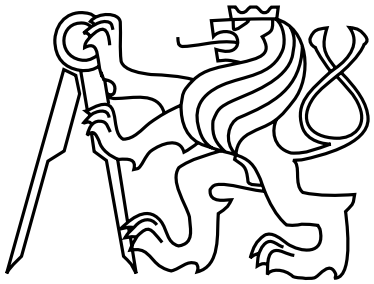
Na tomto místě by mělo být oficiální zadání práce

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Nástroj pro syntézu sekvenčních logických obvodů

Petr Kraus

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika, strukturovaný, Bakalářský

Obor: Výpočetní technika

31. 12. 2010

Prohlášení

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

V Praze 5. ledna 2011 .....

Abstract

This work aims to create a utility, exploiting structures of the EDuArd system, that would perform a sequential logic synthesis. With consideration of means presented by the EDuArd system, it shall mean a translation from a Mealy machine into a gate level description containing loosely defined flip-flops. Development of this utility will concentrate on establishing a structure for the synthesis algorithms and creation of a basic functional prototype. Second requirement for this utility is existence of a user interface in the form of Tcl/Tk.

This written part of the work will provide theory necessary for understanding operation of this utility. It will elaborate briefly on stages of the synthesis. It will describe relevant structures of the EDuArd system and how to create module for it. And finaly it will also include complete documentation of the utility itself.

Abstrakt

Cílem této práce je vytvořit nástroj využívající struktur systému EDuArd, který by prováděl syntézu sekvenčních obvodů. S přihlédnutím k možnostem systému EDuArd to přesněji bude znamenat překlad z Mealyho automatu na popis na úrovni hradel a z volněji definovaných klopných obvodů. Vývoj nástroje se bude soustředit na vytvoření struktury pro algoritmy syntézy a základního funkčního prototypu. Dalším požadavkem na tento nástroj je existence uživatelského rozhraní ve formě Tcl/Tk.

Tato písemná část poskytne teorii potřebnou k pochopení funkce tohoto nástroje. Bude se stručně zabývat jednotlivými fázemi syntézy. Popíše relevantní struktury systému EDuArd a jak pro něj vytvořit modul. A konečně také obsáhne kompletní dokumentaci nástroje samotného.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Konečné automaty jsou dobrou abstrakcí. Jsou snadno pochopitelné, přesto se jimi dá popsat jakkoli složitý proces. Není pochybu, že když chceme popsat složitější zařízení, zpracovávající vstup na výstup, tak se uchýlíme k popisu formou FSM a jen výjimečně začneme rovnou vymýšlet strukturu hradlel a klopných obvodů.

Problém nastává, když máme hotový popis FSM. Jeho vytvořením jsme si v podstatě jen utřídili myšlenky, ale stále potřebujeme vytvořit elektronické zařízení, o kterém FSM vypovídá jen málo. Převod z FSM je zdlouhavý proces, pokud se ho člověk snaží udělat ručně pro velké množství stavů automatu. Navíc člověk se nebude pokaždé obtěžovat složitějšími optimalizačními algoritmy, a tak na obvod spotřebuje až několikanásobně více součástek. Více součástek znamená větší spotřebu elektrické energie, větší zařízení a také větší cenu. Pokud tedy myslíme náš výrobek vážně, musíme převod z FSM na obvod provádět automatizovaně. Syntetizátor logických obvodů s optimálními algoritmy nepochybně ušetří mnoho prostředků a firmy zabývající se ASIC nebo FPGA takový software často komerčně nabízí.

Systém EDuArd[3], který škola vyvíjí, dokáže uchovávat popis FSM i strukturní hradlový popis, ale zatím žádný z jeho vlastních modulů ani aplikací nedokáže přehledný ale k ničemu jinému nepoužitelný popis FSM převést na hradlový popis, který už má jistou vypovídací hodnotu o tom, jak bude vypadat skutečný hardware. Rozšířit systém o tuto funkci je cílem této práce. Úkolem je připravit platformu pro sofistikované algoritmy syntézy, které pak můžou do systému postupně přidávat další přispěvovatelé. Třešničkou na dortu je rozhraní v Tcl/Tk, takže nejen algoritmy tohoto nástroje budou vyměnitelné, ale bude možné jednoduše vyměnit i GUI.

V této práci se budu zabývat teorií syntézy. Osvětlím použitou podmnožinu systému EDuArd. Nakonec se budu zabývat vlastní syntetizační utilitou.

Kapitola 2

Automaty

2.1 Konečný automat

Konečný automat je druhem Turingova stroje. Má vstup a množinu vnitřních stavů. Obecně platí, že pro každou dvojici vstupu a stavů, kterých konečný automat v daném čase nabývá, lze jednoznačně určit, v jakých stavech se automat ocitne následně. Pro každý vstup, který konečný automat obdrží, také změní množinu stavů ve kterých se právě nachází. Pro definici automatu tedy potřebujeme:

- Konečnou množinu stavů, kterých může nabývat.
- Množinu vstupních symbolů, které umí přijímat.
- Přechodovou funkci, která pro všechny dvojice aktuálních stavů a vstupu určuje následující stav.
- Počáteční stav, ve kterém se automat ocitne, jestliže ho chceme uvést do činnosti.

Tato definice konečného automatu umožňuje tzv. nedeterministický automat, který se vyznačuje tím, že může nabývat více stavů najednou. V této práci se omezíme na deterministické automaty, které se v každém čase během své činnosti smí nacházet pouze v jednom z možných stavu. Důvodem pro to je, že oba automaty řeší stejnou množinu problémů, ale deterministické jsou zásadně jednodušší, a tak se výhradně používají na úkor nedeterministických.

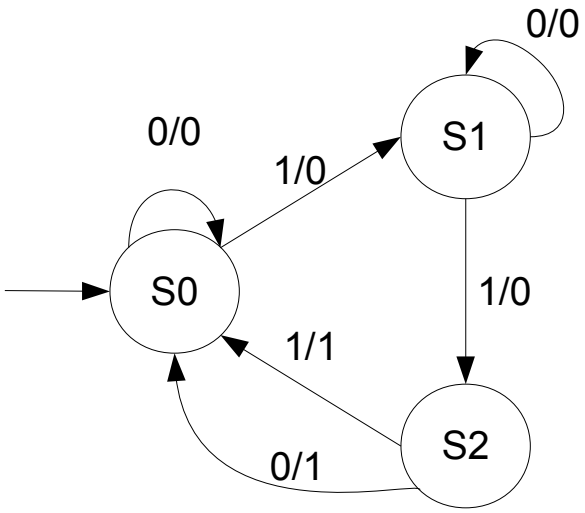
2.2 Mealyho a Mooreův automat

Pro naše účely je dobré také definovat množinu výstupních symbolů a výstupní funkci, kde výstupní funkce ze dvojice vstupního symbolu a aktuálního stavu automatu určí výstupní symbol. Rozšířením deterministického konečného automatu o tyto dva prvky vytvoříme Mealyho automat. Mealyho automat tedy pro každý obdržený vstup a aktuální stav zná výstupní symbol a následující stav.

Mooreův automat narozdíl od Mealyho definuje výstupní funkci závislou pouze na stavu automatu. Nepotřebuje tedy k určení výstupního symbolu znát aktuální vstupní symbol. Z

toho je možné usoudit, že Mooreův automat je speciálním případem Mealyho automatu, kde se pro každý stav ve výstupní funkci nemění výstupní symbol pro všechny různé vstupní symboly. Jakkýkoliv algoritmus, který pustíme na Mealyho automat, bude fungovat i na Mooreův automat s takto upravenou výstupní funkcí. Dále se tedy v tomto textu bude automatem implicitně myslet Mealyho automat.

