VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VÝZKUMNÉ CENTRUM INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY
RESEARCH CENTRE OF INFORMATION TECHNOLOGY

POKROČILÉ TECHNIKY TRANFORMACÍ JAZYKA C DO JAZYKŮ POPISUJÍCÍCH HARDWARE

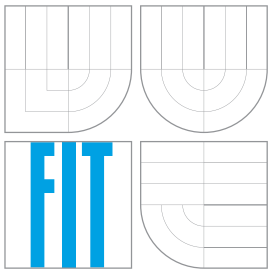
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN MICHALIK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VÝZKUMNÉ CENTRUM INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY
RESEARCH CENTRE OF INFORMATION TECHNOLOGY

POKROČILÉ TECHNIKY TRANFORMACÍ JAZYKA C DO JAZYKŮ POPISUJÍCÍCH HARDWARE

ADVANCED TECHNIQUES OF C-TO-HDL TRANFORMATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN MICHALIK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZDENĚK PŘIKRYL, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce se zabývá návrhem a realizací transformací aplikovaných při převodu jazyka C, použitého k popisu chování v rámci specifikace architektury v jazyce CodAL, do jazyků pro popis hardware. Cílem realizovaných transformací je buď umožnění zkrácení doby výpočtu, zvýšení frekvence nebo zmenšení plochy oproti původnímu řešení. V práci je popsána problematika převodu jazyka C do jazyků pro popis hardware a jsou zde popsány principy a analýza navrhovaných transformací. Výsledky transformací jsou zhodnoceny na základě vizualizace control data flow a register-transfer level grafů, simulace výsledných zdrojových souborů jazyka VHDL v programu ModelSim a pomocí syntézy těchto souborů v programu Xilinx ISE pro cílové FPGA Vertix 5.

Abstract

This thesis deals with proposal and implementation of advanced transformations used during generation HDL from behavior description written in C language, which is part of architecture specification in CodAL language. These transformations focus either on the reduction of time required for execution, increasing frequency or area reduction of target hardware. This thesis discusses main problems of C to HDL transformation and describes principles and analysis of proposed transformations. Transformations results are discussed based on the visualisation of control data flow and register transfer level graphs, simulation of generated VHDL source files in the ModelSim software and synthesis of these source files for target FPGA Vertix 5 in the Xilinx ISE software.

Klíčová slova

Lissom, transformace, AST, CDFG, RTLG, plánování, CodAL

Keywords

Lissom, transformations, AST, CDFG, RTLG, scheduling, CodAL

Citace

Martin Michalik: Pokročilé techniky tranformací jazyka C do jazyků popisujících hardware, bakalářská práce, Brno, FIT VUT v Brně, 2013

Pokročilé techniky tranformací jazyka C do jazyků popisujících hardware

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zdeňka Přikryla, Ph.D. V práci jsem uvedl veškeré literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Michalik
12. května 2013

Poděkování

Rád bych poděkoval vedoucímu své práce, panu Ing. Zdeňku Přikrylovi, Ph.D., za ochotu, odborné vedení a cenné rady poskytnuté při řešení mé bakalářské práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

V dnešní době jsme prakticky všude obklopeni výpočetními systémy, které se staly součástí našeho každodenního života. Největší skupinu výpočetních systémů se kterými se setkáváme, tvoří takzvané *vestavěné systémy*. Jedná se o malé výpočetní systémy se specifickým účelem, které jsou zpravidla zabudovány do určitého zařízení.

Jádro složitých vestavěných systémů mohou tvořit *aplikačně specifické instrukční procesory*,[1] což jsou procesory s upravenou vnitřní strukturou a instrukční sadou odpovídající jejich dané aplikaci. Výroba těchto mikroprocesorů je pak spojena s technologií *systému na čipu*,[2] která umožňuje integraci mikroprocesoru, paměti a dalších periferií na jeden integrovaný obvod.

Cena vestavěného systému je podstatně ovlivněna kromě jeho vlastní výroby, také cenou návrhu, vývoje a dobou následného uvedení na trh. Stále rostoucí trh s těmito systémy si tak žádá snižování doby návrhu a vývoje.

Cílem projektu *Lissom*[3] běžícího na Fakultě informačních technologií Vysokého učení technického v Brně je vytvoření integrovaného prostředí pro návrh a vývoj nových procesorů. Částí projektu je tvorba jazyka *CodAL*,[4] umožnujícího popis architektury a instrukční sady procesoru. Na základě popisu v tomto jazyce je vytvořen model procesoru, pro nějž je pak možné následně generovat další nástroje pro vývoj, optimalizaci a simulaci a také pro generování implementace architektury v hardware. Díky těmto nástrojům je pak možné zkrátit a zefektivnit celý proces vývoje procesoru.

Cílem této bakalářské práce je analýza, návrh a implementace transformací uplatňovaných při převodu jazyka *C* do jazyků pro popis hardware. během generování hardwarové reprezentace architektury popsané v jazyce *CodAL*. Hlavní přínos těchto transformací spočívá v možnostech zkrácení doby výpočtu, zvýšení frekvence nebo zmenšení plochy výsledného hardwaru oproti původnímu řešení.

Bakalářská práce obsahuje 6 hlavních kapitol. Kapitoly 2 a 3 seznamují čtenáře s problematikou popisu architektury a představují možnosti jazyka *CodAL*. V kapitole 4 je popsán proces převodu popisu chování v jazyce *C* do jazyků pro popis hardware a rozveden popis některých prostředků využívaných při tomto převodu. V následující kapitole 5 jsou navrženy a analyzovány transformace pro generátor implementace architektury v hardware a kapitola 6 popisuje podstatné detaily implementace těchto transformací. V kapitole 7 jsou pak zhodnoceny vlastnosti realizovaného řešení.

[1] *ASIP* – Aplication Specific Instruction set Processor
[2] *SoC* – System on a Chip
[3] http://www.fit.vutbr.cz/research/groups/lissom/
[4] http://codasip.com/products/codal/

Kapitola 2

Jazyky pro popis hardware

Vývoj *ASIP* se skládá z několika fází. Jedná se o návrh, simulaci a implementaci architektury procesoru, navržení a implementaci softwaru pro požadovanou aplikaci *ASIP* a nakonec intergace architektury do cílového systému [1]. Každá z těchto fází pak vyžaduje vlastní specifické nástroje, jejichž manuální generování provádí specializovaná skupina vývojářů. Naneštěstí však tyto aspekty vedou ke zvyšování náročnosti a prodlužování návrhu *ASIP* a tím pádem tak také k prodražování celého vývoje. V současné době tak panuje snaha o zvýšení automatizace a sjednocení specifikací pro jednotlivé fáze vývoje na různých úrovních abstrakce.

2.1 Hardware Description Languages

Pro modelování a simulaci procesorů lze použít některý z rodiny jazyků pro popis hardware.[1] Tyto jazyky umožňují popis struktury a chování, zachytit paralelismus a definovat časové závislosti [2]. Jelikož jsou však tyto jazyky primárně zaměřeny pro vývoj hardware, obsahují velké množství hardwarových specifikací, které nejsou při návrhu architektury nutné a zpomalují simulaci. U těchto jazyků je pak také obtížnější vyextrahovat instrukční sadu a popis modelu neobsahuje některé informace, například syntaxi assembleru.

Jako dva největší a nejpoužívanější zástupce *HDL* jazyků lze uvést jazyky *Verilog* a *VHDL*.[2]

2.1.1 Jazyk VHDL

VHDL je typovaný programovací jazyk s prostředky pro zachycení paralelismu, konektivity a explicitní vyjádření času. Jazyk *VHDL* je možné použít pro simulaci obvodů a také pro popis integrovaných obvodů určených výrobě [3].

Popis hardware se v jazyce *VHDL* skládá ze dvou základních částí [4]. První částí je deklarace entity, která popisuje rozhraní mezi danou komponentou a jejím okolím. Definuje vstupní a výstupní signály komponenty a seznam generických parametrů pro předávání dalších parametrů do entity.

Druhou částí je popis architektury. Architektura popisuje chování, případně strukturu komponenty a je svázána s entitou, která definuje její rozhraní. Jedna entita pak může mít více architektur. Architekturu každé komponenty lze popsat na úrovni struktury, chování,

[1] *HDL* – Hardware Description Languages
[2] *VHDL* – VHSIC Hardware Description Language

toku dat nebo lze případně jednotlivé druhy popisu kombinovat.

Behavioralní popis
V případě behaviorálního popisu se popisuje architektura dané entity pomocí jednoho nebo více procesů. Procesy umožňují modelovat sekvenční nebo kombinační logiku a popisovat chování celé entity nebo její části. Jednotlivé procesy mezi sebou komunikují prostřednictvím signálů a jsou tvořeny blokem sekvenčně prováděných příkazů. Cílem je popis chování architektury z hlediska změny výstupů v závislosti na změnách vstupních signálů, není tak zřejmá vnitřní hardwarová reprezentace.

U tohoto popisu však mohou nastat problémy při syntéze, neboť některé procesy mohou být v hardware obtížně realizovatelné.

Strukturální popis
Strukturální popis umožňuje popis architektury z hlediska struktury entity pomocí komponent a jejich vzájemného propojení pomocí signálů. U strukturálního popisu lze vytvořit hierarchický popis, kdy dílčí komponenty mohou být popsány na úrovni struktury pomocí jiných komponent či hradel nebo mohou být popsány na úrovni chování. V případě komponent na nejnižší úrovni hierarchie se používá vždy behaviorální popis. Jednotlivé komponenty reprezentují hotové prvky obvodu (například registr, sčítačka) a umožňují jejich znovupoužití pomocí pojmenovaných instancí.

Data-flow popis
Dataflow popis umožňuje popsat architekturu pomocí reprezentace toku informací. U tohoto popisu je chování kombinační logiky specifikováno pomocí paralelních příkazů, které jsou vyhodnocovány vždy, změní-li se hodnota jednoho ze signálu. U těchto příkazů nezávisí výsledek na pořadí zápisu ve zdrojovém programu.

Bližší informace o jazyku *VHDL* a jednotlivých úrovních popisu lze nalézt v literatuře zabývající se jazykem *VHDL* [2][3][4][5], ze kterých vychází tato kapitola.

2.2 Architecture Description Languages

Nevýhody *HDL* jazyků překonávají jazyky pro popis architektury.[3] *ADL* jazyky umožňují popis *ASIP* na vyšší úrovni abstrakce, kdy dovolují zachytit strukturální uspořádání a zároveň také specifikovat chování architektury procesoru pomocí popisu instrukční sady. Tyto jazyky přinášejí do procesu vývoje vyšší míru automatizace díky možnosti generování softwarových nástrojů pro další vývoj na základě specifikace modelu. Mezi generované nástroje patří například assembler, linker, simulátor apod. Díky těmto nástrojům je možné aplikační programy přeložit a simulovat a následně pak model modifikovat za cílem vytvoření nejvhodnější architektury pro danou aplikaci. ADL jazyky také umožňují generování hardwarové implementace s aplikací určitých omezujících podmínek, například na rozměry plochy, elektrické spotřeby, frekvence nebo doby výpočtu. Bližší informace k *ADL* jazykům lze nalézt například v literatuře a vědeckých publikacích zabývajících se *ADL* jazyky [6][7][8][1], ze kterých vychází kapitola.

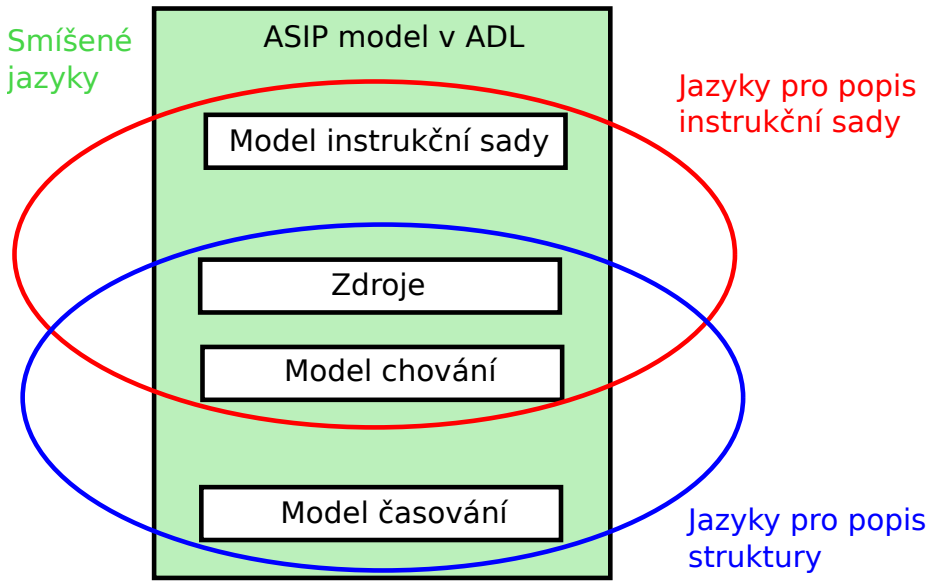
Jazyky ADL lze obecně rozdělit do tří následujících kategorií.

[3] *ADL* – Architecture Description Languages

Jazyky pro popis struktury
Tato skupina jazyků ADL se zaměřuje na popis strukturálních komponent a jejich vzájemné propojení v rámci architektury mikroprocesoru. Pro popis hardware u těchto jazyků se používá nižší úroveň abstrakce popisu například pomocí *meziregistrových přenosů*. Výhodou u těchto jazyků je, že stejný popis může být použit pro syntézu a zároveň pro generování softwarových nástrojů, nejsou však vhodné pro generování přenositelných překladačů. Mezi zástupce těchto jazyků patří například jazyk *MIMOLA* [7].

Jazyky pro popis instrukční sady
Tyto ADL jazyky se zaměřují se na popis instrukční sady procesoru tak, že specifikují sémantiku instrukcí, jejich sekvenci a latence. Detaily hardwarové struktury jsou v případě této skupiny ADL ignorovány. Primární použití těchto jazyků je určeno pro vytváření přenositelných překladačů vyšších programovacích jazyků, nejsou však vhodné pro generování cycle accurate simulátoru a syntézu. Zástupcem této kategorie je například popisný jazyk *nML* [7].

Smíšené jazyky
Skupina smíšených jazyků kombinuje předchozí kategorie a jejich přístupy – tyto jazyky rozšiřují instrukční jazyky o abstraktní popis zdrojů hardware. U těchto jazyků lze provést syntézu architektury a zároveň generovat vývojové nástroje (například simulátor) z jednoho popisu. Mezi zástupce těchto jazyků patří například jazyky *EXPRESSION* [7], *LISA* [7][1] a *CodAL*.



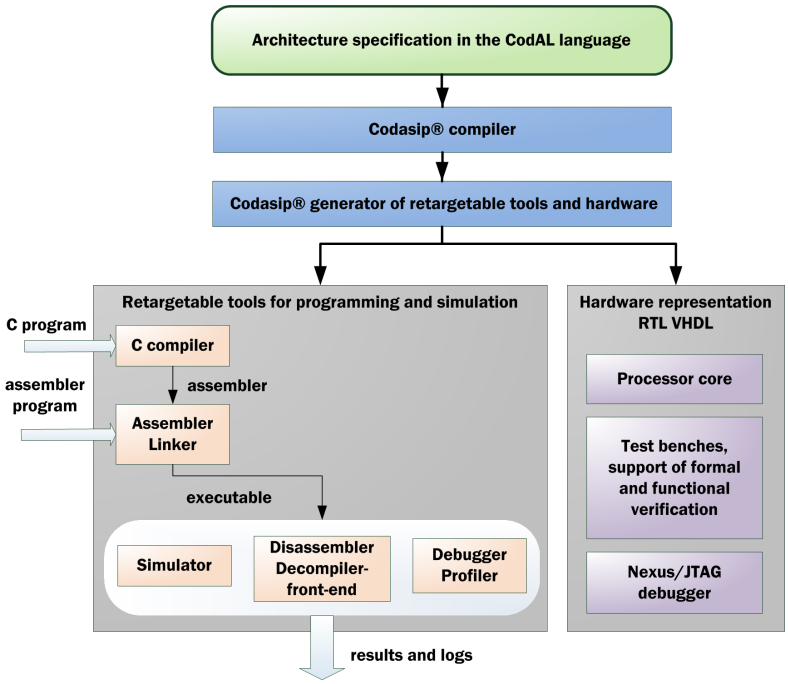
Obrázek 2.1: Rozdělení ADL jazyků [8]

Kapitola 3

Jazyk CodAL

Jazyk *CodAL*, který byl vyvinut v rámci projektu *Lissom*, slouží k rychlému prototypování aplikačně specifických instrukčních procesorů. Tento jazyk patří do kategorie *smíšených jazyků ADL* a částečně sémanticky vychází z jazyka *LISA*. Oproti jazyku *LISA* však jazyk *CodAL* nabízí větší modelovací sílu společně s kvalitnějšími a efektivnějšími generátory sofwarových nástrojů pro práci s instrukční sadou a simulaci architektury. Dále je možné na základě specifikace vytvořit implementaci architektury v hardwaru. Generátory pracující se specifikací architektury v jazyce *CodAL* tak lze rozdělit do dvou kategorií [9]:

- Generátory umožnující vytvoření softwarových nástrojů,
- Generátory realizující vytvoření implementace architektury v hardwaru.



