VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

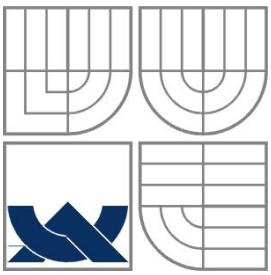
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TRANSFORMACE POPISNÉHO JAZYKA MIKROPROCESORU DO JAZYKA PRO POPIS HARDWARE

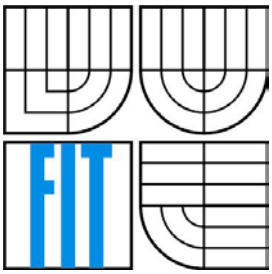
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. TOMÁŠ NOVOTNÝ
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TRANSFORMACE POPISNÉHO JAZYKA MIKROPROCESORU DO JAZYKA PRO POPIS HARDWARE

TRANSFORMATION OF THE MICROPROCESSOR'S DESCRIPTION LANGUAGE TO THE HARDWARE DESCRIPTION LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. TOMÁŠ NOVOTNÝ
AUTHOR

VEDOUCÍ PRÁCE — Prof. Ing. TOMÁŠ HRUŠKA, CSc.
SUPERVISOR

BRNO 2007

Zadání diplomové práce/4747/2006/xnovot65

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání diplomové práce

Řešitel: **Novotný Tomáš, Bc.**

Obor: Informační systémy

Téma: **Transformace popisného jazyka mikroprocesoru do jazyka pro popis hardware**

Kategorie: Překladače

Pokyny:

1. Seznamte se s jazyky pro popis vnitřní struktury mikroprocesorů, zejména z rodin LISA a ISAC, dále s jazyky pro popis hardwaru, zejména s jazykem VHDL.
2. Navrhněte rozšíření popisného jazyka ISAC o konstrukce podporující transformace do hardwarového jazyka VHDL.
3. Návrh z bodu 2 vhodně parametrizujte, zaměřte se na to, aby informace byla úplná, avšak nikoliv duplicitní.
4. Navržená rozšíření a s nimi spojený parametrický generátor VHDL kódu implementujte.
5. Začleňte překladač do vývojového prostředí projektu Lissom, zhodnoťte přínos své práce, diskutujte možná rozšíření, případně nedostatky.

Literatura:

- Dokumentace projektu Lissom,
- Internet,
- dále dle pokynů vedoucího

Při obhajobě semestrální části diplomového projektu je požadováno:

- Body 1-3 zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hruška Tomáš, prof. Ing., CSc.**, UIFS FIT VUT

Datum zadání: 28. února 2006

Datum odevzdání: 22. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Bc. Tomáš Novotný**
Id studenta: 47381
Bytem: Za Lidokovem 433, 679 72 Kunštát
Narozen: 22. 02. 1983, Boskovice
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Transformace popisného jazyka mikroprocesoru do jazyka pro popis hardware
Vedoucí/školitel VŠKP: Hruška Tomáš, prof. Ing., CSc.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☒ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Abstrakt

Diplomová práce Transformace popisného jazyka mikroprocesoru do jazyka pro popis hardware je zaměřena na návrh aplikačně specifických mikroprocesorů s využitím jazyka ISAC. Zabývá se návrhem a implementací transformace, která popis mikroprocesoru v jazyce ISAC převádí na ekvivalentní popis v jazyce VHDL.

Kapitola Přehled o zkoumané problematice popisuje zvolenou problematiku, objasňuje některé pojmy s problematikou související a představuje návrh výše zmiňované transformace. V kapitole nazvané Návrh řešení jsou postupně uvedena nová rozšíření jazyka ISAC, dále je popsán návrh řešení transformace a implementace generátoru popisu v jazyce VHDL, který provádí transformaci. Závěr diplomové práce diskutuje případné rozšíření práce a dosažené výsledky.

Klíčová slova

transformace, procesor, mikroprocesor, vestavěný systém, jazyk ISAC, projekt Lissom, jazyk VHDL, šablony mikroprocesoru, graf procesů, hardware/software co-design

Abstract

The Master's thesis Transformation of the microprocessor's description language to the hardware description language is aimed at design of application specific microprocessors with using ISAC language. It deals with design and implementation of transformation which converts description of microprocessor in ISAC language into equivalent description in VHDL language.

The chapter Summary of research problems describes chosen problems, showing up some notions connected with problems and presents suggestion of transformation mentioned above. The chapter Suggestion of solution presents new extension of ISAC language. There is also described the way of design solution of transformation and solution of implementation of VHDL generator which performs transformation. Conclusion of thesis discusses next points of future work reached results.

Keywords

transformation, processor, microprocessor, embedded system, ISAC language, project Lissom, VHDL language, processor's templates, process graph, hardware/software co-design

Citace

Tomáš Novotný: Transformace popisného jazyka mikroprocesoru do jazyka pro popis hardware, diplomová práce, Brno, FIT VUT v Brně, 2007

Transformace popisného jazyka mikroprocesoru do jazyka pro popis hardware

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Prof. Ing. Tomáše Hrušky, CSc.
Další informace mi poskytli Ing. Karel Masařík, Doc. Dr. Ing. Dušan Kolář a Ing. Roman Lukáš, Phd.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.
Diplomová práce byla vypracována na základě veřejně přístupného kódu programu TinyXml a Eclipse.

.....
Tomáš Novotný
22. 5. 2007

Poděkování

Děkuji Prof. Ing. Tomáši Hruškovi, CSc., vedoucímu diplomové práce, za odborné vedení, rady, konzultace, připomínky, které mi poskytoval v průběhu zpracování této diplomové práce.

Obsah

1 Úvod

S výpočetními systémy se dnes setkáme prakticky všude (v autě, v kuchyni, ve výtahu, v obchodech). Nejčastěji mají podobu vestavěných systémů, které jsou jednou z nejvýznamnějších položek výpočetního trhu. Jde o malý počítačový systém, který je zabudován uvnitř určitého zařízení (elektrického spotřebiče, stroje), jehož „mozkem“ bývá zpravidla procesor.

Mnoho systému tedy potřebuje vestavěné procesory, které zajišťují základní logiku systému. Může se jednat o obecné procesory, digitální signálové procesory nebo aplikačně specifické procesory využívající specifickou množinu instrukcí. Návrháři procesorů vestavěných systémů požadují stále nové prostředky pro zlepšení návrhu a urychlení vývoje.

Návrh procesorů pro vestavěné systémy je třeba urychlit a snížit cenu vývoje, aby se vyplatilo využít nové řešení. Návrh lze rozdělit na dva směry. Prvním z nich je čistě strukturálně orientovaný (hardwarový) pohled na návrh architektury. Již bylo vytvořeno mnoho modelů pro podporu vývoje, kde k nejznámějším patří jazyky VHDL a Verilog. Druhý směr se staví k návrhu z pohledu instrukční sady (softwarový pohled). Snahou je vytvořit nástroj umožňující HW/SW co-design, tedy současný vývoj softwaru pro procesor spolu s vývojem vlastního hardwaru procesoru z jednoho popisu procesoru.

Ve světě běží již několik projektů (např. indický projekt SANKHYA, celosvětový projekt CADENCE s jazykem SIM-nML, projekt Lissom), které se zabývají tvorbou komplexních vývojových nástrojů pro zlepšení návrhu a testování navrhnutého procesoru po hardwarové i softwarové stránce, či tvorbou modelu pro vlastní syntézu (výrobu) procesoru. Tyto nástroje umožní návrháři popsat procesor z hlediska zdrojů, které definují hardwarovou architekturu procesoru, a z hlediska instrukcí a jejich chování. Z těchto popisů jsou generovány nástroje pro tvorbu asembleru a disasembleru pro daný procesor a nástroje pro simulaci procesoru.

Projekt Lissom se vedle tvorby komplexních vývojových nástrojů pro zlepšení návrhu procesoru zabývá i tvorbou univerzálního programovacího jazyka ISAC pro popis struktury procesoru i instrukční sady. Cílem projektu je vytvořit vývojové prostředí, v němž lze navrhovat hardwarovou architekturu procesoru a současně vyvíjet jeho software.

Jazyk ISAC umožňuje popis procesoru na vyšší úrovni abstrakce, která je vhodná spíše pro simulaci software, nikoliv však pro samotnou hardwarovou implementaci. Proto je třeba transformovat popis v jazyce ISAC na popis vhodnější pro hardwarovou syntézu, konkrétně na popis v jazyce VHDL. Mým úkolem v rámci projektu Lissom je tedy navrhnout generátor syntetizovatelného popisu procesoru v jazyce VHDL. Pro generátor bude vstupem popis procesoru v jazyce ISAC. Součástí práce je i návrh potřebných rozšíření jazyka ISAC pro transformaci na syntetizovatelný kód v jazyce VHDL.

2 Přehled o zkoumané problematice

Tato část práce je zaměřena na teoretický úvod ke zkoumané problematice. Obsahuje definice pojmů souvisejících se zkoumanou oblastí, předkládá stručný pohled na problematiku návrhu mikroprocesorů pro vestavěné systémy a podává přehled o jazycích ISAC[1] a VHDL[2]. Současně jsou v kapitole diskutována možná řešení transformace z jazyka ISAC do jazyka VHDL.

2.1 Základní pojmy

Kapitola obsahuje základní přehled pojmů. Další pojmy jsou vysvětlovány v rámci navazujících kapitol.

2.1.1 Procesorové architektury

Kapitola objasňuje vybrané procesorové architektury, s nimiž souvisí tato práce.

2.1.1.1 Procesor

Procesor je zařízení, které transformuje zadaná vstupní data podle definovaného předpisu na data výstupní. Z matematického hlediska vlastně realizuje funkci mnoha vstupních proměnných, jejímž výsledkem jsou další proměnné výstupní. Tato transformace je definována programem, což je posloupnost řídících pokynů pro vykonání konkrétních činností, které je procesor schopen realizovat. Tyto pokyny se nazývají instrukcemi a jim asociované činnosti/operace jsou přesně definovány. Vstupem do procesoru chápeme data uložená v pamětech, ať jsou to již paměti externí, interní nebo paměti vyrovnávací (cache). Dále to mohou být přímé, relativně izolované vstupy, jako jsou například přerušení, vstupní datové či komunikační porty apod. Tyto prostředky mohou být jistým způsobem také do paměti mapovány. Výstup transformační funkce bývá často totožný, resp. má velkou společnou část, s množinou vstupů. Výsledky se totiž taktéž zapisují do paměti, přenášejí se přes vstupně/výstupní porty apod. [4].

Jako synonymum pro procesor se často setkáváme s pojmem *mikroprocesor*. Oba pojmy lze dnes již rovnocenně používat.

2.1.1.2 Generický procesor

Z hlediska návrhu se jedná o procesor, který je nastavitelný sadou parametrů. Mezi tyto parametry lze uvažovat například šířku datové cesty nebo velikost paměťových prvků.

[1] viz. kapitola 2.3.2
[2] viz. kapitola 2.3.3

2.1.1.3 RISC

Koncepce *RISC* (Reduced Instruction Set Computer) obsahuje malý počet jednoduchých instrukcí se stejnou dobou vykonávání. Instrukce mají pevnou délku a jsou jednotně kódovány. Procesor neobsahuje mikroprogramový řadič, je řízen pevnou logikou. Datové operace probíhají pouze nad registry, paměťové operace probíhají pouze pomocí vyhrazených instrukcí přesunů (Load/Store). Koncepce RISC obsahuje větší počet registrů než dále popsané CISC [4].

2.1.1.4 CISC

Koncepce *CISC* (Complex Instruction Set Computer) obsahuje velké množstvím relativně složitých instrukcí. Instrukce používají mnoho adresovacích režimů. Pro jejich vykonávání se typicky používá mikroprogramový řadič, jenž je řízen mikroprogramem, kde má každá instrukce definovánu sekvenci mikrooperací [4].

2.1.1.5 VLIW

Procesory s velmi dlouhým instrukčním slovem (*VLIW*, Very Long Instruction Word) mají blízký vztah k superskalárním procesorům. Obě třídy si kladou za cíl zvýšit výkonnost využitím paralelismu na úrovni instrukcí (ILP, Instruction-Level Parallelism). Obsahují několik funkčních jednotek pracujících paralelně. Instrukce VLIW obsahují řídící pole pro každou funkční jednotku v procesoru. Odpovědnost za výběr instrukcí, které mají být provedeny současně, a jejich naplánování (sestavení do posloupnosti instrukcí VLIW) je pouze na kompilátoru. Každý takt je vydávána jedna instrukce VLIW [3].

2.1.1.6 DSP

DSP (Digital Signal Processor) je procesor specializovaný na výpočty související se zpracováním digitálních signálů. Typický DSP má architekturu s oddělenou pamětí pro program a data. Data a kód programu využívají zvláštní sběrnice, což je důležité pro zpracování velkého množství dat v reálném čase. Dalšího zrychlení výpočtu se dosahuje pomocí specializovaných výpočetních jednotek, které mohou pracovat paralelně. DSP má například rychlou násobičku, která dokáže nezávisle na ALU (Arithmetic Logic Unit) provádět operaci MAC (Multiply-accumulate, násobení s přičítáním) v jednom taktu. Zpravidla také obsahuje dva nezávislé adresní generátory tzv. DAG (Data Address Generator). Ty dokáží adresovat data v lineárních nebo kruhových bufferech, což je důležité pro algoritmy zpracování digitálních signálů [16].

2.1.1.7 ASIP

Podle MEYR[12] lze zkratku *ASIP* použít k popsání dvou rozdílných druhů integrovaných digitálních obvodů.

- Application-Specific Integrated Processor – tento termín zahrnuje všechny druhy aplikačně specifických digitálních integrovaných obvodů používaných pro zpracování dat a nezahrnuje žádný druh instrukčně orientovaných nebo programovatelných aplikačně specifických obvodů pro zpracování dat.
- Application-Specific Instruction set Processor nebo Application-Specific Instruction Processor- tento termín označuje programovatelný aplikačně specifický procesor mající instrukčně orientovanou architekturu pro zpracování dat.

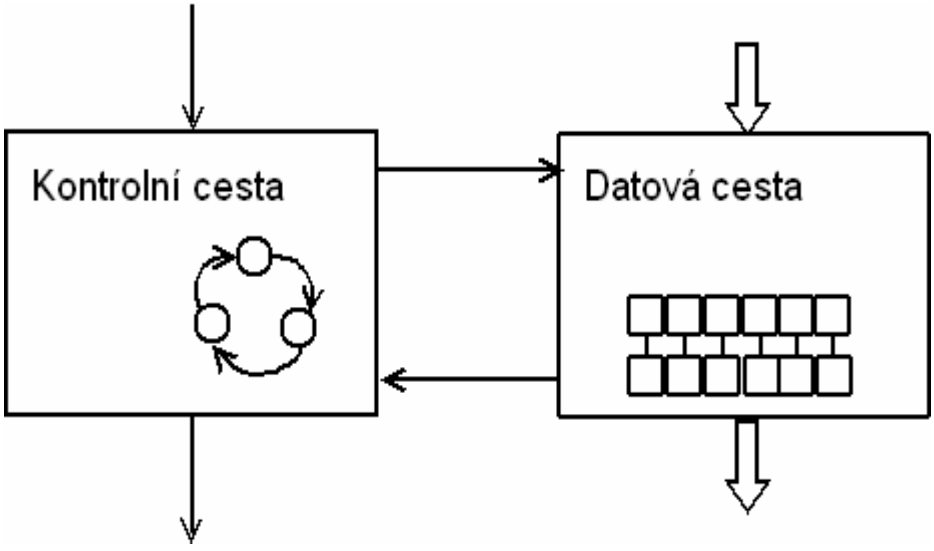
V této práci je zkratkou ASIP míněna architektura odpovídající druhému zmíněnému termínu.

2.1.1.8 Související pojmy

Instruction Set Architecture (ISA) je část procesoru, která je viditelná pro programátora či pro kompilátor.

Processor Architecture (PA) rozšiřuje rozsah ISA přidáním implementačních charakteristik, které jsou rezervovány pro software. PA obsahuje popis procesorových zdrojů (funkční jednotky a paměťové elementy), propojení mezi těmito zdroji, kódování a chování podporovaných instrukcí. Chování instrukcí určuje přechod mezi stavy procesoru a využití funkčních jednotek.

Každý PA obsahuje **datovou cestu** (datapath), která provádí vlastní výpočet (provádí jednotlivé instrukce). Zpravidla je rozdělena do několika stupňů a obsahuje funkční jednotky a paměťové elementy pro zpracování dat. Zbytek architektury představuje **kontrolní cesta**, která „říká“, co má datová část provádět. Generuje řídící signály pro datovou část, obvykle je implementována v podobě konečného stavového automatu. Vzájemná vazba obou částí je vidět na obr. 2.1.



Obr. 2. 1 Datová a kontrolní část PA

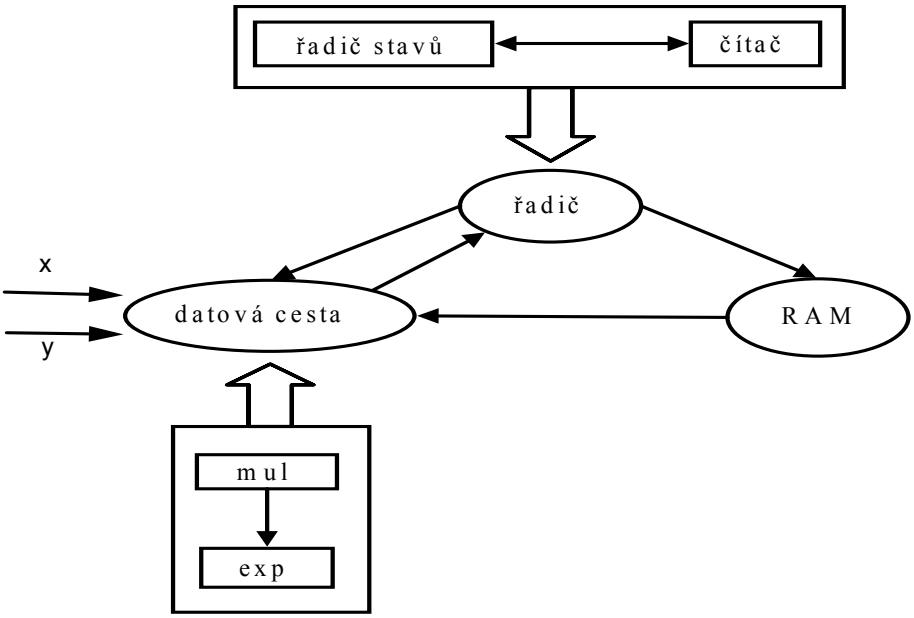
Zřetězený procesor používá registry pro rozdělení výpočetní úlohy na sekvence překrývajících se podúloh. Obvodová logika zajišťující vykonání instrukcí je rozčleněna do několika na sobě nezávislých bloků, které oddělují registry. Instrukce na sobě často závisejí a před započetím vykonání

další instrukce musí být dokončena jedna či více instrukcí předchozích. Podle DVOŘÁKA[3] existují celkem tři typy závislostí, které se mohou projevit jako konflikty při zřetězeném zpracování (pipeline):

- *Datové závislosti* – RAW (Read After Write, čtení po předchozím zápisu), WAW (Write After Write, zápis po předchozím zápisu) a WAR (Write After Read, zápis po předchozím čtení).
- *Strukturní závislosti* – instrukce vyžaduje prostředek, který je ještě používán předchozí instrukcí.
- *Řídící závislosti* – instrukce prováděné po podmíněném skoku závisejí na výsledku testu podmínky, který je součástí skokové instrukce.

2.1.2 Graf procesů

V oblasti návrhu číslicových obvodů je často využíván k popisu hierarchie návrhu PM (Process-Module) graf. Každý uzel tohoto grafu reprezentuje komponentu, modul, proces navrhovaného číslicového odvodu. Hrana koresponduje s propojením těchto komponent. Na obr. 2.2 je vidět grafová reprezentace návrhu číslicového odvodu. Na nejvyšší úrovni se obvod skládá z řadiče, datové cesty a paměti RAM (Random Access Memory). Všechny části (moduly, procesy) lze hierarchicky rozdělit na další části, jak je naznačeno u řadiče či datové cesty.



Obr. 2. 2 Grafová reprezentace návrhu

Graf procesů lze neformálně definovat jako zjednodušený PM graf, který neobsahuje hierarchii modulů a procesů. Každý uzel v grafu procesů reprezentuje jeden proces, který sekvenčně popisuje chování jisté části logického obvodu (uzel odpovídá procesu v jazyce VHDL). Hrany reprezentují propojení mezi jednotlivými procesy (tedy signály na úrovni jazyka VHDL).

2.1.3 Konečný automat

Formálně je *nedeterministický konečný automat M* podle ČEŠKY [1] definován jako uspořádaná pětice:

$$M = (Q, \Sigma, \delta, q_0, F),$$ kde:

1. Q je konečná množina stavů,
2. Σ je konečná vstupní abeceda,
3. δ je přechodová funkce tvaru $\delta : Q \times \Sigma \rightarrow 2^Q$,
4. $q_0 \in Q$ je počáteční stav
5. $F \subseteq Q$ je množina koncových stavů.

Je-li $\delta : Q \times \Sigma \rightarrow Q \cup \{nedef\}$, tj. $| \delta(q,a) | = 1$ pro všechna $q \in Q$ a $a \in \Sigma$, pak M se nazývá *deterministický konečný automat*.

2.2 Konstrukce a návrh vestavěných systémů

Tato kapitola objasňuje pojem vestavěného systému, zabývá se konstrukcí vestavěných systémů a ukazuje metodologii návrhu mikroprocesoru, jakožto hlavního prvku vestavěného systému.

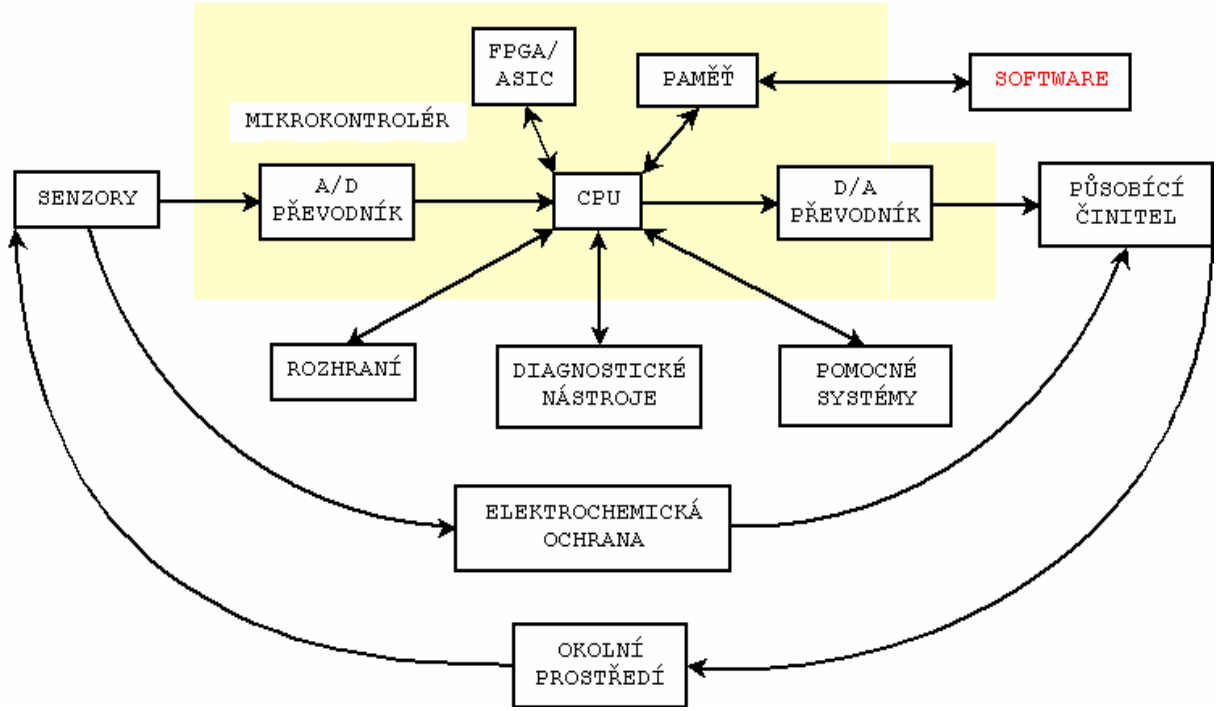
2.2.1 Vestavěné systémy

V literatuře lze nalézt následující definice vestavěných systémů [10]:

- Jde o systém obsahující jeden nebo několik mikroprocesorů, jehož (jejichž) úkolem je řídit činnost displejů, motorů, světel nebo jiných přístrojů. Program prováděný mikroprocesorem je „mozkem“ stroje nebo přístroje.
- Jde o počítač, který je zabudován do systému, ale pro uživatele není jako počítač viditelný.
- Vestavěný systém je dedikovaný počítač pracující v reálném čase jako součást nějakého uceleného systému. Slouží pro řízení celého systému a poskytuje výpočetní služby ostatním částem systému.

Z pohledu uživatele jde zpravidla o jednoúčelové systémy, jejichž reakce na změny v okolí systému musí být v reálném čase bez zpoždění. Typická je pro ně nízká cena a spotřeba, jsou malé a rychlé [10].

Typickou strukturu vestavěného systému podle SCHWARZE [15] znázorňuje obr. 2.3.