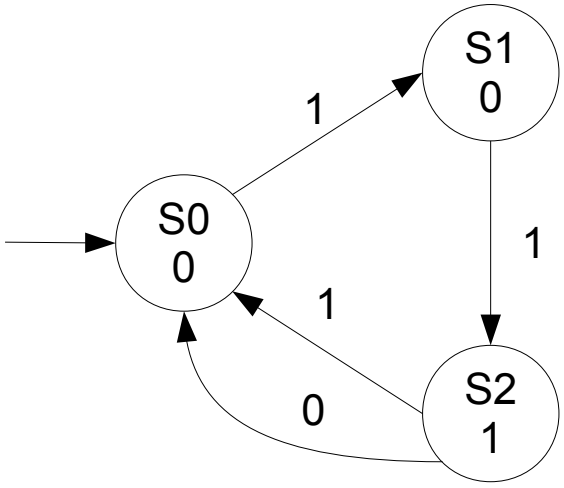
2.3 Reprezentace automatů



Obrázek 2.1: Diagram Mealyho automatu

Mealyho a Mooreův automat se s oblibou reprezentují pomocí diagramu. Mealyho automat se zobrazuje pomocí uzlů reprezentujících stavy a orientovanými hranami mezi nimi reprezentující přechody. Tyto hrany jsou označené vstupním symbolem, který tento přechod vyvolá a výstupním symbolem, pro daný stav a vstup, jak ukazuje obrázek 2.1. Diagram zobrazuje počítadlo do dvou jedniček, kde dalším vstupem se resetuje. Diagram plně reprezentuje Mealyho automat. Najdeme na něm celou množinu stavů, vstupních a výstupních symbolů, definovanou přechodovou i výstupní funkci a počáteční stav. Diagram Mooreova automatu znázorňuje výstupní symbol pod stavem, jak zobrazuje obrázek 2.2. Za povšimnutí stojí, že tentokrát lze s výhodou vynechat z diagramu smyčky s tím, že pokud nějaký vstupní symbol u stavu chybí, pak přechod pro něj stav nemění.

Pro počítačové zpracování budeme s výhodou využívat tabulkový popis automatu. Pro mealyho automat může vypadat například, jak ukazuje tab. 2.1. Běžněji se používá dvourozměrná tabulka přechodů a tabulka výstupů 2.2, kde na jedné ose jsou všechny stavy a na druhé ose všechny vstupní symboly. Obsah buněk je pak následující stav respektive výstupní symbol. Tabulka 2.1 uvádí tu samou informaci kondenzovanou do jedné tabulky a roztaženou do



Obrázek 2.2: Diagram Mooreova automatu

Vstup	Současný stav	Následující stav	Výstup
0	S0	S0	0
1	S0	S1	0
0	S1	S1	0
1	S1	S2	0
0	S2	S2	1
1	S2	S0	1

Tabulka 2.1: Tabulka reprezentující Mealyho automat

řádků. Povšimněte si, že v tabulce chybí informace o počátečním stavu a musí být sdělena zvlášť, abychom měli korektní automat. Případně se za počáteční stav považuje první stav uvedený v tabulce.

2.4 Automaty vzhledem k sekvenčním logickým obvodům

Pokud se z konečného automatu chystáme udělat sekvenční obvod, pak je důležité si uvědomit několik věcí. Zaprvé u automatu si můžeme symboly zvolit, jak chceme. U počítače ale budeme omezeni na znakové řetězce a na úrovni hradel dokonce na pouhá binární čísla, tedy řetězce logických úrovní 0 nebo 1. Zatímco symboly stavů si může syntetizační program přeložit na binární čísla jak chce, symboly vstupů a výstupů by měly vždy od začátku být binární a zadaná od uživatele. Syntetizátor si je nemůže vymyslet, aby u obvodu bylo jasné a následnými syntézami neměnné, co který binární vstup a výstup obvodu znamená.

Druhá zřejmější věc je, že sekvenční obvod bude mít narozdíl od automatu explicitně definovaný synchronizační vstup, který bude obvodu oznamovat, kdy je jeho vstup platný a

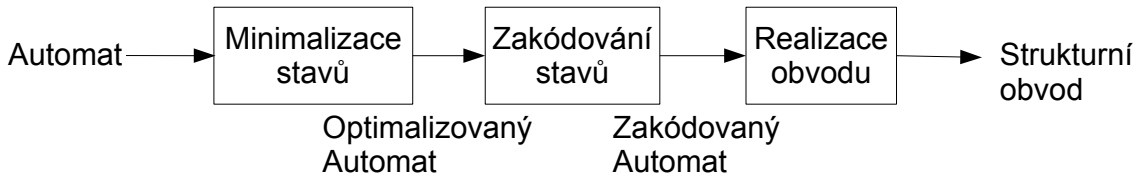
	Vstupy			
Současný stav	0	1	0	1
S0	S0	S1	0	0
S1	S1	S2	0	0
S2	S2	S0	1	1

Tabulka 2.2: Tabulka přechodů a výstupů reprezentující Mealyho automat

ovlivní jeho stav. Dále hotový obvod bude vyžadovat část, která by ho uvedla do počátečního stavu, protože stav paměťových prvků po zapnutí napájení se nemusí automaticky schodovat s počátečním stavem automatu.

Kapitola 3

Syntéza sekvenčních logických obvodů



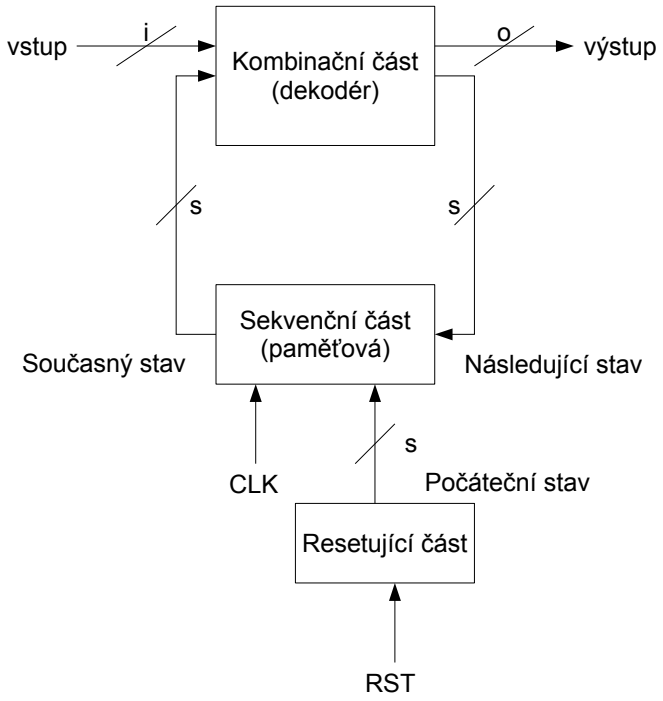
Obrázek 3.1: Blokový diagram průběhu syntézy

Syntéza sekvenčních logických obvodů je proces překladu abstraktního popisu na popis obvodu. Dále tímto termínem budeme myslet překlad z Mealyho automatu na úroveň RTL, tedy úroveň logických hradel a paměťových bloků. Můžeme ho rozdělit na jednotlivé fáze. Rozdělme tedy tento proces na tři fáze. První fází nechť je minimalizace stavů FSM. Druhou nechť je zakódování stavů automatu na binární čísla. Třetí fázi nazvěme realizací zakódovaného automatu. V její rámci dojde k vytvoření obvodu obsahující kombinační a sekvenční část. Touto problematikou se zabývá například staré skriptum ČVUT [1].

3.1 Sekvenční logický obvod

Při syntéze budeme aspirovat k vytvoření obvodu, jehož blokový diagram je na 3.2, kde s bude počet binárních číslic kódu stavů, i bude počet logických vstupů, jež dohromady tvoří binární číslo odpovídající vstupnímu symbolu z teoretického modelu z předchozí kapitoly, o bude se stejnou úvahou počet výstupů.

Hlavní myšlenkou je, že sekvenční část bude reprezentovat stav automatu. Kombinační část bude implementovat přechodovou a výstupní funkci. Chceme-li automat syntetizovat naprosto věrně, musíme přidat také blok, který ho dokáže uvést do počátečního stavu. Zde uděláme v implementaci následující kompromis. Obvod bude po zapnutí napájení pravděpodobně



Obrázek 3.2: Blokový diagram výsledku syntézy

ve špatném stavu a do počátečního stavu se dostane, teprve až obdrží signál na nadbytečném vstupu RST. Jelikož nám jde o vytvoření synchronního obvodu, tak sekvenční část bude synchronizovaná vstupem CLK neboli "hodinami".

3.2 Minimalizace stavů

Při vytváření automatu se snažíme o nejlepší vystihnutí podstaty jeho funkce. Zatímco člověku se může zdát vytvořený automat logický a dobře "čitelný", často bývá neoptimální, obzvláště s přibývajícím počtem stavů. Za optimální označujeme automat, který má co nejmenší počet stavů při plném zachování jeho chování vzhledem ke vstupům a výstupům.

