíc nutné vyjádření požadovaného rozsahu bitů, např. od nejvyššího po nejnižší. Syntax vstupního jazyka tudíž doplníme o seznam operátorů, jejichž význam je zobrazen v tabulce 2.1.

operátor	**význam**
#	zřetězení dvou vektorů (náhrada VHDL operátoru &)
[n : 1]	vybrání konkrétních bitů vektoru (náhrada VHDL operátoru *downto*)
>>	bitový posun doprava
<<	bitový posun doleva

Tabulka 2.1: Úprava vstupní syntaxe

2.5.3 Počet taktů

Bez zásahu do vstupního kódu nejsme schopni ovlivňovat počet taktů VHDL kódu. Taková vlastnost by se mohla hodit například při oddělování přiřazení do registrů nebo při prodlužování doby trvání cyklu.

Program by měl totiž implicitně fungovat tak, že přiřazovací signály, které nejsou odděleny jiným příkazem, se provedou všechny v jednom taktu. To však může způsobovat kolizi při postupném přiřazování různých hodnot do stejného registru. Například pokud chceme generovat signál měnící svou hodnotu po taktech několikrát za sebou. Příklad takovéto možné kolize je znázorněn v následujícím příkladu.

- **vstup:**

```
-----
Ack=1;    // ack nastav '1'
Ack=0;    // ack nastav '0'
Ack=1;    // ack nastav '1'
-----
```

- **výstup:**

číslo taktu	**operace**
1	Ack=1,Ack=0,Ack=1;

Tabulka 2.2: Příklad kolize při zápisu do registru

Pro ovládání počtu taktů VHDL kódu tedy bude vytvořen příkaz *clk*, ukončovaný středníkem. V principu bude fungovat tak, že první výskyt příkazu *clk* oddělí takt následujícího příkazu od taktu předešlého příkazu. Každý další výskyt v řadě vytváří prázdný takt.

Příklad použití:

- **vstup:**

```
Ack=1;    // ack nastav '1'
clk;      // oddělení od předešlého taktu
Ack=0;    // ack nastav '0'
clk;      // oddělení od předešlého taktu
ckl;      // prázdný takt
Ack=1;    // ack nastav '1'
```

- **výstup:**

číslo taktu	**operace**
1	Ack=1;
2	Ack=0;
3	prázdný takt
4	Ack=1;

Tabulka 2.3: Příklad použití příkazu *clk*

2.5.4 Vytvoření komponenty

Pro jednodušší práci s výsledným kódem je pro uživatele výhodné nadefinovat si vlastní komponentu, což je část VHDL kódu definující rozhraní výsledného programu. Ta se dá následně využít například při tvorbě testbench. Pro její definici bude sloužit funkce *func_proved*(*nazvy_mapovanych_portu*).

Příklad použití:

- **vstup:**

```
func_proved(IN A, OUT B, OUT C);
```

- **výstup:**

```
component func_proved
    port (
        A    : in std_logic;
        B    : out std_logic;
        C    : out std_logic
    );
end component;
```

```
begin

i_func_proved : func_proved
    port map (
    A_R => A_R,
    B => B,
    C => C
    );
```

2.6 Návrh struktury výstupního kódu

Pro ucelení představy o funkci překladače je důležité si stanovit, jakým způsobem chceme, aby výsledný VHDL kód realizoval program zadaný na vstupu, resp. jaká má být cílová struktura. V této části práce je naznačeno, jakými postupy bude překlad probíhat a jaké prvky bude potřeba pro jeho realizaci použít.

Kompletní výstupní VHDL kód je uložen na přiloženém CD.

2.6.1 Vstup v jazyce C

```
CONST n = 8;                         // definice konstanty
REG A[n], B[n], C[n], D[n], i[5];   // definice registrů
WIRE S[n+1], S2[n], AA[n];          // definice signálů (vodičů)

void main(){
    A = 1;
    B = 2;                     // přiřazení do registru
    C = 0;
    for(i=0; i<n; i++){        // cyklus for
        if (B[0] == 1) {       // podmíněný výraz if-else
            AA = A;
        }else {
            AA = 0;
        }

        S = C + AA;
        S2 = S[n:1];           // výběr požadovaných bitů
        C = S2;
        D = S[0]#D[n-1:1];     // zřetězení vektorů
        B = B>>1;              // bitový posun doprava
    }
}
```

2.6.2 Překlad datové části

Datovou částí se má na mysli část vstupního programu obsahující pouze deklarace datových typů. Tato část nemá žádný vliv na posloupnost běhu programu. V našem případě do této části spadá deklarace konstant, vodičů a registrů.

Konstanta
Pro překlad konstanty $CONST\ n = 8$; se nabízí možnost klasické VHDL konstrukce *constant*. Avšak abychom mohli konstantu používat přímo i v ostatních definicích, bude výhodnější použít typ *generic*.
$generic(n : integer := 16);$

Signál
Vodič $WIRE$ bude jednoduše definován jako *signal*. V definici nesmíme zapomínat na snížení počtu bitů o jedna.
$signal\ S : std_logic_vector(n - 1\ downto\ 0);$

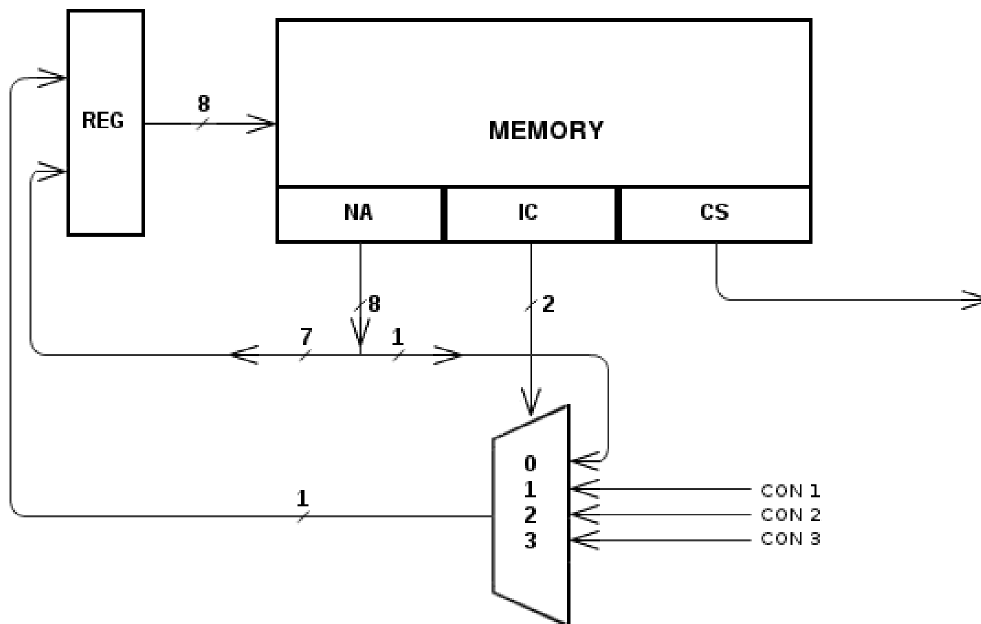
Registr
Registr REG bude definován stejně jako signál. Rozdíl bude pouze v tom, že zápis do registu bude řízen mikrořadičem. (Více v části implementace.)