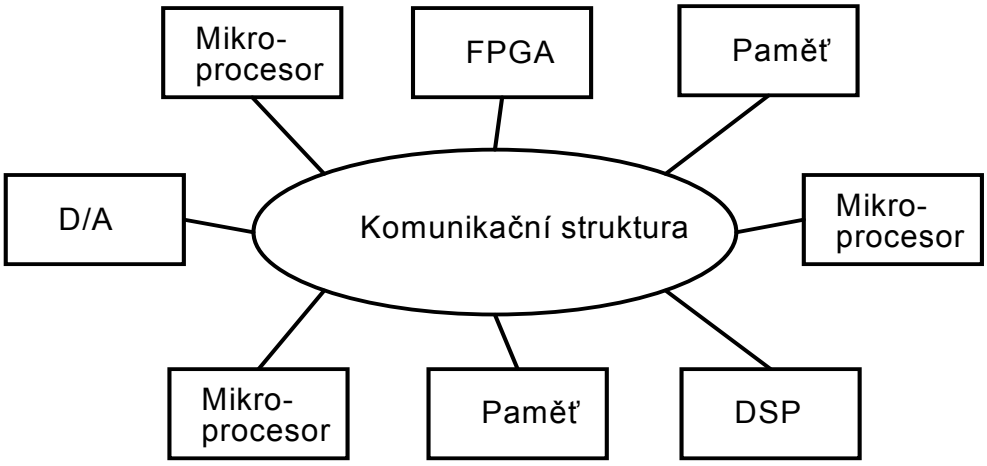


Obr. 2. 3 Typická struktura vestavěného systému

Funkční bloky, kterými jsou senzory, působící činitelé či různé pomocné systémy, jsou řízeny vestavěným mikrokontrolérem. Jádro řídící činnosti obstarává mikroprocesor (CPU, Central Processing Unit), který je součástí čipu mikrokontroléru. S použitím mikrokontroléru odpadá nutnost použití externí paměti, neboť paměť s dalšími přídavnými obvody je přímo součástí čipu mikrokontroléru.

2.2.2 Výroba vestavěných systémů

Pro realizaci vestavěného systému samotný mikroprocesor nestačí. Je třeba jej doplnit přídavnými obvody (paměti, řadič přerušení, obvody vstupů a výstupů, atd.). Pro výrobu je nejčastěji použita technologie SoC (Systém on Chip). Tato technologie umožňuje integraci velkého množství komponent jako mikroprocesorů, DSP, *FPGA* (Field Programmable Gate Array), pamětí, zákaznických obvodů a přídavných obvodů (D/A – digitálně-analogový převodník) na jeden čip (obr. 2.4.). Proces návrhu SoC je velmi komplexní. Od abstraktních modelů pro definici požadavků a systémovou specifikaci se postupným vytvářením dalších modelů dostáváme na nízkou úroveň implementačního modelu popisujícího rozmístění hardwarových komponent a asemblerovský kód. Z hlediska redukce času při návrhu SoC je důležité přesunout úlohy do programovatelných částí systémů. Programovatelné části mohou být popsány pomocí vyšších programovacích jazyků, např. C, C++, Java, atd. Programovatelné části SoC jsou většinou implementovány v ASIP.



Obr. 2. 4 Možná architektura SoC

2.2.3 Konstrukce vestavěných systémů

Při konstrukci vestavěných systémů lze v zásadě vyjít z jednoho ze tří koncepčních přístupů[10]:

- Realizace vestavěného systému pomocí jednoúčelového hardwaru.
- Použití mikroprocesoru speciálně určeného pro danou aplikaci – ASIP.
- Konstrukce založená na použití klasického mikroprocesoru pro všeobecné použití nebo mikrořadiče.

Systémy založené na jednoúčelovém hardwaru obsahují číslicové obvody umožňující provádět jediný jednoúčelový program. Nemají paměť programu. Jejich konstrukce obvykle bývá založena na použití plně zákaznických obvodů s vysokým stupněm integrace (Very Large Scale Integration – VLSI), specializovaných jednoúčelových obvodů (Application Specific Integrated Circuit – ASIC, např. hradlová pole) nebo programovatelných obvodů PLD (Programmable Logic Device). Mezi PLD patří i obvody typu FPGA, které mají z programovatelných obvodů nejobecnější strukturu a obsahují nejvíce logiky. Výhodou této koncepce jsou velká rychlost, malá spotřeba a malé rozměry výsledného systému. Příkladem systémů založených na jednoúčelovém hardwaru může být matematický koprocesor nebo akcelerátor pro grafické operace.

Dedikované mikroprocesory typu ASIP mají speciální funkční jednotky a architekturu optimalizovanou pro příslušný typ úloh. V závislosti na aplikaci jsou zvoleny instrukce a charakteristiky mikroprocesoru.

Systémy s ASIP obsahují paměť programu. Výhodou systémů založených na dedikovaných mikroprocesorech jsou částečná flexibilita, dobrá výkonnost, malé rozměry a malá spotřeba. Funkce implementované v hardwaru jsou zahrnuty v ASIP v podobě nových instrukcí nebo ve formě speciálních funkčních jednotek (integrované na čipu nebo jako periferní zařízení).

Mikroprocesory pro všeobecné použití nemají architekturu optimalizovanou pro specifický typ úloh. Obvykle mají velké pole registrů a obecnou aritmeticko-logickou jednotku. Systémy

s mikroprocesory mají paměť programu. Výhodou jsou krátká doba uvedení na trh, nízké náklady na vývoj a velká flexibilita.

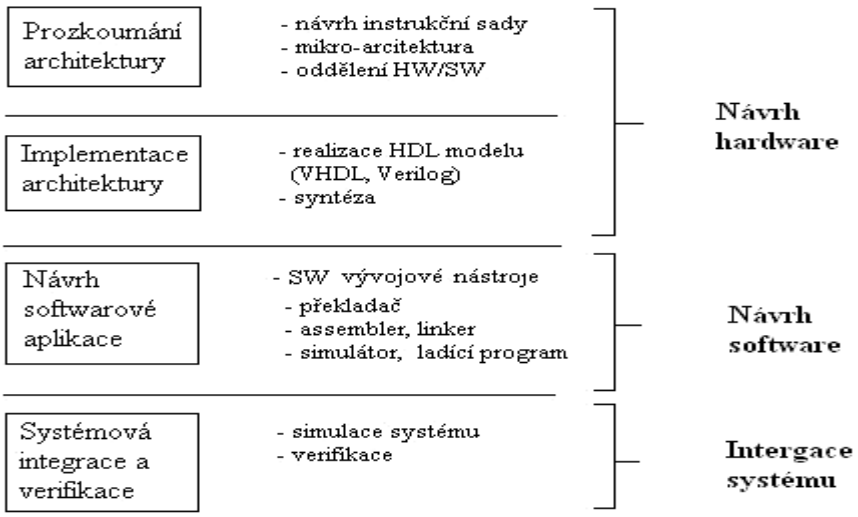
2.2.4 Metodologie návrhu mikroprocesoru

V oblasti výpočetní techniky přicházejí stále požadavky na řešení složitějších a náročnějších úkolů, nejlépe ve specializovaném hardwarovém řešení. Při volbě klasického počítače standardu PC (Personal Computer) můžeme při řešení nastíněných úkolů narazit na jistá omezení. Je třeba PC dovybavovat externími komponentami, jiné komponenty nahrazovat, ale téměř vždy dojdeme k situaci, kdy některé části nebudou pro danou úlohu dostatečně výkonné nebo naopak budou nevyužity. Pokud tedy vyžadujeme speciální prvky a podmínky, zřejmě se obrátíme k vývoji zařízení nového, specializovaného. To však znamená návrh a výrobu speciálních zařízení na zakázku (např. zařízení typu ASIP či typu FPGA). Další podkapitoly popisují tradiční způsoby návrhu těchto zařízení. Jednak pomocí jazyka ADL[1] a jednak pomocí jazyka HDL[1].

2.2.4.1 Průběh vývoje architektury ASIP za pomocí jazyka ADL

Vývoj mikroprocesorů ASIP vyžaduje podporu určitých softwarových nástrojů (překladač vyššího programovacího jazyka, asembler, linker[2], simulátor). Implementace podporujících nástrojů je časově náročná a je tedy požadována její automatizace (pro vyvíjený mikroprocesor se nástroje automaticky generují). Možnost automatického generování nástrojů je implementována v prostředí projektu Lissom, který využívá k popisu mikroprocesoru jazyka ISAC, který je velmi blízký jazyku LISA[3].

Průběh vývoje architektury ASIP lze rozdělit do čtyř fází, které jsou znázorněny na obr. 2.5 [6]:



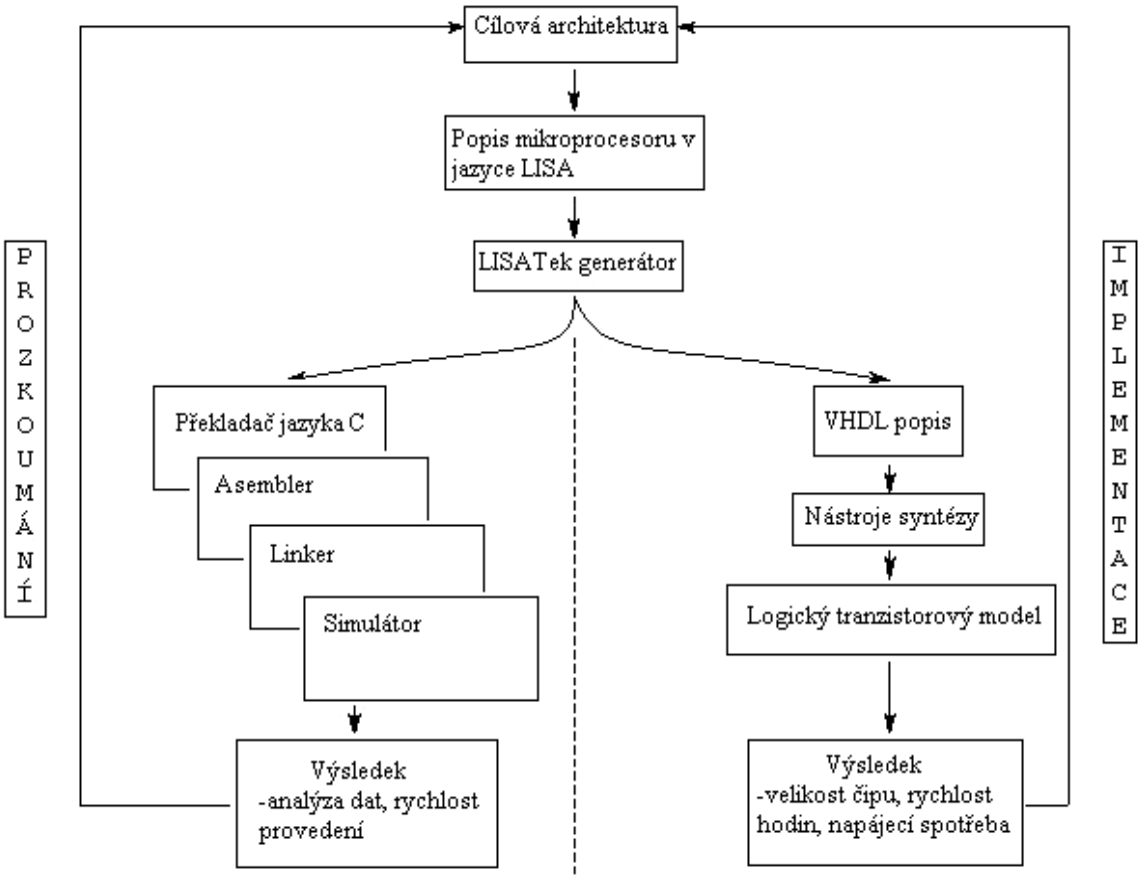
Obr. 2. 5 Čtyři fáze tradičního procesu návrhu mikroprocesoru

[1] viz. kapitola 2.3
[2] linker - sestavovací program
[3] viz. kapitola 2.3.1

- **Prozkoumání architektury** – cílová aplikace je mapována do jednoúčelové architektury mikroprocesoru. Skládá se ze tří úkolů, které jsou úzce svázány:
    1. Zjištění kritických částí aplikace, které vyžadují podporu hardwaru dostupnou přes aplikačně specifické instrukce.
    2. Definování instrukční sady.
    3. Ustálení mikro-architektury, která implementuje instrukční sadu.

  Proces prozkoumání architektury probíhá v několika iteračních krocích, dokud nedosáhneme nejvhodnějšího řešení (každá změna architektury vyžaduje kompletní novou sadu vývojových nástrojů).
- **Implementace architektury** – specifikovaný mikroprocesor je transformován do syntetizovatelného modelu jazyka pro popis hardwaru. Např. jazyk VHDL a Verilog.
- **Návrh softwarové aplikace** – vyžaduje podporu nástrojů pro pohodlné programování architektury.
- **Systémová integrace a verifikace** – návrh rozhraní pro integraci softwarového simulátoru do různých simulačních prostředí.

Standardní proces vývoje architektury za pomoci ADL jazyka LISA je popsán na obr. 2.6.



Obr. 2. 6 Proces vývoje architektury pomocí jazyka LISA

Návrh architektury v jazyce LISA vyžaduje po návrháři práci ve dvou fázích. V první fázi prozkoumání využívá návrhář automaticky generovaných nástrojů jako asembler, sestavovací program, simulátor a překladač. Pomocí simulátoru může získat profilová data o architektuře a zodpovědět si otázky týkající se instrukční množiny, výkonnosti algoritmu, požadované velikosti paměti a registrů.

V druhé fázi již s odsimulovanou architekturou přistupuje k hardwarové implementaci. Z popisu v jazyce LISA chce návrhář získat odpovídající popis v HDL jazyce. Tento kód lze již standardními syntetizačními nástroji transformovat na popis cílové technologie a poté provést např. vlastní vypálení na čip.

2.2.4.2 Návrh mikroprocesoru pro FPGA za pomocí HDL jazyka

Podle FALTÝNKA [4] lze při návrhu programovatelného hardwaru postupovat následujícím způsobem.

Při návrhu mikroprocesoru je nutné nejprve vybrat hlavní směry, kterými se bude návrh ubírat. Jednak v oblasti rozdělení paměti mikroprocesoru a jednak v oblasti uspořádání ústředního výpočetního systému mikroprocesoru.

Von Neumannova koncepce

Mikroprocesor má k dispozici jednu hlavní paměť, která obsahuje jednak program (instrukce), ale i data programu (proměnné, vstupní a výstupní data).

Harvardská koncepce

Mikroprocesor má k dispozici dvě oddělené paměti. První je paměť programu neboli *instrukční cache*. Jedná se v podstatě o paměť typu ROM (Read Only Memory). Druhou je paměť dat neboli *datová cache*. Ta je určena jak pro zápis, tak pro čtení. Při použití Harvardské koncepce odpadají možnosti modifikace programové paměti při běhu programu a je tím odstraněn jeden potencionální zdroj chyb při programování. Jsou přesně definovány a navzájem odděleny datové cesty, kterými jsou vedena instrukční či datová slova. Toto členění napomáhá jednoduššímu návrhu cílového hardwaru.

Druhou nastíněnou možností, kde lze volit směr dalšího návrhu, je organizace centrálního výpočetního systému mikroprocesoru. Zjednodušeně řečeno, je nutné definovat zdroje dat a cíle výsledků pro výpočetní (aritmeticko-logickou) jednotku. Pro tento účel jsou k dispozici tři základní možnosti architektury:

Registrová architektura

K dispozici je konečný počet registrů, ve kterých jsou uloženy zdrojové operandy operace. Výsledek bude uložen opět do jednoho z registrů. Každá instrukce může použít libovolný/é registr(y) ke své

činnosti. Registrová architektura vede k jednoduššímu programování, protože dostatečný počet registrů umožní ukládat proměnné výpočtu lokálně. Šetří se tím také čas, který je potřeba pro přístup do paměti. Nevýhodou registrové architektury je nutnost explicitního uvedení identifikace registrů v instrukčním kódu. Tím narůstá šířka instrukčního slova.

Akumulátorová architektura

Je podobná předchozí registrové v tom smyslu, že opět je k dispozici sada registrů. Počet registrů bývá nižší než u předchozí architektury. Řešení s akumulátorem se snaží zmírnit potřebu velké části instrukčního slova k adresování operandů. Jeden ze zdrojových operandů je vždy určen implicitně a je umístěn v tzv. *akumulátoru*, což je další registr se speciálním významem a zapojením. Akumulátor slouží zároveň jako registr výsledku. Z toho plyne, že v instrukčním kódu bude potřeba adresovat pouze jeden zdrojový registr. Nevýhodou této architektury je složitější přístup k programování. Je třeba psát program tak, aby byl před každou operací jeden zdrojový operand v akumulátoru a zároveň počítat s akumulátorem jako výsledkem.

Zásobníková architektura

Co do využití hardwarových zdrojů nejefektivnější početní architektura. Pro operandy a výsledek operace využívá *zásobníku*. Dva záznamy na vrcholu zásobníku jsou výpočetní jednotkou vyjmuty (POP), je provedena operace a výsledek je uložen do zásobníku (PUSH). Tento způsob je realizací tzv. postfixové notace (někdy též nazývané Polskou notací), kdy operační znaménko následuje až za uvedením zdrojových operandů.

Při návrhu je třeba též specifikovat další pojmy jako:

Datová šířka a šířka instrukčního slova

Nelehkým úkolem při návrhu je stanovit, s jakou *šířkou dat* se bude uvnitř procesoru operovat. Asi prvotní ukazatel, který naznačuje cestu, je dispozice s hardwarovými prostředky. Návrhář může disponovat externími paměťmi nebo interními paměťovými prvky v FPGA. Dále se volí šíře instrukčního slova (v digitálním světě se šíře datových sběrnic pohybuje v mocninách dvou).

Pro volbu šíře instrukčního slova existují principiálně dvě cesty. Buďto budou všechny instrukce zabírat stejný paměťový prostor, tedy šířka instrukčního slova bude pevná, nebo podle potřeby budou mít různé instrukce různou šíři. Každá instrukce pak zabere v paměti programu jiný prostor. Šířku instrukčního kódu určuje datová šířka.

Kódování instrukcí

Pro tuto část návrhu je nutné si stanovit, jaké instrukce by měl procesor provádět. Podle požadavků pak lze instrukce dle typu a počtu operandů rozdělit do několika skupin (např. instrukce se dvěma

zdrojovými operandy a jedním cílovým – typicky aritmetické a logické instrukce, s jedním zdrojovým a jedním cílovým operandem – instrukce přesunu, instrukce pracující s pamětí).

Každá instrukce nese veškeré informace potřebné k provedení přisouzené operace ve svém instrukčním kódu.

Komponenty procesoru

- *Programový čítač* – dodává na výstup aktuální adresu instrukce a po jejím načtení z paměti programu je obsah programového čítače zvýšen obvykle o jednu instrukci (nemusí platit pro skokové instrukce).
- *Zásobník návratových adres* – je zodpovědný za uchování návratových adres. Ty jsou užívány instrukcí, která vrací řízení programu na místo, odkud se volal podprogram, aniž by bylo explicitně nutné udávat bod, kam se má skočit.
- *Aritmeticko-logická jednotka (ALU)* – jde o ústřední výpočetní prvek procesoru.
- *Instrukční dekodér* – z načtených instrukčních kódů dekóduje informace potřebné pro provádění instrukcí.
- *Registry* – sada registrů je speciálnější případ paměti (registrové pole). Dalo by se využít i přímo hardwarových registrů tvořených sadou D-klopných obvodů, avšak takovéto registry by byly z pohledu hardwarových zdrojů příliš drahé.
- *Paměti* – jedná se o registrové pole, paměť instrukcí a paměť dat.

Aby návrhář byl schopen navržené komponenty procesoru implementovat, potřebuje sadu podpůrných programových nástrojů (asembler, simulátor, syntezátor). Tyto nástroje překlenou cestu od zdrojového tvaru v jazyce HDL až k binárnímu formátu vhodnému k zavedení do hardwaru.

2.2.4.3 Společné rysy návrhu

V předchozích dvou kapitolách byly představeny dva pohledy na návrh mikroprocesoru. Rozdíl v přístupu k návrhu je zřejmý. V prvním případě je k návrhu použit jazyk, který poskytuje rychlou a přehlednou simulaci vyvíjené architektury (zejména instrukční sady), avšak přináší vysokou úroveň abstrakce pro hardwarovou implementaci (konkrétní propojení dílčích komponent). V druhém případě je nutné při návrhu jít na nižší úroveň abstrakce a zabývat se právě konkrétními komponentami a jejich propojením.

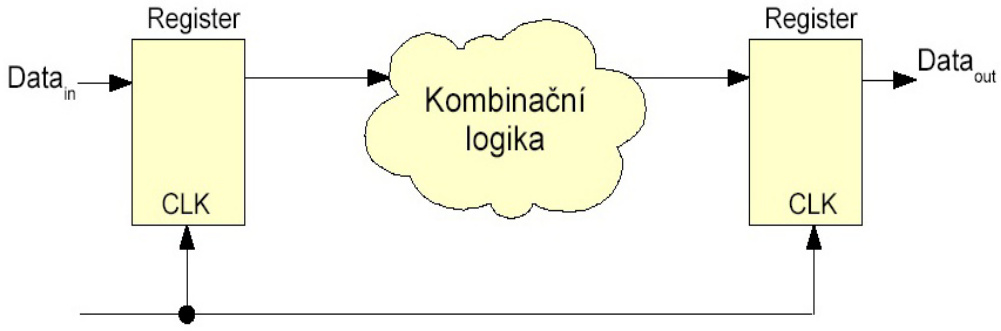
Aby byl návrh za pomoci jazyka ADL úspěšný je třeba automatických nástrojů, které vytvoří z ADL popisu popis na úrovni RTL (Register Transfer Level). Tento popis je vhodný pro vlastní syntézu mikroprocesoru a nejčastěji je implementován v HDL jazyce.

2.2.5 Popis architektury na úrovni RTL

U jazyků HDL se setkáváme převážně s popisem struktury číslicových systémů na úrovni abstrakce, která se označuje jako úroveň *meziregistrových přenosů* (RTL). Při ní se vychází

z představy, že u synchronních subsystémů, s nimiž nejčastěji pracujeme, je jejich struktura složená z registrů, mezi něž jsou vloženy kombinační celky. Jako registr může být označena libovolná část číslicového systému, která obsahuje nebo uchovává binární informaci.

Typicky lze popis na úrovni RTL chápat tak, jak znázorňuje obr. 2.7. (obrázek byl převzat z [9]).



Obr. 2. 7 Popis na úrovni RTL

Mezi základní funkční elementy, které se využívají v popise na úrovni RTL, lze zařadit logická hradla (AND, OR, NOT, XOR), registry (D-klopné obvody) a vyšší funkční celky (paměti, aritmeticko-logické jednotky, funkční jednotky, multiplexory, dekodéry).

2.3 Jazyky ADL a HDL

Se vzrůstajícím počtem tranzistorů na čipu mikroprocesorů již nelze při návrhu použít klasické logické schéma, ale návrhář popisuje obvod pomocí vhodného jazyka. Jazyky a k nim příslušející nástroje pro vývoj architektury mikroprocesoru lze rozdělit na dva typy.

Prvním typem jazyků jsou jazyky ADL (Architecture Description Languages). Jde o jazyky s vysokou úrovní abstrakce, což umožňuje rychlejší aplikaci změn do modelu architektury. Ačkoliv je vyšší míra abstrakce základem úspěchu těchto jazyků, ztrácí se tu spojitost architektury s fyzickými parametry jako je plocha na čipu, frekvence hodin či výkon. Pokud by návrháři ignorovali fyzické parametry během fáze vývoje architektury, vedl by vývoj jen k nějakému suboptimálnímu řešení. Kombinace vysoké míry abstrakce a fyzických parametrů během vývoje je nezbytná. Jazyky ADL lze rozdělit na tři skupiny:

- **Jazyky zaměřené na instrukční sadu** - charakterizují mikroprocesor prostřednictvím architektury instrukční sady. Obsahují pouze informaci o instrukční sadě, nikoliv další informace o mikro-architektuře. Používají se zejména pro vývoj softwarových nástrojů.
- **Jazyky zaměřené na architekturu –** zaměřují se na strukturní komponenty a propojení architektury mikroprocesoru. Mají blízko k popisu na úrovni RTL.

- **Jazyky zaměřené na instrukční sadu i architekturu -** zahrnují v sobě oba předchozí přístupy pro popis architektury mikroprocesoru. Syntéza architektury a generování vývojového softwaru jsou provedeny z jednoho popisu.

Z představené skupiny jazyků, je pro tuto práci nejpodstatnější skupina poslední, do které lze zařadit jazyk ISAC.

Druhým typem jazyků jsou jazyky HDL (Hardware Description Language). Typickými představiteli jsou jazyky VHDL a Verilog, které jsou zaměřeny na modelování a simulaci procesoru. Hlavním cílem těchto jazyků je hardwarová implementace procesoru. Tyto jazyky zahrnují velké množství hardwarových implementačních detailů, které nejsou potřeba zejména pro cycle-based simulaci (simulaci prováděnou po cyklech procesoru) či softwarovou verifikaci.

2.3.1 Jazyk LISA

Jazyk LISA (Language for Instruction Set Architecture), popsaný detailněji v [5], je jazykem smíšené úrovně (strukturální a behaviorální), který vede k formalizovanému popisu programovatelných architektur, jejich periferií a rozhraní. Jazyk zamýšlí zacelit mezeru mezi striktně strukturálně orientovanými jazyky (VHDL, Verilog) a jazyky pro popis instrukční sady. Dřívější verze jazyka LISA se zaměřovaly hlavně na generování kompilovaných simulátorů. Dnešní verze jazyka LISA zahrnuje model instrukční sady a poskytuje vysokou flexibilitu pro popis různých mikroprocesorů. Rovněž mikroprocesory s komplexním zřetězením mohou být v jazyce LISA snadno modelovány. Jazyk LISA je silně orientován na programovací jazyk C.
Jazyk LISA se skládá ze dvou základních částí. První část je tvořena deklarací zdrojů, z nichž se modelovaný mikroprocesor skládá. Druhá část je tvořena popisem operací.