	Vstupy			
Současný stav	0	1	0	1
S0	S0	S1	0	0
S1	S0	S1	0	0
S2	S0	S1	1	0
S3	S1	S0	1	0

Tabulka 3.1: Tabulka přechodů a výstupů ukazující ekvivalentní stavy

	S0	S1	S2
S3	nikdy	nikdy	pokud S0==S1
S2	nikdy	nikdy	
S1	vždy		

Tabulka 3.2: Implikační tabulka pro automat v tab. 3.1

Z toho vyplývá, že pokud nepožadujeme optimální výsledek, pak můžeme tuto fázi syntézy s klidem vynechat.

Neoptimální automat má vždy alespoň dva stavy, které jsou si ekvivalentní. Ekvivalentní stavy jednoduše poznáme z tabulky přechodů a výstupů jak ukazuje tab. 3.1 a její první a druhý řádek. Poznáme je tak, že mají naprosto stejné přechody i výstupy v tabulce. Při optimalizaci všechny stavy z množiny vzájemně si ekvivalentních stavů nahradíme pouze jedním stavem. Tím se mohou objevit další ekvivalence, jak ukazuje třetí a čtvrtý řádek tabulky 3.1.

Pro výuku se často používá algoritmus implikační tabulky. Spočívá v tom, že se vytvoří tabulka, kde jsou buňky pro všechny možné dvojice stavů. Do buněk se vepíše za jakých podmínek by byla daná dvojice stavů ekvivalentní, neboli které stavy si musí být ekvivalentní, aby dvojice této buňky byla také ekvivalentní. V prvním průchodu tabulkou můžeme nepochybně vyloučit dvojice, které mají odlišné řádky ve výstupní tabulce, jako neekvivalentní. V dalších průchodech můžeme vyloučit dvojice, jejichž ekvivalence závisí na již vyloučených dvojicích. Až dojde na to, že po celém průchodu tabulkou nelze vyloučit další dvojice, pak dvojice všech zbylých buněk si je ekvivalentní. Implikační tabulka 3.2 pro příklad z 3.1 objeví obě ekvivalentní dvojice (S0=S1 a S2=S3) při následujícím průchodu, kdy nebude možno žádnou dvojici vyloučit.

Dá se nalézt algoritmus s lepší algoritmickou složitostí než má výše uvedený, například Hopcroftův algoritmus pro minimalizaci.

U automatu se mohou vyskytnout další neoptimality, jako třeba dva ekvivalentní vstupy nebo výstupy či stavy, které jsou nedosažitelné, protože k nim od počátečního stavu nevede žádná cesta. Vzhledem k jejich povaze je lze považovat spíše za chybné zadání, než pouhou neoptimalitu.

3.3 Zakódování stavů

Logické hodnoty jsou dnešním standartem pro reprezentaci signálu na úrovni teorie a překládají se následně na fyzikální signál. Naše sekvenční část se samozřejmě touto konvencí bude držet a bude uchovávat pouze binární čísla, narozdíl od člověka, kterému by spíše vyhovovaly textové identifikátory stavů. Z toho vyplývá, že zakódováním stavů budeme myslet překlad z textu na binární číslo.

Z hlediska funkce výsledného obvodu může být zakódování zcela libovolné. Výběr správného kódu ale zásadně ovlivňuje optimálnost výsledného obvodu. Ovlivňuje počet paměťových prvků v sekvenční části, složitost kombinační části obvodu a i energetickou spotřebu jednotlivých prvků sekvenční části. Tyto vlastnosti, jak jsou uvedeny, jsou důsledkem počtu

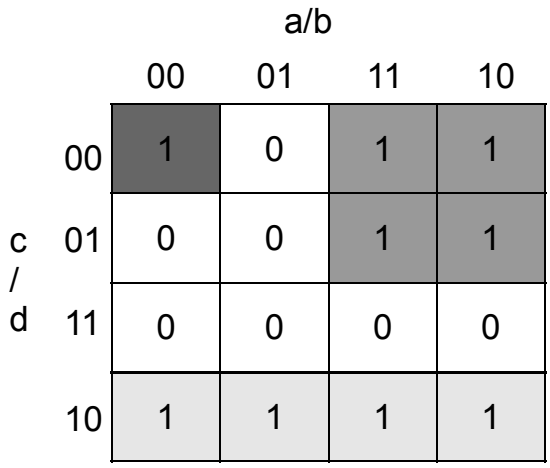
binárních číslic v kódu stavu, snadnosti dekódování kódu stavu na stavy následující a výstup a podobnosti nebo lépe Hammingově vzdálenosti stavu od stavů na které bude přecházet.

Podrobně se jednotlivými algoritmy kódování a jejich výhodami zabývá práce [2].

3.4 Realizace

Zakódovaný automat už vypovídá více o tom, jak bude vypadat výsledný obvod a lze ho s několika dalšími informacemi na něj převést. Realizační algoritmy se můžou lišit tím, jakou architekturu zvolí. Pokud se budou držet blokového schématu 3.2, pak v něm můžou sekvenční část realizovat synchronní s hranou nebo úrovní hodinového signálu, můžou si vybrat různé typy paměťových prvků s různými vstupy. Můžou mít synchroní či asynchronní resetovací signál. Dekodér pak může být realizován prostými hradly, nebo statickou předprogramovanou ROM pamětí.

V algoritmech pro realizaci je další místo pro optimalizaci, tentokrát vzhledem k počtu tranzistorů nebo dostupných součástek obecně. Úroveň hradel neví nic o tom, z čeho a jak jsou jednotlivá hradla vyrobená, ale pokud to víme my, pak určíme algoritmus, který výhradně používá při realizaci kombinační části daný typ hradla na úkor ostatních.



Obrázek 3.3: Karnaughova mapa pro čtyři vstupy

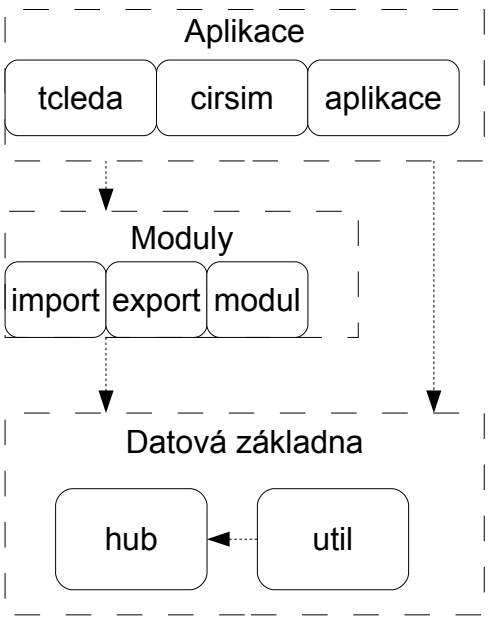
Vytvoření booleovské funkce kombinační části je kapitolou sama pro sebe. Pro ruční určení této funkce se používá Karnaughova mapa. Pro obvod potřebujeme jednu booleovskou funkci pro každý výstup a každou číslici kódu stavu. Realizační algoritmus začíná s pravdivostními tabulkami vytvořenými z tabulky přechodů a výstupů, které pak vyplní Karnaughovu mapu. Z karnaughovy mapy můžeme jednoduše vyčíst jednu booleovskou funkci v konjunktivní nebo disjunktivní normální formě. Pro tabulku na obrázku 3.3 by jsme dosáhli disjunktní formy $a \wedge \neg c + c \wedge \neg d + \neg a \wedge \neg b \wedge \neg c \wedge \neg d$ nebo neoptimálně pro každou jedničku v

karnaughově mapě zvlášť by jsme dosáhli logického součtu devíti mintermů. Pro počítačové zpracování je oblíbenější algoritmus Quine–McCluskey.

Kapitola 4

Systém EDuArd

Systém EDuArd je systémem orientovaným na návrh a simulaci digitálních obvodů. Je to framework nabízející základní struktury pro tyto účely. Součástí systému se za jeho existenci stalo mnoho studentských prací. Hlavní částí systému je datová základna obsahující základní programové struktury. Její součástí je projekt hub a util. Na datové základně jsou závislé moduly a aplikace systému. Aplikace využívají ke svojí práci služeb modulů k obsloužení uživatele. Přibližnou představu o architektuře systému si můžeme udělat z obrázku 4.1, kde šipky zobrazují závislosti.