Obrázek 3.1: Generátory fungující na základě modelu v jazyce *CodAL* [10]

3.1 Model procesoru v jazyce CodAL

Architektura různých aplikačně specifických procesorů se z důvodů jejich různé aplikace liší. Každý popis modelu procesoru v jazyce *CodAL* však musí obsahovat společné dvě části – popis zdrojů a popis instrukcí a událostí. Následující kapitoly, zaměřující se na vlastnosti a strukturu jazyka *CodAL*, vycházejí z manuálu k jazyku *CodAL* [10] a manuálu k v vývojovému prostředí *Codasip Studio* [11].

3.1.1 Popis zdrojů

Sekce popisu zdrojů obsahuje popis všech hadwarových komponent a jejich uspořádání v rámci architektury. Mezi komponenty, které je možné v rámci této sekce specifikovat patří například registry, pamětové prvky nebo prvky pro zřetězené zpracování instrukcí. V závislosti na typu daného prvku lze blíže specifikovat mimo jiné například počet čtecích a zápisových portů, typ uložených dat a nad nimi povolené operace nebo latenci čtení a zápisu dat.

Sekce popisu zdrojů musí vždy specifikovat programový čítač, paměťové prvky a jejich mapování.

Níže je uveden příklad deklarace zdrojů v jazyce *CodAL*, konkrétně programového čítače a paměti RAM [11]:

```
program_counter[32] pc;

memory bit[32] mem {
    .endianess = big,           // pořadí bytů
    .lau = 8,                   // nejmenší adresovatelná jednotka - 8 bitů
    .flags = {r, w, x},         // příznaky
    .size = 0x4000,             // velikost paměti - 64kB
    .latency = {1,1}           // latence čtení a zápisu
};
```

3.1.2 Popis instrukcí a událostí

Pro popis istrukční sady, chování a struktury slouží v jazyce *CodAL* bloky `event` a `element`. Popis těchto bloků je pak členěn do několika sekcí, které například definují textovou a binární reprezentaci instukcí (sekce `assembler` a `binary`), návratovou hodnotu (sekce `return`) nebo popis chování (sekce `semantics`).

Právě sekce `semantics`, určená pro popis chování bloků `event` a `element`, je z hlediska této bakalářské práce je nejpodstatnější. Pro popis chování v této sekci je použita omezená podmnožina jazyka *ANSI C*. Tato varianta *ANSI C* má určitá omezení, kdy mimo jiné například není možné v jakékoliv podobě používat ukazatele, příkaz `goto`, struktury a výčtové typy. Nicméně je možné vytvářet vlastní funkce v oddělených souborech a následně vkládat jejich hlavičkové soubory a pro potřeby simulace používat funkce ze standardní knihovny jako například funkce `printf`.

Níže je uveden příklad popisu chování pomocí `semantics` sekce [11]:

```
register bit[16] r1;
register bit[16] r2;

element swap_regs {
    use ... ;
    assembler {...};
    binary {...};

    semantics {      //popis chování v omezené podmnožině ANSI C
        int temp;
        temp = r1; r1 = r2; r2 = temp;
        printf("Swapping values %d <-> %lld",r1,r2);
    };
}
```

3.2 Generování hardwarové reprezentace architektury

V okamžiku, kdy během vývoje architektura splňuje požadavky aplikace a je dostatečně stabilní, je možné využít generátoru realizujícího vytvoření implementace architektury v hardwaru. Na základě modelu v jazyce *CodAL* je vygenerován hardwarový popis na úrovni meziregistrových přenosů, který je specifikován v několika zdrojových souborech jazyka *VHDL*.

Tyto vygenerované soubory obsahují mimo jiné například popis hlavní entity, která reprezentuje komunikační rozhraní procesoru, entity jednotlivých kontrolérů a dekodérů a popis funkčních jednotek.

Právě funkční jednotky jsou z hlediska této bakalářské práce nejdůležitější. Pro každý `event` nebo `element` v jazyce *CodAL* je vytvořena jedna funkční jednotka, a to i v případě, že je v modelu daný blok instanciován vícekrát. Existuje několik druhů funkčních jednotek:

- funkční jednotka pro funkce,
- funkční jednotka pro sekci `semantics` jednotlivých bloků `event` a `element`,
- funkční jednotka pro sekci `return`,
- funkční jednotka pro podmínku v sekci `timing` nebo `decoder`.

Každá z funkčních jednotek je synchronní konečný automat. V případě, že funkční jednotka obsahuje pouze kombinační logiku, je automat reprezentující funkční jednotku tvořen pouze jedním stavem, který je vyhodnocen během aktuálního hodinového cyklu. V případě, že je k vyhodnocení zapotřebí více cyklů, je automat reprezentující příslušné funkční jednotky tvořen více stavy. Stav je pak změnen při každé nástupné hraně hodinového signálu.

3.2.1 Časování a plánování jazyka C

Tělo funkčních jednotek je vytvořeno na základě popisu chování pomocí podmnožiny jazyka *ANSI C*. Popis chování se skládá z výpočetních příkazů a řídících struktur (podmíněné větvení, cykly) reprezentujících tok dat. Zápis v jazyce *C* je tak sekvencí příkazů. Cílový

hardware však pracuje paralelně a skládá se ze zdrojů, funkčních bloků a jejich vzájemného propojení. Z tohoto důvodu je nutné při generování hardwarové implementace z popisu architektury v jazyce *CodAL* provést paralelizaci a časování příkazů jazyka *C* pomocí *plánovače*.

Jednotlivým příkazům jsou pro zpracování přiděleny hardwarové zdroje. Každý z těchto zdrojů má definované latence čtení a zápisu, na které musí být brán zřetel během plánování. Tabulka latencí jednotlivých zdrojů je uvedena níže.

Zdroj	**Latence čtení**	**Latence zápisu**
signál	0	0
port	0	0
registr/registrové pole	0	1
paměť/ bankovaná paměť	specifikováno v popisu paměti (výchozí hodnota 0)	specifikováno v popisu paměti (výchozí a minimální hodnota 1)

Jelikož příkazy jazyka C mohou přistupovat ke stejným zdrojům a hodnoty těchto zdrojů ovlivňovat, vznikají mezi příkazy následující závislosti a konflikty:[1]

- Závislost R (Read),
- Závislost W (Write),
- Konflikt RAW (Read After Write),
- Konflikt RAR (Read After Read),
- Konflikt WAR (Write After Read),
- Konflikt WAW (Write After Write).

Závislost typu Read nebo Write znamená, že daný zdroj je v příslušném příkazu čten, případně jsou do tohoto zdroje zapisovány data. Konflikty pak reprezentují sekvenci operací nad daným zdrojem v rámci více nebo jednoho příkazu (například v případě `x = x+3`).

Konflikty WAR, RAR, WAW jsou podstatné pouze pro adresovatelné zdroje (například registrové pole) a pro zdroje, jejichž latence čtení nebo zápisu je jiná než výchozí hodnota. Konflikt RAW je pak podstatný pro veškeré zdroje.

Další důležitou vlastností zdrojů je počet zápisových a čtecích portů. V případě neadresovatelných prvků není stanoveno omezení na počet čtení v jednom hodinovém cyklu, avšak může být proveden pouze jeden zápis. U adresovatelných prvků (například registrová pole, paměti) je počet čtecích a zápisových portů definován v popisu zdroje.

Plánovač provádí časování jednotlivých příkazů jazyka *C* na základě principu *As Soon As Possible*. Tento princip znamená, že se *plánovač* pokouší vyhodnotit co nejvíce příkazů v jednom hodinovém cyklu. Během plánování však musí brát ohledy na latence jednotlivých zdrojů a na omezení čtecími a zápisovými porty. Pokud některé příkazy nemohou být současně provedeny v jednom taktu, musí být provedení následujícího příkazů odloženo o minimální dobu odpovídající latenci zdroje.

[1] Dále v práci bude pro závislosti a konflikty mezi jednotlivými zdroji používáno souhrnné označení *datové závislosti*

Kapitola 4

Převod jazyka C do jazyků pro popis hardware

Proces převodu sekvenčního popisu chování v jazyce *C* do paralelní hardwarové reprezentace v jednom jazyků *HDL* se skládá z několika kroků. Prvním krokem je převod bloku popisu chování do podoby *nenormalizovaného abstraktního syntaktického stromu.*[1]. Tento strom umožňuje zachytit stukturu popisu chování pomocí uzlů reprezentujících konstrukce jazyka *C* jako jsou například identifikátory, závorkové struktury nebo výrazy. Nenormalizovaný *AST* je následně normalizován pomocí transformací, mezi které patří například úprava příkazů inkrementace a dekrementace, transformace cyklu `for` na cyklus `while` nebo úprava názvů lokálních proměnných na unikátní název.

Z *AST* je v dalším kroku vytvořen *Basic Block Graf.*[2] Tento graf je tvořen bloky souvisejících příkazů bez řídích struktur. Jednotlivé bloky v grafu odpovídají logickým blokům ve zdrojovém kódu jazyka *C* a navazující bloky jsou pak mezi sebou v grafu propojeny. Nad *BBG* je prováděna optimalizace v podobě odstranění příkazů, které nebudou nikdy provedeny (tzv. mrtvý kód).

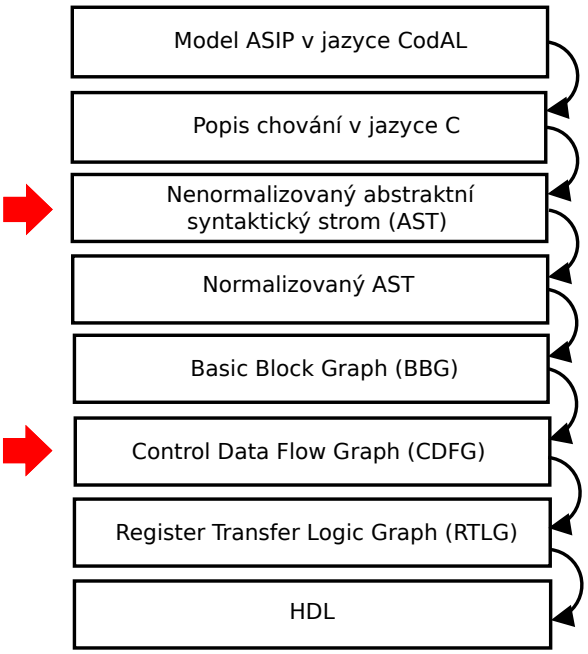
Na základě *BBG* je vytvořen *Control Data Flow Graf.*[3] Obecný *CDFG* je orientovaný graf umožnující zachytit kromě datových toků a operací také řídící konstrukce jako jsou podmíněné větvení nebo cykly. Jednotlivé uzly *CDFG* reprezentují datové operace a hrany mezi nimi pak vyjadřují datové závislosti a určují pořadí, v jakém jsou operace prováděny. Další bližší informace k *CDFG* lze nalézt například v literatuře zabývající se syntézou [12][13]. Nad *CDFG* pak probíhá plánování příkazů.

Po dokončení plánování je *CDFG* vstupem pro vytvoření *Register-Transfer Level Grafu*[4]. Vytvoření *RTLG* probíhá tak, že se sémantické akce přiřazené na hranách *CDFG* přesunují do časovaných uzlů, ze kterých hrany vystupují. Přesunem sémantických akcí do časovaných uzlů dochází ke zkrácení doby výpočtu, neboť není nutné vypočítávat výsledné hrany a následně jednotlivé sémantické akce. *RTLG* je v podstatě konečný automat, kde přechod do dalšího stavu znamená zpoždění o jeden takt.

RLTG je pak vstupem pro *komponentní systém*, který z dalších formálních modelů, jako jsou modely pro popis dekodéru nebo kontroléru, vytváří jednodnotnou reprezentaci. Ta je následně optimalizována a z ní je dále generován hardwarový popis v jazyce *VHDL* nebo *Verilog*.

[1] *AST* – Abstract Syntax Tree
[2] *BBG* – Basic Block Graph
[3] *CDFG* – Control-Data Flow Graph
[4] *RTLG* – Register-Transfer Level Graph

Jelikož se tato práce se zabývá transformacemi nad *AST* a hlavně *CDFG*, budou tyto části blíže představeny.



Obrázek 4.1: Průběh převodu jazyka C do HDL s označením působení bakalářské práce

4.1 Abstract Syntax Tree

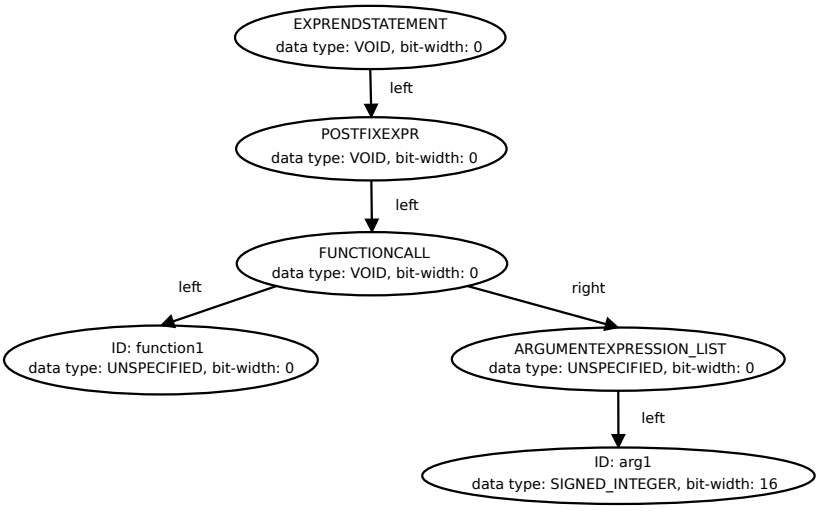
Z každého popisu v jazyce *C* je vytvořen nevyvážený binární strom s jedinečným identifikátorem určujícím příslušející funkční jednotku. Strom je tvořen speciálními uzly reprezentujícími konstrukce jazyka *C*, které obsahují následující informace:

- typ daného uzlu (například identifikátor, volání funkce, závorkové struktury),
- odkaz na rodiče, levého a pravého potomka,
- případná data, jejich typ, bitovou šířku a příslušný datový port,
- informace o zdrojovém souboru (název, číslo řádku),
- případný odkaz na globální zdroj.

AST lze rozdělit do dvou hlavních částí. První z nich je podstrom deklarací, obsahující deklarace proměnných. Každá z deklarací specifikuje identifikátor a datový typ proměnné. Druhou částí je pak část stromu obsahující jednotlivé výpočetní příkazy. Každý z příkazů je reprezentován podstromem specifikujícím typ příkazu, operátory, operandy a další informace o příkazu.

Pořadí uzlů ve stromu je podstatné – určuje pořadí příkazů z obou sekcí. Příkaz jazyka *C*, který je v popisu chování nejvýše, je vždy reprezentován nejlevějším podstromem v sekci deklarací nebo sekci příkazů. Další příkaz je pak pravým potomkem příslušného nadřazeného uzlu. Transformace spouštěné nad *AST* musí brát na toto pořadí zřetel a nesmí jej při běhu narušit, pokud tak nečiní záměrně.