2.6.3 Překlad programové části

Programovou částí se má na mysli část kódu, která definuje průběh programu. V této části musí být zachována posloupnost instrukcí tak, jak je definována na vstupu. Pro zajištění sekvenčního průběhu programu musíme vytvořit řídící jednotku, která bude postupně realizovat jednotlivé vstupní příkazy. Toho můžeme docílit pomocí mikroprogramovatelného řadiče (dále mikrořadiče), definovaného jako pole záznamů v paměti, kde každý záznam bude tvořen informacemi o následující adrese, vstupní podmínce a řídících signálech. Blokové schéma mikrořadiče je zobrazeno na obrázku 2.1.

použitá zkratka	**význam**
NA	Next Address / příští adresa
IC	Input Condition / vstupní podmínka
CS	Control Signals / řídící signály
MEMORY	paměť mikrořadiče
REG	registr obsahující následující adresu
CON	výsledek dané vstupní podmínky

Tabulka 2.4: Zkratky použité v blokovém schématu



Obrázek 2.1: Blokové schéma mikrořadiče

2.6.3.1 Překlad podmínky

Příkazy *for*, *while* a *if-else* obsahují podmínky větvící program na dvě části. Tyto podmínky je potřeba vyhodnotit a podle jejich výsledku dále směrovat běh programu. Rozvětvení programu může být v mikrořadiči snadno realizováno na základě různých následujících adres, jak je tomu naznačeno v tabulce 2.5.

aktuální adr.	**příští adr.**	**vstupní podm.**
"01"	"10" nejnižší bit se změní podle splnění vstupní podmínky (viz blokové schéma)	1
"10"	adresa při nesplnění podmínky	0
"11"	adresa při splnění podmínky	0

Tabulka 2.5: Rozvětvení programu na dvě části

2.6.3.2 Překlad přiřazení

Do registru
Přiřazení do registru musí být řízeno mikrořadičem. Ten bude po taktech

procházet definované záznamy a nastavovat řídící signály, které povolují zápis do registru. Každý registr bude znát počet jemu náležících přiřazení. V případě jediného přiřazení stačí pro povolení zápisu pouze jeden signál, např. A_wr. V případě více různých zápisů do jednoho registru však bude potřeba tyto přiřazení od sebe odlišit. K tomu budou sloužit další signály označené vždy pořadovým číslem daného přiřazení, např. A_1.

Pro realizaci přiřazení následně definujeme speciální proces, který bude obsahovat seznam všech přiřazení vyskytujících se v programu. Každé přiřazení bude podmíněno nastavením řídících signálů odpovídajícím těm z mikrořadiče. Proces bude reagovat na každou změnu taktu, tudíž i na každou změnu řídících signálů. Při nastavení daného řídícího signálu v mikrořadiči se splní daná podmínka a bude provedeno přiřazení. Např.:

```
CHANGE_REG : process(CLK, RESET) is
begin
if RESET = '1' then
    ADR <= (others => '0');
elsif rising_edge(CLK) then
    if ( TX_wr = '1' and TX_0 = '1' ) then
        TX <= "0101";
    end if;
end if;
end process;
```

Do signálu
Přiřazení do signálů se sice nachází v programové části, avšak mikrořadičem být řízeno nemusí. Zde je na rozdíl od registrů vyžadováno, aby signály na změny hodnot reagovaly okamžitě v každé části programu. Přiřazení se tedy provede pomocí tzv. concurrent statement, které se vyskytují mimo proces. Např.:

```
AA <= A when B[0]==1 else 0;
```

2.7 Charakteristika jazyka Java

Pro implementaci překladače jsem zvolil jazyk Java, jelikož je velmi rozšířený a multiplatformní. Vytvořené programy jsou zcela portabilní (program vytvořený pod MS Windows bez problémů funguje pod Unixem a naopak). Vychází z jazyka C++, jemuž se také nejvíce syntakticky podobá. Je objektově orientovaný.(5)

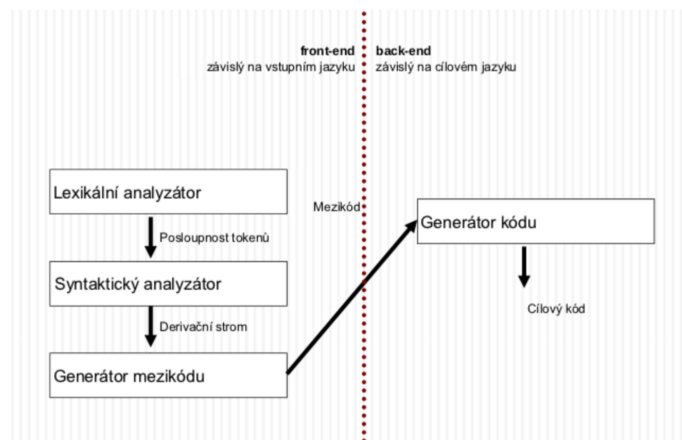
2.8 Vývojové prostředí

Pro vývoj překladače jsem zvolil vývojové prostředí NetBeans IDE, což je český open source projekt sponzorovaný firmou Sun Microsystems (nyní Oracle). Je napsán v jazyce Java, pro jehož vývoj je také primárně určen, nicméně podporuje řadu dalších programovacích jazyků.

Pro testování a simulaci VHDL kódu budou použity programy Xilinx ISE a ModelSim.

2.9 Návrh struktury překladače

Překladač může být navržen několika způsoby. Struktura zpravidla obsahuje dvě nejdůležitější části. První je závislá na vstupním jazyce (tzv. front-end) a druhá závisí na cílové architektuře (tzv. back-end). Překlad bude rozdělen do čtyř fází, jejichž navržená struktura je znázorněna na obrázku 2.2.



Obrázek 2.2: Návrh struktury překladače(6)

2.9.1 Lexikální analyzátor

Jedná se o první fázi překladu vstupního jazyka, jejímž úkolem je číst posloupnost znaků tvořících vstupní program a transformovat ji na posloupnost lexikálních elementů, dále tzv. tokenů, které jsou terminálními symboly pro syntaktickou analýzu.

Každému tokenu obecně odpovídá množina řetězců, jejichž tvar lze zadat regulárním výrazem nebo regulární gramatikou. Lexikální analyzátor je proto realizovaný konečným automatem. Syntax tokenů je stanovena referenční příručkou programovacího jazyka a obvykle to jsou identifikátory, klíčová slova, číselné a jiné literály (např. řetězce) a jedno příp. víceznakové speciální symboly (operátory, omezovače, oddělovače).