Obrázek 4.1: Architektura systému EDuArd

4.1 Hub a jeho struktury

Hub obsahuje struktury pro behaviorálně i sturkturně popsaný obvod. Mezi nimi i pro konečný automat. Stejně tak poskytuje organizaci těchto struktur. Konkrétně hub obsahuje tyto hlavní struktury:

- hub Hub, která je kolekcí knihoven
- knihovny Library, které jsou kolekcí entit
- entity Entity, jež reprezentuje šablonu digitálního obvodu a její instanci Instance.
- architektury Architecture, která popisuje funkci digitálního obvodu
- porty PortProto a jejich implementace Port.
- umístění Location, které jednoznačně určuje pozici v hierarchii složené z architektury, entity a knihovny

Vizuální ilustraci vztahů mezi strukturami poskytuje obrázek 4.2. Stuktury knihoven, entit a architektur nejsou ve světě popisu obvodu neznáme. Jsou součástí mnoha jazyků popisu obvodu včetně například VHDL.

Většina těchto stuktur má pro organizaci:

- jméno
- přístup ke své rodičovské struktuře
- množinu struktur, které obsahuje a její iterátor
- funkce pro vytvoření, přidání a vyhledání v dané množině podle jména

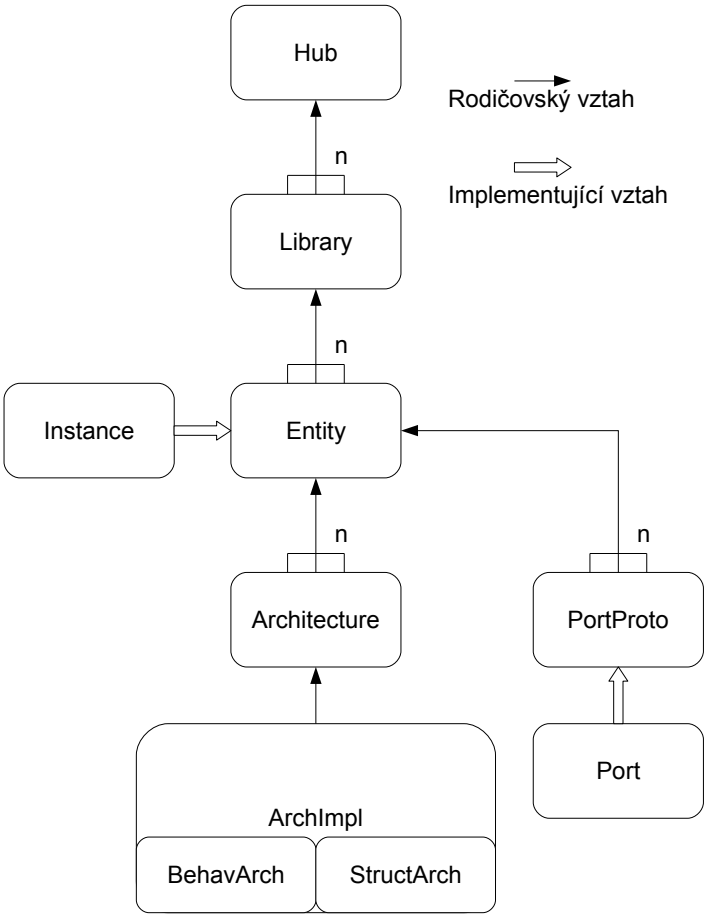
4.1.1 Entity

Entity je šablonou digitálnho obvodu. Obsahuje množinu portů, které reprezentují vstupy a výstupy obvodu. Porty mohou být vstupní, výstupní, obousměrné nebo hodinové. Entita dále obsahuje množinu architektur, z nichž jedna je vybrána jako aktuální. Pokud chceme z entity čili šablony vytvořit komponentu strukturního obvodu, použijeme funkci Instantiate, která nám vrátí strukturu Instance.

4.1.2 Architecture

Architektura Architecture popisuje chování obvodu. Tato třída je víceméně abstraktní a popis chování deleguje na strukturu ArchImpl, která může být typu BehavArch pro behaviorální popis nebo typu StructArch pro strukturní popis. Hierarchii tříd ilustruje obrázek 4.3.

Behaviorální architektura je popsána strukturou BehavDescr. BehavDescr může být typu FsmDescr tedy popisu konečným automatem, typu SymFunc nebo typu RTLReg.

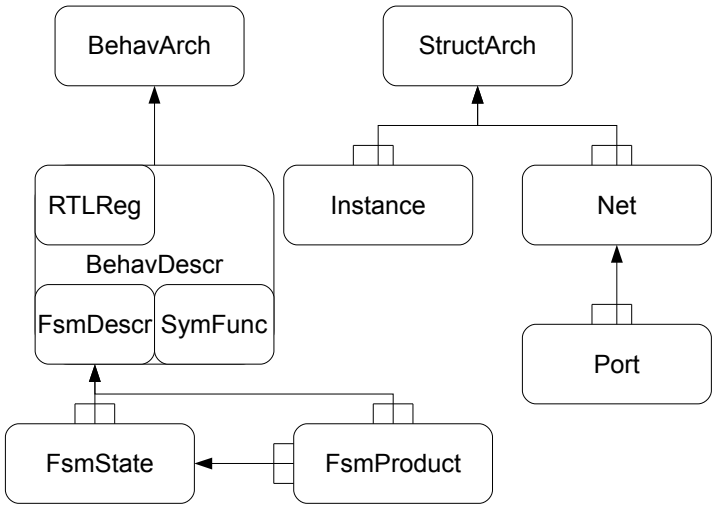


Obrázek 4.2: Hierarchie struktur hubu

Typ FsmDescr má množinu stavů, reprezentované třídou FsmState. Má označený počáteční stav a přechodová a výstupní funkce je implementována třídou FsmProduct. Každý stav FsmState má množinu FsmProductů, které obsahují vstupní symbol, výstupní symbol a následující stav.

Typ SymFunc popisuje symetrické funkce a lze z něj tak vytvořit popis základních kombinačních hradel. Typ RTLReg popisuje registr. Registr je abstrakcí synchronních paměťových prvků. RTLReg blíže nespecifikuje, jak bude synchronizovaný, ani jestli se jedná o klopný obvod typu D nebo jiný. Jeho chování se v systému EDuArd dá odhadnout pouze z toho, jaké má vstupy a výstupy, případně ze jména entity či architektury. V hubu systému EDuArd je tato struktura nejblíže podobná synchronnímu klopnému obvodu a tak jí budeme používat.

Strukturní architektura se skládá z uzlů Net a portů Port. Uzly spojují libovolné množství portů instancí dohromady cestou. V hotovém obvodu tedy bude uzel reprezentovat elektrický vodič spojující porty. Dále se skládá z instancí entit Instance, jejíž porty jsou těmito uzly spojovány. Obrázek 4.4 pro představu ukazuje, jak by v systému EDuArd vypadala entita



Obrázek 4.3: Hierarchie struktur ArchImpl

implementující funkci AND dvěma hradly NAND.