V rámci dalšího zpracování při vytváření *BBG* je *AST* rozdělen na podstromy *AST*, ze kterých jsou vytvořeny například příkazy v *BBG* a *CDFG* nebo jsou použity pro podmínky u hran a příkazů *CDFG*. Níže je zobrazen příklad *AST* pro příkaz volání funkce `function(arg1)` v *CDFG*.



Obrázek 4.2: Ukázka AST pro příkaz volání funkce v CDFG

4.2 Control Data Flow Graph

Pro potřeby plánování příkazů se v rámci generátoru hardwarové reprezentace používá modifikovaný obecný *CDFG*. Tento graf je tvořen následujícím částmi.

První částí jsou orientované hrany reprezentující přechod mezi jednotlivými stavy *CDFG*. Každá hrana může mít přiřazenou podmínku nutnou pro přijetí této hrany a přechod do dalšího uzlu. Tyto podmínky vznikají například z důvodů podmínek provádění cyklů, podmíněných příkazů nebo volání funkcí. Každá hrana pak může mít přiřazeno několik sémantických akcí, reprezentovaných příkazy jazyka *C*. Obdobně pak jako u hran může být provedení každé sémantické akce podmíněno.

Druhou částí jsou uzly grafu reprezentující řídící struktury. Jednotlivé uzly *CDFG* se dají rozdělit do dvou skupin. První skupinou jsou uzly vyvolávající zpoždění o jeden hodinový cyklus. Do skupiny těchto uzlů patří například uzel reprezentující cyklus `while`, volání funkcí nebo speciální uzel pro synchronizaci. Druhou skupinou jsou uzly, které zpoždení o jeden takt nevyvolávají. Jedná se o rozhodovací uzly a patří zde uzly vznikající z konstrukcí `if` nebo `switch` a počáteční a koncový uzel grafu.

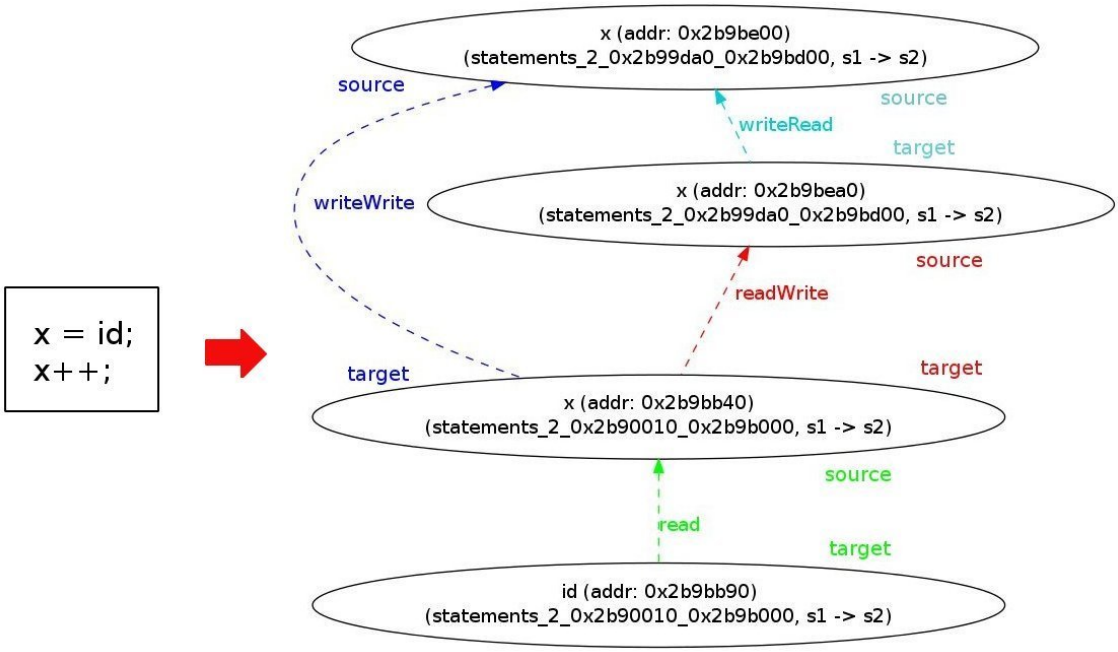
Nad *CDFG* grafem probíhá plánování příkazů. Při vytváření *CDFG* z *BBG* jsou na jednotlivé hrany přiřazovány sémantické akce. Mezi jednotlivými příkazy musí být zachováno pořadí odpovídající původnímu zápisu v jazyce *C* a musí být splněny všechny časové nároky datových závislostí, tedy mezi dvěmi datově závislými příkazy musí být minimálně takový počet uzlů vyvolávajících zpoždění, který odpovídá latenci konfliktní operace.

4.2.1 Datové závislosti v CDFG

Základem analýzy *CDFG* je analýza datových závislostí a konfliktů mezi jednotlivými příkazy. Na základě této analýzy lze pak nad grafem provést transformace vedoucí ke snížení

počtu hodinových cyklů nutných k provedení celé sémantické sekce. Výsledkem těchto transformací může být například paralelní provádění datově nezávislých cyklů.

Narozdíl od obecného *CDFG*, kdy jsou datové závislosti a konflikty reprezentovány pomocí hran, jsou závislosti v *CDFG* používaném pro potřeby plánování zachyceny pomocí speciálního orientovaného grafu, kde jednotlivé uzly grafu reprezentují zdroje a hrany pak určují typ datové závislosti. Jelikož je graf orientovaný, lze určit, který ze zdrojů je původcem závislosti a který je zdrojem závislým, a to i v rámci jednoho příkazu. Graf datových závislostí je přiřazen ke každému příkazu s alespoň jednou datovou závislostí a jeho průchodem lze získat veškeré datové závislosti daného příkazu v rámci celého popisu chování.



Obrázek 4.3: Příklad grafu datových závislostí

Kapitola 5

Analýza a návrh transformací

Hlavním cílem této bakalářské práce bylo navržení a analýza pokročilých transformací pro generátor realizující implementaci architektury v hardware. Navrhované transformace měly umožnit zejména zkrácení doby výpočtu, případně umožnit zvýšení frekvence či zmenšení plochy oproti původnímu řešení. Po konzultaci s vedoucím práce byly pro realizaci zvoleny transformace probíhající nad *AST*:

- Uprava vícenásobného přiřazení,
- Inicializace během deklarace.

A transformace probíhající nad *CDFG*:

- Rozlišení konfliktů sekvenčně prováděných příkazů,
- Rozlišení konfliktů v podmíněných větvích,
- Současné provádění více cyklů,
- Současné provádění více funkcí.

Tyto transformace pracují ve dvou fázích generování hardwarové reprezentace. První část transformací modifikuje příkazy popisu chování na úrovni *AST*. Druhá část transformací pak na úrovni *CDFG* upravuje časování příkazů provedené *plánovačem ASAP*, případně modifikuje samotný graf ještě před samotným *plánovačem* a následně s výhodou využívá jeho vlastností.

5.1 Úprava vícenásobného přiřazení

Norma jazyka *ANSI C* [14] umožňuje uživateli v zápisu zdrojovového kódu použít vícenásobného přiřazení, tedy přiřazení R-hodnoty do více L-hodnot pomocí jednoho příkazu. Příkladem takového přiřazení může příkaz `x = y = z = 5;`. Tento zápis je však pro další zpracování nevhodný a je potřeba jej převést do sekvence jednoduchých přiřazení odpovídajících původnímy příkazu. Cílem této transformace je optimalizace původní transformace vícenásobného přiřazení.

5.1.1 Analýza původního řešení

Původní transformace převádí příkaz vícenásobného přiřazení na sekvenci jednotlivých jednuchých příkazů přiřazení pomocí redukce původního příkazu. Transformace probíhá nad nenormalizovaným *AST* a při běhu mění jeho strukturu. Příkaz vícenásobného přiřazení je transformovován na sekvenci přiřazení odpovídající původnímu příkazu. R-hodnota celého výrazu je tak ve výsledku postupně propagována skrze jednotlivé L-hodnoty.

Nevýhodou tohoto řešení je počet hodinových cyklů nutných k vyhodnocení v případě, že se jedná o zápis do registru. Vzhledem k propagaci původní R-hodnoty, je nutné vždy počkat než je zápis do příslušného zdroje dokončen a až pak je možné jej opětovně číst. Optimálnějším řešením je přiřazení do všech zdrojů provést v jednom taktu.

Takovéto přiřazení však není možné provést ve všech případech, ale pouze pokud je splněna alespoň jedna z následujících podmínek:

- R-hodnota je konstanta,
- R-hodnota je hodnota registru,
- R-hodnota je hodnota signálu,
- R-hodnota je hodnota portu.

Není-li splněna žádná z těchto podmínek, je nutné redukci vícenásobného přiřazení provést původním způsobem.

5.1.2 Postup řešení

Hlavní částí transformace je redukce příkazu vícenásobného přiřazení na jednotlivé přiřazení. Transformace probíhá nad AST a pro průchod stromem je využito přístupu *preorder* [15]. Díky tomuto přístupu je nejdříve získáno nejlevější přiřazení z celého příkazu vícenásobného přiřazení, které musí být provedeno jako poslední. Následně je z celého výrazu získána výsledná přiřazovaná hodnota a určen její typ. Odpovídá-li získaná hodnota jedné z podmínek, je vytvořena kopie a ta nastavena jako R-hodnota nejlevějšího přiřazení. V opačném případě je ponechána původní R-hodnota.

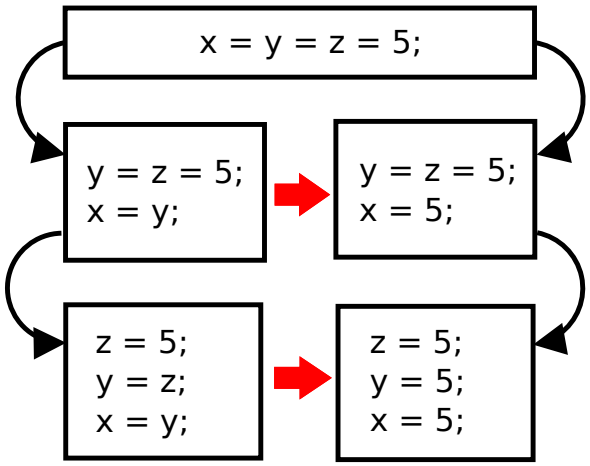
Následně je oddělen zbytek přiřazovaného výrazu a z této části výrazu je vytvořen nový příkaz. Tento příkaz je pak vložen do *AST* před původní příkaz (nyní již pouze nejlevější přiřazení) a transformace se nad změněným *AST* opakuje, dokud není celý původní příkaz zredukován.

Transformace k analýze a modifikaci využívá následující rekurzivní algoritmus:

Algoritmus 1: Transformace vícenásobného přiřazení

```
if typ uzlu == vícenásobné přiřazení then
    získej přiřazovanou hodnotu;
    urči typ přiřazované hodnoty;
    if je možné použít modifikaci then
        nastav přiřazovanou hodnotu jako R-hodnotu prvního přiřazení;
    else
        ponech původní R-hodnotu prvního přiřazení;
    odděl první přiřazení v příkazu vícenásobného přiřazení;
    vlož zbytek příkazu přiřazení před aktuálně zpracovávaný jako nový příkaz;
    opakuj transformaci nad novým AST;
pokračuj v transformaci nad potomky uzlu;
```

Průběh redukce příkazu vícenásobného přiřazení pomocí původní transformace a její modifikace je znázorněn na následujícím obrázku:



Obrázek 5.1: Původní průběh transformace vícenásobného přiřazení a její modifikace

5.2 Podpora inicializace během deklarace

Norma jazyka *ANSI C* [14] umožňuje uživateli deklarovat proměnné a zároveň je inicializovat na požadovanou hodnotu pomocí jednoho příkazu. Příkladem takovéhoto příkazu je například `short i=0;`. Původní podmnožina jazyka *ANSI C* použitá pro popis chování však tento zápis nepodporuje.

Cílem této transformace bylo umožnit podporu takovéhoto zápisu.

5.2.1 Analýza a postup řešení

Původní podmnožina jazyka C použitá pro popis chování neumožňuje použití inicializace během deklarace. Lokální proměnnou je zapotřebí nejprve deklarovat a následně inicializovat na požadovanou hodnotu pomocí dalšího příkazu. Na základě tohoto principu pracuje také tato transformace. Transformace probíhá nad nenormalizovaným *AST* a při běhu mění jeho strukturu.

Je-li při průchodu *AST* nalezena deklarace se současnou inicializací proměnné, je podstrom reprezentující inicializaci oddělen. Z tohoto podstromu je vytvořen nový příkaz, který je následně vložen do *AST* jako první příkaz bloku příkazů. Pro průchod *AST* během transformace je využito přístupu *preorder*. Díky tomuto přístupu je možné vkládat příkazy do bloku příkazů ve správném pořadí.

Průběh transormace umožňující použití inicializace během deklarace využívá následující algoritmus:

Algoritmus 2: Podpora inicializace během deklarace

```
if typ uzlu == blok deklarací then
    ulož tento uzel;
if typ uzlu == deklarace then
    získej výraz inicializace;
    získej první příkaz bloku příkazů;
    vytvoř nový příkaz z výrazu inicializace;
    vlož tento příkaz před první příkaz bloku příkazů;
    opakuj transformaci nad novým AST;
pokračuj v transformaci nad potomky uzlu;
```

5.3 Rozlišení konfliktů sekvenčně prováděných příkazů

Plánovač provádí časování jednotlivých příkazů na základě principu *ASAP* (více v kapitole 3.2.1). Pokud však plánovač zpracovává některý z uzlů vyžadujících více hodinových cyklů ke zpracování jako jsou například uzly pro cykly nebo volání funkcí, jsou další následující příkazy časovány nejdříve do následujícího taktu po zpracování příslušného uzlu. Řada z těchto příkazů však nemá datové závislosti, případně je mezi příkazy v konfliktu dostatečný počet cyklů pro uspokojení závislosti a tyto příkazy by tak mohly být vyhodnoceny v některém z dřívějších taktů.

Cílem této transformace je identifikovat v *CDFG* takovéto příkazy a následně u těchto příkazů provést úpravu časování, tedy přesunout jejich vyhodnocení na některou z předchozích hran.