Lexikální analyzátor je nejčastěji realizován jako podprogram, který je volán syntaktickým analyzátorem a při každém vyvolání čte vstupní text, dokud v něm nerozpozná lexikální element. Společně s jeho hodnotou je syntaktickému analyzátoru předáván tzv. atribut, který předává informaci o typu tokenu (identifikátor, číslo, apod.).(8)

2.9.2 Syntaktický analyzátor - parser

Základní jednotkou syntaktického analyzátoru, tzv. parseru, je token přicházející z lexikálního analyzátoru. Parser v cyklu žádá lexikální analyzátor o sestavení nového tokenu. Z přijatých tokenů sestavuje objekty, které dohromady tvoří jednotlivé příkazy vstupního jazyka. Objekty jsou následně uloženy tak, aby zachovávaly posloupnost, ve které byly na počátku definovány.

Pro ukládání posloupnosti programu se nabízí varianta syntaktického stromu. Jedná se o vícedimenzionální seznam, který oproti jednoduchému zřetězenému seznamu nabízí mnohem vyšší komfort při následném procházení objektů a vyhledávání nadřazených funkcí. Syntaktický strom bude podrobněji popsán v části 3.4.

2.9.3 Generátor mezikódu

V této fázi se na základě struktury syntaktického stromu generuje kód ulehčující operace, které by jinak byly v následném generování čistého kódu příliš komplikované. Nevytváří se přímo kód cílového jazyka, ale kód v určitém pomocném jazyce, tzv. mezikód. V našem případě se tato část bude týkat generování záznamů mikrořadiče, čímž se utvoří základní struktura posloupnosti programu.

2.9.4 Generátor kódu

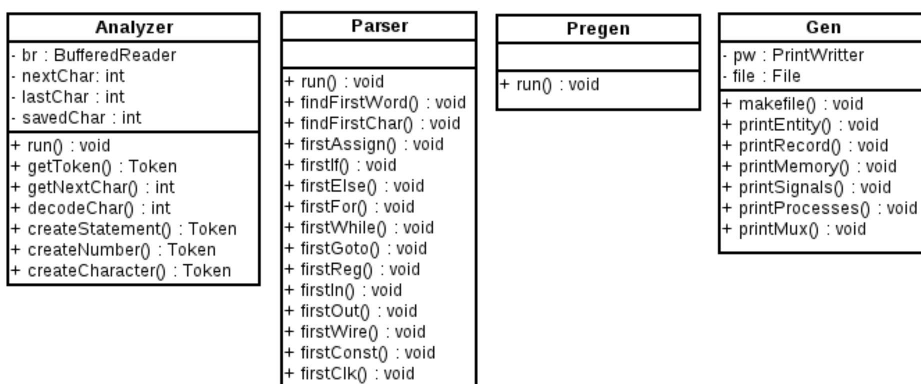
Vytváří soubor podle jména předaného v argumentu při spouštění programu. Na základě dat vygenerovaných v předešlých krocích je do výstupního souboru postupně po jednotlivých částech generován přeložený program vstupního jazyka.

Kapitola 3

Realizace

3.1 Architektura hlavních částí

Podle návrhu z části 2.9 je implementace překladače rozdělena na čtyři hlavní části. Lexikální analyzátor, syntaktický analyzátor, generátor mezikódu a generátor kódu. V programu jsou tyto části tvořeny pomocí čtyř statických tříd, jejichž architektura je znázorněna na obrázku 3.1.



Obrázek 3.1: Architektura hlavních částí programu

3.2 Lexikální analyzátor

3.2.1 Implementace třídy

Proces lexikální analýzy je v překladači implementován pomocí statické třídy *Analyzer*. Obsahuje několik metod, jejichž hlavní funkcí je načítání vstupního souboru a sestavování tokenů pro parser. Třída je definována jako statická, jelikož nevyžaduje vytvoření žádné instance.

3.2.2 Načtení vstupního souboru

Název vstupního souboru je programu předán pomocí argumentu při spuštění. Načtení souboru je provedeno ihned na počátku třídou Analyzer pomocí vstupní funkce *FileReader* do datového typu *BufferedReader*, z něhož může být následně snadněji čten po jednotlivých znacích.

3.2.3 Rozdělení znaků vstupního jazyka

Pro následnou syntaktickou analýzu chceme sestavovat pouze tokeny reprezentující konkrétní syntaktické prvky vstupního jazyka. Bílé znaky a komentáře je potřeba vyřadit, proto znaky vstupního jazyka rozdělujeme na přijímané (budou dále ukládány a zpracovány) a ignorované (budou považovány pouze za oddělovače syntaktických prvků vstupního jazyka a přeskakovány).

3.2.3.1 Přijímané znaky

Rozdělení na písmena, čísla a speciální znaky. Zde se jedná zejména o příkazy vstupního jazyka (např. *if*, *else*, *for*, *while*) a jejich atributy (inicializace, podmínky). Dále ukládáme deklarace datových typů, hodnoty apod. Zjednodušeně řečeno, jde o veškerý text kromě bílých znaků a komentářů.

Rozeznání daných typů je realizováno na základě hodnoty v ASCII tabulce.

3.2.3.2 Ignorované znaky

Jedná se o veškeré bílé znaky (mezera, nový řádek, tabulátor) a komentáře. Bílé znaky jsou v programu rozlišeny pomocí intervalu v ASCII tabulce od hodnoty 0 do hodnoty 32. Komentáře jsou uvozené dvojitým lomítkem.

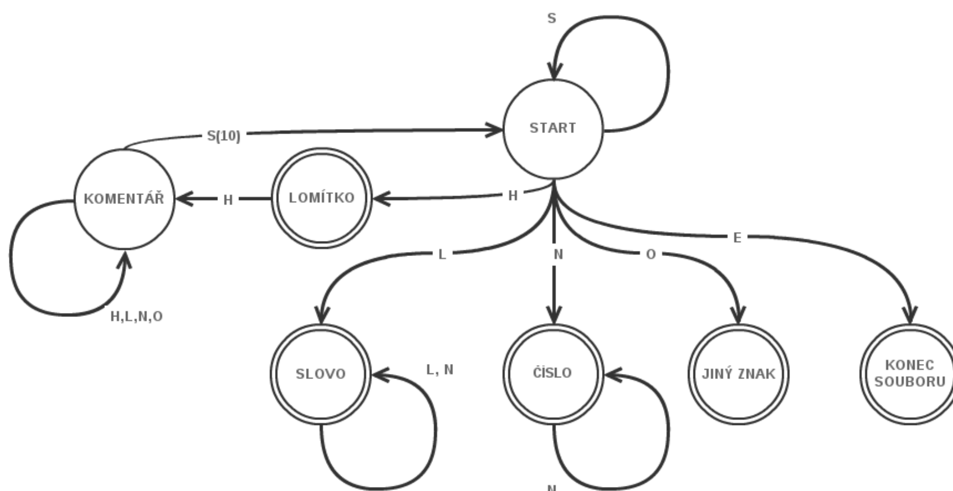
Při rozeznání těchto znaků lexikální analyzátor jednoduše nic neodesílá a čte rovnou další znak.

3.2.4 Sestavování tokenů

Při požadavku syntaktického analyzátoru na sestavení tokenu začne syntaktický analyzátor po jednom načítat znaky ze vstupu.