4.2 Util a logování

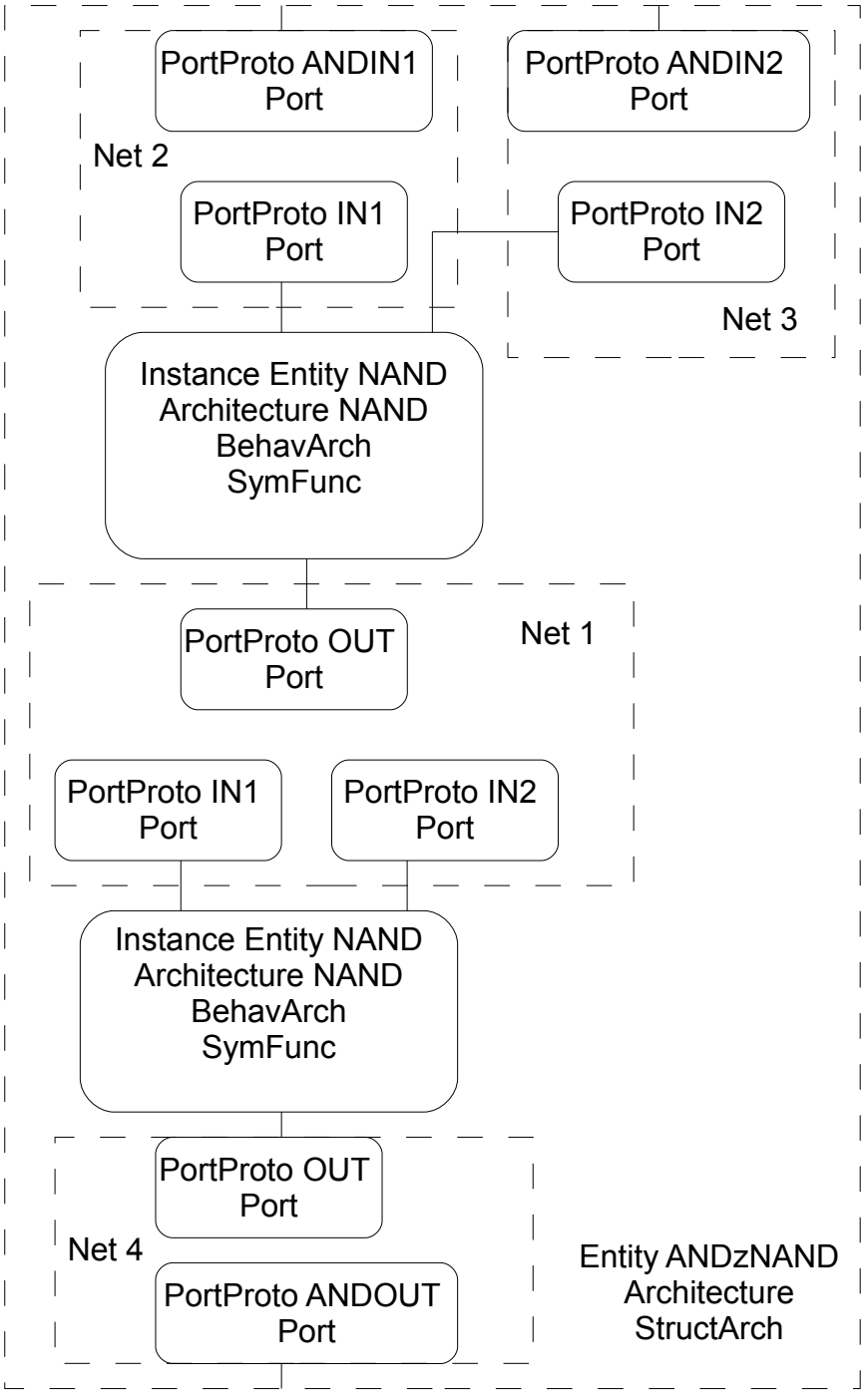
Projekt util definuje čtyři streamy, na které je možné posílat zprávy z aplikací. Jsou to logStream, warnStream, errStream a infStream. V souhře se strukturami vyjímek definovanými v souboru err projektu hub tvoří základní společný systém sloužící všem aplikacím a modulům pro oznamování událostí uživateli.

4.3 Moduly pro systém EDuArd

Datová základna systému EDuArd není nic jiného než statická knihovna, psaná v jazyce c++. A moduly pro tento systém nejsou samy nic jiného než statické knihovny využívající služeb datové základny.

Pro moduly v systému EDuArd platí jisté nepsané konvence. Zdroje systému se podávají jako solution visual studia a moduly i aplikace jsou jeho projekty. Moduly by měly dodržovat adresářovou strukturu systému. Složka jejich projektu by měla být umístěna v podsložce modules. Dále, zkompilované aplikace by měly přijít do podsložky bin a knihovny do podsložky lib. Pro jejich dokumentaci je připravena složka doc.

Jeden modul má většinou na starosti jednu atomickou úlohu. Samotný modul se skládá z třídy, která má přetížený operátor funkce operator(), který bere argumenty potřebné k vykonání úlohy modulu. Aplikace které pak takový modul používají si vytvoří instanci jeho třídy a zavolají onen přetížený operátor funkce.

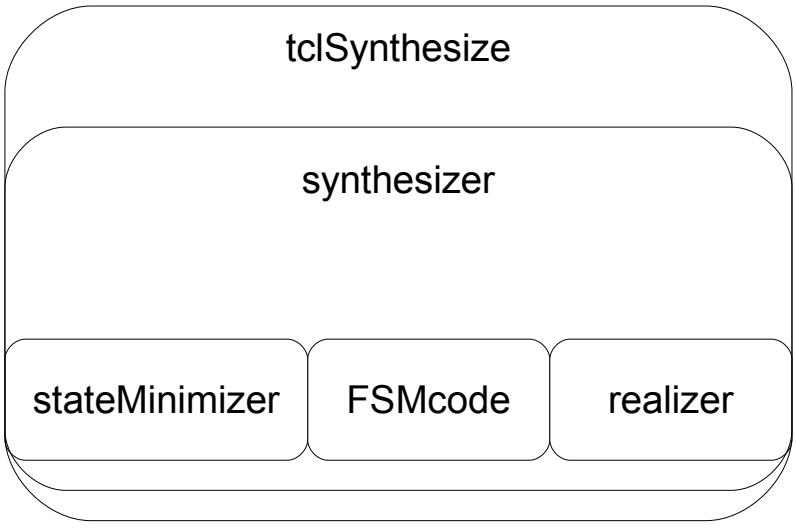


Obrázek 4.4: Příklad jednoduchého obvodu v systému EDuArd

Kapitola 5

Implementace syntetizačního nástroje

Implementace nástroje je vyvíjena v Microsoft Visual Studiu 2008. Systém EDuArd byl ale vyvíjen v systému Microsoft Visual Studiu 2005. Verzi 2008 jsem si vybral hlavně proto, že s ní mám delší zkušenosti a jsem na ní zvyklý a také proto, že v době vývoje byla dostupnější. Microsoft na svých stránkách verzi 2005 zdarma nenabízí a ČVUT FEL nabízí svým studentům také jen verzi 2008 a novější. S tím ale přibývá další komplikace. Je potřeba konverze struktury projektů systému EDuArd na verzi 2008, jestliže chceme novou součást do této struktury zařadit. Nástroj se bude nazývat "synthesize"nebo "tclSynthesize".



Obrázek 5.1: Hierarchie součástí nástroje tclSynthesize

Hierarchie součástí nástroje je na obrázku 5.1. Implementace přidává do systému tři nové moduly a jednu aplikaci:

- modul stateMinimizer, implementující fázi minimalizace stavů
- modul realizer, implementující fázi realizace obvodu
- modul synthesizer, zastřešující stateMinimizer, realizer a zapůjčený modul FSMcode z práce [4]
- aplikaci tclSynthesize

5.1 Moduly implementace

Moduly vždy obsahují hlavičku Algorithms.h obsahující výčet použitelných algoritmů příslušné fáze syntézy. Způsob rozšiřování tohoto výčtu popisuje příloha A. Dále pak obsahují samotný modul ve formě třídy s přetíženým operátorem funkce. Tento přetížený operátor bere zdrojovou a cílovou lokaci v hubu. Dále bere jeden identifikátor algoritmu z výčtu a deleguje vypracování své fáze na danou implementaci algoritmu. Modul synthesize rozšiřuje modul FSMcode o tento výčet algoritmů. Výčet algoritmů je zaprvé v té nejjednodušší formě a to enum, pro výběr implementace algoritmu programem. Zadruhé je výčet algoritmů složitější strukturou, která obsahuje jeho jméno jako řetězec a i jeho popis. Tato struktura je orientovaná na podporu uživatele systému, třeba například přes Tk GUI.

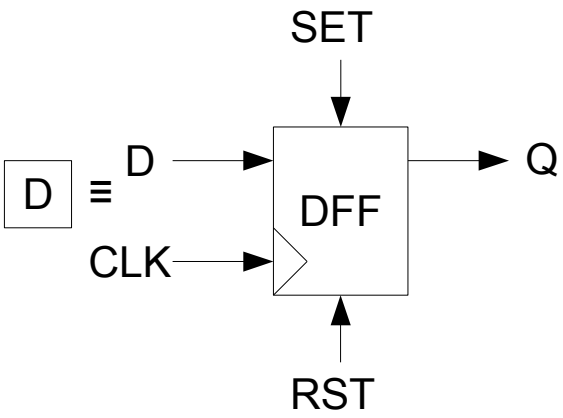
Modul synthesizer by se teoreticky dal také rozdělit na různé algoritmy, ale my ho rozdělovat nebudeme, protože od něj vždy budeme chtít pouze spuštění tří fází syntézy. Výběr algoritmu sytézy tedy bude pro nás znamenat pouze výběr tří algoritmů pro jeho jednotlivé fáze. Zatímco se nevylučuje samostatné použití jednotlivých modulů, doporučuje se používat zastřešující modul synthesizer pro provedení komplétní syntézy, protože zajišťuje lepší ošetření špatných vstupů uživatele.