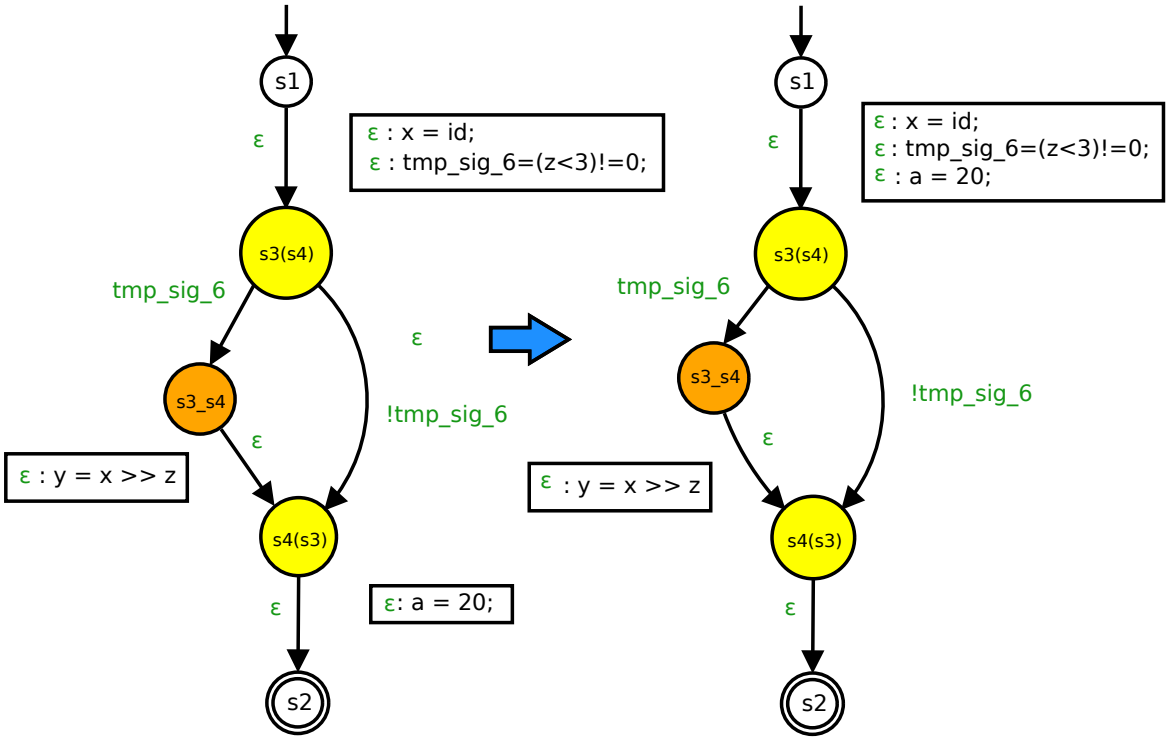
5.3.1 Příklad problému

Výše uvedený problém je možné ilustrovat následujícím příkladem. Uvažujme popis chování v jazyce C a předpokládejme, že všechny zdroje jsou registry (tedy zdroje `a`, `x`, `y`, `z`).

```
x = id;
if (z < 3)
   y = x >> z;
a = 20;
```

Mezi příkazy `x = id` a `y = x >> z` vzniká datová závislost typu RAW na zdroji `x`. Z tohoto důvodu je do jedné z podmíněných větví vložen plánovačem synchronizační uzel vyvolávající zpoždění o jeden takt a následující příkazy jsou časovány nejdříve na hranu následující po podmíněném větvení. Avšak příkaz přiřazení `a = 20` nemá žádnou datovou závislost na předchozích příkazech a bylo by jej tedy možné vyhodnotit ještě před samotným podmíněným větvením. Transformace má za cíl nalézt příkaz `a = 20` a přesunout jej na hranu předcházející podmíněnému větvení.

Na níže uvedeném obrázku 5.2 je zachycen stav *CDFG* před a po průběhu transformace.



Obrázek 5.2: Znázornění transformace rozlišení konfliktů sekvenčně prováděných příkazů

5.3.2 Analýza řešení problematiky

Hlavním problémem transformace je určení nejdřívějšího hodinového cyklu, do kterého lze daný příkaz přečasovat. K rozlišení jednotlivých hodinových cyklů lze využít příslušné uzly *CDFG*, které vyvolávají zpoždění o jeden takt (například synchronizační uzel nebo počáteční uzel cyklu). Transformace se tak dá rozdělit do následujících kroků.

Prvním krokem transformace je získání potencionální hrany pro přesun příkazu. Cílem pro přesun příkazu je vždy bezprostředně předcházející hrana aktuálně zpracovávané hrany, která však musí být součástí aktuálně zpracovávaného podgrafu. Podgrafem je míněna část *CDFG* reprezentující určitou řídící konstrukci nebo celý *CDFG*. Příkaz tedy nesmí být přesunut z nebo do cyklu, případně podmíněného větvení. Nejzazším místem pro přesun příkazu je první hrana v aktuálním podgrafu.

Po získání cílové hrany je nutné ověření datových závislostí zdrojů zpracovávaného příkazu vůči zdrojům příkazů přiřazených k cílové hraně. Nemá-li aktuálně zpracovávaný příkaz žádné datové závislosti na jiném příkazu, je možné jej automaticky přesunout na cílovou hranu. V opačném případě je však nutné rozlišovat typ dané závislosti a její důsledky. Přesun příkazu musí respektovat datové závislosti předcházejících příkazů a nesmí vytvářet konflikty. Aby tedy bylo možné příkaz přesunout, musí být v době zpracování potenciální cílové hrany pro přesun uspokojeny veškeré datové závislosti příslušného příkazu.

Posledním krokem je přesun příkazu v případě, že je splněna výše uvedená podmínka. Jelikož však přesun probíhá pouze o jednu nejbližší možnou hranu, může nastat situace, kdy je možné příkaz dále přesunout. Z tohoto důvodu se transformace nad daným příkazem opakuje a končí až v případě, kdy již není možné příkaz dále přesunout kvůli datovým závislostem nebo pokud byl již příkaz přečasován na první hranu aktuálního podgrafu.

5.3.3 Postup řešení

Během transformace jsou jednotlivé hrany *CDFG* grafu zpracovávány při průchodu grafem shora dolů. Díky tomuto přístupu nedochází k blokování případného přesunu příkazu jiným příkazem, který ještě nebyl zpracován a zároveň lze jednodušeji provádět transformaci v podgrafech reprezentujících cykly nebo podmíněné větvení.

Při získávání potencionální cílové hrany je nutné rozlišovat, ve které části grafu se nachází hrana příslušící k aktuálně zpracovávanému příkazu. Jak již bylo zmíněno, příkaz nesmí být přesunut z cyklu nebo podmíněného větvení. Zároveň je však možné takový řídící blok při získávání cílové hrany s výhodou přeskočit v případě, že se aktuálně zpracovávaný příkaz nachází na hraně bezprostředně následující za tímto blokem.

Rozhodnutí o možnosti přesunu příkazu na cílovou hranu probíhá na základě analýzy grafu závislostí daného příkazu. Při této analýze jsou získány veškeré příkazy obsahující zdroje vytvářející přímou datovou závislost pro zdroje aktuálně zpracovávaného příkazu. Z datových závislostí je dále získána operace vytvářející závislost (například zápis do registru). Aby bylo příkaz možné přesunout, musí být vzdálenost v hodinových cyklech mezi cílovou hranou a hranou s příkazem vytvářejícím závislost rovna nebo větší latenci příslušné operace.

V případě, že je možné příkaz přesunout na jinou hranu, je nutné společně s příkazem přesunou také veškeré spojené povolovací signály pro zápis a čtení jednotlivých zdrojů v příkazu.

Veškeré analýzy této transformace lze provádět přímo nad *CDFG* a celý postup řešení pak odpovídá následujícímu algoritmu:

Algoritmus 3: Rozlišení konfliktů sekvenčně prováděných příkazů

```
foreach všechny hrany CDFG do
    získej příkazy přiřazené k dané hraně;
    foreach všechny příkazy hrany do
        while true do
            získej potencionální hranu pro přesun;
            ověř datové závislosti příkazu vůči příkazům hrany;
            if příkaz je možné přesunout then
                přesuň příkaz na danou hranu;
                přesuň povolovací signály příkazu na danou hranu;
            else
                break;
```

5.4 Rozlišení konfliktů v podmíněných větvích

Řada příkazů vyžaduje ke svému vyhodnocení více než jeden hodinový cyklus. Může se jednat o příkazy reprezentující například čtení z paměti nebo zápis do registru. Tyto příkazy mohou vytvářet datové závislosti pro příkazy následující. Jelikož mohou takovéto příkazy následovat bezprostředně po sobě, musí existovat možnost, jak vynutit příslušný počet hodinových cyklů nutných ke správnému dokončení operace vytvářející datovou závislost. K tomuto účelu slouží v *CDFG* speciální synchronizační uzly, které reprezentují zpoždění o jeden takt.

V případě, že taková datová závislost vzniká v bloku podmíněného větvení, případně před ním, nerozlišuje plánovač konkrétní větev a synchronizační uzel je vložen až za, případně před blok podmíněného větvení. Jelikož však nejsou nikdy provedeny všechny větve tohoto bloku, dochází v řadě případů ke zpoždění i tehdy, není-li to nutné. Optimálním řešením by tedy bylo přesunout příslušné sychronizační uzly pouze do podmíněných větví způsobujících datovou závislost.

Cílem této transformace je nalézt takovéto synchronizační uzly a přesunout je do příslušných větví bloku podmíněného větvení.

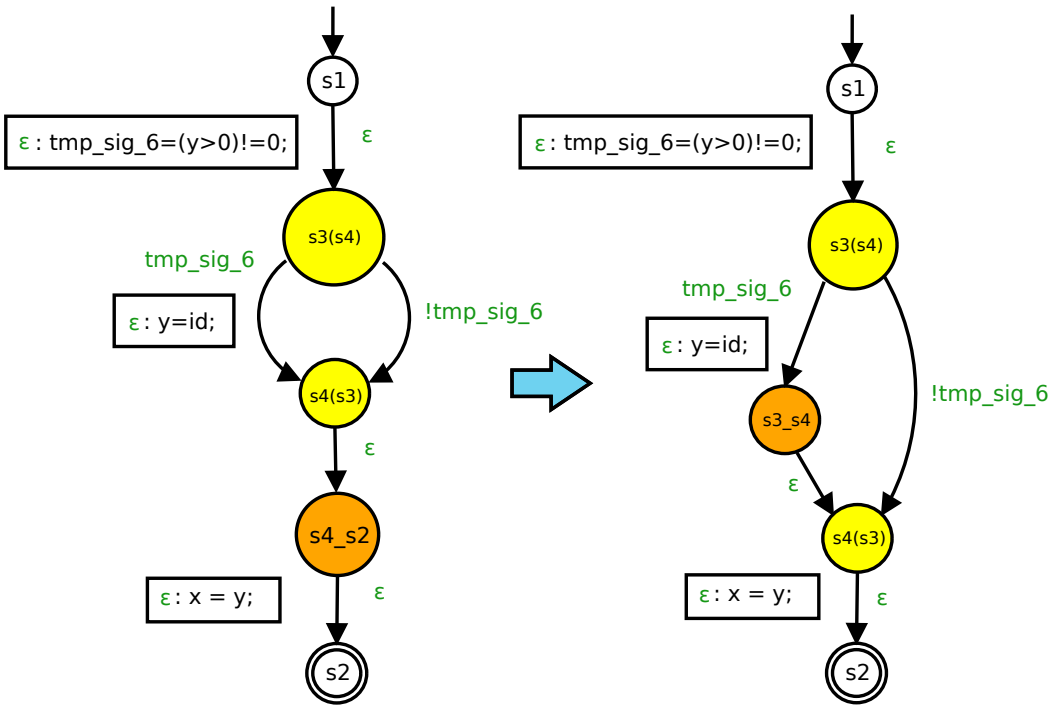
5.4.1 Příklad problému

Výše uvedený problém je možné ilustrovat následujícím příkladem. Uvažujme popis chování v jazyce *C* a předpokládejme, že všechny zdroje jsou registry (tedy prvky `x`, `y`, `id`).

```
if( y > 0 )
    y = id;
x = y;
```

Mezi příkazy přiřazení `y = id` a `x = y` vzniká datová závislost RAW na zdroji `y`, v případě, že bude podmínka splněna. Jelikož zpoždění vyžadované k zápisu hodnoty do registru je jeden takt, není možné oba příkazy provést v jednom taktu bezprostředně za sebou. Z tohoto důvodu plánovač vloží za blok podmíněného větvení synchronizační uzel vyvolávající zpoždění o jeden takt. Toto zpoždění je ale zbytečné v případě, že podmínka není splněna, vyhodnocena je druhá podmíněná větev a příkaz `y = id` tak není proveden. Transformace má za cíl přesunout synchronizační uzel do větve, která přímo ovlivnuje příkaz `x = y`.

Na níže uvedeném obrázku 5.3 je zachycen stav *CDFG* před a po průběhu transformace.



Obrázek 5.3: Znázornění transformace rozlišení konfliktů v podmíněných větvích

5.4.2 Analýza řešení problematiky

Hlavním problémem této transformace je nalezení příslušných hran podmíněného větvení s příkazy vyžadujícími přítomnost synchronizačního uzlu. Synchronizační uzel může být vyžadován kvůli datovým závislostem vznikajících ve více podmíněných větvích (například v příkazu `switch`) a je proto nutné určit tyto hrany v konstrukci podmíněného větvení. Pokud by byl synchronizační uzel vyžadován ve všech větvích selekce, nemá přesun synchronizačního uzlu význam.

Pro tuto transformaci jsou podstatné pouze synchronizační uzly vyskytující se bezprostředně před nebo za blokem podmíněného větvení. Tyto uzly však nenesou žádné doplňující informace o příkazu, který vytváří závislost vyžadující jejich přítomnost v *CDFG*. Rozhodnutí, na které z hran podmíněného větvení bude synchronizační uzel přesunut je nutné provést na základě datových závislostí příkazu na hraně před, respektive za synchronizačním uzlem. Toto řešení však přináší určitá omezení pro transformaci.

Jeden synchronizační uzel se může v grafu vyskytovat z důvodů datových závislostí více příkazů. Může nastat situace, kdy je synchronizační uzel vyžadován jak kvůli příkazu v podmíněné větvi, tak kvůli příkazu na bezprostřední hraně za blokem podmíněného větvení. V tomto případě není možné synchronizační uzel přesunout, neboť by došlo k vytvoření konfliktu. Aby bylo možné jednoznačně určit, zda je synchronizační uzel přítomen kvůli některému z příkazu v bloku podmíněného větvení, musí být na hraně před, respektive za synchronizačním uzlem pouze jeden příkaz. V opačném případě není možné transformaci provést.

Druhým omezením je přesun pouze jednoho synchronizačního uzlu. V případě, že operace vyžaduje více než jeden sychronizační uzel (například čtení z paměti), nemusí se na hraně mezi synchronizačními uzly vyskytovat žádný příkaz. V tomto případě není možné určit příkaz v bloku podmíněného větvení vyvolávající synchronizační uzly a určit tak cílou hranu pro jejich přesun. Řešením je tedy přesunovat pouze jeden uzel, u jehož následující hrany je přiřazen právě jeden příkaz.

Řešením výše uvedené problematiky by bylo doplnit ke každému synchronizačnímu uzlu seznam příkazů vyžadujících přítomnost tohoto uzlu. Tuto možnost ale nebylo možné v době řešení práce realizovat.

5.4.3 Postup řešení

Během transformace jsou jednotlivé hrany *CDFG* grafu zpracovávány při průchodu grafem shora dolů. Tento přístup umožňuje zachovat pořadí hran v grafu a tím pádem i pořadí případných synchronizačních uzlů.

Jak bylo zmíněno získání cílové množiny hran pro přesun synchronizačního uzlu probíhá na základě analýzy datových závislostí. Pro veškeré zdroje příkazu na hraně před, případně za synchronizačním uzlem, je získán graf závislostí. Z tohoto stromu jsou následně získány příkazy vytvářející datové závislosti. Pokud je některý z příkazů přířazen k hraně podmíněného větvení, je tato hrana vložena do cílové množiny hran pro přesun uzlu.

Přesun synchronizačního uzlu spočívá v rozdělení každé hrany z množiny cílových hran a vytvoření kopie původního synchronizačního uzlu. Ten pak vložen mezi nové hrany. Původní synchronizační uzel je na konci transformace odstraněn a jeho okolní hrany sloučeny do jedné.

Celá transformace je realizována na úrovni *CDFG* a celý postup řešení odpovídá následujícímu algoritmu.

Algoritmus 4: Rozlišení konfliktů v podmíněných větvích

```
for všechny hrany CDFG do
    urči okolní uzly hrany;
    if synchronizační stav then
        získej příkazy na předchozí/následující hraně;
        if počet příkazů na hraně != 1 then
            continue;
        ověř datové závislosti a urči konfliktní hrany;
        if počet konfliktních hran != 0 then
            for konfliktní hrany ke zpracování do
                rozděl konfliktní hranu;
                vlož na hranu synchronizační uzel;
            sjednoť hrany a příkazy před a za původním synchronizačním uzlem;
            odstraň původní synchronizační uzel;
```

5.5 Současné provádění více cyklů

Cykly patří k základním řídícím programovým strukturám umožňujícím vytvářet složitější programy. A právě cykly se výraznou měrou podílí na počtu taktů nutných k vykonání bloku příkazů. Většina cyklů je totiž vyhodnocována opakovaně a každé tělo cyklu zpravidla vyžaduje k vyhodnocení více než jeden hodinový takt. S počtem narůstajících cyklů v popisu chování tak narůstá také počet požadovaných hodinových cyklů. Některé z těchto cyklů jsou však vzájemně datově nezávislé a jejich vyhodnocení by mohlo probíhat současně v jednom okamžiku.