Každý čtený znak je nejprve dekódován metodou *decodeChar*() a označen příslušným symbolem. Význam použitých symbolů je zobrazen v tabulce 3.1.

Dekódovaný znak, resp. jemu přiřazená zkratka, je následně zpracována konečným automatem, který realizuje hlavní funkci lexikálního analyzátoru. Jeho stavový diagram je znázorněn na obrázku 3.2.



Obrázek 3.2: Konečný automat pro sestavování tokenů

vstupní znak	označující symbol	popis
’a’..’z’, ’A’..’Z’	L	písmeno (letter)
’0’..’9’	N	číslo (number)
EOF	E	konec souboru (end of file)
’\0’..’ ’	S	mezera a bílé znaky (space)
’/’	H	lomítko (slash)
’.’,’,’,’..’;’	O	ostatní znaky (others)

Tabulka 3.1: Význam symbolů použitých v automatu

Realizace konečného automatu vychází z předpokladů o vstupní syntaxi jazyka C podle analýzy v části 2.

3.2.5 Rozhraní lexikálního analyzátoru

Sestavené tokeny jsou nakonec uloženy jako objekty deklarované třídy $Token$, která o každém tokenu udržuje informaci o typu tokenu a o jeho hodnotě.

Tokeny nejsou na výstupu nijak ukládány, jelikož ihned po jejich sestavení jsou předány na vstup syntaktického analyzátoru, který se sám o sestavení nového tokenu dotazuje funkcí $getToken()$.

3.3 Syntaktický analyzátor

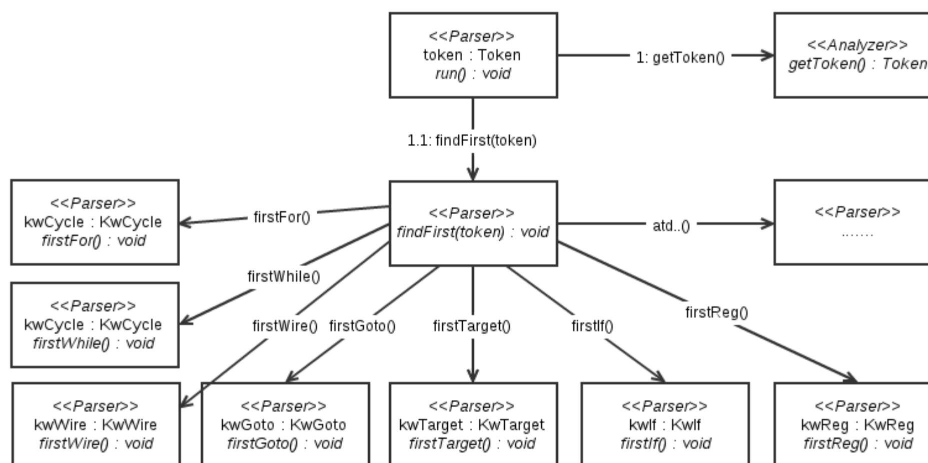
Cílem je načítat tokeny z lexikálního analyzátoru a třídit je do objektů, popisujících syntaktické prvky vstupního jazyka.

3.3.1 Implementace

Syntaktická analýza, neboli parser, je implementována pomocí jedné hlavní statické třídy *Parser* a 13 dalších tříd, kde každá tato třída popisuje jeden příkaz vstupního jazyka.

Těchto 13 tříd bylo vytvořeno na základě analýzy vstupního jazyka z části 2 a popisují následující elementy vstupního jazyka: přiřazení, cyklus, clk, if, else, else-if, goto, návěstí, deklarace registru, deklarace vodiče, deklarace konstanty, konec bloku, konec souboru.

Hlavní funkcí třídy *Parser* je načíst token z lexikálního analyzátoru a na jeho základě vybrat třídu, jejíž instance bude vytvořena. Zbytek tokenů k doplnění celého příkazu si již načte daná třída ve svém konstruktoru.



Obrázek 3.3: Komunikační diagram pro zařazení přijatého tokenu

3.3.2 Sestavení příkazů

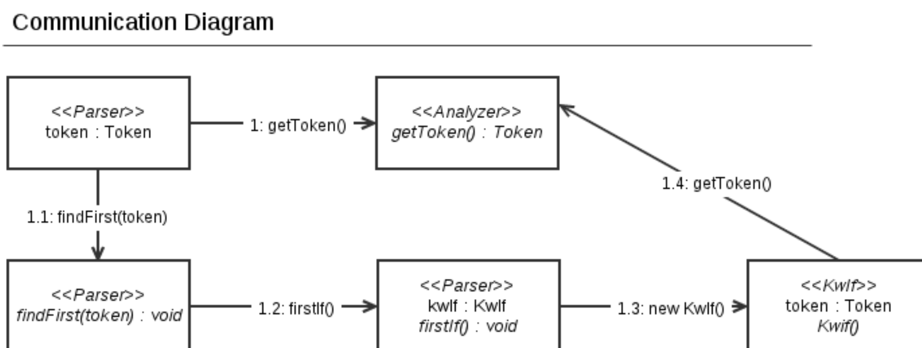
Každý příkaz vstupního jazyka lze podle stanovené syntaxe popsat regulérní gramatikou, tudíž lze sestavení každého příkazu realizovat konečným automatem.

Pro určení třídy, jejíž instance se má vytvořit, stačí stanovit počáteční stavy všech implementovaných automatů a podle prvního přijatého tokenu spustit konstruktor vybrané třídy, který daný příkaz načte.

Obrázek 3.3 znázorňuje komunikační diagram pro zařazení přijatého tokenu do správné třídy. Ta pak obsahuje automat pro načtení zbytku příkazu. Rozhodování funguje vždy na základě prvního přijatého tokenu.

3.3.2.1 Ukázka načítání deklarace funkce if

Obrázek 3.4 znázorňuje komunikační diagram tříd při načtení příkazu if. Obrázek 3.5 pak znázorňuje konečný automat realizující načítání tokenů funkce if. Příkaz je celkem sestaven z následujících tokenů: levá závorka, množina tokenů reprezentující podmínku, pravá závorka, levá složená závorka. Token if v automatu již začleněn není, jelikož na základě jeho dekódování v metodě $run()$ byla rozeznána funkce if a byl spuštěn tento automat.



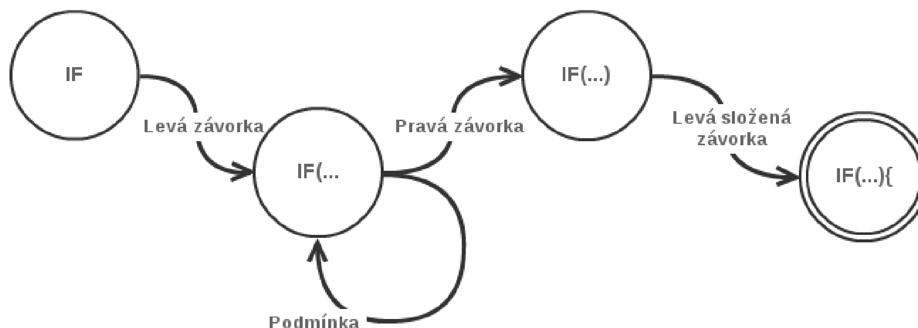
Obrázek 3.4: Komunikační diagram pro načtení funkce If

3.4 Ukládání dat

Jak bylo uvedeno v části 3.3, pro ukládání dat se využívá celkem 13 tříd. Stejně jako bylo zmíněno v části 2.6 o rozdělení dat na dvě skupiny, i tyto třídy lze rozdělit na datové (deklarační část) a programové (průběh programu).