Modul stateMinimizer zatím obsahuje jen jeden algoritmus a to algoritmus naprosto nejjednoduší potřebný k úspěšnému dokončení syntézy. Tento algoritmus neodstraňuje vůbec žádné stavy a jeho efektem je vlastně jen zkopírování FSM z zdrojové lokace do cílové lokace.

Modul FSMcode, jež je již součástí systému EDuArd z dřívějška a nyní je rozšířený o výčet algoritmů. Tento modul již obsahuje slušné množství algoritmů pro zakódování stavů.

5.2 Realizační algoritmus obvodu

Modul Realizer obsahuje v podstatě jednoduchý algoritmus, který se drží návrhu z obrázku 3.2 a vytvoří obvod ilustrovaný obrázkem 5.3. Tento algoritmus vytvoří sekvenční část z registrů typu D 5.2 se vstupem SET a RST, které nastavují hodnotu na logickou 1 respektive 0. Funkce entity registru typu D je popsána strukturou hubu RTLReg. Problém s tím je, že funkce registru se dá odhadnout pouze z popisu jeho portů a podoba výsledného obvodu exportovaného do souboru může být silně závislá na interpretaci tohoto popisu daným exportním modulem. Alternativou by bylo vytvořit tento paměťový element z hradel NAND.



Obrázek 5.2: Forma paměťováho elementu používaná algoritmem realizace

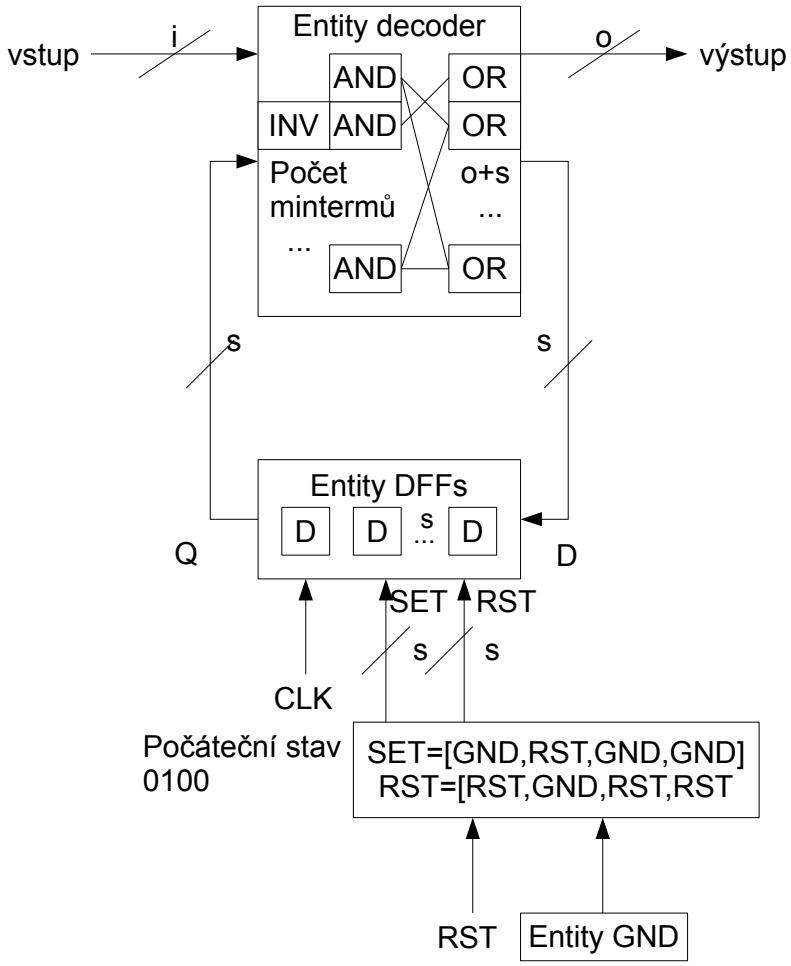
Resetovací obvod je naprosto jednoduchý. Podle toho jestli číslice kódu stavu je log. 0 nebo 1, tak připojí vstup RST na vstup RST respektive SET sekvenční části. Na zbylé vstupy SET nebo RST sekvenční části připojí entitu GND, jejíž popis je struktura hubu SymFunc a která má jeden výstup posílající vždy jen log. 0.

Kombinační část je tvořená invertory, ANDy a ORy implementovanými strukturou SymFunc. ANDy mají počet vstupů rovný počtu vstupů dekodéru. ORy mají počet vstupů rovný počtu řádků tabulky typu 2.1. Na každý výstup dekodéru se připojí jeden OR a na tento OR se připojí mintermy určené logickými 1 ve výstupní tabulce. Tyto mintermy jsou podle potřeby vytvářeny ANDy a invertory. Umožníme uživateli také na vstupu používat speciální logickou úroveň dont care reprezentovanou znakem -. V takovém případě se místo vstupu do mintermu připojí entita VCC neovlivňující výsledek logického součinu. Nevyužité vstupy ORu se připojí na entitu GND.

5.3 Aplikace tclSynthesize

Projekt tclSynthesize se inspiruje existující aplikací systému EDuArd tcleda. Přidává k modulu synthesizer Tcl wrapper, který k příkazové řádce Tcl přidává příkazy synthesize, getStateMinimizationAlgorithms, getEncodingAlgorithms a getRealizationAlgorithms, které zmiňuje příloha B. Aplikace přidává také příkazy ovládající moduly import a export systému. Využívá k tomu již napsaný wrapper pro tyto moduly v aplikaci tcleda. Běžné použití aplikace bude, že uživatel použije import pro přenesení FSM z netlistu v souboru do systému. Když se tak uloží do hubu, uživatel na něj použije příkaz synthesize a následně ho zase exportuje do souboru příkazem export.

Aplikace tclSynthesize interpretuje skript který je mu předán jako parametr při spuštění. Implicitně se spustí skript v souboru pojmenovaném GUI.tcl, který obsahuje jednoduché a přehledné grafické rozhraní. Tento skript je manifestací užitečnosti používání technologie Tcl/Tk v aplikacích. Tento rozsáhlý framework by sám stačil na vytvoření další bakalářské



Obrázek 5.3: Přibližný popis struktur realizovaného obvodu

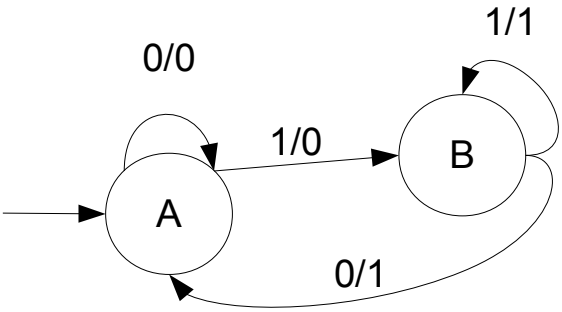
práce a zde se s ním nebudeme více zabývat. O používání aplikace pojednává uživatelský manuál B.

Kapitola 6

Testování implementace

Testy jsou umístěny v podsložce tests projektu. Všechny se provedou se spuštěním tests.bat.

6.1 Základní test funkce



Obrázek 6.1: Jednoduchý automat pro test funkce syntézy

Otestujeme aplikaci na naprosto jednoduchém automatu v souboru easy.kiss2. Pro jednoduchost a větší názornost budeme exportovat do souboru s formátu dbg. Formát dbg je formát systému EDuArd a umožňuje spolehlivě zobrazit obsah interních struktur hubu. Jak vypadá testovaný automat zobrazuje obr. 6.1. Použijeme onehot kódování.

Ve výsledku vidíme původní automat, zakódovaný automat a primitiva potřebná pro výsledný obvod. Stav A je zakódovaný jako 01, stav B jako 10. Stav A je počáteční stav. Když se podíváme na výslednou architekturu seq uvidíme, že je instancována sekvenční i kombinační část a jsou správně propojené. V rámci resetování stavů je správně připojen GND a RST vstup na RST a SET sekvenční části, efektivně nastavující 01 při zmáčknutém RST. Vidíme že sekvenční entita DFF2 je dobře vygenerovaná, složená z dvou prvků DFF připojených na vstupy a výstupy entity.

Nejdelší je popis dekodéru i pro takto jednoduchý automat. Pečlivým pročtením uzlů net níže zjistíme že dekodér odpovídá tabulce 6.1.