Deklarace zdrojů

Deklarované zdroje udržují stav programovatelné architektury (uchované např. v registrech, v pamětech) ve formě uložené datové hodnoty. Deklarace zdrojů zahrnuje čtyři typy objektů:

- Jednoduché zdroje – registry, sběrnice, paměti.
- Zřetězené struktury pro instrukce a datový tok.
- Registry pro pipeline, které umožňují uložení dat mezi jednotlivými stavy pipeline.
- Mapování pamětí, které umístí zdroje do adresového prostoru.

Tyto třídy jsou použity pro pozdější ladění a generování (např. generátor HDL jazyka).

Popis operací

Operace je základní stavební objekt v jazyce LISA. Obsahuje kolekci popisů různých vlastností systému. Např. chování operací, informace o instrukční sadě, časování. Operace je popsána v následujících sekcích:

- Sekce CODING popisuje binární obraz instrukčního slova.
- Sekce SYNTAX zachycuje syntaxi asemblerovských instrukcí a jejich operandů.
- Sekce BEHAVIOR A EXPRESSION popisuje komponenty modelu chování. Během simulace je chování operace prováděno a jsou modifikovány hodnoty zdrojů. Tím se systém dostává do nového stavu.
- Sekce ACTIVATION definuje časování jiných operací vzhledem k popisované operaci.
- Sekce DECLARE obsahuje lokální deklarace identifikátorů, skupin operací a odkazů na jiné operace.

Operace mohou být buď atomické nebo se skládají z jiných operací. Model tedy podporuje hierarchické strukturování operací.

Generované nástroje využívají jednotlivé části těchto sekcí. Např. asembler využívá informaci obsaženou v sekci CODING a SYNTAX k mapování syntaxe instrukčního slova na binární obraz instrukčního slova. Simulator je závislý na informacích obsažených v sekci BEHAVIOR.

2.3.2 Jazyk ISAC

V této kapitole je uveden zestručněný popis jazyka ISAC. Jeho kompletní popis je k dispozici v [7] .

Jazyk ISAC (Instruction Set Architecture C) je speciální jazyk pro popis architektury mikroprocesoru i instrukční sady. Jak bylo uvedeno výše, jde o představitele z rodiny jazyků ADL. Je nadstavbou nad vyšším programovacím jazykem ANSI C. Jazyk ISAC je odvozen od jazyka LISA. Používá některá stejná klíčová slova. Stejně jako jazyk LISA jde o jazyk smíšené úrovně popisu (strukturální a behaviorální), který vede k formalizovanému popisu programovatelných architektur, jejich periferií a rozhraní.

Jazyk ISAC se rovněž skládá ze dvou základních částí. První část je tvořena deklarací zdrojů, z nichž se modelovaný mikroprocesor skládá a které určují stav modelu. Druhá část je tvořena popisem operací.

Zdroje procesoru jsou popsané ve zdrojové sekci uvedené klíčovým slovem RESOURCE. Instrukční repertoár a chování operací je popsané v sekci operační, která následuje bezprostředně za zdrojovou sekcí .

2.3.2.1 Zdrojová část

V sekci zdrojů jsou definovány všechny zdroje modelu jazyka ISAC jako pipeline (zřetězené zpracování), paměti, registry atd. Zdroje definují stav mikroprocesoru a specifikují jisté třídy zdrojů pro pozdější použití v ladění a generování. Pořadí zdrojů není povinné. Zdroje jsou generovány z popisu v jazyce ISAC ve formě proměnné abstraktního datového typu.

Jednoduché zdrojové prvky

Mezi jednoduché zdrojové prvky patří všechny typy registrů. Speciálním typem registru je programový čítač. Pro jeho deklaraci je použito navíc klíčové slovo **PC**. Jako programový čítač lze deklarovat pouze jediný zdrojový prvek. Dalším speciálním typem registru je kontrolní registr, pro jehož deklaraci je použito klíčové slovo **CR**. Bitová šířka programového čítače stejně tak kontrolního registru nesmí být jedna. Pokud se bitová šířka registru rovná jedné, pak jde o speciální zdrojový prvek PIN, neboli jednobitový registr.

Paměťové prvky

Jazyk umožňuje popisovat ideální paměti (**MEMORY**), stejně jako neideální komponenty **RAM**, vyrovnávací paměti (**CACHE)** a sběrnice (**BUS**). U paměťových prvků umožňuje jazyk specifikovat řadu parametrů:

BLOCKSIZE - specifikuje bitovou šířku bloku a podbloku uvažované paměti. Šířka bloku musí být násobkem podbloku. Šířka bloku musí odpovídat bitové šířce zdroje. U MEMORY musí být šířka bloku i podbloku stejná.

ENDIANESS - definuje uspořádání (endianess – způsob uložení bajtů, pořadí bajtů) v případě, že je přistupováno k podblokům. Povolené hodnoty parametru jsou **LITTLE** a **BIG**, kde **LITTLE** je implicitní hodnotou, která je nastavena, pokud není parametr definován. **LITTLE** a **BIG** nejsou klíčová slova.

SIZE - definuje počet bloků paměti. Parametr není povolen pro sběrnice.

LATENCY - lze použít pouze pro paměti **CACHE** a **RAM**. Specifikuje latenci pro operaci čtení a zápisu (první a druhý parametr). Implicitní hodnota (1,1) je nastavena, pokud není parametr definován.

FLAGS - specifikuje, zda je paměť čitelná, zapisovatelná nebo zda může obsahovat proveditelný kód. To je indikováno třemi hodnotami **R,W,X**. V tomto parametru jsou tato písmena oddělena čárkou. Obecně by měly být programové paměti **(R,X)** (read-only, executable) a datové paměti **(R,W)** (read-write, non-executable). **R, W, X** nejsou klíčová slova. Pro FLAGS parametr jsou povoleny pouze hodnoty **R, W, X.**

Parametry použitelné pro paměti CACHE

NUM_WAYS - specifikuje asociativitu cache, tj. počet cest pro set-asociativní cache (cache s omezeným stupněm asociativity). Implicitní hodnota je 1, což znamená přímo mapovaná cache. Implicitní hodnota je nastavena, pokud není parametr definován.

LINESIZE - specifikuje počet bloků na řádek cache. Implicitní hodnota je 1. Je nastaven, pokud není parametr definován. Musí to být mocnina 2.

WA_POLICY - definuje strategii write-allocate (strategii jíž se řídí řadič paměti) pro cache, pokud jsou řádky cache alokovány v případě write miss (výpadek při zápisu do paměti). Povolené hodnoty jsou **NEVER** a **ALWAYS.** Pokud není parametr definován, je implicitně nastaven na **NEVER. NEVER** a **ALWAYS** nejsou klíčová slova.

WB_POLICY - definuje strategii write-back (odložený zápis) pro cache. Povolené hodnoty jsou **NEVER** a **ALWAYS.** Pokud není parametr definován, je implicitně nastavena na **NEVER. NEVER** a **ALWAYS** nejsou klíčová slova.

RPL_POLICY - definuje strategii replacement (strategie přepisu) pro cache. Povolená hodnota je **LRU** (*least recently used*). Pokud není parametr definován, je implicitně nastaven na **LRU. LRU** není klíčové slovo.

WRITEBUFSIZE - definuje velikost volitelné zápisové vyrovnávací paměti (v řádcích cache).

CONNECT - definuje paměťový modul, ke kterému je cache připojena. Může to být zdroj typu RAM, CACHE nebo BUS. Parametr **CONNECT** definuje další úroveň cache. Připojený zdroj musí mít stejnou bitovou šířku a velikost bloku. Parametrem je identifikátor paměťového modulu reprezentující další úroveň paměti.

Mapování pamětí

Pro spuštění simulátoru je nezbytné zobrazení adresového prostoru mikroprocesoru s definovanými paměťmi. Zobrazení musí být jednoznačné, výjimkou jsou stránky (**PAGE**).

Může existovat jedno nebo více mapování paměti, z nichž jedno musí být implicitní. Identifikátorem implicitního mapování je **defaultmap** (není to klíčové slovo, ale má na tomto místě speciální význam). Je-li identifikátor vynechán, je to ekvivalentní uvedení identifikátoru **defaultmap**. Alespoň jedno mapování paměti s identifikátorem **defaultmap** musí existovat.

Každá položka mapování paměti zobrazuje obvykle koherentní rozsah paměťových adres do paměťového zdroje, který musí pochopitelně existovat. V prostém mapování užijeme pouze podklauzule **RANGE**, případně můžeme paměť rozdělit na stránku klauzulí **PAGE**.

Konektivita sběrnic

Abychom připojili paměťový modul s jistým rozsahem adres ke sběrnici, užijeme, dodatečné vzhledem k prostému mapování, klíčové slovo **BUS** v kombinaci s popisem paměťového rozsahu. Musí platit, že rozsah indexů na pravé straně nesmí být větší, než je bitová šířka sběrnice.

Struktury se zřetězeným zpracováním

Cílem zřetězeného zpracování je zlepšení průchodu instrukcí procesorem. Je-li provedení instrukce rozděleno do několika stavů, každá část vykonává část celkového provedení. Zřetězené zpracování si

musí pamatovat stavy instrukcí v různém stavu zpracování. Kontext instrukce je udržován v oddělovacích registrech pro zachycení mezivýsledku.

Zřetězené zpracování má svůj identifikátor. Za ním následují složené závorky, ve kterých jsou definovány jednotlivé stavy zřetězeného zpracování. Stav je definován identifikátorem, za nímž následuje dvojtečka. Za dvojtečkou jsou uvedeny oddělovací registry (v hranatých závorkách) a stavové registry (v kulatých závorkách). Jako oddělovací a stavové registry lze použít veškeré registry deklarované jako **REGISTER.** Registr s bitovou šířkou jedna použít nelze.

Aliasing

Procesory mohou mít množství virtuálních zdrojů, které nemají exaktní ekvivalent v hardwaru. Příkladem mohou být paměťově mapované registry. Pro modelování takových zdrojů slouží aliasing.

2.3.2.2 Operační část

Na nejvyšší úrovni definice jsou nabízeny dvě možnosti:

- operace **OPERATION**, definují nonterminální symboly (symboly, které lze dále přepisovat na řetězce jiných symbolů) gramatik do obou směrů překladu.
- skupiny **GROUP**.

Skupiny jsou definice A-pravidel (variant). Skupina popisuje společným jménem různé varianty analýzy. Omezení proti standardním gramatikám je, že varianty jsou pouze pojmenované operace, tedy nonterminály. Tedy variantou je pouze pojmenovaný nonterminál. To má sice nevýhodu, že je zde více nonterminálů, ale výhodu, že každá varianta má jméno (jméno operace) a toto jméno je v rámci skupiny jednoznačné. Je jasné, že jedna operace se smí ve skupině vyskytovat pouze jednou. Každá operace má jedinečné jméno.
Operace je popsána v následujících sekcích:

- Sekce **CODING** popisuje binární obraz instrukčního slova.
- Sekce **ASSEMBLER** zachycuje mnemoniku a jiné syntaktické komponenty jazyka asembleru jako jsou operandy a prováděcí módy.
- Sekce **BEHAVIOR** a **EXPRESSION** popisují chování operace. Během simulace je chování operace prováděno a jsou modifikovány hodnoty zdrojů a tím se systém dostává do nového stavu. Pro popis chování je použit jazyk ANSI C, který je ochuzen o některé konstrukce.
- Sekce **CODINGROOT** určuje zdroje mikroprocesoru, jež obsahují instrukce, které budou dekódovány pomocí specifikovaných operací (*OPERATION*) jazyka ISAC. Operace nalézající se v rámci jedné sekce se provádí souběžně v jednom taktu mikroprocesoru (sekce umožňuje definovat mikroprocesory typu VLIW). V rámci sekce je definováno podmíněné dekódovaní instrukcí pomocí řídících konstrukcí SWITCH a IF.

- Sekce **ACTIVATION** provádí plánování operací vzhledem k současné operaci. Konstrukce umožňuje vytvářet simulátory založené na cyklech. Je pro ni vyžadována podpora temporálních operátorů, jež umožňují definovat zpoždění mezi operacemi. Provedení plánované operace se muže uskutečnit v rámci aktuálního taktu mikroprocesoru, a nebo lze specifikovat odloženou aktivaci pomocí temporálního operátoru (operace opožděné aktivace). V rámci sekce je definováno podmíněné aktivování operací pomocí řídících konstrukcí SWITCH a IF.

Sekce nesmí být v rámci operace deklarována více než jedenkrát. Ke každé sekci ASSEMBLER musí existovat párová sekce CODING a naopak. V každé operaci je třeba deklarovat (v sekci **INSTANCE**), které všechny skupiny a operace se v definované operaci budou používat. Mohou se použít pod původním jménem (to je pak v operaci lokálně deklarováno) nebo je možné za klíčovým slovem **ALIAS** deklarovat seznam jmen, pod kterými ještě navíc může skupina nebo operace lokálně vystupovat. To je nutné v případě, kdy se v rámci deklarace **CODING** sekce vyskytuje více výskytů téže operace nebo skupiny. Pak musí mít každá své jiné jméno. Sekcí **INSTANCE** může být i několik.

2.3.3 Jazyk VHDL

V této kapitole uvádím jen stručný popis jazyka. Kompletní popis je k dispozici např. v [2].

VHDL (z akronymu VHSIC[1] Hardware Description Language) je jedním ze základních představitelů HDL jazyků. Je určen k použití ve všech fázích návrhu logických zařízení (obvodů). Podporuje vývoj, syntézu a testování hardwarových návrhů. Popsaný obvod je možné modelovat a simulovat. Jazyk VHDL umožňuje tři základní úrovně popisu:

- Behaviorální (popis chování)
- Strukturní
- Data-flow (popis datovými toky)

Jednotlivá zařízení a jejich části jsou v jazyce VHDL popisovány pomocí komponent. Komponenta je popsána pomocí entity a architektury.

2.3.3.1 Entita

Entita specifikuje rozhraní komponenty, tj. vstupní a výstupní porty, kterými může komponenta komunikovat s okolím. Je specifikována pomocí klíčového slova **entity**. Mimo porty může rozhraní obsahovat i generické parametry. Porty (vnější signály entity) jsou charakterizovány jménem, módem činnosti a datovým typem. Mód činnosti určuje směr šíření dat, tj. IN (data lze pouze číst), OUT (data odcházejí ven z entity) nebo INOUT (obousměrný tok). Lze spojovat porty pouze se stejným datovým typem.

[1] VHSIC – Very High Speed Integrated Circuit

2.3.3.2 Architektura

Naproti tomu architektura definuje funkci, tedy vnitřní popis komponenty, a je vždy svázána s entitou. Je specifikována pomocí klíčového slova **architecture.** Funkce může být popsána buď chováním nebo strukturou s použitím dalších komponent. Komponenty lze tedy používat hierarchicky. Každá entita může mít víc alternativních popisů (architektur).

Architektura nabízí dvě různá prostředí, ve kterých ji můžeme popsat. Jedná se buď o prostředí sekvenční nebo prostředí paralelní. Sekvenční prostředí se využívá k popisu chování (algoritmicky, behaviorální popis). Lze využít proces, podprogram nebo funkci. Paralelní prostředí popisuje vazby mezi bloky strukturním popisem, tj. hierarchií dílčích komponent, nebo data-flow popisem (paralelní příkazy).

Behaviorální popis

Pro behaviorální popis se využívá procesů (klíčové slovo **process**). Cílem je popsat jak se mění výstupy v závislosti na změnách vstupních signálů. Proces je tedy souhrnem sekvenčních příkazů popisujících chování dané komponenty. Architektura může obsahovat více procesů komunikujících vzájemně pomocí signálů. Jednotlivé procesy jsou při simulaci vykonávány paralelně. Hlavním problém behaviorálního popisu je fakt, že hardwarová realizace nemusí být zřejmá a může být tedy problém se syntézou do cílové technologie. Popis je vhodný zejména pro simulace.

Strukturní popis

Popisuje z čeho se komponenta skládá (jakou má strukturu). Vyjadřuje konkrétní vazby mezi dílčími komponentami, popisuje propojení komponent pomocí signálů. Umožňuje vytvořit hierarchický popis, tzn. že zapojená komponenta může mít opět strukturu složenou z dalších dílčích komponent. Komponenty na nejnižší úrovni jsou vždy popsány behaviorálně.

Každá komponenta může být v architektuře nainstalována vícekrát, v zapojení jsou z nich vytvořeny instance, které jsou označeny jménem. Mapování portů komponenty se realizuje příkazem **port map,** který připojuje porty komponenty k portům entity nebo k lokálním signálům. Obecně platí, že strukturní popis má blíž k hardwaru, protože je možné systém dekomponovat až na základní stavební bloky a ty pak přímo mapovat na cílovou technologii. Výhodou je optimální využití prostředků, ale zápis může být na první pohled nepřehledný a nebude z něj patrná funkčnost, resp. chování systému.

Návrhář v jazyce VHDL nemusí striktně dodržovat ten či onen způsob popisu. Může podle potřeby přecházet od jedné úrovně popisu k druhé. Typickým případem může být dekompozice návrhu na ucelené části, návrh rozhraní mezi těmito částmi a jejich behaviorální zápis. Složení celého hlavního projektu pak bude zapsáno strukturně propojením dílčích celků.

Data-flow popis

Popis architektury datovými toky je realizován paralelním příkazem přiřazení signálu. Každý z těchto příkazů je proveden, když nastane nějaká změna na vstupním signálu obsaženém v tomto příkazu. Příkazy mohou být podmíněné nebo nepodmíněné. Na této úrovni je nejsnadnější popis pohybu dat v navrhovaném systému. Uvnitř většiny číslicových systémů jsou registry, takže úroveň toku dat popisuje, jak jsou informace předávány mezi registry v obvodu. Tento popis je vhodný zejména pro syntézu.

2.3.3.3 Objekty

Jazyk VHDL rozlišuje základní typy datových objektů:

- Konstanty – nemění hodnoty v sekvenčním i paralelním prostředí.
- Proměnné – jsou lokální v procesech, nepředstavují skutečné signály.
- Signály – fyzické vodiče, mění hodnoty v sekvenčním i paralelním prostředí.
- Soubory – obsahují data určitého typu a používají se např. jako vstupy a výstupy při simulaci.

Nejedná se o objekty jako v klasickém objektově orientovaném jazyce, jejich účelem zde je reprezentovat atributy simulovaného systému. Každý objekt musí být nějakého datového typu.

Vedle datových objektů lze tedy rozlišit 3 základní skupiny datových typů:

- Skalární – diskrétní, real, výčtový
- Složené – pole, záznamy
- Přístupové – obdoba ukazatele z jiných jazyků

2.3.3.4 Syntéza

Důležitou součástí návrhu v jazyce VHDL je kromě simulace syntéza. Jde o proces transformace zdrojového kódu v jazyce VHDL na zapojení na úrovni logických hradel. Z této úrovně je poté vytvořen *Netlist* (detailní propojení komponent cílové technologie) prvků cílové technologie. Během syntézy je prováděna optimalizace či případná minimalizace obvodu. Při popisu obvodu je nutné brát zřetel právě na syntézu, protože ne všechny konstrukce mohou být syntetizovatelné (zejména behaviorální popis).

Případ popisu v jazyce VHDL, kdy není definovaná hodnota výstupu pro některou kombinaci na vstupu vede při syntéze na zachování původní hodnoty pomocí záchytného registru (latch). Vznik těchto registrů je většinou nežádoucí, protože způsobuje zpoždění a tím může být potenciálně snížena maximální frekvence obvodu. Proto se vždy snažíme pokrýt všechny možnosti výstupu pro každou vstupní kombinaci (platí zejména pro řídící příkazy **if** a **case**).

2.3.3.5 Generická komponenta a příkaz generate

Pomocí příkazu **generate** lze jednoduše navrhovat obvody s pravidelnou strukturou. Tento příkaz se nachází mimo proces a počet opakování nebo podmínka musí být konstantní. Ve spojení s parametrem **generic** u entity lze vytvářet generické komponenty potažmo generické obvody. Např. snadno pomocí existence komponenty invertoru můžeme generovat invertor s libovolnou šířkou bitů.

2.4 Transformace jazyka ISAC

Jazyk ISAC má návrháři mikroprocesoru poskytnout prostředí pro prozkoumání a implementaci nové architektury procesoru. Ve fázi prozkoumání má k dispozici sadu generovaných nástrojů (asembler, simulátor) pomocí nichž může architekturu navrhnout na vysoké úrovni abstrakce. Bude navržena instrukční sada, definovány hardwarové zdroje, simulováno chování, časování, zřetězené zpracování, dekódování instrukcí. Avšak návrhář v této fázi bude abstrahovat od konkrétního fyzického zapojení hardwarových zdrojů, či dekodéru instrukcí. Problémem skutečného zapojení se zabývá fáze implementace. V ní chceme získat Netlist pro konkrétní technologii (ASIC, FPGA). Abychom mohli Netlist získat, musíme použít jiného popisu architektury a to popisu na nízké úrovni abstrakce, který nabízejí jazyky HDL. Cílem je tedy provést programovou transformaci z jazyka ISAC do jazyka HDL, konkrétně jazyka VHDL. Jazyk VHDL již má standardní syntetizační nástroje, pomocí nichž kýžený Netlist získáme. Fáze prozkoumání a implementace znázorňuje obr. 2.8.

Nejprve tedy objasním pojem programové transformace, poté budu diskutovat možnosti řešení programové transformace mezi jazyky ISAC a VHDL.

2.4.1 Programová transformace

Program můžeme chápat jako strukturovaný objekt se sémantikou. Obvykle je program zapsán v nějakém programovacím jazyce (poté program nazýváme zdrojovým kódem), který určuje jeho syntaxi. Díky pevné syntaxi (pevné struktuře) můžeme program transformovat na jiný program.

Programová transformace je tedy nějaká operace, která z programu v jednom jazyce generuje program v jiném jazyce. Jazyk, ve kterém je popsán transformovaný program se nazývá zdrojový, a jazyk, ve kterém je popsán program výsledný se nazývá cílový. Přičemž platí, že zdrojový a cílový jazyk může být shodný nebo rozdílný.

Velmi často je důležité, aby zdrojový program byl sémanticky ekvivalentní s programem cílovým. Sémantika programu nám dává možnost *validovat* výsledek transformace.

Dalším důležitým aspektem transformace je úroveň abstrakce popisu. Transformace může buď snižovat, zachovávat či zvyšovat úroveň abstrakce zdrojového programu. Poté transformaci, která snižuje úroveň abstrakce nazýváme programovou syntézou, transformaci, která zachovává úroveň

abstrakce programovou migrací a transformací zvyšující úroveň abstrakce označujeme jako Reverse Engineering.[1]

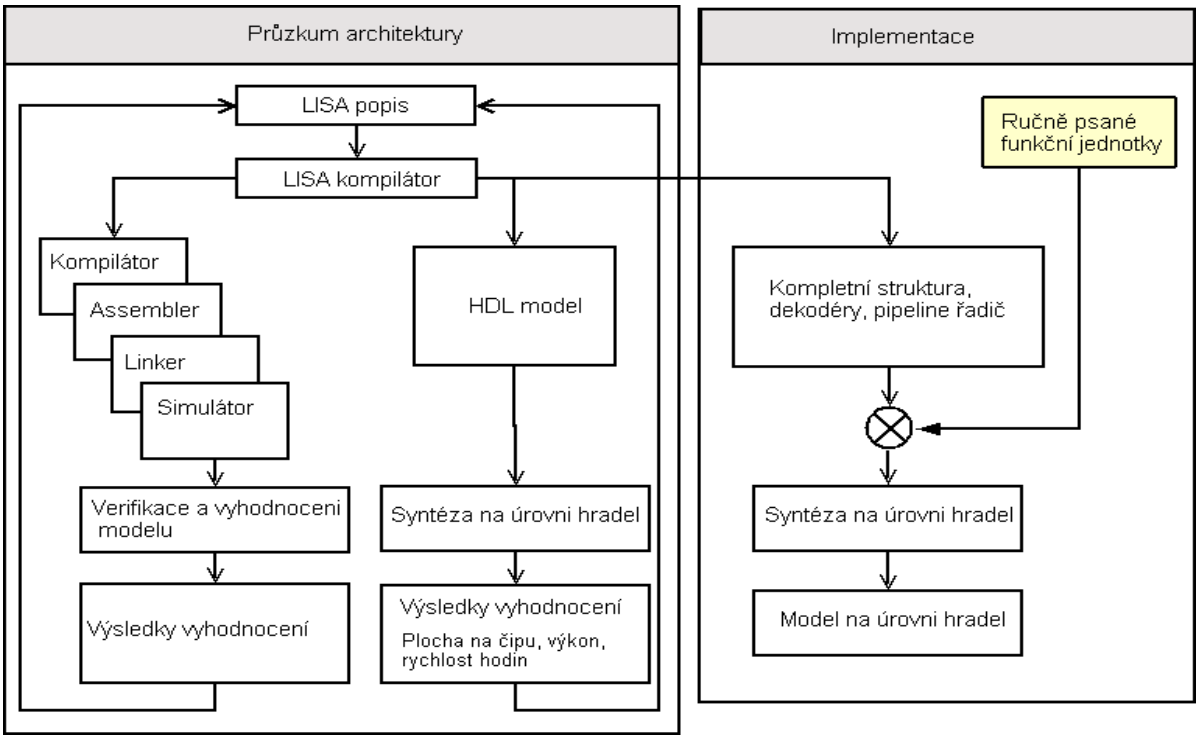
Termínem programová transformace můžeme také rozumět formální popis algoritmu, který implementuje transformaci.