3.4.1 Datové třídy

Deklarace registru
Uchováváme informace o názvu registru, jeho délce, směru, počtu vstupů a výskytu resetu registru.

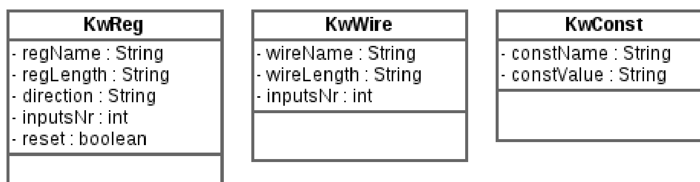


Obrázek 3.5: Konečný automat načítání tokenů funkce If

Deklarace vodiče
Uchováváme informace o názvu vodiče, jeho délce a počtu vstupů.

Deklarace konstanty
Uchováváme informace o názvu konstanty a její hodnotě.

Diagram tříd je znázorněn na obrázku 3.6.



Obrázek 3.6: Class diagram datových tříd

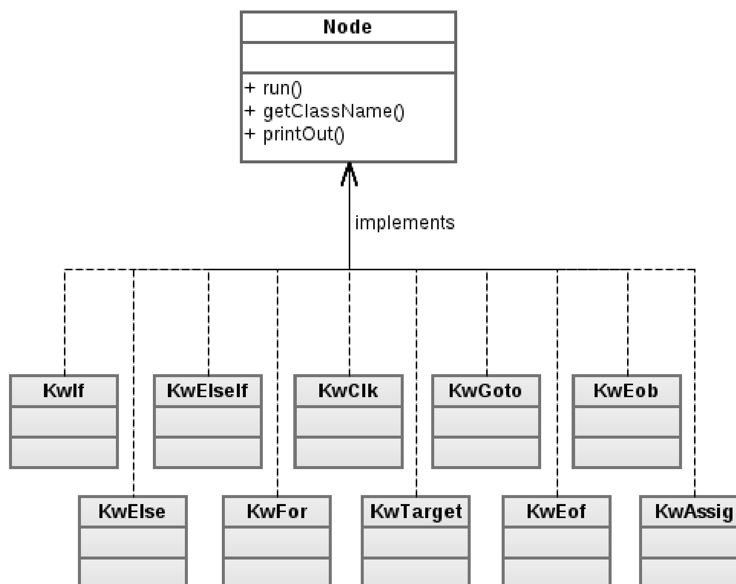
Instance datových tříd jsou uloženy ve statické třídě *List*, která reprezentuje úložiště všech dat načtených v parseru, které budou později potřebné při generování výstupního kódu.

3.4.2 Programové třídy

Jedná se o třídy popisující následující příkazy: přiřazení, cyklus, if, else-if, else, clk, goto, návěstí, konec bloku a konec souboru. Pozice těchto příkazů v programu ovlivňuje výsledný proces a při pozdějším generování výstupního kódu musí být zpracovávány ve stejném pořadí, ve kterém byly načteny.

Pro zjednodušení a sjednocení architektury těchto tříd je využito tzv. rozhraní. Výše zmíněné programové třídy implementují rozhraní $Node$, které po každé podřazené třídě vyžaduje implementaci metod $run()$, $getClassName()$ a $printOut()$. Díky těmto metodám nepotřebujeme při práci s objekty zjišťovat jaké třídě náleží, což je velmi výhodné při procházení uložených objektů, které jsou seřazeny na základě posloupnosti příkazů na vstupu, tudíž jejich výskyt je nepravidelný.

Diagram tříd je znázorněn na obrázku 3.7.



Obrázek 3.7: Class diagram programových tříd

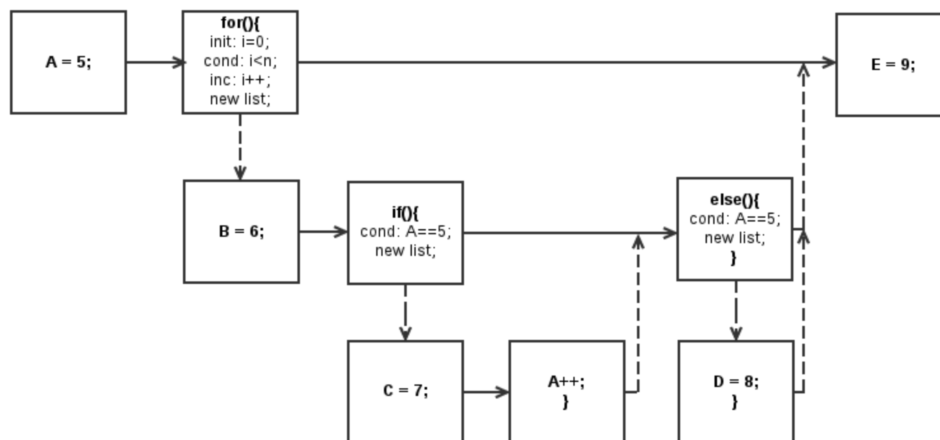
3.4.2.1 Uložení programových tříd do syntaktického stromu

Smysl syntaktického stromu spočívá v rozdělení posloupnosti programu na jednotlivé úseky. V případě vstupního jazyka C lze pro rozdělení úseků snadno využít složené závorky, které uvozují části programu vnořené ke konkrétnímu

příkazu. V praxi to znamená, že při každém výskytu levé složené závorky je z aktuálního pole vytvořen odkaz na další pole (další dimenzi), která realizuje tělo složených závorek. Při následném výskytu pravé složené závorky je aktuální pole ukončeno a ukazatel na aktuální pole je změněn opět na pole nadřazené, které tím pokračuje tam, kde při vytvoření nové dimenze skončilo.

Na obrázku 3.8 je znázorněn příklad syntaktického stromu pro následující vstupní kód:

```
A = 5;
for(i=0;i<n;i++){
    B = 6;
    if(A == 5){
        C = 7;
        A++;
    }else{
        D = 8;
    }
}
E = 9;
```



Obrázek 3.8: Ukázka syntaktického stromu

3.5 Generátor mezikódu

3.5.1 Mikrořadič

Stěžejní částí výstupního VHDL kódu je mikrořadič definovaný jako pole záznamů. Pořadí záznamů v mikrořadiči následně udává posloupnost provádění instrukcí VHDL programu. Pro vygenerování této části je tedy potřeba vycházet z posloupnosti načtených dat, která byla parserem uložena do syntaktického stromu (programová část vstupních dat).