Cílem této transformace je nalézt v *CDFG* skupiny datově nezávislých cyklů a tyto cykly upravit pro paralelní vyhodnocování.

5.5.1 Příklad problému

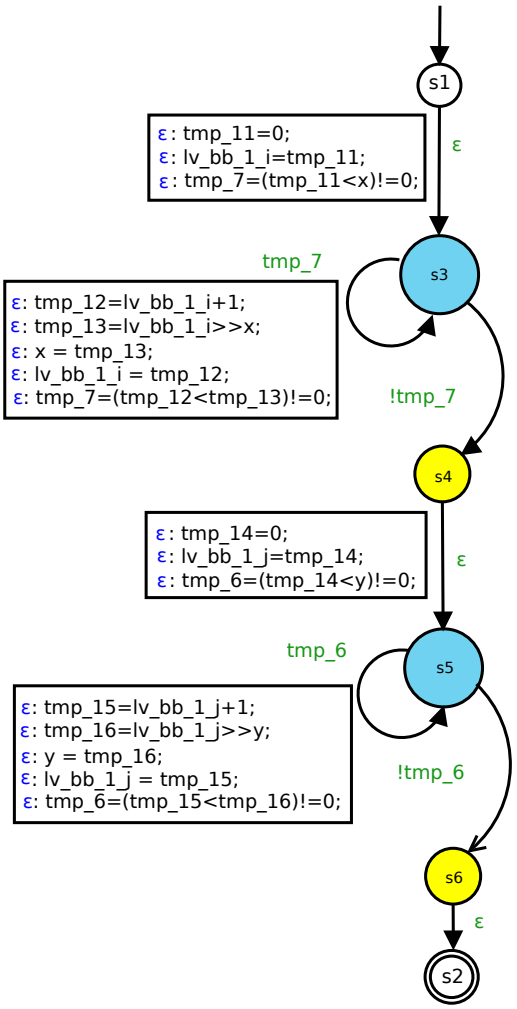
Zmiňovaný problém lze ilustrovat následujícím příkladem. Uvažujme popis chování v jazyce *C*, kdy předpokládejme, že veškeré zdroje v příklady jsou deklarovány jako registry:

```
i = 0;
while(i < x){
   x = i >> x;
   i++;
}
j = 0;
while(j < y){
    y = j >> y;
    j++;
}
```

Oba z uvedených cyklů jsou ekvivalenty cyklu `for` s inicializací a iteračním výrazem. Příkazy těla druhého cyklu jsou vyhodnocovány až po ukončení cyklu prvního. Jelikož

však oba cykly inkrementují jinou proměnnou, přistupují k jiným zdrojům a jsou datově nezávislé. Příkazy těchto cyklů tak mohou být prováděny současně v jedné cyklící smyčce. Cílem transformace je tak vytvoření nové cyklící smyčky a sjednocení těl cyklů. Většina cyklů je však složitějších než uvedený ilustrační příklad a je nutná pokročilejší analýza, zda-li je možné cykly sjednotit.

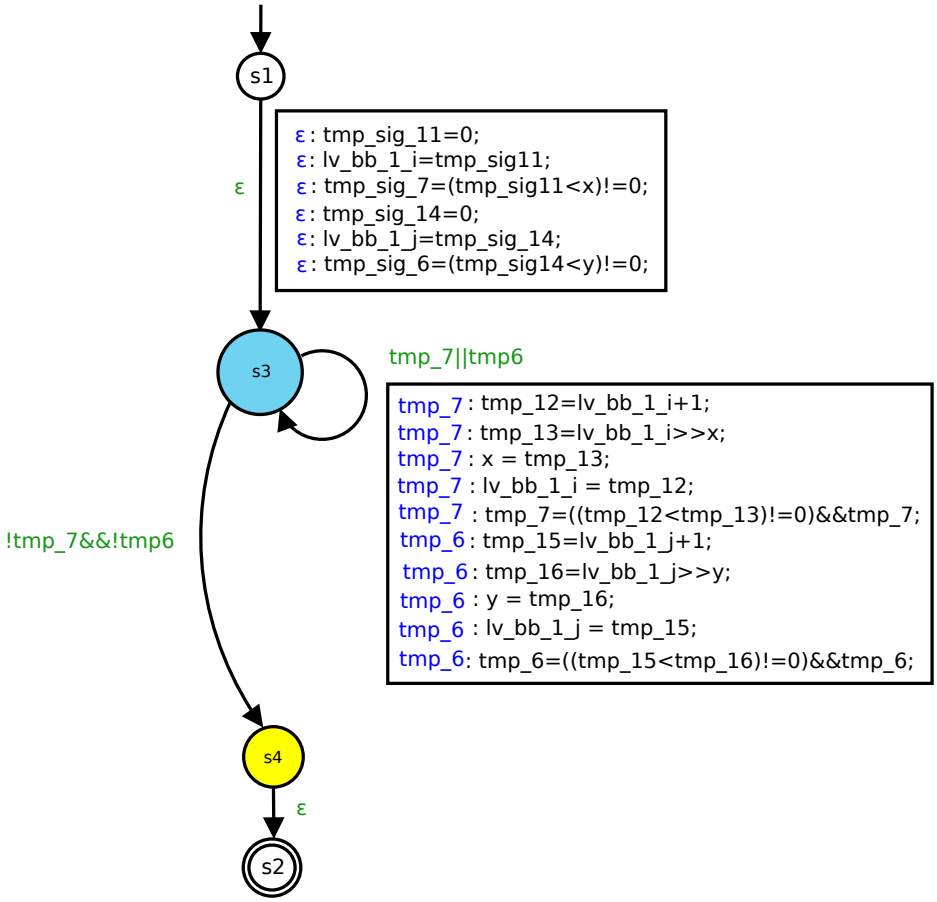
Na obrázku 5.4 je pak zachycen stav *CDFG* před a na obrázku 5.5 po průběhu transformace.



Obrázek 5.4: CDFG před transformací sjednocení cyklů

5.5.2 Analýza řešení problematiky

Každý cyklus je v *CDFG* grafu reprezentován dvojicí uzlů. Prvním z nich je cyklící uzel. Z tohoto cyklu vychází 2 hrany, kdy první je prováděna vždy, je-li splněna podmínka cyklu. K této hraně jsou přiřazeny příkazy těla cyklu, jejichž provádění není dále podmíněno. Druhá hrana tvoří přechod do druhého uzlu cyklu. Jedná se o synchronizační uzel, sloužící pro dokončení zápisů z poslední iterace cyklu. Přechod do tohoto cyklu je proveden v případě, že již není splněna podmínka cyklu. Součástí cyklu jsou také hrany vznikající díky příkazům `break` a `continue`.



Obrázek 5.5: CDFG po transformaci sjednocení cyklů

Hlavním problémem této transformace je ověření datové nezávislosti jednotlivých dvojic cyklů. Aby bylo možné dva cykly sjednotit, nesmí první cyklus nijak ovlivňovat průběh vykonávání cyklu následujícího. Z tohoto důvodu musí být na příkazech prvního cyklu také nezávislá inicilizace cyklu druhého. Inicializací jsou v tomto případě myšleny příkazy předcházející samotný cyklus, které následně ovlivnují příkazy uvnitř těla cyklu. Cykly je tedy možné sjednotit, nevytváří-li příkazy prvního cyklu datové závislosti pro inicializaci a příkazy druhého cyklu.

Paralelní vyhodnocení cyklů musí mít z hlediska dat stejný efekt jako by byly cykly vyhodnoceny sekvenčně. Sjednocované cykly se však zpravidla liší v počtu iterací a nový cyklící stav tyto iterace musí respektovat. Počet opakování nového cyklu tak odpovídá vyššímu z dvojice počtu iterací původních cyklů. Aby nedocházelo k chybnému vyhodnocení příkazů po sjednocení v nadbytečných iteracích je nutné podmínit vykonávání příkazů. Podmínkou pro každý příkaz je podmínka původního cyklu. Nový cyklus tak bude vykonán vždy, bude-li splněna alespoň jedna z původních podmínek cyklů, ale příslušné příkazy budou vyhodnoceny pouze, pokud bude splněna jejich nová podmínka.

5.5.3 Postup řešení

Sjednocení cyklů probíhá vždy na jedné úrovni *CDFG* grafu. V případě, že cyklus obsahuje vnořené cykly, je transformace nejdříve spuštěna nad těmito vnořenými cykly. Cykly jsou

tak sjednocovány od nejhlubší úrovně zanoření.

Pro ověření datové nezávislosti jsou získány veškeré příkazy těla a inicializace cyklu, který je z dané dvojice v *CDFG* následníkem druhého uzlu z dvojice. U těchto získaných příkazů jsou pak ověřovány datové závislosti vůči příkazům těla prvního cyklu.

Pro vytvoření nové cyklící smyčky není nutné vytvářet novou dvojici uzlů. Lze výhodně využít uzly prvního cyklu a ty upravit. Samotné sjednocení cyklů se pak skládá z několika kroků.

Prvním krokem je úprava podmínek cyklu a příkazu pro jejich vyhodnocení. V původním řešení cyklů je podmínka reprezentována hodnotou signálu. Jelikož je však při sjednocení nutné podmínit veškeré příkazy původní podmínkou cyklu, tedy i příkazy nastavující tuto podmínku, není možné použít signál, jelikož ten neumožňuje současný zápis a čtení hodnoty v jednom taktu a docházelo by k uváznutí. Řešením tohoto problému je transfomace podmínkového signálu (v uvedených příkladech se jedná o `tmp_sig_..`) na registr, který již umožňuje podmínit veškeré příkazy v cyklu. Zároveň je nutné upravit příkaz pro vyhodnocení podmínky tak, aby po sjednocení při dokončení iterací původního cyklu nedocházelo k nežádoucímu dalšímu vyhodnocení.

Druhým krokem je úprava podmínek nové cyklící smyčky. Tyto podmínky vzniknou kombinací podmínek původních cyklů. Novou podmínkou pro vykonání další iterace cyklu bude disjunkce původních podmínek pro vykonání iterace, podmínkou pro ukončení cyklu pak bude konjunkce podmínek pro ukončení cyklu. Jelikož je však podmínkový signál transformován na registr, je ukončení vyhodnocení cyklu penalizováno zpožděním o jeden takt, odpovídajícímu latenci zápisu do registru.

Posledním krokem transformace je sjednocení těl cyklů a přesun inicializace druhého cyklu před nový cyklus. Sjednocení probíhá napojením podgrafu reprezentující tělo druhého cyklu na podgraf cyklu prvního. Je však nutné upravit cílové uzly speciálních hran vznikajících kvůli příkazům `break` a `continue`. Ostatní úpravy zajistí samotný plánovač. Následně jsou odstraněny původní uzly druhého cyklu.

Jelikož při transformaci dochází ke změně struktury *CDFG* grafu, je transformace nad daným grafem opakována. Celý postup řešení je možné zachytit následujícím algoritmem:

Algoritmus 5: Sjednocení provádění cyklů

```
najdi veškeré uzly reprezentující cyklus v aktuálním podgrafu;
pro každý z těchto uzlů spusť znovu transformaci;
vytvoř veškeré možné dvojice uzlů reprezentující cykly;
foreach dvojice uzlů cyklů do
    urči pořadí cyklů v aktuápním podgrafu;
    získej všechny příkazy obou cyklů;
    získej všechny inicializační příkazy druhého cyklu;
    ověř datové závislosti příkazů druhého cyklu vůči prvnímu cyklu;
    ověř datové závislosti inicializace druhého cyklu vůči prvnímu cyklu;
    if cykly jsou datově nezávislé then
        proveď podmínění příkazů obou cyklů původní podmínkou cyklu;
        proveď sjednocení podmínek obou cyklů jako novou podmínku prvního cyklu;
        přesuň příkazy druhého cyklu do prvního cyklu;
        přesuň inicializaci druhého cyklu před první cyklus;
        odstraň uzly druhého cyklu;
        opakuj transformaci nad aktuálním podgrafem;
```

5.6 Současné provádění více funkcí

Na počtu hodinových cyklů nutných k vyhodnocení bloku příkazů se významnou měrou kromě cyklů podílí také volání funkcí, jejichž tělo není tvořeno pouze kombinační logikou. Takovéto funkce ke svému vyhodnocení vyžadují více než jeden hodinový cyklus. Při volání funkce je využíváno jiné funkční jednotky, přičemž různá volání funkcí mohou mít rozdílné délky vyhodnocení. Po zavolání funkce se čeká, až je daná funkce vyhodnocena a sekvenční volání funkcí má za následek nárůst počtu potřebných hodinových cyklů. Řada z těchto funkcí by však mohla být vyhodnocena současně.

Cílem této transformace je nalézt v *CDFG* skupiny datově nezávislých funkcí a volání těchto funkcí upravit pro paralelní vyhodnocování.

5.6.1 Příklad problému

Zmiňovaný problém lze ilustrovat následujícím příkladem. Uvažujme popis chování v jazyce C a funkce `factorial` a `fibbonacci`, které nerekurzivně počítají faktoriál a fibbonacciho posloupnost daného čísla.

```
a = factorial(5);
b = fibbonacci(5);
```

Obě funkce ke svému volání potřebují z důvodu charakteru výpočtu více než jeden hodinový takt. V řešení plánovače jsou funkce volány sekvenčně – funkce `fibbonacci` je volána až po skončení výpočtu funkce `factorial`. Jelikož však dané funkce nepotřebují k výpočtu globální zdroje ani mezi nimi nevzniká datová závislost, bylo by možné vyhodnocení těchto volání provádět současně.

Na obrázku 5.6 je zachycen stav *CDFG* před a po průběhu transformace.

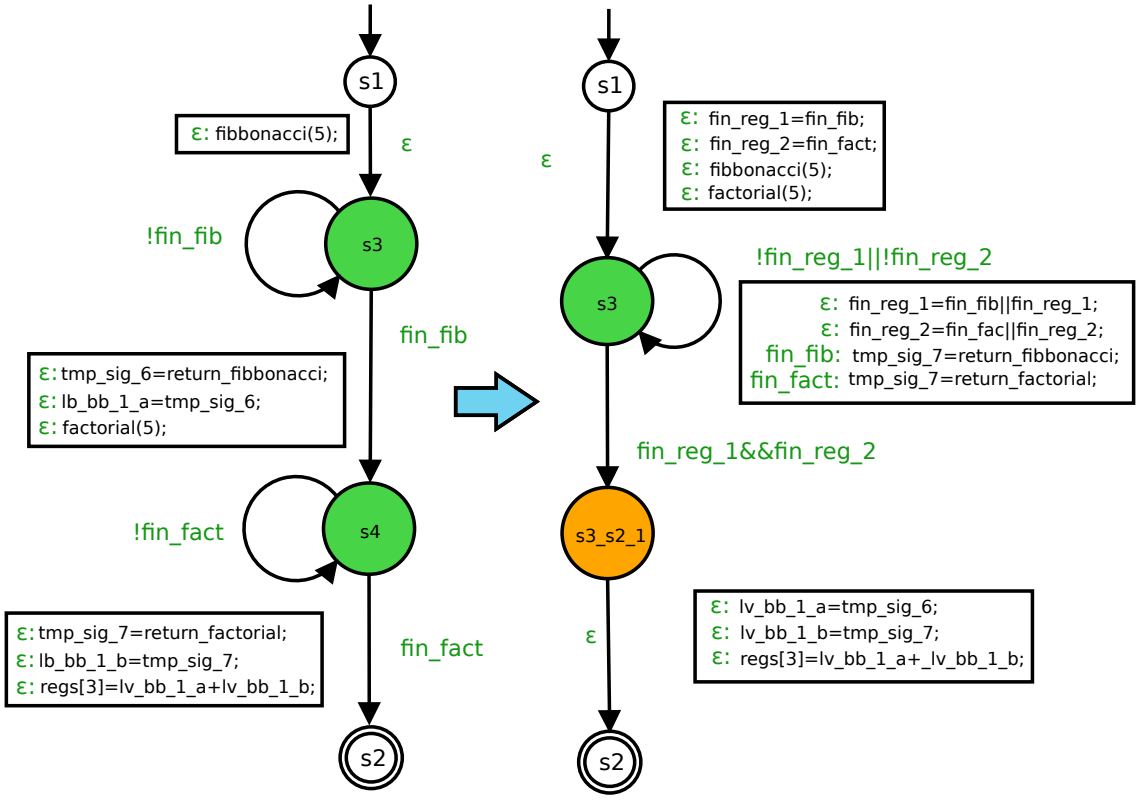
5.6.2 Analýza řešení problematiky

Hlavním problémem této transformace je určení, kdy je možné sjednotit volání funkcí. Každé volání funkce je v *CDFG* reprezentováno pomocí speciálního uzlu. Tento uzel v *CDFG* následuje za příkazem, ve kterém je daná funkce volána. Tento uzel je obdobou uzlu reprezentujícího cyklus, neboť obsahuje cyklící smyčku generující zpoždění do doby než je dokončeno volání funkce. Sjednocení volání funkcí pak spočívá ve sjednocení těchto cyklících uzlů. Aby však bylo možné volání funkcí sjednotit je nutné aby bylo splněno několik podmínek.