V další části práce budeme termínem transformace rozumět proces programové transformace z jazyka ISAC do jazyka VHDL, která snižuje úroveň abstrakce.

2.4.2 Plně generovaná transformace

Plně generovaná transformace znamená, že syntetizovatelný popis architektury v jazyce VHDL bude generován z popisu v jazyce ISAC bez použití šablon hardwarových entit. Jazyk ISAC je odvozen od jazyka LISA. Lze tedy uvažovat nad řešením transformace obdobně, jako je tomu v jazyce LISA. Tato kapitola popisuje řešení transformace podle [5], [13] a [14].

Na obr. 2.8. je vidět jakým způsobem přistupuje LISA k fázi prozkoumání architektury a k fázi hardwarové implementace. Řešení transformace do jazyka HDL vychází z vlastnosti jazyka, který poskytuje různé modely. K modelům jazyka LISA lze vytvořit odpovídající modely jazyka HDL, z nichž lze provádět syntézu na úrovni hradel. Kompilátor však může z jazyka LISA přímo generovat popis v jazyce HDL, z něhož bude prováděna syntéza na úrovni hradel. Tato transformace však vyžaduje, aby návrhář specifikoval všechny funkční jednotky ručně. Obdobným způsobem lze transformaci provádět i v rámci jazyka ISAC.



Obr. 2.8. Prozkoumání a implementace architektury založená na jazyku LISA

[1] českým ekvivalentem může být zpětné inženýrství

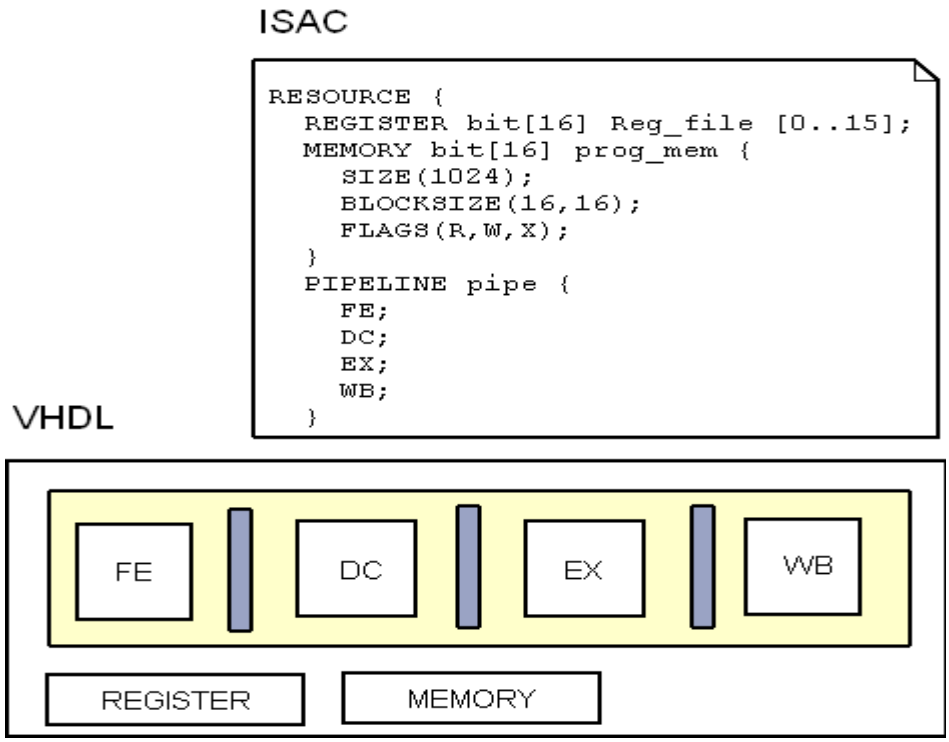
Jazyk ISAC nabízí tři druhy hardwarového popisu. Explicitní, implicitní a neformalizovaný.

2.4.2.1 Implicitní hardwarový popis

Hardwarová sémantika prvků jazyka ISAC není příliš zřetelná, je více obecná a vyžaduje hlubší analýzu. Je zapotřebí přídavných informací, které nejsou v modelu. Například jsou zapotřebí informace z plánovače simulace pro vytvoření RTL dekodéru (ACTIVATION sekce jazyka ISAC).

2.4.2.2 Explicitní hardwarový popis

Sem patří prvky jazyka ISAC s jasnou hardwarovou sémantikou. Korespondující hardware na úrovni RTL může být generován přímo z modelů jazyka ISAC. Obr. 2.9. ukazuje zdrojovou sekci jazyka ISAC a z ní přímo generované struktury architektury v jazyce VHDL.



Obr. 2.9. Explicitní hardwarový popis

2.4.2.3 Neformalizovaný hardwarový popis

Datová cesta architektury je modelována v jazyce ISAC pomocí BEHAVIOR sekce. Tato sekce může být považována za neformalizovaný hardwarový popis.

Obdobně jako jazyk LISA, lze i jazyk ISAC rozdělit na několik modelů, podle typu poskytovaných informací:

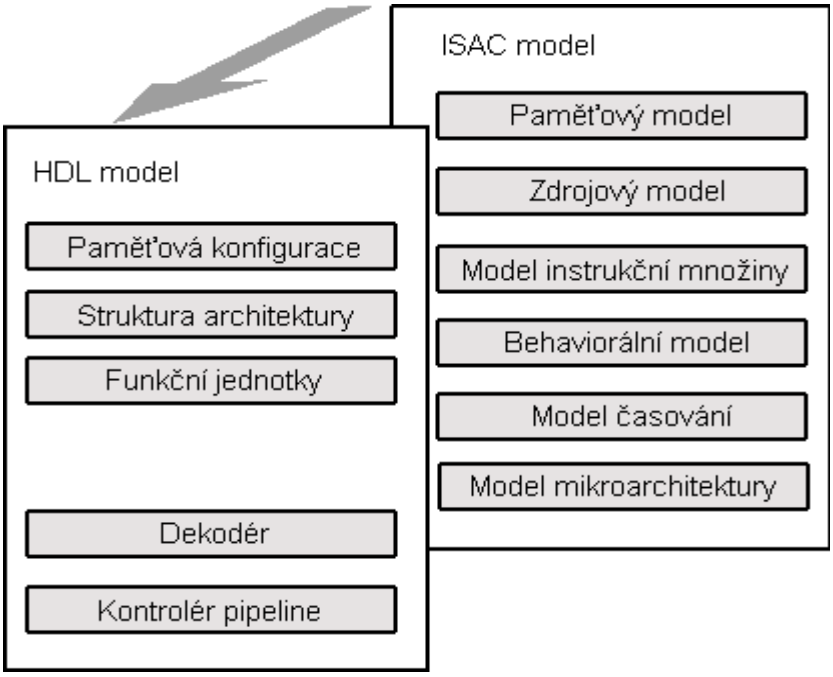
- **Paměťový model** – seznam pamětí a registrů systému s jejich bitovou šířkou, rozsahem a aliasy.

- **Zdrojový model** - dosažitelné hardwarové zdroje – např. funkční jednotky nebo registry používané v operacích.
- **Model instrukční množiny** - hardwarové operace a povolené operandy. Je popsán asemblerovskou syntaxí a kódováním.
- **Behaviorální model** – změna stavu mikroprocesoru je dána operacemi, které popisují aktivity hardwarové struktury.
- **Model časování** – aktivační sekvence hardwarových operací a jednotek.
- **Model mikroarchitektury** - seskupení operací do funkčních jednotek, obsahuje přesnou implementaci některých komponent jako sčítačky, multiplexory, atd.

Kompilátor jazyka ISAC musí získat z těchto modelů další nezbytné informace pro sestavení odpovídajících modelů jazyka HDL (VHDL). Některé modely jazyka HDL lze získat přímo z modelů jazyka ISAC, pro ostatní je nezbytná hlubší analýza. Korespondující modely jazyka HDL lze popsat následovně:

- **Paměťová konfigurace** – lze ji přímo získat z paměťového modelu jazyka ISAC. Sumarizuje registry, paměti a konfiguraci sběrnice.
- **Struktura architektury** – stavy a registry zřetězeného zpracování lze získat ze zdrojového a behaviorálního modelu a modelu mikroarchitektury.
- **Funkční jednotky** – jsou generovány z modelu mikroarchitektury.
- **Dekodér** – generovaný na základě informací (zejména informace o hardwarových operacích a jejich průběhu vykonání) z instrukčního modelu a modelu časování a mikroarchitektury.
- **Kontrolér zřetězeného zpracování (pipeline)** - generovaný na základě informací z instrukčního modelu a modelu časování.

Obr. 2.10. ukazuje modely jazyka ISAC ve srovnání s odpovídajícími modely jazyka HDL.



Obr. 2.10. Modely jazyka ISAC a korespondující modely jazyka HDL

Návrhář má plnou kontrolu nad oběma generovanými modely. Z modelu jazyka HDL lze již generovat odpovídající popis architektury v jazyce VHDL. Avšak v případě optimalizací zaměřených na plochu čipu či frekvenci hodin nezbývá, než aby návrhář přepsal model jazyka HDL do příslušného kódu v jazyce VHDL ručně. Speciálně datová cesta architektury je velmi kritická a v mnoha případech musí být ručně optimalizována. Z těchto důvodů by generovaný kód v jazyce VHDL mohl být limitován pouze na určité části architektury:

- **Hrubá struktura mikroprocesoru** jako jsou registry, zřetězené zpracování, registry zřetězeného zpracování.
- **Instrukční dekodér** nastavující datové a řídící signály, které jsou přenášeny skrze struktury zřetězeného zpracování a aktivují příslušné funkční jednotky.
- **Kontrolér zřetězeného zpracování**, který podporuje přerušení zřetězené linky, vyprazdňování registrů zřetězené linky nebo řešení konfliktů.

Hardwarové operace by mohly být seskupovány do funkčních jednotek. Tyto funkční jednotky by však byly generovány pouze jako prázdné bloky. Tedy obsahovaly by vstupní a výstupní porty pro připojení na další funkční jednotky či registry zřetězeného zpracování, avšak jejich vnitřní popis by se musel doplnit ručně. Nevýhoda ručního přepsání datové cesty popsané v jazyce VHDL spočívá v tom, že chování hardwarové operace uvnitř funkčních jednotek by bylo popsáno dvakrát – jednou v modelu jazyka ISAC a podruhé v modelu jazyka HDL cílové architektury.

Dalším úskalím je behaviorální model, který je v jazyce ISAC popsán pomocí upraveného ANSI-C, jehož sémantika je obtížně převoditelná do jazyka VHDL.

2.4.3 Transformace založené na šablonách

V předchozí kapitole bylo nastíněno řešení, které vychází z jazyka LISA. Jak bylo uvedeno, aplikace tohoto řešení na jazyk ISAC by byla možná. Avšak řešení představené jako plně generovaná transformace z popisu v jazyce ISAC, v sobě skrývá několik úskalí.

Pro jazyk ISAC je charakteristická vysoká úroveň abstrakce. Z pohledu transformace je nutné jej rozšířit o konstrukce blízké HDL jazykům (např. informace o zdrojových a cílových operandech funkční jednotky). Dalším podstatným problémem je duplicita zápisu funkčních jednotek. Jednou na úrovni modelu jazyka ISAC a podruhé na úrovni modelu jazyka HDL. V neposlední řadě lze vidět problém s generováním řadiče zřetězeného zpracování a kontrolní logiky obecně. To platí zejména pro složitější architektury s větším počtem zřetězených linek, zpracováním mimo pořadí, atd.

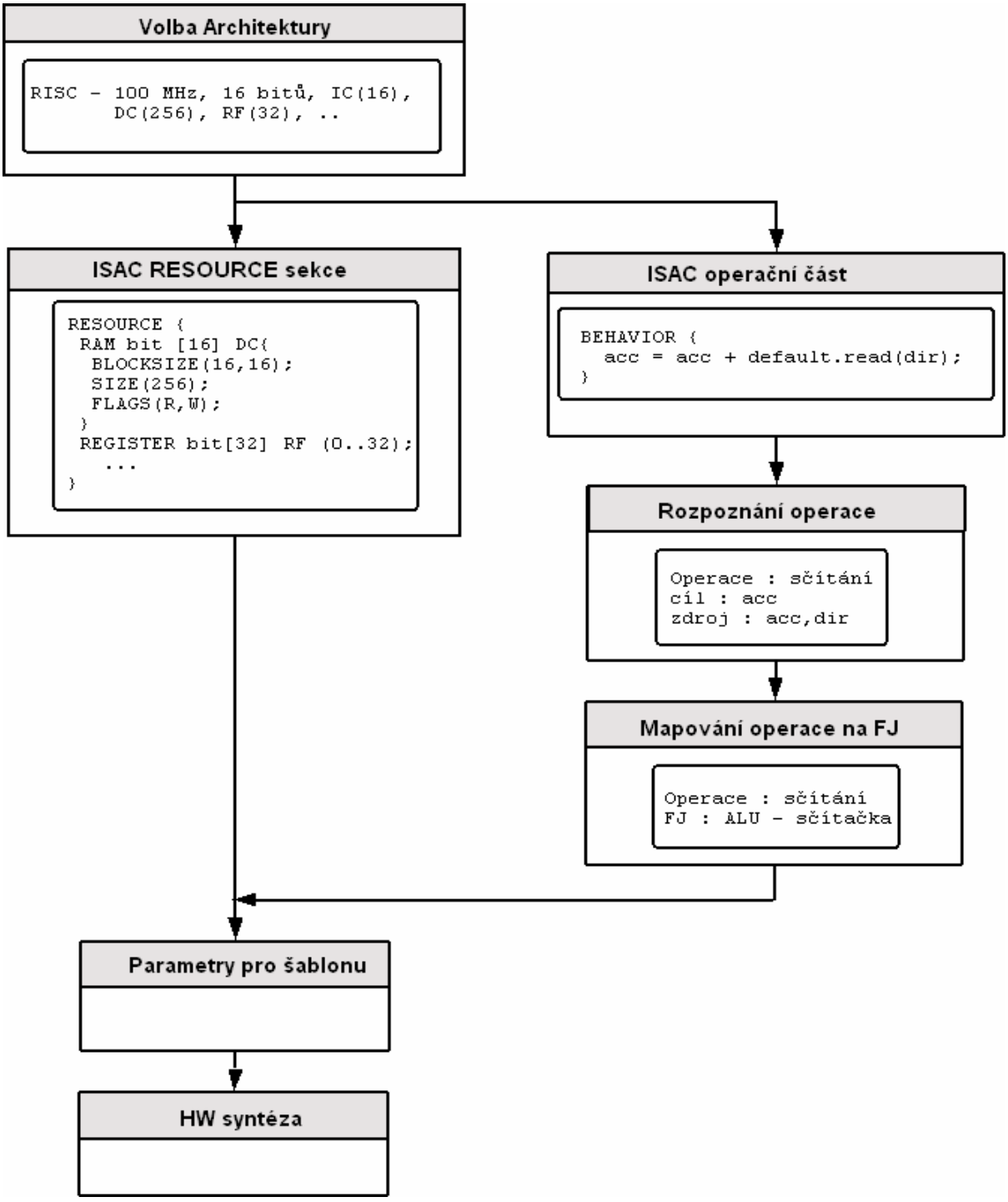
Nyní představím nové řešení *transformace založené na šablonách*. V transformaci založená na šablonách se nevyhneme potřebě rozšířit jazyk ISAC o nové konstrukce. Řešení se však snaží odstranit problém s duplicitním zápisem funkčních jednotek. Problém s generování řadiče a kontrolní logiky obecně zůstává nadále.

Transformaci založenou na šablonách lze provádět dvěma způsoby.

2.4.3.1 Generická šablona architektury

Cílem tohoto řešení je poskytnout návrháři sadu *parametrizovaných šablon* různých architektur mikroprocesorů popsaných v jazyce VHDL. Šablony představují generické mikroprocesory (založené na generickém VHDL). Návrhář tedy bude mít kompletní představu o tom, jak bude vypadat hardwarová realizace architektury (hardwarová syntéza by měla být jednoduší a efektivnější) a jaké bude mít specifikační parametry (frekvence, plocha na čipu, výkonnost). Podle specifikačních parametrů, které bude od řešení vyžadovat, může tedy návrhář zvolit vhodnou šablonu. Šablony budou připraveny pro nejrůznější mikroprocesorové technologie (VLIW, RISC, DSP). Podstatou řešení pomocí šablon generických mikroprocesorů je, že pro konkrétní mikroprocesor bude existovat sada volitelných parametrů, které budou mít vliv na konkrétní podobu cílové architektury. Např. u šablony typu RISC se bude volit instrukční šířka slova, velikost pamětí a registrového pole, endianess, instance zásobníku, atd.

Obr. 2.11 ukazuje, jakým způsobem by transformace založená na generické šabloně architektury mohla probíhat.



Obr. 2.11. Transformace založená na generické šabloně architektury

V první fázi zvolí návrhář architekturu procesoru (RISC – 100MHz, …), na kterou chce svůj návrh mapovat. V další fázi je ze zadaných parametrů sestavena RESOURCE sekce v jazyce ISAC. Tato sekce bude generována automaticky. Šablona mikroprocesoru má již zdrojové části popsané, lze tedy měnit jen jejich počet či velikost. Je nežádoucí zdrojové prvky vytvářet přímo v jazyce ISAC, neboť by nemusely odpovídat prvkům v šabloně a byla by nutná jejich speciální kontrola. Vedle RESOURCE sekce existuje v jazyce ISAC sekce operační. Zde bude mít návrhář svůj vlastní prostor pro inovaci. Zvolí svůj vlastní popis kódování instrukcí a popis chování. Pro popis chování

(BEHAVIOR sekce) je v jazyce ISAC použito upraveného ANSI-C. Toto upravené ANSI-C neobsahuje ukazatele, typy s pohyblivou řádovou čárkou a je syntakticky stejné s tělem funkce či procedury. Z behaviorálního popisu operace bude rozpoznána její sémantika. Tedy o jakou operaci se jedná (sčítání, odčítání, atd.), jaké jsou zdrojové a cílové operandy, zda bude proveden zápis výsledku do paměti či do registrů. Jakmile je operace rozpoznána, je návrháři nabídnuta možnost mapování operace na funkční jednotku (FJ), která je v šabloně pro tuto operaci připravena. Pokud nebude operace rozpoznána nebo nebude pro operaci v šabloně připravena odpovídající funkční jednotka, musí mapování operace provést návrhář ručně. Ruční mapování spočívá v tom, že návrhář bude mít pro šablonu připraveny speciální funkční jednotky popsané v jazyce VHDL. Tyto jednotky zakomponuje do šablony a poté na ně může mapovat operace popsané v jazyce ISAC.

Jak již bylo řečeno, šablona představuje generický mikroprocesor, který má volitelnou sadu parametrů. Parametry jsou sestaveny na základě volby architektury (zdrojová část) a na základě rozpoznání operací. Parametrizovaná šablona je poté připravena pro hardwarovou syntézu.

2.4.3.2 Šablony komponent architektury

Řešení pomocí generické šablony architektury nutí návrháře, aby popis architektury v jazyce ISAC zcela přizpůsobil šabloně. Jinak řečeno, dává mu jen omezenou možnost vnést do návrhu nějakou inovaci a navrhnout libovolnou architekturu. Generická šablona je tak říkajíc „napevno zadrátovaná“. Jednotlivé komponenty architektury jsou pevně propojené a lze měnit jen šířky instrukčního a datového slova, případně velikosti paměťových prvků. Řídící logika však zůstává neovlivnitelná z pohledu jazyka ISAC. To znamená, že popis řízení na úrovni jazyka ISAC by nebyl transformován do popisu v jazyce VHDL. Rovněž analýza jazyka ANSI-C v rámci generické šablony architektury by nepřinesla optimální hardwarové řešení na úrovni jazyka VHDL.

Z těchto důvodů se lépe jeví řešení pomocí šablon komponent architektury. Toto řešení neposkytuje návrháři kompletní popis funkční architektury, ale jen popis jejich komponent (programový čítač, registrové pole, paměťové prvky, registry) v podobě šablon. Každá komponenta je reprezentována jednou entitou jazyka VHDL. Entity samy o sobě představují jen funkční bloky a je nutné je správně propojit v celkovou funkční architekturu.

Cílem řešení je na základě popisu v jazyce ISAC vybrat patřičné entity z šablony. Tyto entity mohou mít generické parametry, které jsou nastaveny hodnotami z popisu v jazyce ISAC. Dále pak vytvořit řídící a dekódovací logiku a propojení komponent v funkční popis architektury v jazyce VHDL.

V této kapitole je uvedena pouze základní myšlenka transformace. Její konkrétní popis je uveden v kapitole 4. Návrh řešení. Neboť myšlenka šablon komponent je použita pro řešení transformace z jazyka ISAC do jazyka VHDL.

3 Cíl práce a metodika

Cílem této diplomové práce je seznámit se s projektem Lissom[1]. S jeho dosavadní činností a s jeho cíli. Cíle projektu již byly zmíněny v úvodu práce. Dosavadní činností byl v rámci projektu navržen nový jazyk ISAC a jeho překladač do jazyka XML[2], z něhož jsou generovány komponenty pro podporu vývoje (asembler, spojovací program, disassembler, ladící nástroj, simulátor). Další výzkum v projektu je nyní podporován grantem MPO ČR FT-TA3/128 – Jazyk a vývojové prostředí pro návrh mikroprocesoru. Nyní tedy probíhá vývoj integrovaného vývojového prostředí, nových verzí asembleru a disasembleru a generátoru popisu procesoru v jazyce VHDL. Tato práce se věnuje jen určité části projektu Lissom, konkrétně části týkající se vývoje generátoru popisu procesoru v jazyce VHDL.

Dílčím cílem práce je seznámení se s různými procesorovými architekturami a metodikou jejich návrhu. A to jak s návrhovými modely, tak s jazyky určenými pro popis architektur, zejména pak s jazyky z rodin LISA a s jazykem VHDL.

Hlavním cílem práce je *navržení a implementace transformace z jazyka ISAC do jazyka VHDL*. Je nutné podotknout, že práce si neklade za cíl kompletní vyřešení této problematiky, neboť takové řešení by zřejmě převyšovalo rozsah diplomové práce. Cílem tedy je navrhnout jedno z možných řešení transformace a implementovat část navrženého řešení. Navržená transformace by měla být v obecné míře použitelná pro různé typy architektur popsaných v jazyce ISAC. Při implementaci je nutné počítat s jejím pozdějším rozšiřováním.

Po stanovení tématu a cíle diplomové práce bylo nutné prostudovat odbornou literaturu vztahující se ke zkoumané oblasti. Na základě studia byla vypracována část nazvaná Přehled o zkoumané problematice. Tato část byla vypracována v rámci semestrálního projektu, na který tato práce navazuje. Na základě výsledků semestrálního projektu je vypracována část nazvaná Návrh řešení.

Při zpracování praktické části je nutné vycházet z již hotových částí projektu Lissom, neboť práce musí být do projektu vhodně začleněna.

Na základě znalostí získaných z první fáze je navržena transformace z jazyka ISAC do jazyka VHDL. Dále je charakterizována implementace generátoru, který provádí zmíněnou transformaci.

[1] Více informací o projektu lze zjistit v [19]
[2] XML (eXtensible Markup Language) je obecný značkovací jazyk, který umožňuje strukturovaně popisovat data. Jazyk je určen např. pro výměnu dat mezi aplikacemi.

4 Návrh řešení

Tato část práce se věnuje řešení navržené transformace z jazyka ISAC do jazyka VHDL pomocí šablon komponent procesorové architektury. Nejprve jsou popsány rozšíření jazyka ISAC o nové konstrukce, poté je popsán návrh transformace, implementace transformace a nakonec je popsáno začlenění práce do projektu Lissom.

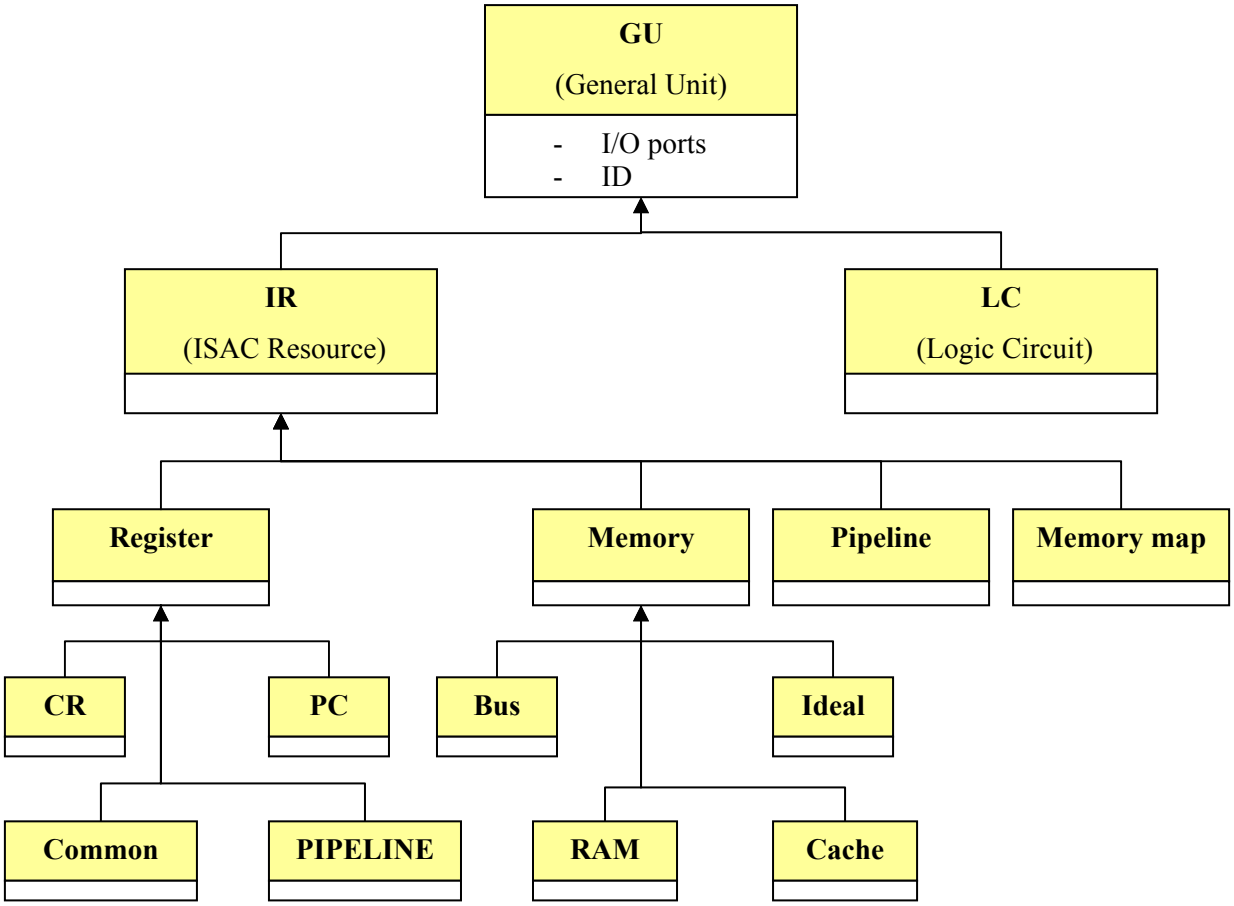
4.1 Rozšíření jazyka ISAC

Transformace do jazyka pro popis hardwaru vyžaduje, aby byl jazyk ISAC rozšířen o nové konstrukce. Bez rozšíření by nebylo možné transformaci provést.