3.5.1.1 Procházení programových dat

Programová data jsou uložena v syntaktickém stromu obsahujícím instance různých tříd. Bylo by velmi obtížné při procházení stromu zjišťovat, o potomka jaké třídy se u každé instance jedná, proto je každá třída obsahující programová data implementována jako potomek rozhraní $Node$, které definuje funkci $run()$.

Toho lze nyní snadno využít k procházení syntaktického stromu a volání funkce $run()$ na každý prvek stromu. Konkrétní tělo funkce $run()$ se mírně liší u každé funkce, avšak všechny funkce mají za úkol vytvořit svůj záznam do paměti mikrořadiče.

3.5.1.2 Generování záznamů mikrořadiče

3.5.1.2.1 Podmíněné funkce Funkce, které na základě určité podmínky větví program na dvě možnosti. Jedná se tedy o funkce if, $else-if$, for, $while$. V tabulce 3.2 můžeme vidět syntax pro realizaci rozskoku podmíněné funkce, jehož je docíleno pomocí různých následujících adres.

index adresy	NA	IC	komentář
"001"	"010"	1	vyhodnocení podmínky
"010"	?	0	podmínka nesplněna
"011"	"100"	0	podmínka splněna
"100"	"101"	0	tělo podmínky
"..."	"..."	0	tělo podmínky
?	"..."	0	konec těla

Tabulka 3.2: Ukázka rozskoku podmíněné funkce

Stěžejní problém zde nastává při nesplnění podmínky, kdy je potřeba skočit až za tělo části realizující splněnou podmínku. V této části kódu však ještě nevíme, kde tělo splněné podmínky končí.

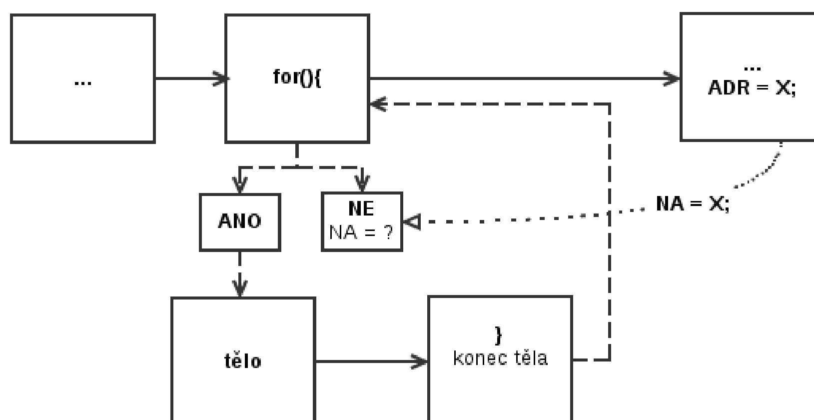
Jednoduché řešení spočívá v ponechání cílové adresy skoku prozatím prázdné, dokud nenarazíme na pravou složenou závorku, uzavírající tělo splněné pod-

mínky. Pro ukončující závorku také vytvoříme instanci záznamu do paměti. Adresa této instance nám bude sloužit jako cíl skoku při nesplněné podmínce.

Pro změnu cílové adresy nesplněné podmínky využijeme vlastností struktury syntaktického stromu. Jelikož se právě nacházíme v části vnořené pod podmíněným příkazem, pro nalezení onoho prázdného záznamu nám stačí získat první nadřazenou instanci (resp. instanci o uzel výše).

Cyklus

Při vyhodnocování cyklu (*for*, *while*) na adresu uzavírající závorky těla nastavíme adresu podmínky (*for*), čímž zajistíme cyklus. Adresu nesplněné podmínky nejdříve ponecháme prázdnou. Zapíšeme ji až v případě, kdy známe adresu příkazu následujícího po cyklu *for*. Tím zajistíme, že je při nesplněné podmínce celý cyklus přeskočen. Průběh zápisu vynechané adresy je znázorněn na obrázku 3.9. Vyplnění paměti je znázorněno v tabulce 3.3.



Obrázek 3.9: Realizace cyklu

ADR	NA	IC	komentář
"001"	"010"	1	vyhodnocení podmínky cyklu
"010"	?	0	podmínka nesplněna, NA = X
"011"	"100"	0	podmínka splněna
"100"	"..."	0	tělo podmínky
"..."	"001"	0	konec těla, cyklus na začátek
X	"..."	0	pokračování kódu

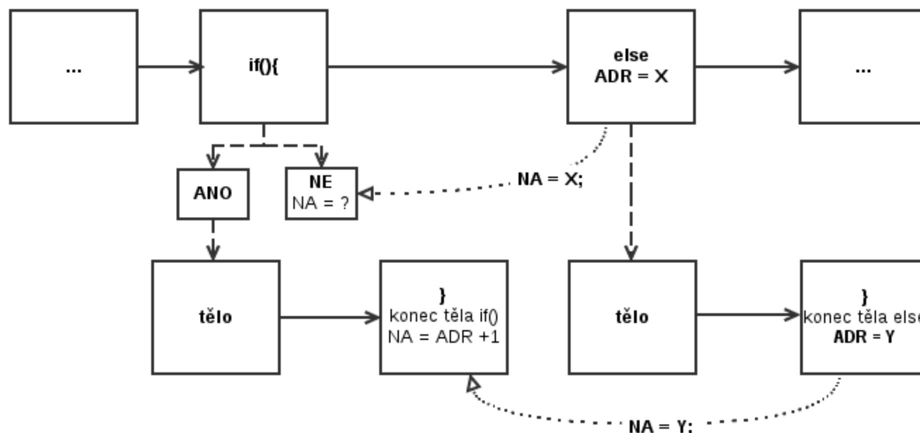
Tabulka 3.3: Ukázka vyplnění paměti pro cyklus

Podmíněný výraz

Při vyhodnocování podmíněných výrazů (*if*, *else-if*, *else*) se na adresu nesplněné podmínky stejně jako u cyklu zapisuje adresa bezprostředně následujícího záznamu za uzavírající závorkou. Podle pravidel syntaxe zde následuje buď další *else-if*, *else*, nebo zde pokračuje posloupnost programu jiným příkazem.

Na příští adresu uzavírající závorky se zatím zapisuje pouze inkrement této adresy a k přepsání adresy se vracíme až tehdy, pokud následuje *else-if* nebo *else*. V tom případě na následující adresy všech konečných závorek nastavujeme adresu poslední konečné závorky funkce *else*.

Průběh zápisu neznámých adres je znázorněn na obrázku 3.10. Vyplnění paměti je znázorněno v tabulce 3.4.