První podmínkou je, aby volané funkce byly součástí stejného podgrafu v rámci *CDFG*. Není možné sjednotit funkce, které jsou součástí různých řídících bloků. Například tak není možné sjednotit funkci uvnitř cyklu s funkcí vně cyklu.

Druhou podmínkou pro sjednocení volání funkcí je zamezení konfliktů volání stejné funkce. Jelikož je každá funkce reprezentována pouze jednou funkční jednotkou, je v jednom okamžiku možné provádět pouze jedno volání této funkce. Při sjednocení volání tak nesmí nastat situace, kdy dva různé příkazy na jedné hraně volají tutéž funkci. Nesmí docházet také k nepřímým konfliktům volání, kdy funkce volaná v rámci prvního příkazu ve svém těle volá funkci další, která je konfliktní s funkcí volanou v příkazu druhém.

Třetí podmínkou je datová nezávislost sjednocovaných funkcí, kdy výsledek volání jedné funkce nesmí být použit jako parametr funkce jiné. Zároveň nesmí docházet ani k nepřímé závislosti, kdy výsledek není předáván přímo, ale nejprve modifikován v rámci jiných příkazů a následně předán.



Obrázek 5.6: Znázornění transformace sjednocení volání funkcí

Poslední podmínkou je pak zamezení přístupu ke stejným globálním zdrojům. Pokud jsou splněny veškeré uvedené podmínky, je možné volání funkcí sjednotit. Jelikož se však sjednocované funkce mohou lišit v délce vyhodnocení, je nutné zajistit uchování návratové hodnoty příznaku ukončení funkce a zamezit opakovanému volání stejné funkce.

5.6.3 Postup řešení

Analýzy zabývající se konflikty volání stejné funkce a přístupu ke stejným globálním zdrojům není možné provádět přímo nad *CDFG*, neboť tento graf obsahuje pouze příkaz volání funkční jednotky a neobsahuje bližší informace o funkcích. Tyto analýzy je nutné provádět nad abstraktním syntaktickým stromem příslušných funkcí.

Rozlišení konfliktů volání stejné funkce spočívá v získání všech *AST* pro veškeré příkazy volání funkcí příslušných uzlů volání funkcí, včetně zanořených volání. Tím získáme dvě množiny *AST*, jednu pro každý uzel volání funkcí, kde každý *AST* obsahuje identifikační klíč. Pokud existuje neprázdný průnik množin *AST* na základě identifikačního klíče, není možné provést sjednocení. Rekurzivní volání funkcí není zapotřebí řešit, jelikož rekurze není povolena již na úrovni modelu.

Rozlišení konfliktů přístupů ke stejným globálním zdrojům probíhá obdobně jako předchozí analýza. Pro příkazy volání funkcí jsou získány *AST* pro dané funkce a z nich pak veškeré identifikátory používaných globálních zdrojů. Opět tak vznikají dvě množiny pro příslušné uzly volání funkcí a jejich neprázdný průnik neumožnuje provést sjednocení.

Analýzu podmínky datové nezávislosti volání funkcí je již možné řešit na úrovni *CDFG*. Tato analýza probíhá obdobně jako analýza inicializačních příkazů u sjednocení cyklů.

Pro příkazy volání funkcí z druhého uzlu volání funkcí jsou získány předcházející příkazy a u těchto příkazů jsou ověřeny datové závislosti a konflikty vůči výsledku některé z funkcí prvního uzlu volání funkcí.

Nejsou-li splněny veškeré podmínky pro sjednocení, zůstává *CDFG* nezměněn a transformace se opakuje s další dvojicí uzlů volání funkcí. V opačném případě dochází k sjednocení dvojice uzlů. Sjednocení volání se skládá z několika kroků. Je nutné přesunout příkazy volání a případnou inicializaci volání na jednu hranu *CDFG*. Jelikož se mohou funkce v délce vyhodnocení lišit a návratová hodnota společně s informací o ukončení výpočtu je reprezentována signálem, je nutné tyto informace uchovat. Pro uchování informace o příznaku ukončení výpočtu je vytvořen speciální registr, jehož hodnota je poté použita v podmínkách hran vycházejících z uzlu volání funkce. Pro uchování návratové hodnoty je původní dočasný signál, do něhož byla návratová hodnota zapisována, transformován na registr. Příkazy pro vyhodnocení podmínky a případné uložení návratové hodnoty jsou přesunuty na cyklící hranu uzlu volání. Jelikož se tyto příkazy vyhodnocují v každém taktu, je nutné podmínit příkaz pro zápis návratové hodnoty příznakem ukončení funkce, aby byla zapsána správná hodnota. Vedlejším efektem této úpravy je to, že se volání funkce prodlouží o jeden takt, který odpovídá latenci zápisu vyhodnocení podmínky do registru. Po vytvoření těchto příkazů je možné provést sjednocení uzlů volání funkcí společně s příslušnými příkazy a podmínkami.

Celý postup řešení pak odpovídá následujícímu algoritmu:

Algoritmus 6: Sjednocení volání funkcí

najdi v CDFG veškeré uzly reprezentující volání funkce;
foreach *dvojice uzlů volání funkcí* **do**
    urči pořadí uzlů volání funkcí v CDFG;
    získej všechny příkazy volání příslušných funkcí;
    ověr zda-li jsou funkce součástí stejného podgrafu;
    ověř konflikty volání stejných funkcí;
    ověř konflikty přístupů ke globálním zdrojům;
    ověř datové závislosti funkcí;
    if *funkce je možné sjednotit* **then**
        vytvoř registr pro uchování příznaku ukončení výpočtu a příslušný příkaz;
        přesuň příkaz na cyklící hranu uzlu volání funkce;
        nastav podmínky hran uzlu volání funkce na hodnotu tohoto registru;
        transformuj signál pro uložení návratové hodnoty na registr;
        přesuň příkaz zápisu návratové hodnoty na cyklící hranu uzlu volání funkce;
        přesuň případnou inicializaci volání funkce;
        proveď sjednocení podmínek obou uzlů volání funkcí jako nové podmínky prvního uzlu;
        proveď sjednocení příkazů na cyklících hranách;
        odstraň druhý uzel volání funkcí;
        opakuj celou transformaci nad změněným CDFG;

Kapitola 6

Implementace

Implementace transformací navrhovaných v kapitole 5 byla realizována v jazyce *C++*. Volba jiného programovacího jazyka nebyla možná, neboť transformace jsou spouštěny během běhu generátoru hardwarové reprezentace architektury, implementovaného právě v jazyce *C++*. Tento jazyk však umožňuje multiplatformní vývoj a použití principů objektově orientovaného a generického programování.

Pro vývoj bylo použito prostředí *Eclipse* s pluginem *CDT*, umožňujícím vývoj v *jazyce C++* a které zároveň podporuje integraci systému správy verzí *CVS* používaného členy vývojového týmu *Lissom*. Na základě prostředí *Eclipse* je také postaveno integrované vývojové prostředí *Codasip Studio*, které bylo použito pro vytvoření testovacích modelů v jazyce *CodAL*.

Každá z transformací je reprezentována vlastní třídou a je spouštěna statickou metodou `Run`. Implementace jednotlivých transformací popsaných v kapitole 5 odpovídá algoritmu, který je uveden vždy v sekci *Popis řešení* dané transformace, a proto zde nebudou popisovány v jazyce *C++*, jelikož by se jednalo shodné algoritmy přepsané v poměru 1:1. V této kapitole jsou však dále popsány další podstatné detaily implementace.

6.1 Knihovna Standart Template Library

Podstatnou náplní této bakalářské práce bylo vytváření algoritmů nad datovými kontejnery z knihovny *Standart Temlate Library*.[1] Knihovna *STL* je kolekcí *generických šablon* kontejnerů, iterátorů a algoritmů, které programátorům umožňují jednoduše vytvářet a používat často používané datové struktury jako jsou například seznamy, fronty nebo vektory určitých datových typů [16]. Kontejnery lze rozdělit do tří skupin: *asociativní kontejnery*, *sekvenční kontejnery* a *adaptéry sekvenčních kontejnerů* [17]. Během práce byly používány nejčasteji tyto kontejnery: `std::set`, `std::map`, `std::vector` a `std::queue`.

[1] *STL* – Standart Template Library

6.2 Reprezentace AST

AST používaný v rámci generátoru hardwarové reprezentace je realizován pomocí několika tříd:

`CASTTree` reprezentuje jeden *AST*. Obsahuje odkaz na kořen stromu a odkaz na objekt s informacemi o daném stromu.

`CASTKey` obsahuje informace pro rozlišení typu *AST*. Uchovává informace o původu *AST* a jeho unikátní identifikaci.

`CASTNode` reprezentuje uzel *AST* a identifikuje jeho typ. Obsahuje informaci o rodičovském uzlu a případných potomcích. Umožňuje získat a měnit další informace o daném uzlu.

`CASTFactory` umožňuje vytváření *AST*. Obsahuje prostředky pro speciální operace nad uzly *AST* a umožňuje vyhledávání v *AST*.

6.3 Reprezentace CDFG

Obdobně jako *AST* je také *CDFG* v generátoru hardwarové reprezentace realizován pomocí několika tříd:

`CDFG` reprezentuje příslušný *CDFG* jako celek. Umožňuje vkládání a odebírání uzlů a hran v grafu.

`CDFGNode` reprezentuje jednotlivé uzly *CDFG* a identifikuje jejich typ.

`CDFGEdge` reprezentuje jednotlivé hrany mezi uzly *CDFG*. Zpřístupňuje příkazy přiřazené k dané hraně a umožňuje jejich přidávání a odebírání. U každé hrany lze nastavit podmínku provedení a počáteční a cílový uzel *CDFG*.

`CDFGStatement` reprezentuje jednotlivé příkazy, kdy každý z nich je vždy přiřazen k jedné z hran *CDFG*. Umožňuje vytvoření podmínky pro provádění příkazu. Zpřístupňuje datové závislosti daného příkazu a umožňuje jejich vytváření.

6.3.1 Reprezentace datových závislostí

Jak bylo zmíněno v kapitole 4.2.1, nejsou datové závislosti a konflikty mezi jednotlivými příkazy jazyka C součástí *CDFG*, ale jsou reprezentovány pomocí speciálního grafu. Jelikož součástí jednoho příkazu může být více zdrojů, které mohou mít více různých závislostí, je nutné rozlišovat, o který typ závislosti se jedná a která dvojice zdrojů v této závislosti figuruje. Graf datových závislostí je tak v rámci jednoho příkazu tvořen seznamem trojic složených z typu závislosti, zdroje vytvářející datovou závislost a datově závislého zdroje. Pro potřeby rozpoznávání těchto trojic a jejich analýzu slouží následující struktury:

`dependence_target_t` je struktura reprezentující příkaz vytvářející datovou závislost. Tato struktura obsahuje identifikátor zdroje vytvářejícího datovou závislost a příkaz jehož jehož je daný zdroj součástí.

`dependence_source_t` reprezentuje zdroj, který je datově závislý. Tato struktura obsahuje pouze identifikátor příslušného zdroje.

`dependences_t` je mapa všech datových závislostí vztahujících se k jednomu příkazu. Klíčem této mapy je příslušný závislý zdroj příkazu (struktura `dependence_source_t`), hodnotou je pak pak kontejner `dependence_targets_t`. Tato mapa je přiřazena každému příkazu majícímu alespoň jeden zdroj s datovou závislostí.

`dependence_targets_t` je kontejner pro uchovávání závislostí s ohledem na jejich typ. Jedná se o mapu, jejíž klíčem je typ datové závislosti (například RAR) a hodnotou je pak vektor zdrojů vytvářejících datovou závislost příslušného typu (vektor struktur `dependence_target_t`).

Kapitola 7

Experimenty

Ověření správné funkčnosti implementace transformací bylo provedeno na sadě příkladů popisu chování zapsaných v jazyce *CodAL*. Každá transformace byla ověřena samostatně a následně byla otestována kompatibilita s ostatními transformacemi plánovače. Ověření správnosti transformací bylo provedeno na několika úrovních.

Pro prvotní ověření správnosti implementace jednotlivých transformací byla použita vizualizace *AST*, *CDFG*, *grafu závislostí* a *RTLG*, která byla automaticky vytvářena přímo v rámci generátoru hardwarové reprezentace architektury. Tato vizualizace umožnila otestovat implementaci na sadě příkladů popisu chování typických pro příslušnou transformaci a následně také jejich kombinaci.

Následně byl z příkladů vygenerován popis v jazyce *VHDL* a ten odsimulován v programu *Modelsim*, díky čemu bylo možné odhalit skryté problémy na úrovni logických signálů v hardware. Po ověření správnosti pomocí simulace byly zdrojové soubory vygenerované po aplikaci transformace sjednocení cyklů a volání funkcí vysyntetizovány pomocí nástroje *Xilinx ISE*. Cílem syntézy bylo potvrdit předpoklad, že transformace kromě zkrácení doby výpočtu vedou také ke zmenšení celkové plochy čipu.

V této kapitole jsou popsány příklady experimentů prováděných pro ověření správnosti transformací nad *CDFG*. Správnost ransformací probíhajících nad *AST*, tedy transformace pro podporu inicializace během deklarace a pro úpravu vícenásobného přiřazení byly otestovány na základních příkladech a poté v rámci transformací nad *CDFG*. Další ověření nebylo vzhledem k povaze těchto transformací potřeba.

7.1 Transformace rozlišení konfliktů sekvenčně prováděných příkazů

Cílem experimentů pro transformaci rozlišení konfliktů sekvenčně prováděných příkazů bylo ověření, že transformace nevytváří konflikty vůči stávajícím příkazům a nenarušuje logiku výpočtu. Ověření bylo provedeno na úrovni *CDFG* a *RTLG*. Správnost transformace byla otestována například na níže uvedeném příkladu v jazyce *CodAL*. Všechny použité zdroje v příkladu jsou deklarovány jako registry.