Vstup	Současný stav	Následující stav	Výstup
0	01	01	0
1	01	10	0
0	10	01	1
1	10	10	1

Tabulka 6.1: Zakódovaná tabulka pro dekodér

```
+net andinput0-0 IN[2].minterm0, OUT.invininput0-0
-net andinput0-0
+net andinput0-1 IN[2].minterm1, IN[0]
-net andinput0-1
+net andinput0-2 IN[2].minterm2, OUT.invininput0-2
-net andinput0-2
+net andinput0-3 IN[2].minterm3, IN[0]
-net andinput0-3
+net andstate0-0 IN[0].minterm0, OUT.invinstate0-0
-net andstate0-0
+net andstate0-1 IN[0].minterm1, OUT.invinstate0-1
-net andstate0-1
+net andstate0-2 IN[0].minterm2, INSTATE[0]
-net andstate0-2
+net andstate0-3 IN[0].minterm3, INSTATE[0]
-net andstate0-3
+net andstate1-0 IN[1].minterm0, INSTATE[1]
-net andstate1-0
+net andstate1-1 IN[1].minterm1, INSTATE[1]
-net andstate1-1
+net andstate1-2 IN[1].minterm2, OUT.invinstate1-2
-net andstate1-2
+net andstate1-3 IN[1].minterm3, OUT.invinstate1-3
-net andstate1-3
```

Z popisu dekodéru je jasné, že plýtvá uzly na vstupy, plýtvá invertory a zbytečně vytváří mnohavstupové ORy, ale jako základní algoritmus postačuje.

Použijeme automat easy.kiss2 s větším počtem vstupů a výstupů a s použitím logické úrovně dont care. Binárním kódováním docílíme toho, že bude prozměnu méně kódů stavu. Znovu se vše zdá být v pořádku. Níže vidíme, že dont carey byly správně zpracovány.

```
+net andinput0-4 IN[1].minterm4, OUT.decVCC
-net andinput0-4
+net andinput1-4 IN[2].minterm4, OUT.decVCC
-net andinput1-4
```

6.2 Náročný test

Použijeme automat bbara.kiss2, který je součástí distribuce systému EDuArd a je poměrně složitý. Z výsledků se zdá, že sekvenční část, vygenerovaná primitiva a resetovací obvod jsou správně. V tomto případě ruční kontrola vygenerovaného dekodéru je nad lidské síly.

6.3 Test exportu do vhdl

Chceme-li dosáhnout užitečného výsledku, pak je dobré mít funkční export do zavedeného formátu vhdl. Zkusíme tedy první příklad s easy.kiss2 exportovat do vhdl.

```
architecture AND3architecture of AND3 is
  begin
    O <= I1 and I2 and I3;
  end architecture AND3architecture;
```

Výsledek je celkem negativní. Z kódu výše je vidět, že funkci některých primitiv exportní modul sice vytvořil správně, ale jméno vstupů a výstupů ignoroval a vymyslel si vlastní. Paměťový element DFF, GND a VCC nemají popis vůbec. Modul při exportu entity sekvenční nebo kombinační části z neznámých důvodů spadne. Nejlépe na tom je export nejvrchnější entity výsledného obvodu, která se zdá být s překvapením naprosto správně vyexportovaná.

6.4 Test výpisu algoritmů

Další test zkouší funkci výpisu algoritmů přes příkazy typu ”eda synthesize getEncodingAlgorithms”. Dá se říct, že tato část programu funguje bezchybně, jak ukazuje výpis z příkazové řádky:

```
noMinimization super
own binary onehot s_onehot zerohot s_zerohot random brute twohots johnson
NaiveRTLDFFbased
```

Kapitola 7

Závěr

Podařilo se vytvořit platformu pro algoritmy syntézy v rámci systému EDuArd. Platforma je v duchu velice simplistická, ale naprosto postačující po dobu, kdy systém budou využívat zkušenější uživatelé a kdy zdrojový kód systému bude volně dostupný.

Podařilo se vytvořit také aplikaci s rozhraním Tcl provádějící syntézu. Jako demonstrace síly vyjadřovací schopnosti skriptovacího jazyka Tcl bylo pro tuto aplikaci nad rámec zadání práce vytvořeno jednoduché grafické uživatelské rozhraní. Toto GUI odhaluje možnosti této platformy, když dynamicky dává na výběr z algoritmů, které jsou momentálně v systému. Aplikace dodává základní, silně neoptimální algoritmy syntézy. Testy naznačují, že tyto algoritmy jsou korektní. Užitečnost aplikace je ale bohužel momentálně omezena možnostmi exportního modulu systému EDuArd, jež má problém vyexportovat části obvodu, ozvlášť jeho sekvenční prvky.

Rozvoj systému EDuArd se zdá dost pomalý, protože toto je jediná studenská práce za poslední dva roky, která ho obohacuje. Nicméně má-li se systém rozvíjet, bylo by dobré vylepšit exportní modul a následně přidat sofistikované algoritmy syntézy do nových modulů z této práce. Výsledkem by byla už skutečně hotová aplikace, která by dokázala převést netlisty konečného automatu na netlisty optimalizovaného strukturního obvodu jako je oblíbený formát VHDL.

Literatura

[1] BLAžEK, K. H. Z. *Logické systémy.* Vydavatelství ČVUT, 1996.

[2] EGERT, J. *Algoritmy pro kódování stavů konečného automatu.* Vydavatelství ČVUT, 2007.

[3] Ing. Jan Schmidt Ph.D. a Ing. Petr Fišer Ph.D. a kolektiv studentů. EDuArd, 2007. EDuArd - příručka programátora.

[4] Jan Egert. fsmcode, 2007. modul systému EDA pro kódování stavů.

Příloha A

Vývojářská příručka

A.1 Úvod

Řešení Synthesize usnadňuje programátorům přidávání algoritmů týkajících se syntézy. Algoritmy se přidávají jako jednoduché c++ funkce. Je možné přidat tři typy algoritmů:

- Algoritmus minimalizace stavů, který musí umět zpracovat zdrojový objekt FsmDescr a vrátit do cílového FsmDescr minimalizovaný automat.
- Algoritmus zakódování stavů, který musí umět zpracovat zdrojový objekt FsmDescr a přidat kódy stavů. Přidání algoritmu kódování je nepatrně odlišné, protože projekt fsmcode [4] je implementován z dřívějška jiným autorem.
- Algoritmus realizace syntézy, který musí umět zpracovat zdrojový objekt FsmDescr se zakódovanými stavy a vytvořit strukturní obvod do StructArch.

Každý typ algoritmu je v řešení implementován jako projekt. Je tak teoreticky možné provádět jednotlivé fáze syntézy odděleně. V souborech, které je třeba upravit pro přidání algoritmu je to výrazně naznačeno v komentářích a vždy obsahují alespoň jeden příklad.

A.2 Požadavky

- Ideálně Systém Windows s Visual studio 2008 ve kterém jsou zdroje vytvořeny a který je potřeba pro jejich zkompilování s nastavením v sln souboru.
- Přístupné statické knihovny a hlavičky projektů EDuArdu util, hub, import, export, fsmcode a tcleda - POZOR vyžaduje aplikaci tcleda jako .lib a ne .exe, protože využívá její část přidávající tcl k modulu fsmcode.
- Přístupné statické knihovny a hlavičky zlib(zdll.lib), Tcl(tcl85.lib) a Tk(tk85.lib). Doporučuje se zlib verze 1.2.3 a Tcl\Tk 8.5.
- Pro spuštění jsou potřeba dynamické knihovny zlib1.dll, Tcl85.dll a Tk85.dll.

A.3 Přidání algoritmu

Přidání algoritmu probíhá ve čtyřech krocích:

- V následujících krocích přidávejte své části vždy před prvky se jménem ”last”.
- Do Algorithms.h projektu daného typu algoritmu zadáme do enumerace ”typ”ID jeho identifikátor. Kódovací algoritmy mají tento soubor v projektu Synthesizer pojmenovaný FSMEncodeAlgorithms.h .
- V témže souboru přidáme do pole ”TYP”_ALGORITHMS_CHOICE instanci ”typ”AlgorithmIdentificator, jejíž konstruktor bere jako parametry dlouhý popis, stringový identifikátor a identifikátor, který jsme zadali do enumerace. Nezapomeňte zakončovat řádky \ - jedná se o makro preprocesoru.
- Do souboru a třídy modulu, kam se snažíme algoritmus přidat, přidáme privátní funkci implementující tento algoritmus. Samotná Implementace algoritmu může být klidně uplně jinde, musí jí ale být možno z této třídy zavolat.
- Do stejného souboru, přidáme do přetíženého operátoru() třídy, do switch rozhodování nový case s identifikátorem algoritmu z enumerace volající naší novou funkci.