Obrázek 3.10: Realizace funkce if-else

ADR	NA	IC	komentář
"0001"	"0010"	1	vyhodnocení podmínky If
"0010"	?	0	nesplněna, NA = X
"0011"	"0100"	0	splněna
"0100"	"...."	0	tělo podmínky If
"...."	?	0	konec těla If, NA = Y
X	"...."	0	tělo Else
Y	"...."	0	konec těla Else

Tabulka 3.4: Ukázka vyplnění paměti pro funkci if-else

Vyhodnocení vstupní podmínky

Třetí pole v záznamu mikrořadiče indikuje číslo podmínky, která se má pro rozskok programu vyhodnotit. Toto číslo se inkrementuje při každém výskytu

podmínky. Při ostatních operacích je jeho hodnota nulová.

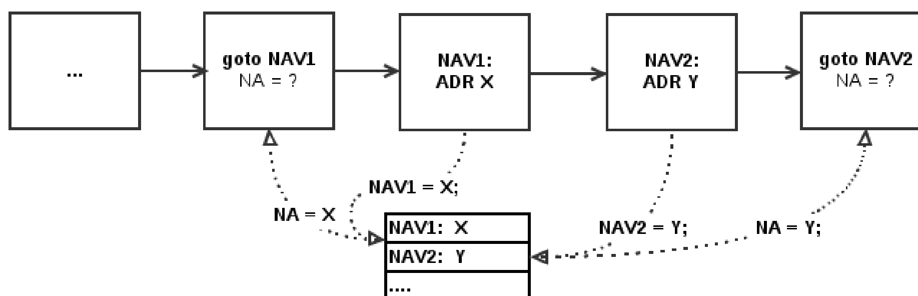
Ve výstupním VHDL kódu se na základě těchto podmínek generuje proces, který má za úkol všechny podmínky vyhodnocovat. Pomocí multiplexoru je vybrána aktuální podmínka a její výsledek je nastaven jako nejnižší bit následující adresy.

3.5.1.2.2 Přiřazení Zde je potřeba řešit případ, kdy se v programu vyskytuje více přiřazení za sebou. V tom případě totiž chceme šetřit se stavy mikrořadiče a všechny řídící signály pro přiřazení vyvolat rovnou v jednom záznamu.

K tomuto je využita metoda *seeNextNode*(). Pokud metoda vidí, že za aktuální instancí následuje opět instance přiřazení, program si pouze zapamatuje použité signály, žádný záznam negeneruje a přechází na následující instanci. Takto pokračuje, dokud nenarazí na poslední přiřazení v řadě. Zde se vytvoří záznam do paměti, který obsahuje i všechny předchozí signály. Stejná problematika byla uvedena již v části 2.5 pro příkaz *clk*.

3.5.1.2.3 Goto V problematice implementace funkce *goto* se vyskytuje stejný problém jako u podmínkových funkcí: potřeba skákat na adresu, kterou ještě neznáme. Hlavní složitost je však v tom, že u funkce goto zpočátku nevíme, zda se návěští nachází před *goto*, nebo až po něm. Navíc výskytů *goto* může být v programu více než jeden.

Jednochuché řešení spočívá ve využití dvou průchodů programu. Při prvním průchodu stromem pomocí funkce *run*() se adresy návěští spolu s jejich názvy uloží do předpřipraveného seznamu. Pro *goto* se zatím vytvoří pouze záznam s prázdnou následující adresou. Takto je program připraven adresaci dokončit při dalším průchodu v části generátoru výsledného kódu, kdy při vypisování záznamu goto vyhledá požadované návěští v seznamu připraveném ve třídě *List*. Průběh je znázorněn na obrázku 3.11.



Obrázek 3.11: Realizace funkce goto

3.6 Generátor kódu

Cílem generátoru je sestavit data upravená v části generátor mezikódu do takové podoby, která se následně vytiskne do cílového souboru.

Proces generování výsledného kódu je implementován pomocí jedné statické třídy, která je rozdělena na následující metody:

makeFile
Vytváří výstupní soubor na základě názvu předaného v argumentu programu při spuštění. Souboru je přiřazena přípona *.vhd*.

printEntity
Na začátek souboru tiskne výčet importovaných knihoven a entitu doplněnou o deklarace konstant (*generic*) a registrů definovaných jako vstupní nebo výstupní port. Zároveň jsou vytvořeny vstupní signály $RESET$ a CLK.

printRecord
Tiskne definici jednoho záznamu mikrořadiče, který udává velikost vektorů pro uložení následující adresy, vstupní podmínky a řídících signálů.

printMemory
Tiskne pole výše zmíněných záznamů, resp. paměť mikrořadiče, která udává průběh výsledného programu.

printSignals
Tiskne definice signálů. Kromě signálů zadaných na vstupu jako typ $WIRE$ je potřeba vytisknout také vnitřní registry (definované jako *REG*). Pro práci s pamětí je potřeba ke každému vektoru záznamu vytvořit signály, do kterých se budou hodnoty aktuálního záznamu ukládat. Také je potřeba vytvořit samotné přiřazení záznamu z paměti do signálu.

printProcesses
Tiskne proces pro změnu následující adresy a proces pro řízení zápisu do registrů. Oba procesy jsou doplněné o podmínku zohledňující signál $RESET$.

printMux
Tiskne proces pro vyhodnocení vstupních podmínek a multiplexor, který podle aktuální vstupní podmínky nastavuje nejnižší bit následující adresy.

printWire
Tiskne přiřazení do signálů definovaných jako $WIRE$ pomocí tzv. concurent statements.

Kapitola **4**

Testování

4.1 Použitý software

Pro překlad a simulaci práce byly postupně použity následující softwarové nástroje:

- NetBeans IDE pro překlad vstupního kódu, definovaného v souboru *input.c*, do výstupního kódu, generovaného do souboru *output.vhd*.
- Xilinx ISE pro kontrolu správnosti syntaxe vygenerovaného VHDL kódu a pro sestavení testbench.
- ModelSim pro odsimulování vygenerovaného VHDL kódu na základě sestaveného testbench.

Podrobný návod pro překlad (bez použití NetBeans) a simulaci se nachází na přiloženém CD spolu s vygenerovanými výstupními VHDL kódy a sestavenými testbench kódy.

4.2 Příklad 1: Násobička

První testovaný příklad je násobička. Jejím úkolem je vynásobit dvě celá čísla uložená v registrech A a B, výsledek uložit do registru C.

4.2.1 Vstup

Vstupní kód v jazyce C byl zadán následovně:

```
CONST n = 8;                        // definice konstanty
REG A[n], B[n], C[n], D[n], i[5];   // definice registrů
WIRE S[n+1], S2[n], AA[n];          // definice signálů (vodičů)

void main(){
```

```
    A = 1;
    B = 2;                      // přiřazení do registru
    C = 0;
    for(i=0; i<n; i++){         // cyklus for
        if (B[0] == 1) {        // podmíněný výraz if-else
            AA = A;
        }else {
            AA = 0;
        }

        S = C + AA;
        S2 = S[n:1];            // výběr požadovaných bitů
        C = S2;
        D = S[0]#D[n-1:1];      // zřetězení vektorů
        B = B>>1;               // bitový posun doprava
    }
}
```

4.2.2 Výstup a simulace

Výstupní VHDL kód je uveden na přiloženém CD.