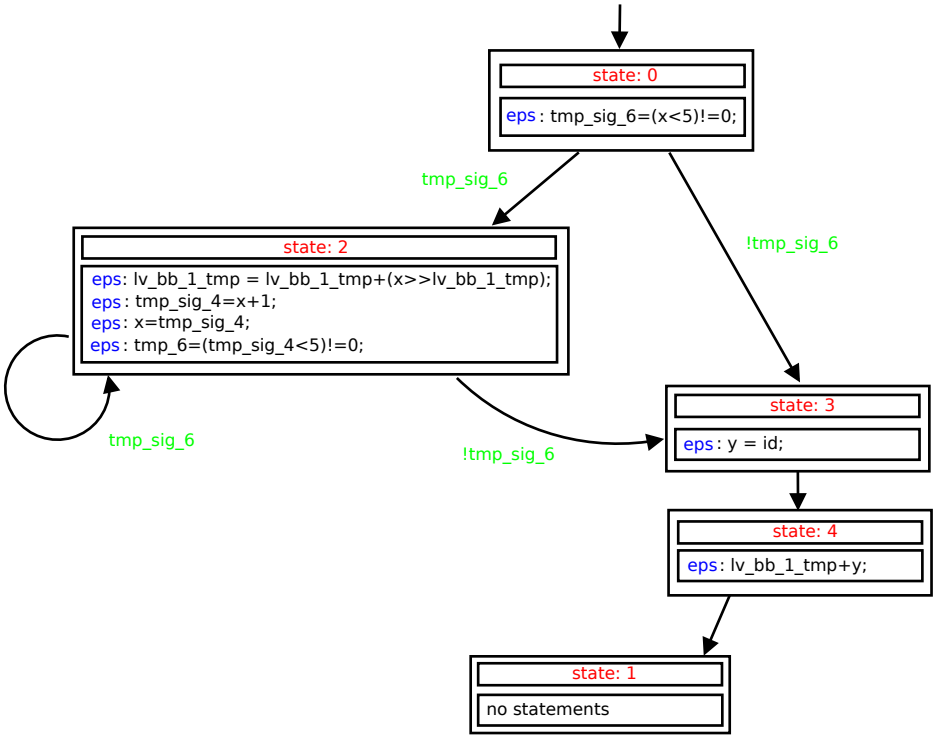
```
semantics {
     while ( x < 5 )
     {
       tmp += x >> tmp;
       x++;
     }
     y = id;
     x = tmp + y;
};
```

Na obrázku 7.1 je zachycen *RTLG* před aplikací transformace. Jelikož příkazy `y = id;` a `x = tmp+y;` přistupují ke stejnému zdroji `y`, do kterého je nejprve hodnota zapisována a následně čtena, nemohou být příkazy provedeny v jednom taktu. Z toho důvodu je *RTLG* tvořen 5 uzly (stavy automatu). Nejpodstatnější jsou stavy označené v obrázku jako `state:3` a `state:4`. Ve stavu `state:3` je proveden zápis hodnoty do příslušného zdroje. Jelikož se jedná o registr, je nutné k dokončení operace zpoždění o jeden takt a čtení hodnoty zdroje tak probíhá až v následujícím stavu `state:4`.

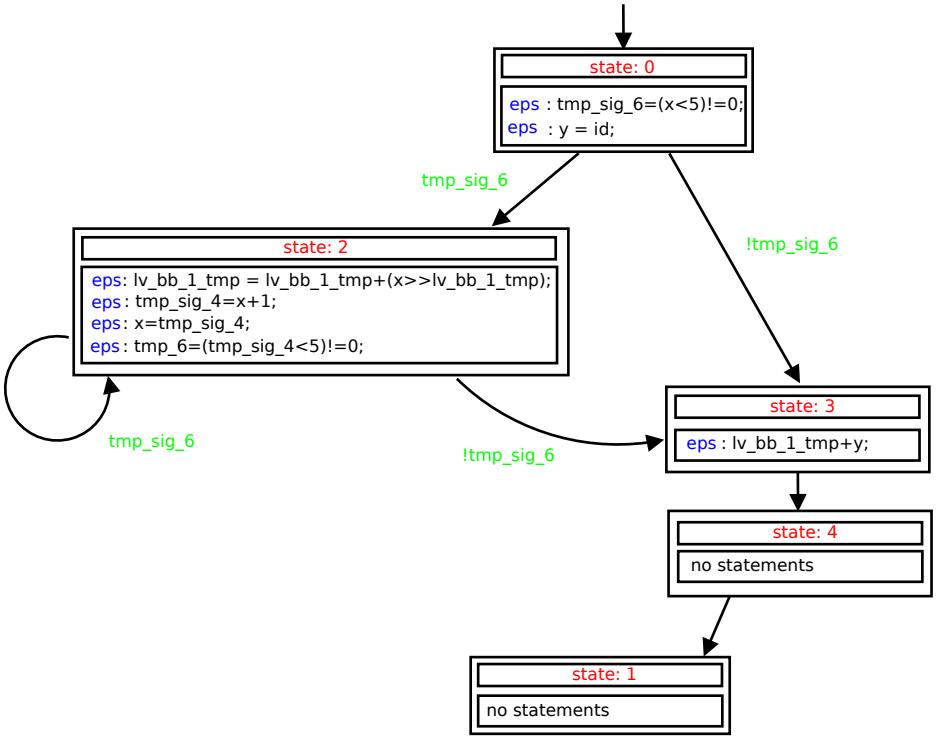
Na obrázku 7.2 je pak zachycen *RTLG* po aplikaci transformace. V tomto obrázku jsou podstatné stavy *RTLG* označeny jako `state:0`, `state:3` a `state:4`. Během transformace je příkaz `y = id;` určen jako příkaz, který je možné vyhodnotit v dřívějším taktu. Příkaz je v *CDFG* přesunut na hranu předcházející cyklus `while`, což se v *RTLG* projeví tak, že se příkaz nyní vyskytuje ve stavu `state:0` namísto původního `state:3`. Přesunutím tohoto příkazu se zvýšil počet taktů mezi závislými příkazy a příkaz `x = tmp+y;` je tak následně možné také přesunou do dřívějšího taktu. Přesunutí příkazu v *CDFG* se v *RTLG* projeví přesunem příkazu ze stavu `state:4` do stavu `state:3`.

Po dokončení transformace zůstávají v *RTLG* dva stavy neobsahující příkazy – stav `state:4` a koncový stav `state:1`. Doba výpočtu tak není zkrácena, nicméně transformace umožňuje dodržení principu plánovače *ASAP*, tedy, že příkazy jsou vyhodnoceny v nejbližším možném taktu. Ke snížení potřebné doby výpočtu by mohla přispět další transformace, která by například odstranila z *CDFG* synchronizační uzly, které díky přečasování příkazu v rámci transformace již nejsou potřeba.

Během experimentů bylo ověřeno, že transformace nevytváří žádné konflikty vůči stávajícím příkazům ani nenarušuje logiku výpočtu přesunem příkazů z nebo do jiných logických bloků. Transformace tak funguje správně.



Obrázek 7.1: RTLG před transformací rozlišení konfliktů sekvenčně prováděných příkazů



Obrázek 7.2: RTLG po transformací rozlišení konfliktů sekvenčně prováděných příkazů

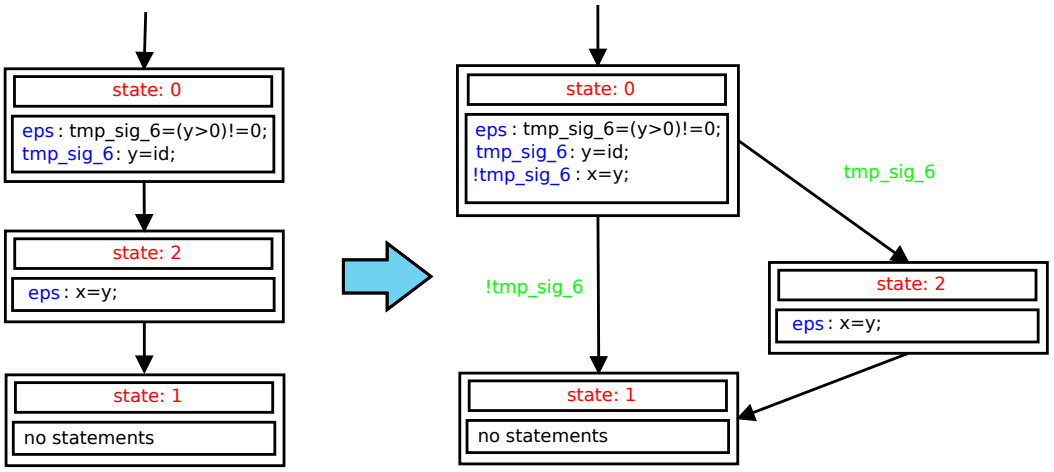
7.2 Transformace rozlišení konfliktů v podmíněných větvích

Cílem experimentů pro transformaci rozlišení konfliktů v podmíněných větvích bylo ověření, že transformace nenarušuje logiku vyhodnocení a její aplikace vede ke snížení doby potřebné na výpočet. Správnost transformace byla ověřena na úrovni *CDFG* a *RTLG* například na následujícím příkladu v jazyce *CodAL*. Všechny zdroje v příkladu jsou deklarovány jako registry.

```
semantics {
    if(y > 0){
        y = id;
    x = y;
};
```

Na obrázku 7.3 je zachycena změna *RTLG* vlivem transformace, kdy přechod do dalšího stavu *RTLG* znamená zpoždění o jeden takt. V levé části je znázorněno původní řešení. Vyhodnocení trvá dva takty, jelikož ve stavu `state:0` dochází k vyhodnocení podmínky větvení a zároveň k případnému přiřazení hodnoty. Jelikož toto přiřazení vytváří závislost pro další příkaz, byl v *CDFG* vložen za podmíněné větvení vložen synchronizační stav, který způsobí, že příkaz `x=y` je provede až v dalším stavu *RTLG*. Výpočet této části trvá vždy 2 takty.

Během transformace je synchronizační uzel přesunut do bloku podmíněného větvení. Tento přesun vyvolá změnu struktury *RTLG*. Ve stavu `state:0` je na základě vyhodnocení podmínky větvení provedeno přiřazení. V případě, že podmínka byla splněna, jedná se o `y=id`, v opačném případě o přiřazení `x=y`. Na základě vyhodnocení podmínky je také proveden přechod do dalšího stavu. Není-li splněna, je proveden přechod do koncového stavu `state:1` a výpočet tak trvá jeden takt. Je-li podmínka splněna je proveden přechod do stavu `state:2` a výpočet trvá 2 takty.



Obrázek 7.3: Změna RTLG vlivem transformace rozlišení konfliktů v podmíněných větvích

Aplikace transformace vede ke snížení celkové doby potřebné na výpočet. V současném řešení je doba výpočtu zkrácena vždy o jeden takt v případě, že podmínka větvení není splněna a větev obsahující přesunutý synchronizační uzel není vyhodnocena. Další zkrácení doby výpočtu by bylo možné při přesunu více synchronizačních uzlů, což však není v současné době možné z důvodů popsaných v kapitole 5.4.2.

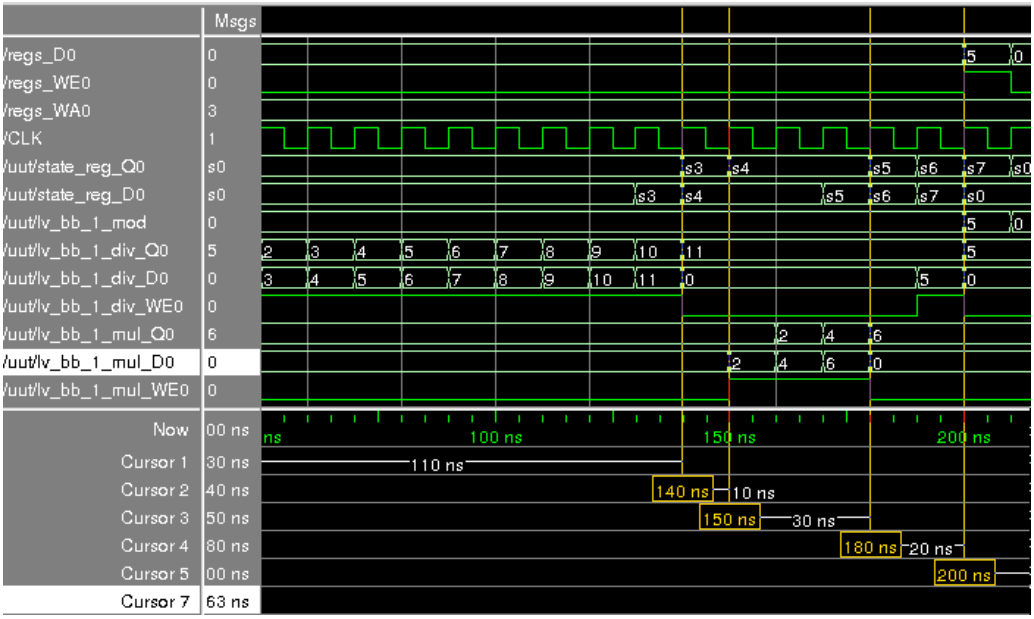
7.3 Transformace sjednocení cyklů

Cílem experimentů pro transformaci sjednocení cyklů byla kontrola správnosti sjednocení a ověření, že transformace zkracuje dobu potřebnou na výpočet. Pro ověření správnosti byl použit mimo jiné také následující příklad v jazyce *CodAL*. V uvedeném příkladu je zdroj `regs` je deklarován jako registrové pole, ostatní zdroje jsou deklarovány jako lokální proměnné (signály), přičemž zdroje `i`, `div` a `mul` jsou během dalších transformací *plánovače* transformovány na registry s asynchronním zápisem.

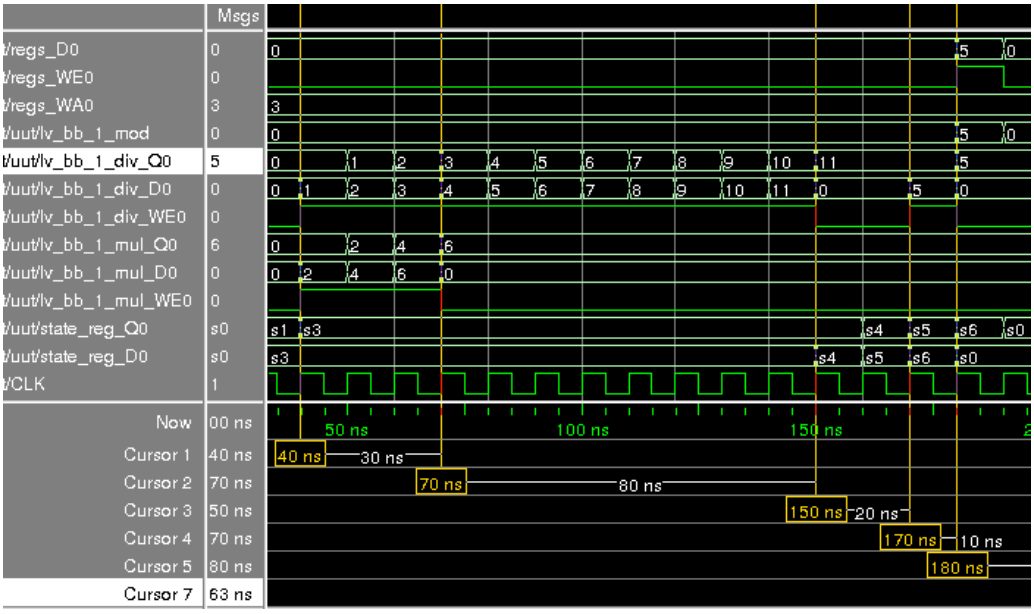
```
semantics {
    int i, div, mul, mod, a, b, c, d;
    i = div = mul = mod = 0;
    a = 55;
    b = 5;
    c = 3;
    d = 2;
    while(a >= b){
        a -= b;
        ++div;
    }
    while(i < c){
        i++;
        mul += d;
    }
    while(div >= mul){
        div -= mul;
    }
    mod = div;
    regs[3] = mod;
};
```

Na obrázku 7.4 je zachycen průběh simulace VHDL vygenerovaného bez aplikace transformace. Vyhodnocení prvního z cyklů, realizujícího dělení, je zahájeno v čase `t = 30 ns`. Vyhodnocení tohoto cyklu trvá 11 taktů do času `t = 140 ns`. Výpočet dalšího cyklu je pak zahájen o jeden takt později, kdy toto zpoždění vzniká kvůli synchronizačnímu uzlu prvního cyklu. Výpočet tohoto cyklu je ukončen v čase `t = 180 ns` a opět dochází ke zpoždění dalších výpočtů o jeden takt. Následně je zahájen výpočet posledního z trojice cyklů, který je ukončen v čase `t = 200 ns`, kdy je zároveň aktivován zápis do registrového pole `regs[3]`. Celkový výpočet tak trvá 21 taktů.

Na obrázku 7.5 je zachycen průběh simulace *VHDL* vygenerovaného s aplikací transfomace. Při transformaci jsou sjednoceny první 2 cykly, které jsou vzájemně datově nezávislé. Poslední cyklus nemůže být také sloučen, neboť jeho výpočet závisí na hodnotách obou předcházejících cyklů. Jelikož se při transformaci přesouvá také inicializace cyklů, je zahájení výpočtu sjednocených cyklů zpožděno o jeden takt z důvodu zápisu hodnoty inicializace druhého cyklu do registru. Výpočet obou sjednocených cyklů je zahájen v čase `t = 40 ns` a ukončen o 11 taktů později v čase `t = 150 ns`. Výpočet posledního cyklu je zahájen v čase `t = 170 ns`, kdy zpoždění o dva takty je způsobeno synchronizačním uzlem a změnou podmínkových signálů cyklů na registry s latencí zápisu jeden takt. Zápis výsledků do registrového pole je proveden v čase `t = 190 ns`. Celkový výpočet tak trvá 19 taktů.