Přidáním algoritmů tímto způsobem se tak stanou automaticky i součástí výběrů v GUI.

Například výsledný kód s přidaným minimalizačním algoritmem ”pridany”vypadá takto:

Algorithms.h

```
enum StateMinimizationAlgorithmID { noMinimization, pridany, lastM };

#define STATE_MINIMIZATION_ALGORITHMS \
const StateMinimizationAlgorithmIdentificator STATE_MINIMIZATION_ALGORITHMS_CHOICE[]\
  = { StateMinimizationAlgorithmIdentificator( "without state minimalization", "noMinimizatio
      StateMinimizationAlgorithmIdentificator( "pridany algoritmus", "pridany", pridany ),\
      StateMinimizationAlgorithmIdentificator( "end of array", "lastM", lastM )\
    };
```

StateMinimizer.h

```
class StateMinimizer{
  hub::Hub* theHub;

  void doNoMinimization( const hub::FsmDescr* srcFSM, hub::FsmDescr* destFSM );
  void pridany( const hub::FsmDescr* srcFSM, hub::FsmDescr* destFSM );
  ...
}
```

StateMinimizer.cpp

```
operator()(...){
  ...
  switch( algorithmID ){
    case noMinimization: this->doNoMinimization( srcFSM, destFSM ); break;
    case pridany: this->pridany( srcFSM, destFSM ); break;
  }
}

void StateMinimizer::pridany( const hub::FsmDescr* srcFSM, hub::FsmDescr* destFSM ){
  //implementace
}
```

A.4 Kompilace

V mém Solution je nastavená jak Debug, tak i Release konfigurace. Solution poskytuje Property sheets visual studia paths a debug, které nastavují cesty k závislostem z požadavků. Před kompilací je potřeba je nastavit, aby odpovídaly danému prostředí.

Příloha B

Uživatelská příručka

B.1 Úvod

Program tclSynthesize.exe provádí syntézu sekvenčních logických obvodů. Umí interpretovat Tcl skript, nebo poskytuje Grafické rozhraní, které je taky skriptem v souboru GUI.tcl. Systém EDA v současné době importuje popisy obvodů z mnoha formátů, ale většinou zná jenom podmnožinu jazyka daného formátu. Nejspolehlivější je importovat z .kiss souboru a exportovat do debugovacího .dbg souboru.

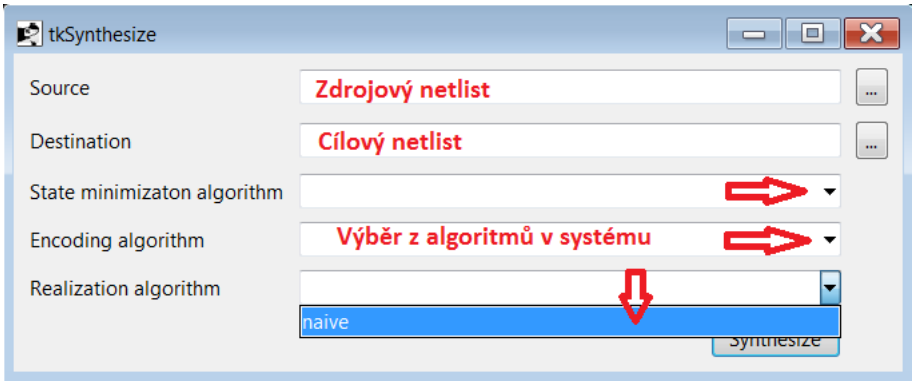
B.2 Požadavky

- Tento program nevyžaduje žádnou instalaci
- Systém Windows
- skript GUI.tcl definující GUI ve složce programu
- přístupnou knihovnu zlib1.dll ze složky programu
- přístupné knihovny Tcl, například instalací ActiveTcl

B.3 Spuštění

Program tclSynthesize.exe lze spustit:

- Bez parametru, nebo s nesmyslnými parametry. Takto se program spustí s grafickým rozhraním, které usnadní práci. Grafické rozhraní používá Tcl skript GUI.tcl, který musí být ve stejném adresáři s tclSynthesize.exe.
- S jedním parametrem a to s cestou a jménem Tcl skriptu, který chceme spustit. Například tclSynthesize.exe skript.tcl. Program v konzoli informuje o průběhu syntézy.



Obrázek B.1: Grafické rozhraní programu tclSynthesize

B.4 Grafické rozhraní

Obrázek B.1 ukazuje vzhled grafického rozhraní po spuštění a naznačuje jeho použití. V okně grafického rozhraní musíme vybrat zdrojový a cílový soubor. Můžeme si pomoci stisknutím trojtečky, která vyvolá dialog výběru souboru. Dialogy obsahují filtr přípony souboru. I v dialogu výběru musíme zadat příponu v názvu souboru, pokud vytváříme nový soubor. Je možné zadat i příponu, kterou program nenabízí a zkusit zda jí systém EDA dokáže importovat, či exportovat. POZOR - import či export mnoha formátů není v systému plně implementován obzvláště pro sekvenční obvody. .

Dále je třeba vybrat z výběru algoritmus pro každou fázi syntézy. Pak stačí už zmáčknout tlačítko Synthesize. Pokud nastal problém zobrazí se zpráva s popisem chyby. Pokud je vše v pořádku, zobrazí se zpráva informující o tom, s jakými parametry byla syntéza spuštěna a vytvoří se výsledný soubor.

B.5 Spouštěcí skript

Program je samozřejmě možné ovládat tcl skripty přímo. Tento program rozšiřuje jazyk Tcl o základní příkazy:

- eda import zdrojový_soubor cílové_umístění
- eda export cílový_soubor zdrojové_umístění ; Pokud chceme exportovat celý strom hubu do .dbg souboru stačí zadat eda export cílový_soubor.dbg
- eda synthesize zdrojové_umístění cílové_umístění minimizační_algoritmus kódovací_algoritmus realizační_algoritmus

a o příkazy vypisující seznam algoritmů:

- eda synthesize getStateMinimizationAlgorithms
- eda synthesize getEncodingAlgorithms

- eda synthesize getRealizationAlgorithms

Při spuštění konzolového rozhraní se musí uživatel postarat o import a export sám. Spouštěcí skript bude tedy nejčastěji vypadat například takto:

```
eda import fsm.kiss2 main:work..fsm
eda synthesize main:work..fsm work.circuit.seq noMinimization binary naive
eda export fsm.dbg work.circuit.seq
exit
```

Tento skript importuje konečný automat fsm.kiss2 do knihovny work, entity odvozené od názvu souboru a architektury fsm. Následně ho syntetizuje a výsledek nalezneme v knihovně work, entitě circuit a architektuře seq. Syntéza se provede bez minimalizace stavů, provede se binární kódování a realizuje se algoritmem naive. Výsledná architektura seq v entitě circuit se uloží do fsm.dbg .

Příloha C

Seznam použitých zkratek

EDA Electronic design automation neboli systém pro automatizovaný vývoj elektronických systémů

EDuArd Projekt ČVUT FEL snažící se o vytvoření EDA systému

Tcl/Tk Tool Command Language - skriptovací jazyk a framework pro tvorbu uživatelských rozhraní s domovskou stránkou www.tcl.tk

FSM Finite State Machine neboli konečný automat

GUI Graphical User Interface neboli grafické uživatelské rozhraní

ASIC Application specific integrated circuit neboli integrovaný obvod pro konkrétní aplikace

FPGA Field programmable gate array aneb programovatelné pole hradel

RTL Register transfer level tedy úroveň přenosu signálů s registry

Příloha D

Obsah přiloženého CD

readme.txt
text/ - obsahuje bakalářkou práci v pdf
latex/ - obsahuje latexové zdroje bakalářské práce
latex/figures - obsahuje obrázky použité v práci
synthesize/ - obsahuje implementaci práce
synthesize/apps/ - obsahuje zdroje aplikace tclSynthesize
synthesize/modules/ - obsahuje zdrojové kódy modulů
synthesize/bin/ - obsahuje zkompilovanou aplikaci tclSynthesize
synthesize/lib/ - obsahuje zkompilované knihovny
synthesize/tests/ - obsahuje testy aplikace z příslušné kapitoly