Překlad programem NetBeans proběhl bez chyby. Výstupní VHDL kód byl importován do nově vytvořeného Xilinx projektu, kde proběhla kontrola syntaxe bez chyby. Pro odsimulování projektu v prostředí ModelSim bylo potřeba vytvořit testbench, který je rovněž na přiloženém CD. Simulace proběhla také bez chyby.

4.3 Příklad 2: Transmitter

Druhý testovaný příklad je transmitter, neboli vysílač, který simuluje odesílání po sériové lince.

4.3.1 Vstup

Vstupní kód v jazyce C byl zadán následovně:

```
// serial TX for clk = 50MHz and speed 9600
CONST n=16;

IN Data[n], Start;
OUT Ack, Done, TX;
OUT  B, C;
REG A_R[n+2];
REG  div[n];
```

```
REG     i[5];

void main(){
    TX=1;
    Ack=0;
beg:
    if (Start == 1) {goto go;}
    else {goto beg;}
go:
    clk;
    Ack = 1;
    Done = 0;
    clk;
    Ack = 0;
    A_R = 0 # Data # 1;
    for(i=0;i<(n+2);i++){
        TX = A_R[n+1];
        A_R = A_R[n+1:1] # 1;
        func_proved(IN A_R, OUT B, OUT C);
        div=0;
        while(div!=5208){
            div++;
        }
    }
    TX = 1;
    Done = 1;
    clk;
    clk;
    clk;
    goto beg;
}
```

4.3.2 Výstup a simulace

Výstupní VHDL kód je uveden na přiloženém CD.

Překlad programem NetBeans proběhl bez chyby. Výstupní VHDL kód byl importován do nově vytvořeného Xilinx projektu, kde proběhla kontrola syntaxe bez chyby. Pro odsimulování projektu v prostředí ModelSim byl vytvořen testbench, který je rovněž na přiloženém CD.

Při kompilaci v programu ModelSim vzniklo pouze jedno chybové hlášení kvůli špatně namapované konstantě v testbench. Po opravě chyby v testbench proběhla kompilace v pořádku. Program byl však kvůli příliš dlouhé délce trvání násilně ukončen definovaným procesem *timebomb*, který má za úkol

simulaci ukončit po 100 μs. To bylo způsobeno vysokými hodnotami podmíněných cyklů v zadání. Pro přehledné odsimulování je vhodné tyto hodnoty snížit řádově o stovky, jelikož několikanásobná iterace registru *div* od nuly do 5208 zabírá v simulaci zbytečně mnoho času. Po této úpravě hodnot proběhla simulace v pořádku.

Závěr

Zadání práce bylo úspěšně splněno a byl vytvořen překladač generující VHDL kód na základě vstupu v jazyce C. Výsledný program byl otestován na řadě příkladů, z nichž dva jsou uvedeny v části 4. Testování.

Hlavní motivací pro dokončení práce bylo vytvořit nástroj, který by mohl značně ušetřit čas při návrhu komplexních hardwarových projektů. Překladač je nyní schopen ušetřit práci vygenerováním algoritmů, které jsou oproti zdlouhavému hardwarovému popisu snadněji popsatelné v jazyce C. Až na výjimky, kdy vygenerovaný kód vyžaduje dodatečné manuální doladění uživatelem, je překladač plně funkční a použitelný pro návrh hardwarových projektů. Program byl zároveň navržen se zohledněním možnosti pokračujícího rozvoje a doplnění dalších vlastností.

V práci byly detailně popsány algoritmy pro lexikální a syntaktickou analýzu jazyka C, které je možno použít v řadě jiných aplikací zabývajících se překladem či parsováním jazyka tohoto typu. Dále byly vytvořeny algoritmy pro generování mikroprogramovatelného řadiče a pro sestavení dílčích procesů, které dohromady tvoří základní řídící jednotku výsledného mikrokódu.

Na základě odlišností datových typů jazyků VHDL a C byl vstupní kód doplněn o definice registrů a signálů, což nicméně uživatele nijak neomezuje, naopak to usnadňuje predikci požadovaného VHDL kódu. Pro jednodušší manipulaci s bitovými operacemi byl vstupní kód C doplněn také o operátory realizující bitový posun a zřetězení vektorů.

Navíc byly nad rámec zadání implementovány příkazy pro vygenerování komponenty a pro ovládání počtu taktů mikrořadiče.

Literatura

(1) Verilog.com [online]. 2011, [cit. 2012-05-14]. Dostupné z WWW: <http://www.verilog.com>

(2) CatapultC. In Wikipedia, the free encyclopedia [online]. 2012, [cit. 2012-05-14]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Catapult_C>

(3) SystemC. In Wikipedia, the free encyclopedia [online]. 2012, [cit. 2012-05-14]. Dostupné z WWW: <http://en.wikipedia.org/wiki/SystemC>

(4) Balík, M.: Programování a algoritmizace 1 [online]. 2012, [cit. 2012-05-14]. Dostupné z WWW: <https://edux.fit.cvut.cz/courses/BI-PA1/lectures/>

(5) Balík, M.: Programování v jazyku Java [online]. 2012, [cit. 2012-05-14]. Dostupné z WWW: <https://edux.fit.cvut.cz/courses/BI-PJV/lectures/>

(6) Janoušek, J.: Programovací jazyky a překladače [online]. 2011, [cit. 2012-05-14]. Dostupné z WWW: <https://edux.fit.cvut.cz/courses/BI-PJP/lectures/>

(7) Kernighan, B. W.; Rithie, D. M.: *Programovací jazyk C*. Computer Press, 2006, ISBN 80-251-0897-X, 285 s.

(8) Müller, K.: *Programovací jazyky*. České vysoké učení technické, 2002, ISBN 80-010-2458-X, 10–12 s.

(9) Novotný, M.; Bečvář, M.: Praktika návrhu obvodů [online]. 2006–2011, [cit. 2012-05-14]. Dostupné z WWW: <https://edux.fit.cvut.cz/courses/BI-PNO/lectures/>

(10) Rotem, N.: C to Verilog [online]. 2009, [cit. 2012-05-14]. Dostupné z WWW: <http://www.c-to-verilog.com>

Příloha A

Seznam použitých zkratek

VHDL VHSIC Hardware Description Language

VHSIC Very-High-Speed Integrated Circuit

FPGA Field Programmable Gate Array

ASIC Application-specific integrated circuit

ADR Address

NA Next Address

IC In Condition

CS Control Signal

REG Register

CON Condition

ASCII American Standard Code for Information Interchange

MUX Multiplexor

ISE Integrated Software Environment

IDE Integrated Development Environment

Příloha **B**

Obsah přiloženého CD

- readme.txt .... stručný popis obsahu CD
- exe .... adresář se spustitelnou formou implementace
- src
  - impl .... zdrojové kódy implementace
  - thesis .... zdrojová forma práce ve formátu LaTeX
- text .... text práce
  - thesis.pdf .... text práce ve formátu PDF
  - thesis.ps .... text práce ve formátu PS
- test .... adresář s testovacími příklady
  - howto.txt .... návod na spuštění a testování