Obrázek 7.4: Průběh simulace před transformací sjednocení cyklů



Obrázek 7.5: Průběh simulace po transformaci sjednocení cyklů

Výpočet byl pomocí transformace zkrácen o 2 takty. Další zkrácení výpočtu by bylo možné odstraněním synchronizačního uzlu u sjednocených cyklů. Zpoždění, které bylo nutné k zápisu hodnot z poslední iterace cyklu bylo zajištěno změnou podmínkových signálů cyklu na registry – cyklus je ukončen o jeden tak později a synchronizační uzel již není potřeba.

Celkovou dobu pro vyhodnocení příkazů cyklů před a po aplikaci lze vyjádřit níže uvedenými vztahy. Rovnice 7.1 charakterizuje dobu potřebnou pro výpočet n datově nezávislých cyklů t_{sum} před aplikací transformace. Tato doba je tvořena součtem dob vyhodnocení inicializace cyklu t_{init} a dob vykonávání příkazů těl cyklů t_{loop}. Rovnice 7.2 pak charakterizuje

dobu pro výpočet n datově nezávislých cyklů po aplikaci transformace. Ta je tvořena součtem maximální doby nutných pro výpočet jednotlivých cyklů a jejich inicializace a penalizací v podobě zpoždění o jeden takt vlivem synchronizačního uzlu.

$$t_{sum} = \sum_{0..n}^{i}(t_{init_i} + t_{loop_i}) \tag{7.1}$$

$$t_{sum} = max(t_{init_0}, t_{init_1}, ..., t_{init_n}) + max(t_{loop_0}, t_{loop_1}, ..., t_{loop_n}) + 1 \tag{7.2}$$

Transformace funguje správně a její aplikace vede ke snížení celkové doby potřebné na výpočet. Výrazné snížení doby výpočtu se projeví zejména u cyklů s podobnou délkou vyhodnocení vyžadují větší množství iterací.

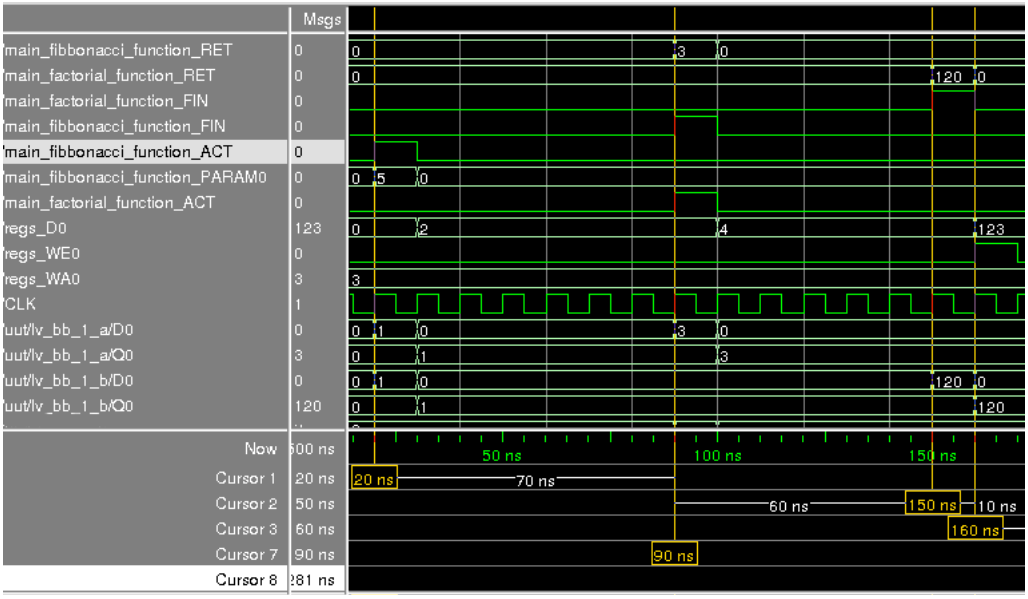
7.4 Transformace sjednocení volání funkcí

Cílem experimentů pro transformaci sjednocení volání funkcí bylo ověřit správné sjednocení volání a potvrdit, že transformace snižuje dobu nutnou na výpočet. Funkčnost této transformace byla otestována například na sjednocení volání funkcí pro výpočet n-tého čísla *Fibbonacciho posloupnosti* a *faktoriálu*. Výsledek těchto funkcí byl následně sečten a uložen do registru. Uvedený popis chování odpovídá níže uvedenému zápisu v jazyce *CodAL*:

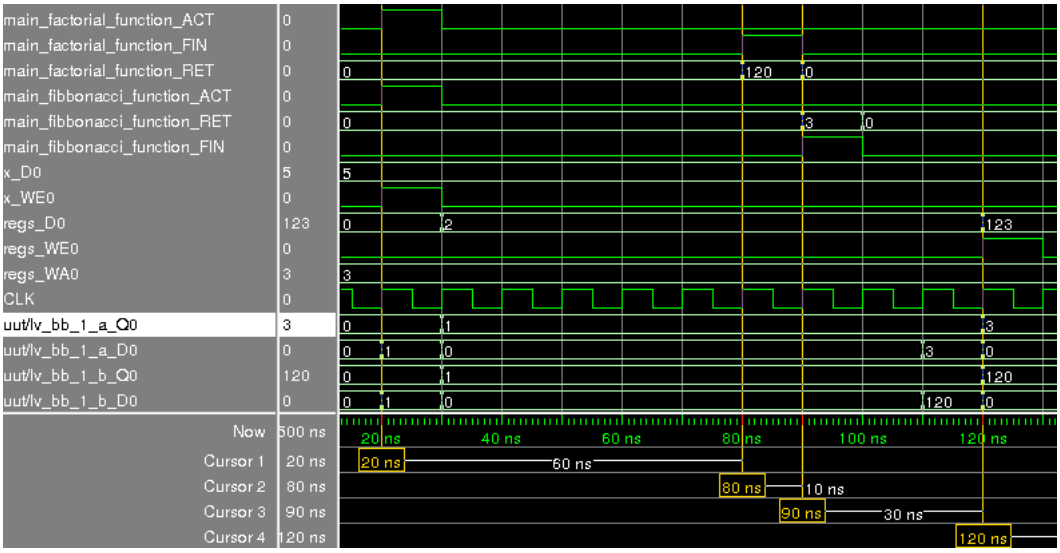
```
semantics {
    long a;
    long b;
    a = fibbonacci(5);
    b = factorial(5);
    regs[3] = a + b;
};
```

Na obrázku 7.6 je zachycen průběh simulace VHDL vygenerovaného bez použití transformace. V obrázku jsou zaznačeny důležité okamžiky simulace. V čase `t = 20 ns` je aktivována funkční jednotka pro funkci `fibbonacci` příslušným signálem `ACT`. Ta provádí výpočet 7 hodinových cyklů do času `t = 90 ns`. V tomto čase je aktivován `FIN` signál této funkční jednotky, reprezentující ukončení výpočtu a je vrácena výsledná hodnota. Zároveň je v tomto čase aktivován výpočet funkční jednotky pro funkci `factorial`. Výpočet této funkční jednotky je ukončen v čase `t = 150 ns`. Součet výsledků obou funkcí je zapsán do registru v následujícím taktu, tedy v čase `t = 170 ns`. Výpočet tak trvá 17 taktů.

Na obrázku 7.7 je zachycen průběh simulace VHDL vygenerovaného s aplikací transformace. V čase `t = 20 ns` dochází k aktivací obou funčních jednotek funkcí `fibbonacci` a `factorial`. Výpočet obou funkcí tak probíhá současně. V čase `t = 80 ns` je ukončen výpočet funkce `factorial`. Ta byla v původním řešení volána až po ukončení volání funkce `fibbonacci`, která je nyní ukončena o takt později po funkci `factorial`. Součet výsledků obou funkcí je zapsán do registru v čase `t = 120 ns`. Zpoždění před zápisem výsledku do registrového pole je způsobeno transformací podmínkových signálů na registry. I s tímto zpožděním však bylo vyhodnocení volání funkcí zkráceno o čtyři takty.



Obrázek 7.6: Průběh simulace před transformaci sjednocení volání funkcí



Obrázek 7.7: Průběh simulace po transformaci sjednocení volání funkcí

Obdobně jako v případě transformace pro sjednocení cyklů lze celkovou dobu výpočtu n datově nezávislých volání funkcí vyjádřit níže uvedenými vztahy. Rovnice 7.3 odpovídá době vyhodnocení n datově nezávislých volání funkcí před aplikací transformace a je tvořena součtem všech dob inicializace volání funkce t_{init} a samotného volání t_{fnc}. Doba vyhodnocení po aplikaci transformace (rovnice 7.4) je pak tvořena součtem maximální doby vyhodnocení inicialaze volání a samotného volání. K celkové době je však nutné připočítat penalizaci v podobě zpoždění o jeden takt vlivem ukládání příznaků ukončení výpočtu funkce do registru.

$$t_{sum} = \sum_{0..n}^{i}(t_{init_i} + t_{fnc_i}) \tag{7.3}$$

$$t_{sum} = max(t_{init_0}, t_{init_1}, ..., t_{init_n}) + max(t_{fnc_0}, t_{lfnc_1}, ..., t_{fnc_n}) + 1 \tag{7.4}$$

Experiment potvrdil snížení doby nutné na výpočet vlivem transformace. Obdobně jako u transformace pro sjednocení cyklů platí, že efekt transformace se výrazněji projeví při sjednocení volání funkcí s delší dobou výpočtu.

7.5 Syntéza zdrojových VHDL souborů

Po ověření transformací programem *ModelSim* byly zdrojové soubory v jazyce *VDHL* vysyntetizovány pomocí nástroje *Xilinx ISE* verze 13.4 pro cílové *FPGA Virtex 5*. Cílem syntézy bylo porovnat řešení s aplikovanými transformacemi s původním řešením z hlediska plochy a frekvence. Výsledky syntézy jsou uvedeny v následující tabulce:

Transformace	**Registry**	**LUTs**	**Frekvence [Mhz]**
před sjednocením cyklů	136	564	232
po sjednocení cyklů	137	507	232
před sjednocením funkcí	168	551	195
po sjednocení funkcí	168	568	195

U transformace pro sjednocení cyklů, došlo vlivem transformace k podstatnému snížení počtu *LUT* o 10%. Toto snížení počtu LUT je způsobeno snížením počet stavů RTLG a logika výsledného automatu je tak menší. U transformace pro sjednocení volání funkce naopak dochází k nárůstu počtu LUT, a tím i k nárůstu plochy. Tento nárůst je způsoben zejména vložením dvou nových registrů pro každé volání funkce, které mohou být reprezentovány právě pomocí LUT. Obě transformace mají však minimální nebo žádný vliv na frekvenci a počet registrů.

Kapitola 8

Závěr

V rámci této bakalářské práce byly navrženy a implementovány transformace, které jsou aplikovány při převodu jazyka *C* z popisu chování specifikace architekturyv jazyce *CodAL*, do jazyka *VHDL*. Jedná se o transformace úpravy vícenásobného přiřazení, podpory inicializace během deklarace, rozlišení konfliktů sekvenčně prováděných příkazů, rozlišení konfliktů v podmíněných větvích, současného vyhodnocení více cyklů a současného vyhodnocení více funkcí. U transformací pro rozlišení konfliktů v podmíněných větvích a pro současné vyhodnocení více cyklů a funkcí bylo dokázáno, že jejich aplikace vede ke snížení doby potřebné na výpočet. Zároveň bylo pomocí syntézy výsledných zdrojových souborů po aplikaci transformací dokázáno, že transformace pro sjednocení cyklů přispívá také ke snížení celkové plochy. Naopak transformace pro sjednocení funkcí vede k mírnému nárůstu plochy.

Další pokračování práce v budoucnu by se mohlo zabývat další optimalizací při převodu jazyka *C* do *VHDL*, kdy implemenované transformace by mohly být dále rozšířeny či doplněny o nové. Transformace pro rozlišení konfliktů v podmíněných větvích by například mohla být rozšířena o podporu přesunu více synchronizačních uzlů. Také by mohla být provedena úprava dvojice uzlů *CDFG* reprezentujících cyklus tak, aby v případě sjednocení cyklů nebyl již vyžadován nadbytečný synchronizační uzel. Nově by pak mohla být implementována transformace, která by v *CDFG* grafu odstranila přebytečné synchronizační uzly, které již nejsou potřeba vlivem ostatních transformací.

Literatura

[1] Hoffmann, A. *Architecture Exploration for Embedded Processors With LISA*. 1. vyd. [b.m.]: Springer, 2010. 238 s. ISBN 978-1-4419-5334-6.

[2] Vašíček, Z. *Materiály k přednáškám z předmětu Seminář VHDL*. Fakulta informačních technologií, Vysoké učení technické v Brně, 2010.

[3] Sekanina, L. *Studijní opora k předmětu Návrh počítačových systémů: Úvod do jazyka VHDL* [online]. Brno: FIT VUT v Brně, 2006 [cit. 7.4.2013]. Dostupné na: <TODO>.

[4] Musil, V., Kolouch, J. a Prokop, R. *Návrh digitálních obvodů a jazyk VHDL* [online]. 2011 [cit. 7.4.2013]. Dostupné na: <http://www.umel.feec.vutbr.cz/METMEL/studijnipomucky/METMEL_16_S_BNDI_Navrh_digitalnich_integrovanych_obvodu_a_jazyk_VHDL.pdf>.

[5] *An Introductory VHDL Tutorial* [online]. 1995 [cit. 22.3.2013]. Dostupné na: <http://www.gmvhdl.com/dataflow.htm>.

[6] Mishra, P. a Dutt, N. Architecture description languages for programmable embedded systems. *IEE Proceedings - Computers and Digital Techiques* [online]. Květen 2005, roč. 152 [cit. 29.3.2013]. S. 285–297. Dostupné na: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1468678>.

[7] Mishra, P. a Dutt, N. *Processor description languages: applications and methodologies*. 1. vyd. Burlington: Morgan Kaufmann, 2008. 432 s. ISBN 978-0-12-374287-2.

[8] Masařík, K. *Materiály k přednáškám z předmětu Principy programovacích jazyků a OOP*. Fakulta informačních technologií, Vysoké učení technické v Brně, 2012.

[9] Masařík, K. *Systém pro souběžný návrh technického a programového vybavení počítačů*. Fakulta informačních technologií, Vysoké učení technické v Brně, 2008. Disertační práce. 156 s. ISBN 978-80-214-3863-7.

[10] APS Brno s.r.o. *CodAL Manual: reference guide*. 2013.

[11] APS Brno s.r.o. *Codasip Framework Manual: User's guide*. 2013.

[12] Gajski, D. D. et al. *High-level synthesis: introduction to chip and system design*. 2. vyd. [b.m.]: Springer, 1994. ISBN 0-7923-9194-2.

[13] Elliott, J. P. *Understanding behavioral synthesis: a practical guide to high-level design*. 2. vyd. Norwell: Kluwer Academic Publishers, 1999. 338 s. ISBN 0-7923-8542-X.

[14] Stroustrup, B. *The C Standard: incorporating technical corrigendum 1.* [b.m.]: John Wiley, 2003. 538 s. ISBN 978-0470845738.

[15] Sedgewick, R. *Algorithms in C.* 1. vyd. [b.m.]: Addison-Wesley, 1990. 657 s. ISBN 0-201-51425-7.

[16] Silicon Graphics International. *Introduction to the Standard Template Library* [online]. 2009 [cit. 24.3.2013]. Dostupné na: <`http://www.sgi.com/tech/stl/stl_introduction.html`>.

[17] Josuttis, N. M. *The C++ standard library: a tutorial and handbook.* [b.m.]: Addison-Wesley, 1999. 799 s. ISBN 0-201-37926-0.