4.1.1 Návrh obecného zdrojového prvku

Před vlastním rozšířením jazyka ISAC o nové konstrukce bylo nutné navrhnout obecný zdrojový prvek procesoru, který bude popsán v jazyce ISAC.
Při návrhu obecného zdrojového prvku jsem vycházel z myšlenky popisu obecného hardwarového prvku v jazyce VHDL, kterým je pravděpodobně entita. Jazyk VHDL pomocí entity umožňuje popsat libovolný prvek z hlediska jeho chování, hardwarové struktury a komunikace s okolím. Chování a hardwarová struktura entity je specifikováno pomocí těla entity (předpokládáme RTL úroveň popisu). Komunikace s okolím je zajištěna pomocí vstupních a výstupních portů.

Nově tedy bude v rámci jazyka ISAC modelován obecný zdrojový prvek, jak je vidět na obr. 4.1. Všechny zdrojové prvky jazyka ISAC jsou modelovány jako obecný prvek (*General Unit - GU*). Tento prvek má stejně jako entita jazyka VHDL definovány vstup/výstupní porty a jedinečný identifikátor. Obecným prvkem může být buď zdrojový prvek jazyka ISAC (ISAC Ressource - IR) nebo libovolný číslicový obvod (*Logic Circuit - LC)*. Prvky IR jsou na úrovni jazyka ISAC rozlišeny pomocí klíčových slov (REGISTER, RAM, atd.). Jako prvek IR lze tedy modelovat registr (*Register*), paměť (*Memory*), zřetězené zpracování (*Pipeline*) a mapování pamětí (*Memory map*). Registry dále můžeme rozdělit na běžný registr (*Common*), programový čítač (*PC*), kontrolní registr (*CR*) a registr zřetězeného zpracování (*Pipeline*). Paměti lze rozdělit na ideální paměť (*Ideal*), sběrnici (*Bus*), paměť typu RAM (*RAM*) a vyrovnávací paměť (*Cache*). Sémantika prvků IR je dána. Jejich chování je známé simulátoru. Sémantika a chování prvků LC známa není. Z tohoto důvodu je v jazyce ISAC zavedena možnost, přiřadit prvku LC funkčnost pomocí operace jazyka ISAC. Operace může být přiřazena prvku LC, čímž je řečeno, že je vykonávána touto hardwarovou entitou.



Obr. 4.1. Návrh obecného zdrojového prvku

4.1.2 Nové konstrukce

Návrh obecného zdrojové prvku nově zavádí prvek LC. Pro tento prvek je tedy nutné navrhnout nové konstrukce v jazyce ISAC. Dále je nutné navrhnout konstrukce, které definují vzájemnou interakci mezi jednotlivými komponentami návrhu. Rovněž je nutné vytvořit konstrukce, které umožní specifikovat šablonu architektury a její jednotlivé prvky.

4.1.2.1 Specifikace konfiguračního souboru hardwarové šablony

Jazyk ISAC byl rozšířen o klíčové slovo **map,** za nímž následuje název konfiguračního souboru hardwarové šablony. Název souboru je umístěn v uvozovkách. Před vlastní deklarací zdrojů (RESOURCE) lze v jazyce ISAC uvést makrogenerátor nebo specifikaci konfiguračního souboru hardwarové šablony. Pokud jsou uvedeny obě varianty, musí být první uvedena specifikace konfiguračního souboru hardwarové šablony.

Podrobný popis syntaxe viz. příloha č.1.

Příklad:

```
map "cnf_RISC.xml";
```

```
  RESOURCE {
    PC REGISTER bit [8] pc;
}
```

Definován je konfigurační soubor *cnf_RISC.xml*. Šablona procesoru, k níž konfigurační soubor patří, bude použita pro generování popisu v jazyce VHDL.

4.1.2.2 Deklarace prvku LC

Jazyk ISAC byl rozšířen o prvek LC reprezentující libovolný číslicový obvod. Prvky LC jsou deklarovány pomocí klíčového slova **LC.** Prvkům LC mohou být přiřazeny operace pomocí operační **IN** sekce. LC prvky jsou řazeny mezi prvky popisující hardwarovou strukturu mikroprocesoru a jsou tedy deklarovány v rámci RESOURCE sekce.

Podrobný popis syntaxe viz. příloha č.1.

Příklad:

```
RESOURCE {
  REGISTER bit[8] reg;
  LC       alu;
  }
```

Definován je jeden registrový prvek *reg* a LC prvek *alu.*

4.1.2.3 IN sekce operací

Každá operace jazyka ISAC má možnost volitelně definovat v rámci své IN sekce, v kterém prvku LC bude vykonávána nebo ve kterém stavu pipeline bude provedena. Přiřazení operací ke stavům pipeline nebo do prvku LC je provedeno v záhlaví odpovídající operace za klíčovým slovem **IN.**

Podrobný popis syntaxe viz. příloha č.1.

Příklad:

```
RESOURCE {
  PIPELINE pipe = { FE: ;DC: ;EX: ; WB: ; };
  LC       alu;
  }

OPERATION decode IN pipe.DC { … }
OPERATION add IN alu { … }
```

V pipeline *pipe* je operace *decode* přiřazena stavu *DC*. Operace *add* je přiřazena LC prvku *alu.*

4.1.2.4 Hlavička operačních sekcí

Hlavička je u operací zavedena pro definování interakce prvku s okolím. Je využívána jen u sekce BEHAVIOR, do budoucna je možné její využití i u jiných operačních sekcí. V rámci sekce BEHAVIOR je možné pracovat s různými zdrojovými prvky jazyka ISAC, kromě prvku LC. Vzhledem k tomu, že pracujeme se zdrojovými prvky jako s proměnnými jazyka C (nespecifikujeme

přes které vstup/výstupní porty jednotka komunikuje s okolím) je nutné toto explicitně specifikovat. Snahou je správně mapovat jednotlivé vstup/výstupní porty jednotek, konkrétně porty datové.

Pro zdrojové prvky, s nimiž pracujeme v rámci sekce BEHAVIOR, přidáváme do hlavičky sekce mapování na *místní porty*. Místními porty rozumíme porty prvku LC, ke kterému je operace přiřazena. Jednotlivé porty identifikujeme typem. První pár hranatých závorek s klíčovým slovem **IN** definuje mapování místních vstupních portů, druhý pár s klíčovým slovem **OUT** definuje mapování místních výstupních portů. Rovněž je definováno pořadí portů v rámci jednoho typu.

Podrobný popis syntaxe viz. příloha č.1.

Příklad:

```
RESOURCE  {
   REGISTER bit[8] rf [0..63];
   REGISTER bit[8] src,dst;
   LC alu;
}
OPERATION ADD IN alu (IN [rf,rf] OUT [rf]){
   BEHAVIOR {
      rf[dst]=rf[src]+rf[dst]
   };
   }
```

V rámci hlavičky sekce BEHAVIOR je 1. výstupní port prvku *rf* mapován na 1. vstupní port prvku *alu*. 2. výstupní port prvku *rf* je mapován na 2. vstupní port prvku *alu*. 1. výstupní port prvku *alu* je mapován na 1. vstupní port prvku *rf*.

4.1.2.5 Identifikace zdrojového prvku v konfiguračním souboru

Každý prvek z RESOURCE sekce popisující hardwarovou strukturu mikroprocesoru může volitelně obsahovat svoji identifikaci v hardwarové šabloně. Pokud tuto identifikaci obsahuje, bude mapován na odpovídající entitu hardwarového jazyka VHDL.

Podrobný popis syntaxe viz. příloha č.1

Príklad:

```
RESOURCE {
   PC REGISTER (cnf_pc)  bit[32]pc;
   REGISTER  (cnf_areg)  bit[8] areg [0..15];
   LC      (cnf_alu)     alu;
   }
```

Programový čítač *pc* je mapován na hardwarovou entitu identifikovanou v konfiguračním souboru pomocí identifikátoru *cnf_pc,* register *areg* je mapován na hardwarovou entitu identifikovanou v konfiguračním souboru pomocí identifikátoru *cnf_areg,* LC prvek *alu* je mapován na hardwarovou entitu identifikovanou v konfiguračním souboru pomocí identifikátoru *cnf_alu.*

4.1.3 Vnitřní model jazyka ISAC

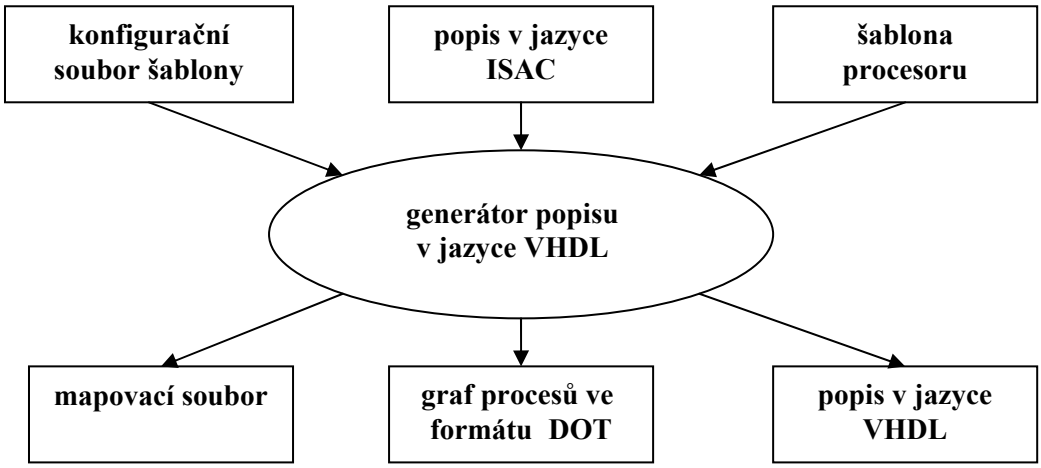
Díky nově navrženým konstrukcím v jazyce ISAC bylo nutné tyto konstrukce rovněž přidat i do vnitřního modelu jazyka ISAC, neboť z vnitřního modelu generuje překladač jazyka ISAC kód

v jazyce XML, který je základem pro řešenou transformaci. Rozšíření vnitřního modelu jsou uvedená v příloze č. 3

4.2 Návrh transformace

V kapitole 2.4.3.2 byla uvedena základní myšlenka transformace za pomoci šablon komponent architektury. Šablony jsou psány na strukturální RTL úrovni. Předpokládá se tedy, že šablonu tvoří komponenty, které mají být vzájemně propojené v jedné hlavní entitě. Hlavní entita představuje popisovanou architekturu procesoru. V této kapitole bude podrobněji diskutován proces transformace z jazyka ISAC do jazyka VHDL na základě šablon komponent architektury.

Na obrázku 4.2 je vidět základní schéma transformace. Program, jenž provádí celou transformaci nazýváme generátor popisu v jazyce VHDL a pod tímto názvem bude používán dále v této práci. Program nese označení *Generátor popisu v jazyce VHDL*, neboť jeho hlavním výstupem je generovaný popis architektury procesoru v jazyce VHDL. Jeho vstupem jsou tři typy souborů, které jsou podrobněji popsány v následujících podkapitolách. Generátor ze vstupů vytvoří popis procesoru v jazyce VHDL odpovídající popisu v jazyce ISAC. Rovněž generuje mapovací soubor, který uchovává informace o procesu mapování a soubor ve formátu DOT, který popisuje tvořený graf procesů.



Obr. 4.2. Základní schéma transformace

Proces transformace lze rozdělit do dvou fází:

- **Fáze mapování** – v této fázi budou jednotlivé zdrojové prvky jazyka ISAC (včetně prvků LC) mapovány na patřičné entity šablony popsané v jazyce VHDL. Tato fáze je zaměřena na transformaci RESOURCE sekce jazyka ISAC.

- **Fáze generování** – v této fázi dochází k generování nových entit, které tvoří řídící logiku transformované architektury (řadič, dekodér). Tato fáze je zaměřena na transformaci operační části jazyka ISAC

Obě fáze budou podrobněji popsány v kapitole Implementace transformace.

4.2.1 Vstup transformace

V této kapitole jsou podrobněji popsány vstupy transformace, kterými jsou konfigurační soubor šablony, šablona popsaná v jazyce VHDL a vnitřní model jazyka ISAC.

4.2.1.1 Konfigurační soubor šablony

Je třeba, aby pro každou šablonu procesoru existoval konfigurační soubor, který popisuje všechny její komponenty, tedy jednotlivé entity šablony popsané v jazyce VHDL. Konfigurační soubor zprostředkovává propojení mezi šablonou procesoru popsanou v jazyce VHDL a popisem procesoru v jazyce ISAC. Za jeho správné vytvoření zodpovídá návrhář šablony.

Konfigurační soubor je tvořen v jazyce XML. Jeho úplná specifikace je uvedena v příloze č.2. Soubor obsahuje identifikaci šablony. Tato identifikace musí být rovněž uvedena v jazyce ISAC, pokud má být provedeno mapovaní na šablonu popsanou v jazyce VHDL. Konfigurační soubor obsahuje popis jednotlivých entit z šablony procesoru. Entita je buď typu IR nebo LC. Každá entita má definovaný počet vstup/výstupních portů, identifikátor z jazyka ISAC, pomocí něhož je entita svázána se zdrojovým prvkem jazyka ISAC. Pokud je v jazyce ISAC tento identifikátor uveden u zdrojového prvku, provede se mapování tohoto prvku na příslušnou entitu šablony. Konfigurační entita dále obsahuje informace o popise v jazyce VHDL. Konkrétně tedy název entity jazyka VHDL, název souboru, ve kterém je entita umístěna, a nakonec nepovinný seznam generických parametrů, které mají být nastaveny.

Příklad:

Popis v jazyce ISAC:

```
map "RISC.xml";
RESOURCE {
  REGISTER (cnf_reg) bit [8] register;
}
```

Odpovídající část konfiguračního souboru v jazyce XML:

```
<TEMPLATE>
  <ID>RISC</ID>
  <IR>
    <ID>cnf_reg</ID>
    <PORT_CNT>
      <IN>3</IN>
      <OUT>2</OUT>
      <INOUT>0</INOUT>
    </PORT_CNT>
    <VHDL>
```

```
        <ENTITY>Register</ENTITY>
        <FILE>register.vhd</FILE>
        <GENERIC_LIST>
          <ITEM>
            <ID>width</ID>
            <IMELEMENT>SIZE_OF_BITS</IMELEMENT>
          </ITEM>
        </GENERIC_LIST>
      </VHDL>
    </IR>
  </TEMPLATE>
```

Jedná se o šablonu procesoru *RISC,* která pro demonstraci obsahuje pouze jednu entitu prvku IR, tedy zdrojového prvku jazyka ISAC. Entita registru obsahuje 3 vstupní porty a dva výstupní porty. Identifikátor *cnf_reg*, je uveden v jazyce ISAC před zdrojovým prvkem, který má být mapován na tuto entitu. Entita se nachází v souboru *register.vhd* a má jeden generický parametr *width*, který bude doplněn bitovou šířkou (SIZE_OF_BITS) zdrojového prvku jazyka ISAC.

4.2.1.2 Šablona popsaná v jazyce VHDL

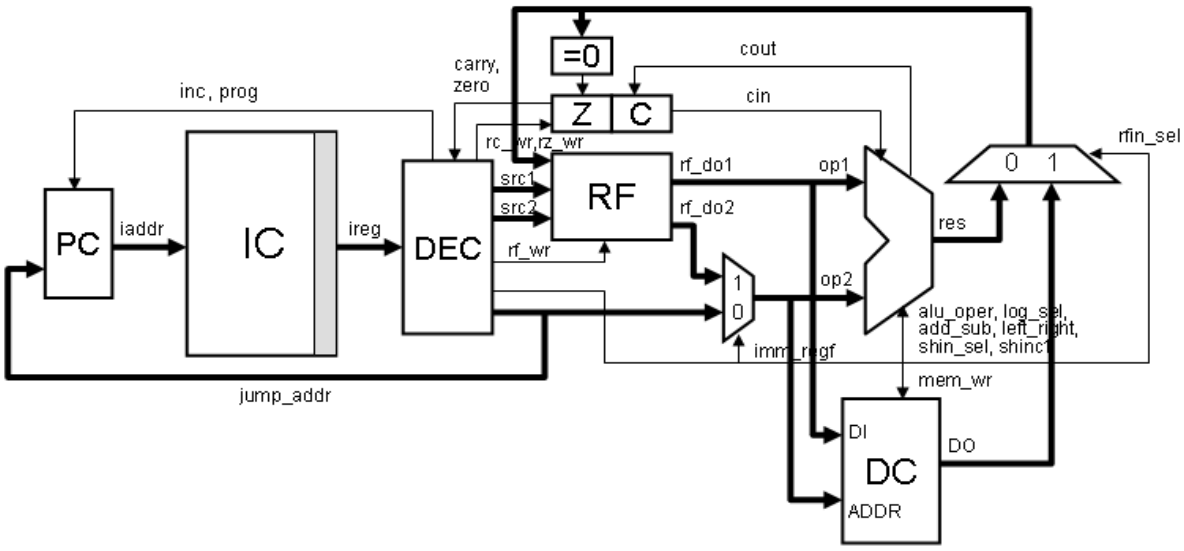
Šablona procesoru popsaná v jazyce VHDL je tvořena sadou připravených komponent popsaných v jazyce VHDL na strukturní RTL úrovni.

Zásady pro tvorbu šablony popsané v jazyce VHDL

- Každá entita jazyka VHDL je umístěna v samostatném souboru.
- Celá šablona je psána strukturálně na RTL úrovni. To znamená, že v rámci jedné entity, která představuje celý procesor, lze provést mapování (propojení) jednotlivých komponent, čímž vznikne funkční popis procesoru.
- Zdrojové soubory jedné šablony jsou umístěny v rámci jednoho adresáře ve vývojovém prostředí.
- Ke každé šabloně existuje jeden konfigurační soubor.
- Entity jazyka VHDL v rámci šablony mohou obsahovat generické parametry.

Ukázka šablony popsané v jazyce VHDL

Na obrázku 4.3 je vidět návrh jednoduchého RISC procesoru popsaného na úrovni RTL (obrázek byl převzat z [11]).



Obr. 4.3. Návrh jednoduchého RISC procesoru na úrovni RTL

Pro tuto architekturu jsem vytvořil šablonu popsanou v jazyce VHDL, jejíž části se objevují v příkladech uváděných v dalších kapitolách této práce. Šablona byla rovněž použita pro testování generátoru popisu v jazyce VHDL.

Šablonu tvoří komponenty uvedené v tabulce 4.1:

Komponenta	Entita	Název souboru

PC	`entity pc is`
	` generic (`
	`  width : integer := ;`
	` );`
	` port(`
	`  INC  : in std_logic;`
	`  PROG  : in std_logic;`
	`  J_ADDR : in std_logic_vector(width-1 downto 0);`
	`  CLK  : in  std_logic;`
	`  RESET : in  std_logic;`
	`  IADDR : out std_logic_vector(width-1 downto 0)`
	` );`
	`end pc;`
		pc.vhd

RF	`entity reg_file is`
	` generic (`
	`  width : integer := ;`
	` );`
	` port(`
	`  CLK  : in  std_logic;`
	`  WRA  : in  std_logic;`
	`  ADDRA : in  std_logic_vector(3 downto 0);`
	`  DIA  : in  std_logic_vector(width-1 downto 0);`
	`  DOA  : out std_logic_vector(width-1 downto 0);`
	`  ADDRB : in  std_logic_vector(3 downto 0);`
	`  DOB : out std_logic_vector(width-1 downto 0));`
	`end reg_file;`
		rf.vhd

ALU	`entity alu is`
	` port(`
	`  A  : in std_logic_vector(7 downto 0);`
	`  B  : in std_logic_vector(7 downto 0);`
		alu.vhd

Komponenta	Entita	Název souboru
	CIN : in std_logic; Q : out std_logic_vector(7 downto 0); COUT : out std_logic; OPER : in std_logic_vector(1 downto 0); INC_CARRY : in std_logic; ADD_SUB : in std_logic; LOG_OPER : in std_logic_vector(1 downto 0); LEFT_RIGHT : in std_logic; SHINC : in std_logic; SHIFT_SEL : in std_logic_vector(1 downto 0)); end alu;	
Z/C	entity reg is generic (width : integer := ;); port (CLK : in std_logic; RESET : in std_logic; DIN : in std_logic_vector(width-1 downto 0); REG_WE : in std_logic; DOUT : std_logic_vector(width-1 downto 0)); end reg;	reg.vhd
DC	entity dc_mem is generic (width : integer := ;); port(CLK : in std_logic; WR : in std_logic; ADDR : in std_logic_vector(5 downto 0); DI : in std_logic_vector(width-1 downto 0); DO : out std_logic_vector(width-1 downto 0)); end dc_mem;	dc.vhd
IC	entity ic is generic (width : integer := ;); port(ADDR : in std_logic_vector(7 downto 0); DO : out std_logic_vector(width-1 downto 0) CLK : in std_logic); end ic;	ic.vhd

Tabulka 4.1 Popis entit šablony procesoru

Tabulka ukazuje, jakým způsobem budou vytvořeny šablony komponent procesoru. Všechny komponenty je možné vzájemně mapovat, doplnit některé multiplexory realizující řízení a dostaneme tak funkční popis jednoduchého procesoru. Komponenta nazvaná DEC představuje dekodér, který má v rámci této architektury plnit rovněž funkci řadiče. Řadič a dekodér bude obecně pro každou architekturu generován z popisu v jazyce ISAC. Pro tyto komponenty tedy nebude připravena šablona.

Šablona je parametrizovaná. Jednotlivé entity obsahují generické parametry, jak je vidět v tabulce 4.1 (příkaz **generic**). Tyto parametry budou doplněny hodnotou z vnitřního modelu jazyka ISAC. Jaká hodnota má být doplněna je uvedeno v konfiguračním souboru v elementu

<GENERIC_LIST>. Při psaní generických parametrů do šablony v jazyce VHDL je třeba dodržet tuto syntaxi:

```
"generic" "("
   generic_name ":" type ":=" ";"
   { generic_name ":" type ":=" ";" }
 ")"
```

- generic_name – jméno generického identifikátoru (stejné jméno uvedeno i v konfiguračním souboru)
- type – typ generického identifikátoru (např. integer)

4.2.1.3 Popis v jazyce ISAC

Popis architektury v jazyce ISAC je překladačem jazyka ISAC transformován na kód v jazyce XML. Tento kód strukturovaně uchovává veškerá data z popisu v jazyce ISAC. Předchozí vstupy slouží především pro kontrolu procesu transformace nebo tvoří základ generovaného popisu v jazyce VHDL. Nad vstupním kódem v jazyce XML obsahujícím popis v jazyce ISAC je prováděn vlastní proces transformace.

4.2.2 Výstup transformace

V této kapitole jsou podrobněji popsány výstupy transformace, kterými jsou mapovací soubor, graf procesů ve formátu DOT a popis procesoru v jazyce VHDL.

4.2.2.1 Mapovací soubor

Tento soubor je výsledkem procesu transformace. Uchovává informaci o procesu mapování na šablonu procesoru popsanou v jazyce VHDL. Konkrétně je zde obsažena informace o tom, které entity z šablony byly mapovány a které byly použity pro vygenerování výsledného kódu v jazyce VHDL. Rovněž je zde uchována informace o nastavených generických hodnotách. Tento soubor může být využit v rámci zpětné transformace nebo pro kontrolu případných změn provedených v rámci vygenerovaného kódu v jazyce VHDL. Mapovací soubor je generován v jazyce XML.

Příklad - mapovací soubor pro prvek IR

Popis v jazyce ISAC:

```
RESORCE {
   MEMORY (memory) bit[8] mem{
      SIZE (256);
      BLOCKSIZE (8,8);
      SIZE (R,W);
   };
}
```

Možný výstupní mapovacího souboru v jazyce XML:

```
<MAPPING>
   <IR_NODE>
```

```
      <ISAC_ID>mem</ISAC_ID>
      <ENTITY>memory8</ENTITY>
      <GENERIC_LIST>
        <ITEM>
          <ID>width</ID>
          <VALUE>8</VALUE>
        </ITEM>
        <ITEM>
          <ID>adress_width</ID>
          <VALUE>256</VALUE>
        </ITEM>
      </GENERIC_LIST>
    </IR_NODE>
  </MAPPING>
```

Zdrojový prvek *mem* je transformován na entitu s názvem *memory8*. U entity byly nastaveny dva generické parametry *width* a *adress_width* na hodnoty 8 a 256. Nastavení generických parametrů závisí na konfiguračním souboru (entita v konfiguračním souboru musí vyžadovat nastavení generických parametrů).

Příklad - mapovací soubor pro prvek LC

Popis v jazyce ISAC:

```
RESOURCE {
  LC (ALU) alu;
  MEMORY bit [8] mem{...};
  REGISTER bit [8] R[0..255];
  REGISTER bit [8] DST_REG;
}
OPERATION op_A IN alu {
  BEHAVIOR ( [ R, mem ] [ R ] ){
    R[DST_REG] = R[mem] + R[DST_REG];
 };
};
```

Možný výstupní mapovacího souboru v jazyce XML:

```
<MAPPING>
  <LC_NODE>
    <ISAC_ID>alu</ISAC_ID>
    <ENTITY>ALU</ENTITY>
    <OPERATION_LIST>
      <ITEM>op_A</ITEM>
    </OPERATION_LIST>
  </LC_NODE>
</MAPPING>
```

Zdrojový prvek *alu* je transformován na entitu s názvem *ALU*. Prvku LC byla přiřazena operace *op_A*.

4.2.2.2 Graf procesů v DOT formátu

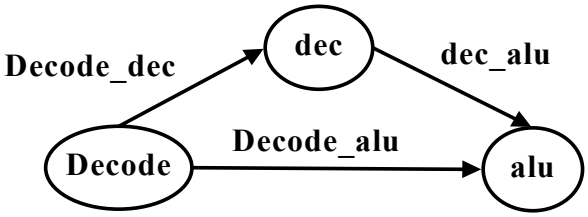
DOT je textový formát pro popis grafu. Existuje mnoho nástrojů, které umí tento popis převést do grafické podoby. Tento fakt není stěžejním důvodem toho, proč byl tento formát vybrán. Formát je poměrně jednoduchý a lze nad ním provést kontrolu ekvivalence dvou grafů. Tato kontrola může probíhat během zpětné transformace z jazyka VHDL do jazyka ISAC.

Příklad

Popis grafu v DOT formátu:

```
digraph "risc-part-example" {
   Decode -> dec [ label = "Decode_dec" ];
   dec -> alu [ label = "dec_alu" ];
   Decode -> alu [ label = "Decode_alu"];
}
```

Graf generovaný z popisu je na obrázku 4.4.

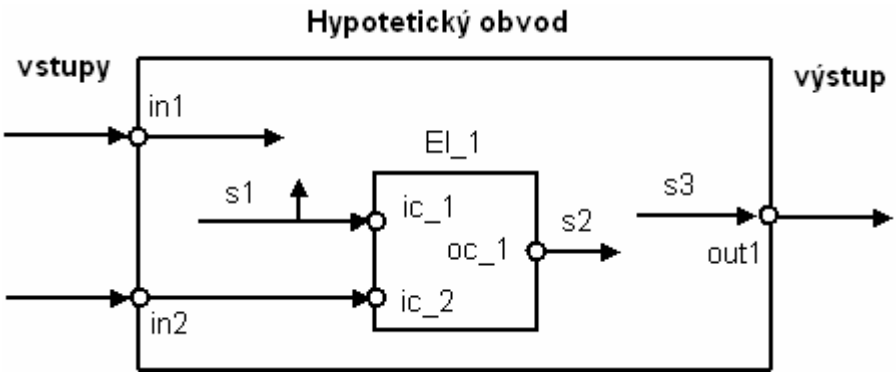


Obr. 4.4. Graf z formátu DOT

4.2.2.3 Popis v jazyce VHDL

Hlavním výstupem transformace je popis v jazyce VHDL. Tento popis by měl být sémanticky ekvivalentní s popisem v jazyce ISAC. Navrhované řešení transformace však nevytvoří plně ekvivalentní popis.

Cílem je, aby generátor vytvořil popis procesoru v jazyce VHDL na úrovni RTL. Takový popis se skládá ze tří částí. První část obsahuje deklaraci knihoven, druhá část popisuje vstup/výstupní porty modelovaného procesoru a třetí část popisuje na strukturální úrovni chování procesoru (architekturu procesoru). Třetí část lze dále rozdělit na další tří části. První z nich obsahuje porty všech komponent procesoru (typy všech elementů procesoru), druhá část obsahuje vnitřní propojovací signály a v třetí části jsou popsány všechny elementy procesoru (jako element je chápána instance komponenty). Přičemž platí, že první část z popisu architektury procesoru nemusí být uvedena, pokud budou komponenty umístěny v rámci jedné knihovny. Knihovnu je poté třeba specifikovat v konstrukcích *port map*. Na obrázku 4.5 je vidět hypotetický logický obvod, k němuž je dále uveden zdrojový kód jazyka VHDL na úrovni RTL.



Obr. 4.5. Hypotetický logický obvod

Ukázka popisu v jazyce VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
•
•
entity Obvod is
port (in1 : in std_logic_vector(3 downto 0);
      in2 : in bit;
      out1 : out bit
);
end Obvod;
architecture structural of Obvod is
  component Comp_1
    port (ic_1, ic_2 : in bit;
          oc_1 : out bit);
  end component;
•
•
  signal s1, s2, s3, ... : bit; ...
begin
  El_1 : Komp_1
  port map (s1, in2, s3);
•
•
end structural;
```

Hypotetický obvod má 2 vstupy (porty *in1* a *in2*) a 1 výstup (port *out1*). V příkladu je ukázán pouze jeden element *El_1* komponenty C*omp_1*. Tento element má 2 vstupy (porty *ic_1* a *ic_2*) a 1 výstup (port *oc_1*). Naznačeny jsou propojovací signály *s1*, *s2* a *s3*, které by v případě kompletního popisu byly přivedeny k dalším elementům.

Generovaný popis tedy bude rozdělen do několika souborů generovaných do jednoho výstupního adresáře. Každý soubor bude popisovat jednu RTL komponentu architektury procesoru. Bude tedy obsahovat jednu entitu jazyka VHDL a k ní příslušnou architekturu jazyka VHDL. Soubor bude vždy nazván podle jména entity. Architektury některých entit budou vyžadovat doplnění kódu návrhářem. Předpokladem je, že komponenty budou součástí jedné knihovny.

Současně bude rovněž vytvořen soubor obsahující hlavní entitu jazyka VHDL, která popisuje celý modelovaný procesor. Tato entita bude v popisu architektury obsahovat sadu propojovacích signálu s patřičnými komentáři. Dále budou připraveny konstrukce *port map,* které však budou muset být doplněny návrhářem. V těchto konstrukcích však bude návrhář používat dříve popsané signály.

Aby byl kód v jazyce VHDL syntetizovatelný, musí být návrhářem doplněn (konstrukce *port map*). Tedy z vygenerovaného popisu v jazyce VHDL nelze přímo provést syntézu.

4.3 Implementace transformace

Implementace generátoru popisu v jazyce VHDL proběhla v jazyce C++ s využitím standardní knihovny šablon STL (Standard Template Library). Implementace využívá graf procesů. První část transformace spočívá ve vytvoření grafu procesů z popisu v jazyce ISAC. Vytvořený graf je třeba

reprezentovat ve formě konečného automatu. Generování popisu v jazyce VHDL probíhá ve dvou fázích, ve fázi mapování a generování.

4.3.1 Graf procesů

Graf procesů tvoří vnitřní model transformace. Z tohoto vnitřního modelu je generován výstupní popis v jazyce VHDL. Graf je využit jak pro fázi mapování, tak pro fázi generování.

4.3.1.1 Vytvoření grafu procesů

Z popisu v jazyce ISAC je nejdříve vytvořen graf procesů v následujících krocích:

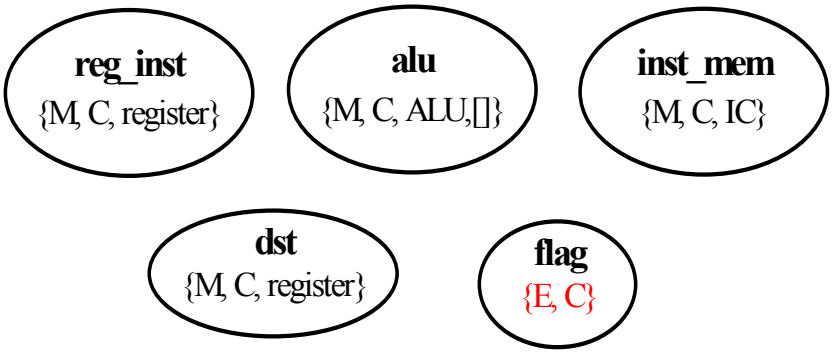
1. **Uzly zdrojových prvků**
   - Pro každý zdrojový prvek jazyka ISAC (tedy prvky IR a LC) je vytvořen uzel grafu N. Název uzlu je vytvořen z identifikátoru zdrojového prvku jazyka ISAC.
   - Podle existence identifikátoru konfigurace u zdrojového prvku označíme každý uzel značkou:
     - **M** – reprezentuje zdrojové prvky, které mají uveden konfigurační identifikátor. Pro tyto prky bude provedeno mapování na entitu jazyka VHDL označenou v konfiguračním souboru pomocí zmíněného konfiguračního identifikátoru.
     - **E** – reprezentuje prvek, který nemá uveden konfigurační identifikátor.
   - Každý uzel rovněž obsahuje identifikátor konfigurace pokud je uveden u odpovídajícího zdrojového prvku.
   - Každý uzel rovněž obsahuje informace nutné k nastavení případných generických parametrů šablony popsané v jazyce VHDL. Uzly tedy obsahují bitovou šířku zdrojového prvku, horní a dolní index, u paměťových prvků počet bloků a podbloků a další parametry z vnitřního modelu jazyka ISAC.
   - Každý uzel vytvořený pro prvek LC navíc obsahuje pole pro uchování identifikátorů přiřazených operací.

Příklad

Popis zdrojových prvků v jazyce ISAC:

```
RESOURCE {
  REGISTER (IC) bit[8] inst_mem [0..15];
  REGISTER (register) bit[16] reg_inst;
  REGISTER (register) bit[8] dst;
  REGISTER bit[2] flag;
  LC (ALU) alu;
}
```

Uzly grafu procesů tvořené z popisu v jazyce ISAC jsou znázorněny na obr. 4.6.



Obr. 4.6. Uzly grafu procesů odpovídající zdrojovým prvkům

Pro zdrojové prvky *inst_mem, reg_inst, dst, alu* jsou vytvořeny uzly se značkou M a implicitní typovou značkou C. Každý uzel obsahuje konfigurační identifikátor. Uzel zdrojového prvku *alu* navíc obsahuje pole, do kterého mohou být v průběhu tvorby grafu přidány identifikátory operací. Pro zdrojový prvek *flag* je vytvořen chybový uzel se značkou E.

2. Uzly operací

Pro každou operaci:

- Nemá-li operace IN sekci, je pro ni vytvořen uzel N, který je pojmenován identifikátorem operace z jazyka ISAC. Uzel je však tvořen pouze pro operace obsahující sekci ACTIVATION. Uzel je označen značkou G, která reprezentuje generovanou entitu jazyka VHDL. Nově vytvořenému uzlu je přiřazen identifikátor operace z jazyka ISAC, který je umístěn do pole identifikátorů operací.
- Má-li operace IN sekci, pomocí niž je přiřazena prvku LC, je identifikátor operace přiřazen uzlu, který reprezentuje prvek LC.
- Všechny uzly (uzly operací i zdrojových prvků) jsou označeny typovou značkou:
  - **C** – je implicitní hodnotou pro všechny vytvářené uzly. Může být změněna na značku S.
  - **S** – reprezentuje synchronní logický prvek. Uzel obsahující operaci *main* je označen značkou S.

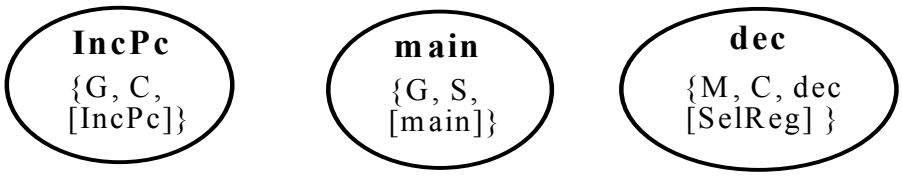
Příklad

Popis operací v jazyce ISAC:

```
RESOURCE {
   LC (dec) dec;
}
OPERATION IncPc{
   BEHAVIOR{
      pc+=1;
   };
}
OPERATION Fetch
{
   INSTANCE IncPc ALIAS { inc };
   ACTIVATION {
```

```
      inc;
   };
}
OPERATION SelReg IN dec
{
   ASSEMBLER { val=#U };
   CODING { val=0bx[8] };
   EXPRESSION { val; };
}
OPERATION main
{
   INSTANCE Fetch ALIAS { fetch };
   ACTIVATION {
      fetch;
   };
}
```

Uzly grafu procesů tvořené z popisu v jazyce ISAC jsou znázorněny na obr. 4.7.



Obr. 4.7. Uzly grafu procesů odpovídající operacím

Pro operaci *IncPc,* je vytvořen uzel se značkou G a implicitní typovou značkou C. Pro operaci *main* je vytvořen uzel se značkou G a typovou značkou S. Oběma uzlům je přiřazen identifikátor operace. Pro operaci *SelReg* není vytvořen uzel, pouze je identifikátor operace přiřazen do uzlu *dec* reprezentující zdrojový prvek.

3. Vytváření řadiče a řídících cest

Předchozí dva kroky tvorby grafu procesů zajišťovaly pouze vytvoření uzlů grafu. V dalších krocích jsou kromě uzlů tvořeny i hrany mezi uzly grafu.

Informace pro tvorbu řadiče a řídících cest jsou uvedeny v aktivační sekci jazyka ISAC. Pro každou operaci obsahující aktivační sekci[1] jsou provedeny následující změny v grafu procesů:

- Pokud sekce obsahuje aktivace instance jiných (aktivovaných) operací, je pro každou aktivovanou operaci vytvořen uzel N, pokud aktivovaná operace již není přiřazena některému existujícímu uzlu grafu. Aktivovaná operace je přiřazena novému uzlu.
- Pokud sekce obsahuje aktivace instance aktivovaných operací, popřípadě podmíněné aktivování, je vytvořena orientovaná hrana grafu. Hrana je vytvořena z uzlu, který obsahuje aktivační operaci (aktuálně zpracovávaná operace s aktivační sekcí) do uzlu obsahujícího aktivovanou operaci (operace jejíž instance je uvedena v aktivační sekci).

[1] Analyzují se pouze operace se sekcí ACTIVATION

- V případě podmíněné aktivace je vytvořena orientovaná hrana pro instanci operace či instanci zdrojového prvku uvedeného v podmínce. Tedy hrana je vytvořena buď z uzlu, který obsahuje operaci jejíž instance je uvedena v podmínce, nebo z uzlu, který reprezentuje zdrojový prvek v podmínce. Hrana vede k uzlu aktivační operace.
- Pokud je instance operace aktivovaná v operaci *main*, pak zdrojový prvek, se kterým je operace spojena, je synchronní (každý nový hodinový takt mění svůj stav). Typová značka takového prvku je změněna na S.
- V případě podmíněné aktivace uchovává uzel v podobě vnitřního modelu i obsah a strukturu podmíněného příkazu, aby tento podmíněný příkaz později mohl být generován do jazyka VHDL.

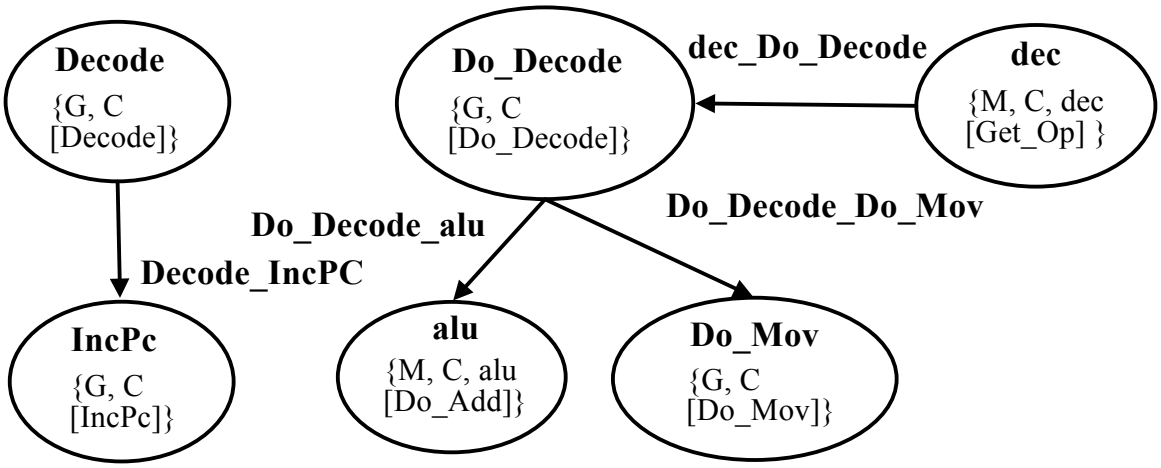
Příklad

Popis části architektury v jazyce ISAC:

```
RESOURCE {
   LC (dec) dec;
   LC (alu) alu;
}
OPERATION Do_Add IN alu
{
   ...
}
OPERATION Do_Mov
{
   ...
}
OPERATION Get_Op IN dec
{
   ...
}
OPERATION Do_Decode
{
   INSTANCE  Get_Op ALIAS { opc };
   INSTANCE  Do_Add ALIAS { do_add };
   INSTANCE  Do_Mov ALIAS { do_mov };
   ...
   ACTIVATION{
      SWITCH(opc)
      {
         CASE Do_Add: do_add;
         CASE Do_Mov: do_mov;
      };
   };
}
OPERATION IncPC{
   BEHAVIOR{
      pc+=1;
   };
}
OPERATION Decode
{
   INSTANCE IncPC ALIAS {incpc};
   ACTIVATION
   {
```

```
      incpc;
   };
}
```

Uzly grafu procesů tvořené z popisu v jazyce ISAC jsou znázorněny na obr. 4.8.



Obr. 4.8. Uzly grafu procesů odpovídající popisu v jazyce ISAC

Z analýzy aktivační sekce operace *Do_Decode* vznikne uzel *Do_Mov* pro aktivovanou operaci *Do_Mov*. Rovněž vznikají hrany *dec_Do_Decode, Do_Decode_Do_Mov* a *Do_Decode_alu.* Z analýzy aktivační sekce operace *Decode* vznikne uzel *IncPc* a hrana *Decode_IncPc.*

4. Vytváření datových cest – interakce obecného prvku s okolím

V této části tvorby grafu procesů vycházíme z předpokladu, že všechny výpočetní operace budou součástí nějaké funkční jednotky. Tedy operace budou přiřazeny prvku LC. Na základě tohoto předpokladu provádíme svázání LC prvku s jinými zdrojovými prvky. Toto svázání je realizováno pomocí hlavičky sekce BEHAVIOR.

- Pro všechny operace, které jsou přiřazeny nějakému prvku LC a obsahují BEHAVIOR sekci s hlavičkou, tvoříme orientované hrany grafu následně:
  - Pro všechny zdrojové prvky uvedené v IN sekvenci hlavičky BEHAVIOR sekce tvoříme hrany z uzlů reprezentujících zdrojové prvky do uzlu reprezentujícího prvek LC.
  - Pro všechny zdrojové prvky uvedené v OUT sekvenci hlavičky BEHAVIOR sekce tvoříme hrany z uzlu reprezentujícího prvek LC do uzlů reprezentujících zdrojové prvky.

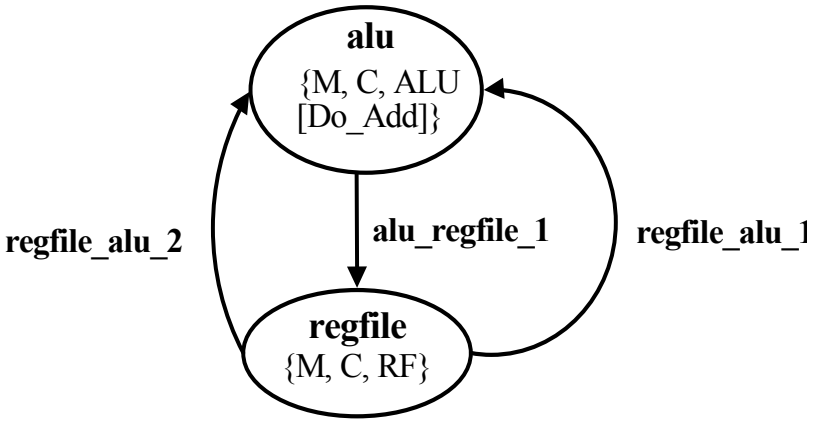
Příklad

Popis části procesoru v jazyce ISAC:

```
RESOURCE {
  RAM (RF) bit[8] regfile {
    ENDIANESS   (BIG);
    BLOCKSIZE   (8, 8);
    SIZE (255);
```

```
      FLAGS (R,W);
   };
   LC (ALU) alu;
}
OPERATION Do_Add IN alu
{
   BEHAVIOR (IN[regfile,regfile] OUT[regfile])
   {
      regfile.write(regfile.read(0)+ regfile.read(1));
   };
}
```

Uzly grafu procesů tvořené z popisu v jazyce ISAC jsou znázorněny na obr. 4.9.



Obr. 4.9. Uzly grafu procesů odpovídající popisu v jazyce ISAC

Analýzou hlavičky sekce BEHAVIOR vznikají pro uzel *alu* dvě vstupní hrany *regfile_alu_2* a *regfile_alu_1* a jedna výstupní hrana *alu_regfile_1*. Čísla v názvu hran udávají pořadí portů.

5. Vytváření instrukčního dekodéru

Uzly grafu procesů reprezentující instrukční dekodér vznikají na základě analýzy sekcí CODINGROOT a CODING. Sekce CODINGROOT určuje kořen dekódovacího stromu. Dekódovací strom má podobu AND/OR grafu a vzniká na základě analýzy CODING sekcí. AND uzly reprezentují operace, které v sekci CODING volají instance jiných operací. OR uzly reprezentují skupiny (v jazyce ISAC označované klíčovým slovem GROUP).

Pro každou operaci obsahující sekci CODINGROOT jsou provedeny následující změny v grafu procesů:

- Je-li v sekci CODINGROOT uvedena instance operace, pak vytváříme pro operaci, jejíž instance je uvedena, nový uzel grafu a přiřadíme do něj identifikátor operace. Takto vytvořený uzel označíme speciální značkou **D** (ve významu dekodér). Pokud operace, jejíž instance je uvedena, již patří do nějakého uzlu grafu, označíme pouze tento uzel značkou **D** a nový uzel pro operaci nevytváříme.
- K nově vytvořenému či modifikovanému uzlu vytvoříme hranu vedoucí z uzlu reprezentujícího zdrojový prvek, který je uveden v instanci operace.

Tato část grafu procesů nebyla v rámci práce implementována, neboť bude vyžadovat další rozšíření a modifikaci původně navrženého postupu tvorby grafu procesů. O zmíněném rozšíření bude pojednáno v rámci kapitoly Další rozvoj práce. V nynějším řešení, které je prezentováno i na příkladech, je dekodér chápán jako prvek LC, pro který je připravena šablona v jazyce VHDL. Z tohoto důvodu není v této části uveden příklad popisující vytváření instrukčního dekodéru.

4.3.1.2 Reprezentace grafu procesů

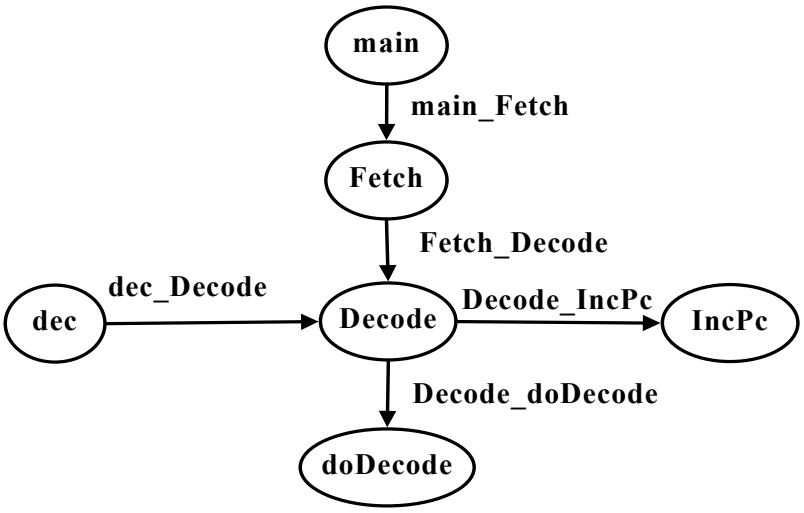
Vnitřní reprezentace grafu procesů je obtížně implementovatelná, proto byl pro jeho reprezentaci vybrán konečný automat [1]. Graf procesů tedy bude reprezentován pomocí konečného automatu následovně:

1. Konečnou množinu stavů Q tvoří uzly grafu.
2. Konečnou vstupní abecedu Σ tvoří názvy hran grafu.
3. Přechodová funkce δ je v grafu vyjádřena orientovanou hranou mezi dvěma uzly. Pro reprezentaci přechodové funkce je použita matice přechodů (tabulka).
4. Počátečním stavem q_0 je uzel, ve kterém je operace *main*.
5. Množinu koncových stavů F tvoří uzly grafu, ze kterých nevycházejí žádné hrany, a uzly, jejichž výstupní hrany končí v uzlu označeného značkou S.

Přechodová funkce je uložena ve formě tabulky. Sloupce tvoří prvky konečné množinu vstupních symbolů. Řádky tvoří prvky konečné množiny stavů. Průsečíky obsahují podmnožinu množiny Q (tedy stavy do kterých se konečný automat dostane z aktuální konfigurace C = (q, w), (q,w) ∈ Q × Σ*, kde *q* je aktuální stav a *w* je doposud nezpracovaná část vstupního řetězce.)

Příklad

Obr. 4.10 ukazuje graf procesů, pro který je vytvořena reprezentace pomocí konečného automatu.



Obr. 4.10. Ukázka grafu procesů

Formální popis konečného automatu reprezentujícího graf procesů:

```
Q = {main, Fetch, Decode, IncPc, dec, doDecode}
Σ = {main_Fetch, Fetch_Decode, Decode_IncPc, Decode_doDecode,
    dec_Decode}
q0 = main
F = {IncPc, doDecode}
```

Reprezentace konečného automatu ve formě tabulky:

Q/Σ	main_Fetch	Fetch_Decode	Decode_IncPc	Decode_doDecode	dec_Decode
main	Fetch				
Fetch		Decode			
Decode			IncPc	doDecode	
IncPc					
doDecode					
dec					Decode

Tabulka. 4.2. Tabulka reprezentující konečný automat

4.3.2 Generování popisu v jazyce VHDL

Výsledný popis architektury v jazyce VHDL je vytvořen z vnitřního modelu transformace, kterým je graf procesů. Uzly a hrany grafu, které jsou v rámci implementace reprezentovány odděleně. Před generováním popisu v jazyce VHDL jsou převedeny na reprezentaci konečným automatem. Konečný automat, který reprezentuje vytvářený graf je poté vstupem do fáze mapování a generování. Tyto dvě fáze provedou vlastní transformaci z jazyka ISAC do jazyka VHDL. Součástí generování je však i fáze propojení komponent, kdy je vytvořena hlavní entita procesoru, jehož popis je transformován.

4.3.2.1 Fáze mapování

Než začne vlastní fáze mapování či generování, je kontrolováno, zda graf procesů neobsahuje uzly se značkou E. Jde o uzly zdrojových prvků, které nemají uveden konfigurační identifikátor. Pokud graf obsahuje nějaký uzel se značkou E, není transformace provedena a uživatel je informován chybovým hlášením.

V samotné fázi mapování jsou z konečné množiny stavů Q konečného automatu vybrány stavy označené značkou M. Podle konfiguračního identifikátoru je v konfiguračním souboru nalezena entita, na kterou má být zdrojový prvek mapován. Entita obsahuje informace o počtu vstupních a výstupních portů, které mohou být využity. Je provedena kontrola, zda není překročen povolený počet vstupních či výstupních portů. Kontrola vychází z faktu, že uzel grafu procesů představuje entitu, kde počet vstupních hran udává počet vstupních portů a počet výstupních hran udává počet výstupních portů.

Tyto počty získáme jednoduše z tabulky, která reprezentujeme konečný automat. Počet výstupních portů udává počet neprázdných průsečíků v řádku daného stavu. Počet vstupních portů udává počet všech průsečíku, ve kterých se daný stav nachází.

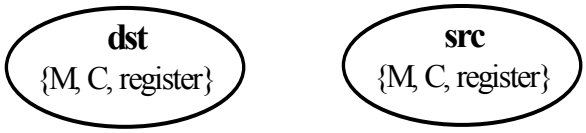
Pokud byla kontrola portů úspěšná, je nalezen zdrojový kód šablony v jazyce VHDL. Pokud jsou v konfiguračním souboru vyžadovány některé generické parametry, jsou jejich hodnoty doplněny do kódu v jazyce VHDL. Takto modifikovaný kód je poté uložen do generovaného souboru.

Příklad

Popis části procesoru v jazyce ISAC:

```
RESOURCE {
  REGISTER (register) bit[8] src, dst;
}
```

Uzly grafu procesů tvořené z popisu v jazyce ISAC jsou znázorněny na obr. 4.11.



Obr. 4.11. Uzly grafu procesů vytvořené z popisu v jazyce ISAC

V první části příkladu je k části popisu v jazyce ISAC vytvořena část grafu procesů. Do transformace nyní vstupuje konfigurační soubor a proběhne fáze mapování.

Část kódu v jazyce XML, který v konfiguračním souboru popisuje entitu identifikovanou konfiguračním identifikátorem:

```
<TEMPLATE>
<ID>RISC</ID>
<IR>
   <ID>register</ID>
   <PORT_CNT>
      <IN>4</IN>
      <OUT>1</OUT>
      <INOUT>0</INOUT>
   </PORT_CNT>
   <VHDL>
      <ENTITY>reg</ENTITY>
      <FILE>register.vhd</FILE>
      <GENERIC_LIST>
         <ITEM>
            <ID>width</ID>
            <IMELEMENT>SIZE_OF_BITS</IMELEMENT>
         </ITEM>
      </GENERIC_LIST>
   </VHDL>
</IR>
</TEMPLATE>
```

Generovaný kód v jazyce VHDL (register8.vhd):

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

entity register8 is
generic (
     width : integer :=8 ;
);
port (
   CLK   : in  std_logic;
   RESET : in  std_logic;
   DIN   : in  std_logic_vector(width-1 downto 0);
   REG_WE :in std_logic;
   DOUT  : out std_logic_vector(width-1 downto 0)
);
end register8;

architecture arch_register8 of register8 is
begin
process (CLK,RESET)
begin
if (RESET='1') then
   DOUT <= '0';
elsif (CLK'event and CLK = '1') then
    if (REG_WE = '1') then
          DOUT <= DIN;
   end if;
end if;
end process;
end  arch_register8;
```

Do kódu šablony je doplněna hodnota generické konstanty, název entity je nově nastaven na *register8* (*register* – konfigurační identifikátor, 8 – bitová šířka). Název generovaného souboru s kódem v jazyce VHDL odpovídá názvu generované entity.

4.3.2.2 Fáze generování

Z konečné množiny stavů Q konečného automatu jsou vybrány stavy označené značkou G. Pro tyto stavy jsou generovány nové entity v jazyce VHDL (pro každý stav jedna entita v jednom samostatném souboru). Vstupní a výstupní porty entity zjistíme z tabulky reprezentující konečný automat stejným způsobem, jakým se ve fázi mapování hledají počty portů. Pokud má entita nějaký výstupní signál je provedeno nastavení jeho hodnoty v architektuře entity, jinak je architektura entity prázdná. Architektura je popsána procesem, který má na svém sensitivity listu uvedeny všechny vstupní signály.

Pokud stav obsahuje typovou značku S, jsou vstupní porty doplněny o signál hodin.

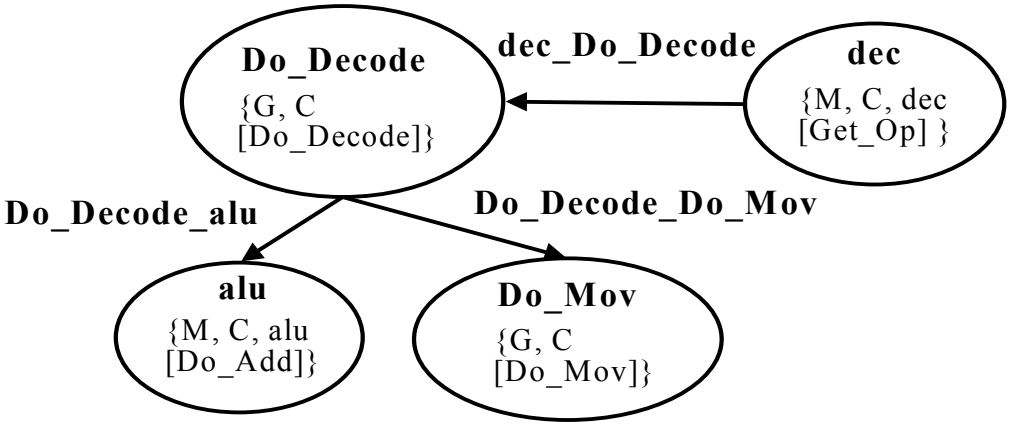
Příklad

Popis části procesoru v jazyce ISAC:

```
RESOURCE {
   LC (dec) dec;
   LC (alu) alu;
}
```

```
OPERATION Do_Add IN alu
{
   ...
}
OPERATION Do_Mov
{
   ...
}
OPERATION Get_Op IN dec
{
   ...
}
OPERATION Do_Decode
{
   INSTANCE  Get_Op ALIAS { opc };
   INSTANCE  Do_Add ALIAS { do_add };
   INSTANCE  Do_Mov ALIAS { do_mov };
   ...
   ACTIVATION{
      SWITCH(opc)
      {
         CASE Do_Add: do_add;
         CASE Do_Mov: do_mov;
      };
   };
}
```

Uzly grafu procesů tvořené z popisu v jazyce ISAC jsou znázorněny na obr. 4.12.



Obr. 4.12. Uzly grafu procesů vytvořené z popisu v jazyce ISAC

V první části příkladu je k části popisu v jazyce ISAC vytvořena část grafu procesů. Na základě informací v grafu procesů proběhne generování entity.

Zdrojový kód jazyka VHDL (Do_Decode.vhd, ukázka pouze pro uzel Do_Decode):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
ENTITY Do_Decode IS
PORT(
   dec_Do_Decode : IN  STD_LOGIC;
   Do_Decode_alu : OUT STD_LOGIC
   Do_Decode_Do_Mov : OUT STD_LOGIC  );
```

```
END Do_Decode;

ARCHITECTURE arch_Do_Decode OF Do_Decode IS
PROCESS(dec_Do_Decode)
BEGIN
   CASE (dec_Do_Decode) IS
      WHEN '0' => Do_Decode_alu <= ;
      -- set signal value for Do_Decode_alu
      WHEN '1' => Do_Decode_Do_Mov <= '1';
   END CASE;
END PROCESS;
END arch_Do_Decode;
```

Vygenerována je nová entita *Do_Decode* (uložena v souboru *Do_Decode.vhd*). Entita odpovídá generovanému uzlu *Do_Decode,* který má jednu vstupní a dvě výstupní hrany. Entita tedy obsahuje jeden vstupní port a dva výstupní. Přičemž výstupní signál na výstupním portu je nastavován jen v případě, pokud odpovídající hrana grafu procesů vede k uzlu se značkou G.

4.3.2.3 Fáze propojení komponent

V této fázi je vytvořena hlavní entita popsaná na strukturální úrovni jazyka VHDL. Vstupním signálem hlavní entity je hodinový signál. Další vstupní a výstupní signály již doplňuje návrhář. Do architektury entity jsou vygenerovány všechny řídící a datové signály s komentářem, který popisuje zavedení jednotlivých signálů do konstrukcí *port map*. Pro každou entitu vygenerovanou v předchozích krocích je vytvořena konstrukce *port map*. Konstrukce je úplná v případě, že jsou vzájemně propojeny dva uzly se značkou G.

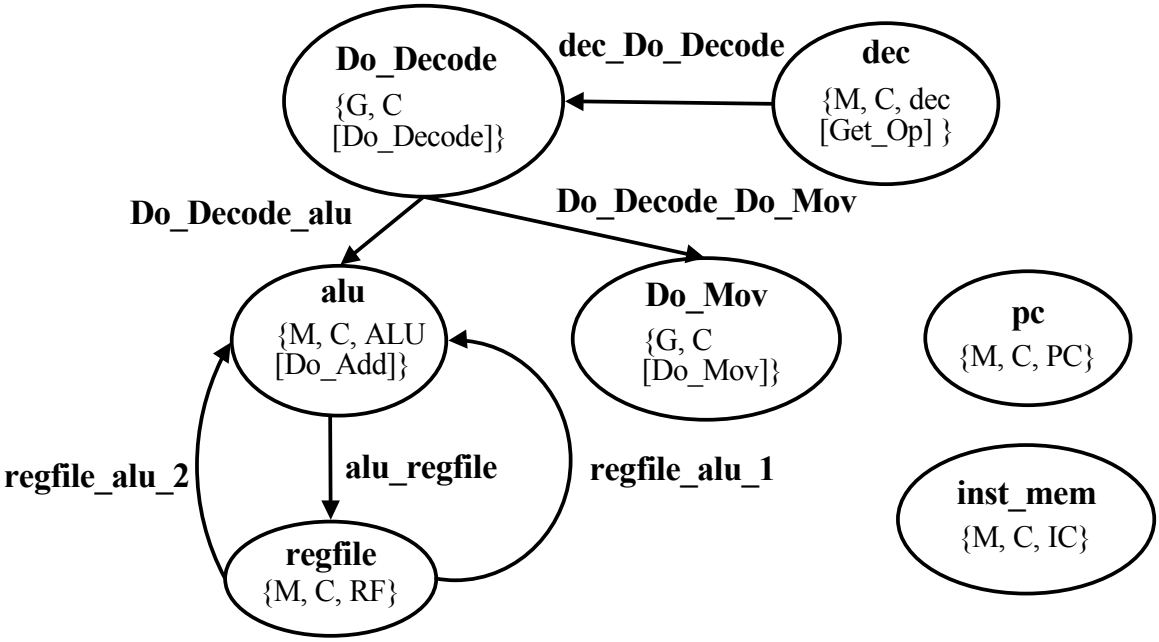
Příklad

Popis části procesoru v jazyce ISAC:

```
map "cnf_RISC.xml";
RESOURCE {
   PC REGISTER (PC) bit[8] pc;
   REGISTER (IC) bit[8] inst_mem[0..15];
   RAM (RF) bit[8] regfile {
      ENDIANESS   (BIG);
      BLOCKSIZE   (8, 8);
      SIZE (255);
      FLAGS (R,W);
   };
   LC (dec) dec;
   LC (ÄLU) alu;
}
OPERATION Do_Add IN alu
{
   ...
}
OPERATION Do_Mov
{
   ...
}
```

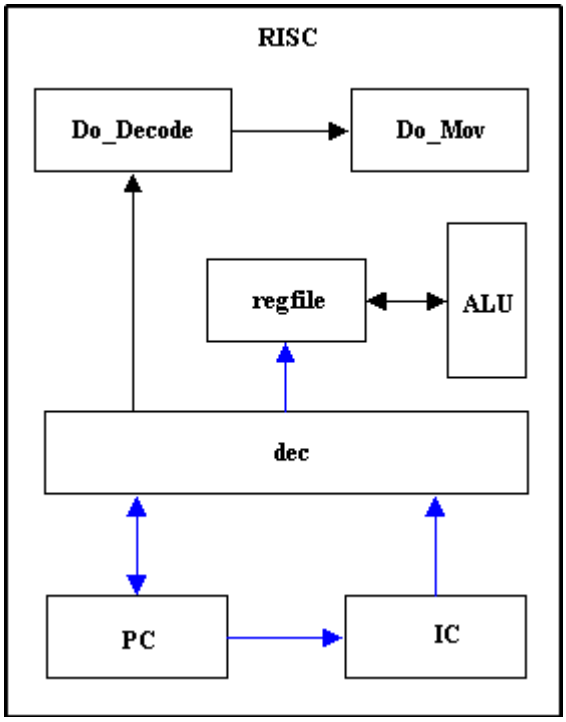
```
OPERATION Get_Op IN dec
{
   ...
}
OPERATION Do_Decode
{
   INSTANCE  Get_Op ALIAS { opc };
   INSTANCE  Do_Add ALIAS { do_add };
   INSTANCE  Do_Mov ALIAS { do_mov };
   ...
   ACTIVATION{
      SWITCH(opc)
      {
         CASE Do_Add: do_add;
         CASE Do_Mov: do_mov;
      };
   };
}
```

Uzly grafu procesů tvořené z popisu v jazyce ISAC jsou znázorněny na obr. 4.13.



Obr. 4.13. Uzly grafu procesů vytvořené z popisu v jazyce ISAC

Hierarchické zapojení entit jazyka VHDL, které jsou generovány z grafu procesů, je uvedeno na obr. 4.14.



Obr. 4.14. Hierarchie generovaných VHDL entit

Šipky v obrázku naznačují propojovací signály, kterými jednotlivé entity komunikují. Modře vyznačené signály musí doplnit návrhář, černě vyznačené signály jsou generovány v architektuře generované entity.

Příklad:

Zdrojový kód jazyka VHDL (RISC.vhd):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
ENTITY RISC IS
  PORT(
    CLK  : IN STD_LOGIC
  );
END RISC;

ARCHITECTURE arch_RISC OF RISC IS
  -- This signal is prepared for connection of components
  -- Do_Decode and Do_Mov. Use it in port map construction.
  SIGNAL Do_Decode_Do_Mov : STD_LOGIC;
  SIGNAL Do_Decode_alu : STD_LOGIC;
  SIGNAL alu_regfile : STD_LOGIC;
  SIGNAL dec_Do_Decode : STD_LOGIC;
  SIGNAL regfile_alu_1 : STD_LOGIC;
```

```
    SIGNAL regfile_alu_2 : STD_LOGIC;
  BEGIN
  -- Port map constructions for IR(ISAC RESOURCES) entities --
    inst_mem : ENTITY work.IC8
    PORT MAP ();
    pc : ENTITY work.pc8
    PORT MAP ();
    regfile : ENTITY work.RF8
    PORT MAP ();
  -- Port map constructions for FU(Function Units) entities --
    alu : ENTITY work.alu
    PORT MAP ();
    dec : ENTITY work.dec
    PORT MAP ();

  -- Port map constructions for controller entities --
    Do_Decode : ENTITY work.Do_Decode
    PORT MAP ();
    Do_Mov : ENTITY work.Do_Mov
    PORT MAP (Do_Decode_Do_Mov);
  END arch_RISC;
```

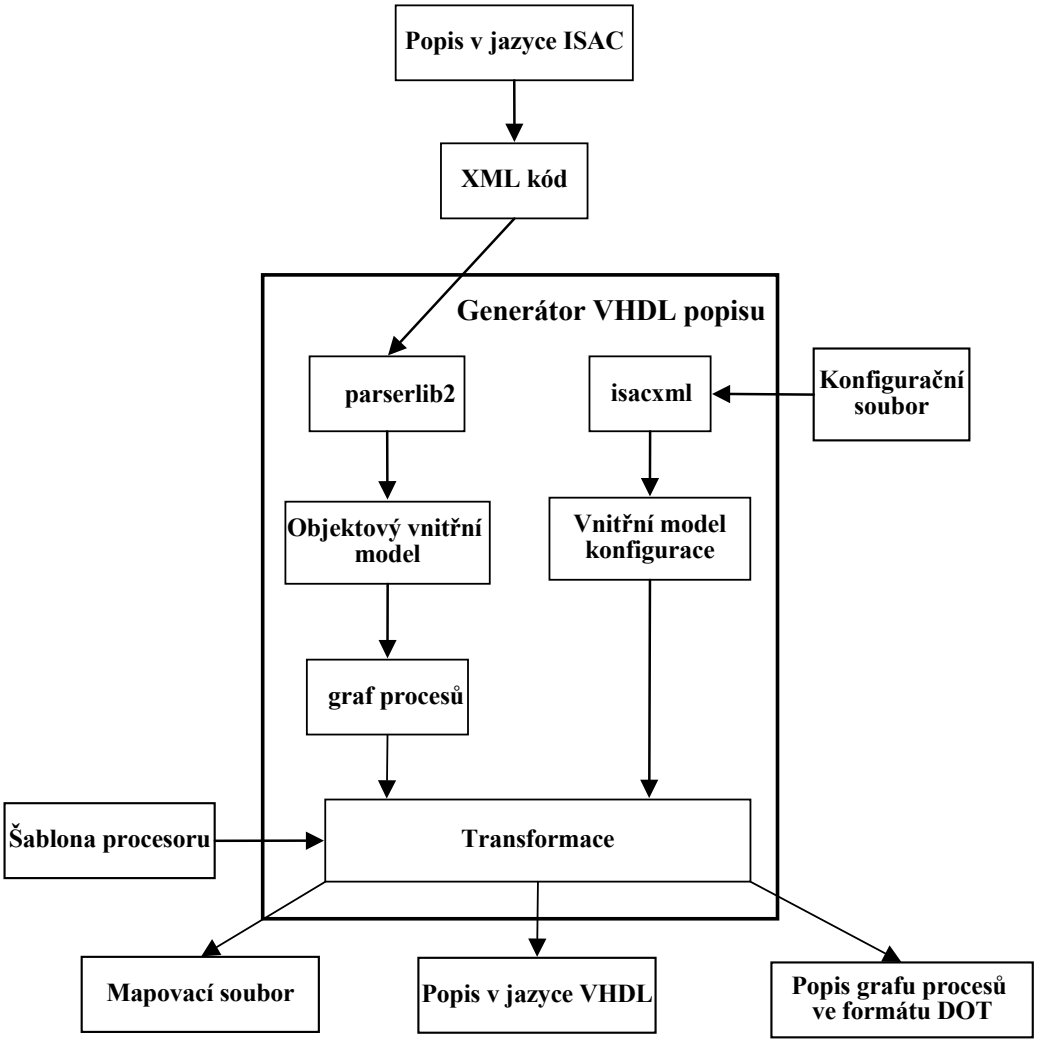
Příklad ukazuje generovanou hlavní entitu procesoru. Komentáře jsou generovány ke všem signálům (v příkladu je jen jeden komentář pro ukázku). Kompletní *port map* konstrukce je vygenerována jen jedna a to konstrukce pro mapování *Do_Mov* entity. Ostatní *port map* konstrukce musí doplnit návrhář. Popřípadně i další signály nutné k propojení entit.

4.3.3 Začlenění generátoru do projektu Lissom

Generátor popisu v jazyce VHDL, jak je nazýván nově vytvořený nástroj provádějící transformaci z jazyka ISAC do jazyka VHDL, využívá několika dílčích knihoven implementovaných v rámci projektu Lissom. Mezi ně patří knihovna *parserlib,* jenž zpracovává kód v jazyce XML, který je výstupem překladače jazyka ISAC. Tato knihovna umožňuje uložit data z XML do objektového vnitřního modelu s nímž poté pracuje generátor popisu v jazyce VHDL. V rámci generátoru je tedy implementován pouze průchod objektovým vnitřním modelem, nikoliv však vlastní analýza vstupního kódu v jazyce XML.

Další knihovnou, kterou využívá generátor popisu v jazyce VHDL, je knihovna *isacxml*. Pomocí ní je analyzován vstupní kód konfiguračního souboru v jazyce XML, který je ukládán do vnitřního modelu konfigurace.

Proces transformace z jazyka ISAC do jazyka VHDL s využitím knihoven je znázorněn na obr. 4.15. Obrázek ukazuje jednotlivé fáze, ve kterých transformace probíhá.



Obr. 4.15. Transformace ISAC -> VHDL s využitím knihoven

Generátor popisu v jazyce VHDL je integrován do vývojového prostředí projektu Lissom. V případě, že chce vývojář provádět transformaci popisu z jazyka ISAC, jsou vývojovým prostředím v rámci aktuálního projektu vytvořeny dva adresáře pro transformaci. První adresář bude obsahovat zdrojové kódy šablony a její konfigurační soubor. Druhý adresář slouží pro generování výstupního popisu v jazyce VHDL. Vývojové prostředí předává generátoru cesty k těmto adresářům.

Před procesem transformace je nutné provést kontrolu existence *map* příkazu v transformovaném popisu v jazyce ISAC a kontrolu validity konfiguračního souboru.

Aby mohla transformace proběhnout, musí zdrojový kód v jazyce ISAC obsahovat konstrukci:

```
map "název_konfiguračního_souboru";
```

Tento soubor rovněž musí být součástí adresáře šablony procesoru.

Před spuštěním transformace je vyžadována kontrola validity konfiguračního souboru šablony v jazyce VHDL. Každá šablona architektury musí mít vytvořen jeden konfigurační soubor. Pomocí

XML schématu[1] je jednoznačně definováno, jakým způsobem může vypadat konfigurační soubor šablony. Díky existenci schématu může být ve vývojovém prostředí snadno proveditelná validace konfiguračního souboru. Proto vývojové prostředí umožňuje uživateli provést validaci vytvořeného konfiguračního souboru v rámci integrovaného tlačítka pro ověření validity.

Kromě popisu v jazyce VHDL jsou výstupem transformace mapovací soubor a soubor obsahující popis grafu procesů ve formátu DOT. Součástí projektu Lissom bude nástroj, provádějící zpětnou transformaci z popisu v jazyce VHDL do popisu v jazyce ISAC. Tato transformace je nutná pro zpětnou validitu změn provedených ve vygenerovaném popisu v jazyce VHDL a přenesení takových změn do popisu v jazyce ISAC. Mapovací soubor pak zpětné transformaci poskytuje informace o mapování zdrojových prvků na entity šablony a přiřazení operací funkčním jednotkám.

Zpětná transformace rovněž generuje z popisu v jazyce VHDL graf procesů ve formátu DOT. Ten bude porovnáván z hlediska kompatibility s grafem procesů generovaným z popisu v jazyce ISAC.

4.3.4 Testování

Pro ověření funkčnosti generátoru byla použita šablona jednoduchého procesoru RISC (uvedená v kapitole 4.2.1.2). Popis v jazyce ISAC pro tuto architekturu je součástí média přiloženého k práci. Výstup generátoru je rovněž součástí tohoto média.

K dispozici je prozatím pouze tato šablona jednoduché architektury procesoru RISC.

[1] XML schéma (XML Schema Definition - XSD) popisuje strukturu XML dokumentu, popisuje metadata XML dokumentu.

5 Další rozvoj práce

Autor práce si je vědom, že jím uvedené řešení práce nepokrývá kompletní transformaci jazyka ISAC. Například zřetězené zpracování, mapování pamětí či aliasy zdrojových prvků transformovány nejsou. Do budoucna se však počítá s rozšířením, ve kterém jistě budou transformovány i tyto konstrukce jazyka ISAC.

Nejproblematičtější částí práce bylo nalezení řešení, které umožňuje propojení komponent popisované architektury procesoru a které umožňuje generovat popis řídící a dekódovací logiky. V této části práce zůstalo několik nedořešených problémů, které by měly být řešeny v dalších etapách vývoje práce. Mezi zmíněné problémy lze řadit propojování komponent a generování dekódovací logiky.

Řešení transformace z jazyka ISAC do jazyka VHDL si v dalším vývoji projektu Lissom vyžádá též rozšíření překladače jazyka ISAC.

V kapitole popisující šablonu procesoru jsem se zmínil o parametrizaci šablony pomocí hodnot získaných z popisu v jazyce ISAC. Další vývoj práce předpokládá nutnou podporu při tvorbě těchto šablon a jejich konfiguračních souborů.

5.1 Propojení komponent

Zdrojové prvky jazyka ISAC jsou mapovány za pomoci identifikátoru v konfiguračním souboru na komponenty šablony popsané v jazyce VHDL. Návrhář má v generovaném popise v jazyce VHDL k dispozici informaci o tom, které komponenty by měl mezi sebou propojit. Současně má k dispozici sadu připravených signálů, za pomoci nichž může propojení realizovat. Schází ovšem informace o tom, na který port patřičné komponenty má být přiveden patřičný signál.

Další vývoj práce by měl spočívat v hledání řešení, které by umožnilo výše zmíněnou informaci do generovaného popisu doplnit. Pokud bude řešení nalezeno, mohl by návrhář dostat kompletní generovaný popis v jazyce VHDL a nemusel by do něj zasahovat a propojovat komponenty ručně. Lze však předpokládat, že nalezení tohoto řešení bude vyžadovat další rozšíření jazyka ISAC o nové konstrukce blízké jazykům pro popis hardwaru. Proto je nutné brát v úvahu, že zavedení těchto konstrukcí by mohlo výrazně snížit vysokou úroveň abstrakce popisu v jazyce ISAC, která je pro něj charakteristická.

5.2 Generování dekódovací logiky

V kapitole popisující tvorbu grafu procesů jsem zmínil, že vytváření instrukčního dekodéru bude znamenat jisté rozšíření a modifikaci postupu, kterým je graf procesů vytvářen.

Popis dekodéru v jazyce VHDL by neměl být generován na základě šablony, ale měl by být generován na základě popisu kódování operací uvedeného v rámci jazyka ISAC. Z tohoto důvodu by tak mělo být modifikováno zpracování aktivačních sekcí operací, kde byl doposud za dekodér považován prvek LC.

Další rozšíření generování instrukčního dekodéru bude vycházet z příbuzné práce prováděné v rámci projektu Lissom. V této příbuzné práci členové projektu Lissom zabývající se vývojem asembleru a disassembleru navrhují vnitřní model reprezentující AND/OR graf, který obsahuje informace o kódování operací jazyka ISAC. Navrženým modelem je deterministický konečný automat. Tato práce by tedy měla v další fázi vývoje zakomponovat automat do procesu generování popisu instrukčního dekodéru v jazyce VHDL.

5.3 Rozšíření překladače jazyka ISAC

Díky zavedení nových konstrukcí do jazyka ISAC je nutné v rámci dalšího vývoje projektu Lissom rozšířit překladač jazyka ISAC. Z hlediska syntaktického již byly nové konstrukce do překladače začleněny, potřeba je však doplnit sémantickou kontrolu prováděnou nad novými konstrukcemi. Překladač by měl provádět tuto sémantickou kontrolu:

- V případě, že má operace u sekce BEHAVIOR uvedenu hlavičku, musí být operace přiřazena prvku LC.
- Pokud popis v jazyce ISAC bude obsahovat příkaz **map** identifikující šablonu, měly by všechny zdrojové prvky uvedené v sekci RESOURCE obsahovat konfigurační identifikátor.
- V případě, že operace v jazyce ISAC obsahuje sekci ACTIVATION, neměla by tato operace být přiřazena prvku LC.
- Pokud je operace jazyka ISAC označena v sekci CODINGROOT jako operace provádějící dekódování (v sekci je uvedena instance této operace), neměla by tato operace být přiřazena prvku LC.

6 Závěr

S rozvojem procesorů a vestavěných systémů se v posledních letech objevují projekty, které se snaží poskytnout návrhářům procesorů co nejefektivnější prostředky pro tvorbu jejich návrhu. Procesor je často modelován s využitím specifikačního jazyka ADL. Tato specifikace se využívá ke generování různých nástrojů, které podporují průzkum a validaci navrhované architektury. Požadovaná plocha na čipu, frekvence hodin či výkonová spotřeba mohou být stanoveny pouze ve spojení se syntetizovatelným modelem v jazyce HDL. Proto je žádoucí generovat odpovídající model v jazyce HDL ze specifikace v jazyce ADL.

Mezi zmiňované projekty patří i projekt Lissom. Jedním z cílů projektu je poskytnout návrháři syntetizovatelný popis modelované architektury v jazyce VHDL. Hlavním úkolem práce bylo navrhnout řešení transformace z jazyka ISAC do jazyka VHDL a takové řešení implementovat.

V práci byl podrobně zmíněn návrh transformace, který prošel několika variantami. V první variantě byl zamítnut návrh s plně generovanou transformací. V druhé fázi bylo zvažováno řešení s generickou šablonou procesoru, které však byly málo flexibilní. V poslední variantě byl přijat návrh transformace na základě šablon (šablon komponent architektury popsaných v jazyce VHDL). Šablony jsou v oblasti návrhu procesoru používané a návrhářům by tak nemělo dělat problém si na tento způsob transformace zvyknout. Rovněž byl v průběhu návrhu transformace zvažován opačný přístup, tedy z generické šablony procesoru popsané v jazyce VHDL generovat odpovídající popis v jazyce ISAC. Tento přístup však bude uplatněn jen pro zpětnou transformaci.

Transformace za pomocí šablon si vyžádala rozšíření jazyka ISAC o nové konstrukce podporující transformaci. Do budoucna se předpokládá, že některé nové konstrukce ještě přibudou.

V práci bylo představeno řešení, pomocí něhož lze generovat popis jednotlivých komponent procesoru v jazyce VHDL a jejich částečné propojení. Řešení není zcela komplexní, neboť výsledkem transformace není plně syntetizovatelný popis. Generovaný popis v jazyce VHDL vyžaduje zásah návrháře. Návrhář v současném řešení musí částečně doplnit propojení některých komponent popisovaného procesoru. V některých případech musí rovněž doplnit tělo architektury jazyka VHDL u generovaných entit jazyka VHDL. Další pokračování práce by se tedy mohlo zaměřit na odstranění tohoto nedostatku a případné optimalizace generovaného popisu v jazyce VHDL.

I když práce neřeší kompletní problematiku transformace, byl jí položen dobrý základ pro další vývoj, jehož cílem je kompletní dokončení transformace založené na šablonách. To jakým směrem se bude další vývoj práce ubírat je zmíněno v kapitole Další rozvoj práce. Tento základ pro další vývoj lze považovat za přínos práce současně s určením konceptu řešení transformace. S výhledem do budoucna by práce měla v dalším vývoji poskytnout návrháři popis architektury procesoru na nízké úrovni abstrakce (popis v jazyce VHDL). Díky tomu bude mít návrhář k dispozici s dřívějším popisem na vyšší úrovni abstrakce (popis v jazyce ISAC) kompletní model procesoru z hlediska hardwaru

i softwaru. Práce pak bude přínosem pro celou oblast Hardware/Software co-design, neboť vznikne kompletní prostředí pro návrh aplikačně specifických procesorů, přičemž takových prostředí na trhu k dispozici mnoho není.

Literatura

[1] ČEŠKA M., VOJNAR T. *Teoretická informatika - (přednáška z předmětu TIN).* Brno: FIT VUT v Brně, 2006

[2] DOUŠA J. *Jazyk VHDL.* Praha: Nakladatelství ČVUT, 2003, 76 str., ISBN 80-01-02670-1

[3] DVOŘÁK V., DRÁBEK V. *Architektura procesorů.* Brno: Vysoké učení technické v Brně, nakladatelství VUTIUM, 1999, 303 str., ISBN 80-214-1458-8

[4] FALÝNEK P. *Podpora výuky hardware na bázi FPGA: diplomová práce.* Brno: Vysoké učení technické v Brně, Fakulta informačních technologií, 2004, 69. str.

[5] HOFFMANN, A., KOGEL, T., NOHL, A., BRAUN G., SCHLIEBUSCH O., WAHLEN O., WIEFERINK A., MEYR H. *A novel methodology for the design of application specific integrated processors (ASIP) using a machinedescription language*. Aachen: Aachen University of technology Germany, 2001.

[6] HOFFMANN, A., MEYR, H., LEUPERS, R. *Architecture exploration for embedded processors with LISA*. Boston: Kluwer Academic Publishers, 2002. 230 s., ISBN 1-4020-7338-0

[7] HRUŠKA, T., *Instruction set architecture C - ISAC*. Brno: Interní materiál FIT VUT v Brně, 2004.

[8] LI L., THORTHON M. A., SZYGENDA S. A.. *Integrated design validation: Combining simulation and formal verification for digital integrated circuit.* Dallas : Dept. of Computer Science and Engineering Southern Methodist University, 2005.

[9] KOŘENEK J. *Syntéza – Pokročilé číslicové systémy (přednáška z předmětu PCS)*. Brno : FIT VUT v Brně, 2006.

[10] MACHO, T. *Vestavné systémy.* http://www.automa.cz/automa/2002/au120205.htm, 2002.

[11] MARTÍNEK T. *Návrh mikroprocesoru v FPGA – Pokročilé číslicové systémy (přednáška z předmětu PCS)*. Brno : FIT VUT v Brně, 2006.

[12] MEYR H., GLŐKLER T. *Design of energy-efficient application-specific instruction set processors.* Boston: Kluwer Academic Publishers, 2004, 221 str., ISBN 1-4020-7730-0

[13] SCHLIEBUSCH O., HOFFMANN A., NOHL A., BRAUN G., MEYR H. *Architecture implementation using the machine description language LISA.* Aachen : Aachen University of technology Germany, 2001.

[14] SCHLIEBUSCH O., CHATTOPADHYAY A., LEUPERS R., ASCHEID G., MEYR H. *RTL processor synthesis for architecture exploration and implementation.*. Aachen : Aachen University of technology Germany, 2004.

[15] SCHWARZ, J. *Vestavěné systémy – úvod (přednáška z předmětu IMP)*. Brno: FIT VUT v Brně, 2004.

Další použité zdroje

[16] PŘISPĚVATELÉ Wikipedie. *Digitální signálový procesor.* Wikipedie : Otevřená encyklopedie, získáno 19. 4. 2007 z http://cs.wikipedia.org/wiki/DSP

[17] PŘISPĚVATELÉ The Program Transformation Wiki. *Program transformation.*. The Program Transformation Wiki : Otevřená encyklopedie, získáno 19. 4. 2007 z http://www.program-transformation.org/Transform/ProgramTransformation

[18] PŘISPĚVATELÉ Wikipedie. *Program transformation.* Wikipedie : Otevřená encyklopedie, získáno 19. 4. 2007 z http://en.wikipedia.org/wiki/Program_transformation

[19] http://merlin.fit.vutbr.cz/lissom/index.html, 2007

Přílohy

Příloha č. 1 – Specifikace jazyka ISAC – rozšíření spojená s transformací

Příloha č. 2 – XML schéma pro konfigurační soubor

Příloha č. 3 – Rozšíření vnitřního modelu jazyka ISAC o nové konstrukce

.

Příloha č. 1

Konvence pro popis syntaxe

Neterminální symboly jsou jednoslovné a začínají velkými písmeny. Pokud se skládají z více podtermínů, každý z nich začíná velkým písmenem. Terminální symboly jsou uzavřeny do apostrofů.

Symbol " | " je použit pro varianty, závorky { } pro opakování 0 až n-krát, závorky [ ] pro volitelnost (opakování 0 až 1-krát) a závorky ( ) pro vyjádření skupiny se stejnými vlastnostmi.

Definice neterminálu začíná jeho jménem a je oddělena od pravé strany pravidla dvojtečkou. Zpravidla je na vlastním řádku a vlastní definice je zanořena tabelátorem.

Příklad:

```
PokusnyNeterminal:
   { [ SpolecnyZacatek ] } ( 'A' | NecoJineho ) }
```

Pokusný neterminál je definován jako 0 až n výskytů sekvence, která začíná nepovinným společným začátkem a za ním následuje buďto písmeno A nebo úsek popsaný neterminálem NecoJineho.

Nová klíčová slova jazyka ISAC – rozšíření spojená s transformací

Tabulka č.1 – Nová klíčová slova jazyka ISAC

IN	map	OUT

Popis syntaxe jazyka ISAC – rozšíření spojená s transformací

Poznámka: Nové konstrukce jazyka ISAC jsou zvýrazněny.

```
Description:
   [ SpecificationInclude ][ ResourceSection ] { OperationSection }

SpecificationInclude:
   { MacroGenerator } |
   HWTempalte |
   HWTempalte { MacroGenerator }

HWTempalte:
   'map' '"'Identifier '"'  ';'

ResourceSection:
   'RESOURCE' '{'
      { ResourceElement |
        MemoryElement |
        MemoryMap |
        PipelineResource|
```

```
        AliasStatement |
        LC
      }
   '}'

ResourceElement:
   ResourceSpecifier [ ConfigSpecification ] Declarator ';'

ConfigSpecification:
   '(' Identifier ')'

LC:
   'LC' [ ConfigSpecification ] IdentifierList ';'

OperationSection:
   Operation |
   Group

Operation:
   'OPERATION' Identifier [INSection ] '{'
        { OperationBody }
 '}'

INSection:
   'IN' Identifier '.' Identifier |
   Identifier

OperationBody:
   { InstanceSection } { CodeSection }

CodeSection:
   ( CodingSection ';' |
     AssemblerSection ';' |
     BehaviorSection ';' |
     ActivationSection ';' |
     ExpressionSection ';' |
     CodingrootSection ';' |
     IfElseSection |
     SwitchCaseSection
   )
BehaviorSection:
```

```
  ['('Head ')' ]

  '{' BehaviorCode '}'

Head:
  'IN' '[' IdentifierList ']'|

  'OUT' '[' IdentifierList  ']' |
  'IN' '[' IdentifierList ']' 'OUT' '[' IdentifierList  ']'
```

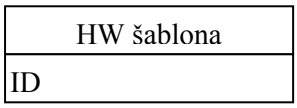
Příloha č. 2

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema
  xmlns="urn:isac:schemas:config:1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:NS="urn:isac:schemas:config:1.0"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
  targetNamespace="urn:isac:schemas:config:1.0"
  elementFormDefault="qualified">
    <xs:element name="TEMPLATE">
      <xs:complexType>
        <xs:sequence>
         <xs:element name="ID" type="xs:string"/>
         <xs:sequence minOccurs="1" maxOccurs="unbounded">
           <xs:choice>
             <xs:element name="IR" type="typEntitaIR" minOccurs="1"
    maxOccurs="unbounded" />
             <xs:element name="LC" type="typEntitaLC" minOccurs="1"
    maxOccurs="unbounded" />
           </xs:choice>
         </xs:sequence>
       </xs:sequence>
     </xs:complexType>
     <xs:unique name="TEMPLATEIdLc" msdata:PrimaryKey="true">
       <xs:selector xpath=".//NS:LC" />
       <xs:field xpath="NS:ID" />
     </xs:unique>
     <xs:unique name="TEMPLATEIdIR" msdata:PrimaryKey="true">
       <xs:selector xpath=".//NS:IR" />
       <xs:field xpath="NS:ID" />
     </xs:unique>
    </xs:element>

    <xs:complexType name="typEntitaIR">
      <xs:sequence>
        <xs:element name="ID" type="xs:string" nillable="true" />
        <xs:element name="PORT_CNT" type="typPort" />
        <xs:element name="VHDL">
          <xs:complexType>
            <xs:sequence>
             <xs:element name="ENTITY" type="xs:string" nillable="true" />
             <xs:element name="FILE" type="xs:string" nillable="true" />
             <xs:element name="GENERIC_LIST" type="typList" minOccurs="0"
       maxOccurs="1" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
       </xs:sequence>
   </xs:complexType>

   <xs:complexType name="typEntitaLC">
     <xs:sequence>
        <xs:element name="ID" type="xs:string" nillable="true" />
        <xs:element name="PORT_CNT" type="typPort" />
        <xs:element name="VHDL">
         <xs:complexType>
           <xs:sequence>
            <xs:element name="ENTITY" type="xs:string" nillable="true" />
```

```
            <xs:element name="FILE" type="xs:string" nillable="true" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="typPort">
    <xs:sequence>
      <xs:element name="IN" type="xs:int" nillable="true" />
      <xs:element name="OUT" type="xs:int" nillable="true" />
      <xs:element name="INOUT" type="xs:int" minOccurs="0" maxOccurs="1"
     nillable="true" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="typList">
    <xs:sequence>
      <xs:element name="ITEM" minOccurs="1" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="ID" type="xs:string" nillable="true" />
            <xs:element name="IMELEMENT" type="xs:string" nillable="true"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
   </xs:complexType>
</xs:schema>
```

Příloha č. 3

Identifikace HW šablony



Obr. 1. Vnitřní model pro identifikaci šablony

Příklad:

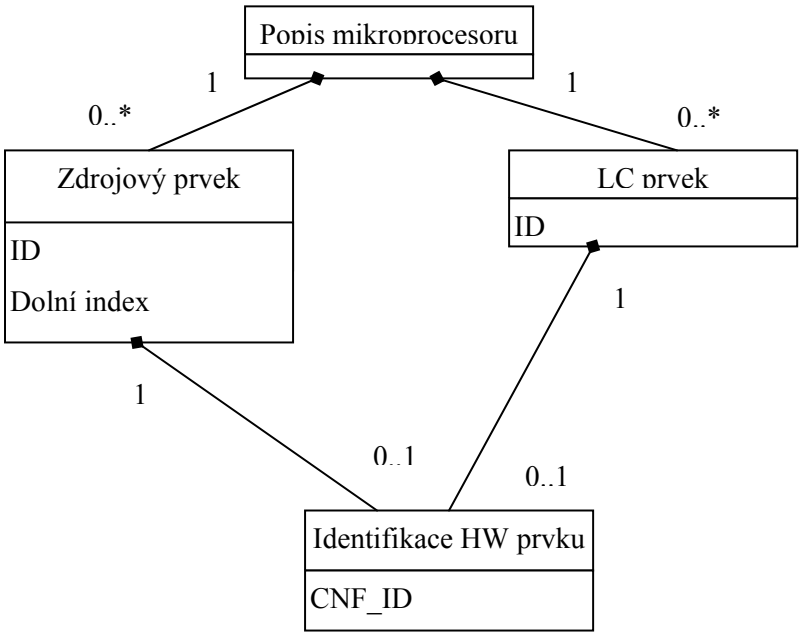
Popis v jazyce ISAC:

```
map "RISC.xml";
```

Odpovídající část XML kódu:

```
<TEMPLATE>
  <ID>RISC.xml</ID>
</TEMPLATE>
```

Prvky LC a identifikace prvku v konfiguračním souboru hardwarové šablony



Obr. 2. Vnitřní model pro prvek LC identifikaci v konfiguračním souboru

Příklad:

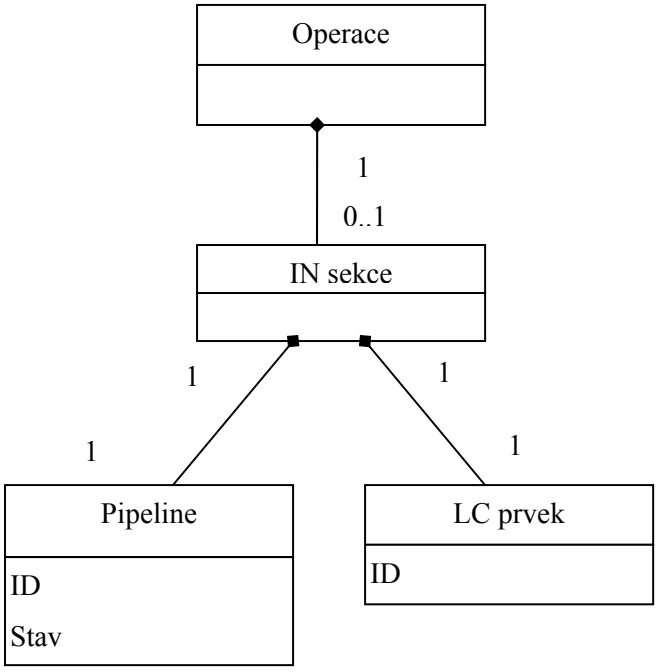
Popis v jazyce ISAC:

```
LC (ALU) Alu;
LC Decoder;
REGISTER (register) bit[8] reg;
```

Odpovídající část XML kódu:

```
<LC>
   <CNF_ID>ALU</CNF_ID>
   <ID>Alu</ID>
</LC>
<LC>
   <ID>Decoder</ID>
</LC>
<REGISTER>
   <CNF_ID> register </CNF_ID>
   <ID> reg </ID>
   <SIZE_OF_BITS> 8 </SIZE_OF_BITS>
</REGISTER>
```

IN sekce operace



Obr. 3. Vnitřní model pro IN sekci operace

Příklad:

Popis v jazyce ISAC:

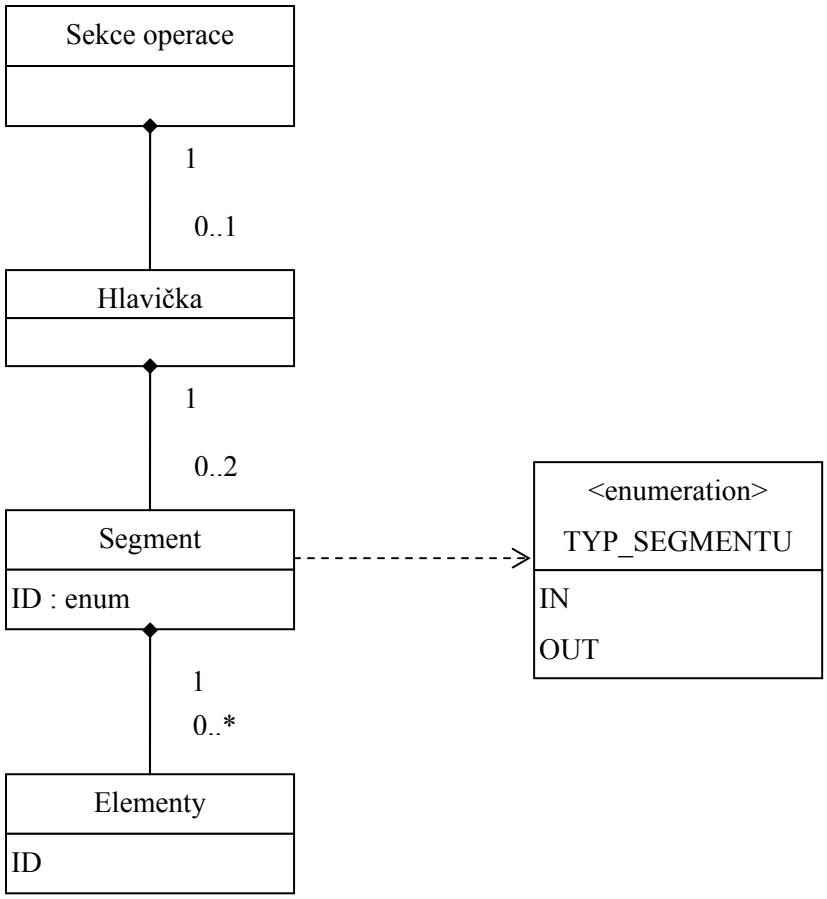
```
LC (ALU) Alu;
PIPELINE pipe {
   FE:;
}
OPERATION add IN Alu {...}

OPERATION decodeu IN pipe.FE{...}
```

Odpovídající část XML kódu:

```
   <OPERATION>
      <ID>add</ID>
      <IN_LC>
         <ID>Alu</ID>
```

```
      <IN_LC>
      ...
   </OPERATION>
   <OPERATION>
      <ID>add</ID>
      <PIPELINE_IN>
         <PIPELINE>
            <ID>pipe</ID>
         </PIPELINE>
         <STATE>
            <ID>FE</ID>
         </STATE>
      </PIPELINE_IN>
      ...
   </OPERATION>
```

Hlavička sekce



Obr. 4. Vnitřní model pro hlavičku sekce

Příklad:

Popis v jazyce ISAC:

```
REGISTER bit[8] rf [0..63];

OPERATION add IN Alu
{
   BEHAVIOR (IN[rf,rf] OUT[rf])
```

```
   {...};
}
```

Odpovídající část XML kódu:

```
<OPERATION>
     <ID>add</ID>
     <IN_LC>
        <ID>Alu</ID>
     <IN_LC>
     <BEHAVIOR>
        <HEAD>
           <SEGMENT>
              <ID>IN</ID>
              <ELEMENTS>
                 <ID>rf</ID>
                 <ID>rf</ID>
              </ELEMENTS>
           </SEGMENT>
           <SEGMENT>
              <ID>OUT</ID>
              <ELEMENTS>
                 <ID>rf</ID>
              </ELEMENTS>
           </SEGMENT>
        </HEAD>
     </BEHAVIOR>
  </OPERATION>
```