České vysoké učení technické v Praze

Fakulta elektrotechnická

Bakalářská práce

# Algoritmy pro kódování stavů konečného automatu

*Jan Egert*

Vedoucí práce: Ing. Petr Fišer

Studijní program: Elektrotechnika a informatika strukturovaný bakalářský

Obor: Informatika a výpočetní technika

červen 2007

## Poděkování:

Chtěl bych poděkovat Ing. Petru Fišerovi za jeho cenné informace, postřehy a pomoc při shánění studijních materiálů. Dále Ing. Janu Schmidtovi, Ph.D. za pomoc při integraci algoritmu do EDA systému a Doc. Ing. Haně Kubátové, CSc. za informace o algoritmu dříve vyvinutém na katedře počítačů.

# Prohlášení

Prohlašuji, že jsem svou bakalářskou 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).

Ve Vodňanech dne .................................. ............................................

# Abstract

In this document, I'm trying to present various methods of FSM state assignment from the easy ones to more difficult. My target was to create simple, fast but effective algorithm for state assignment of FSM. The algorithm is trying to reach the lowest average bit (Hamming) distance between each pair of states in transitions, which should lower the switching activity at the latches of sequential circuit and it's power dissipation. Some of the simple methods were also implemented. The result of this work will be used in EDA system EDuArd being developed at the department of computer science at CTU in Prague.

# Abstrakt

V tomto dokumentu se zabývám různými metodami kódování stavů FSM od jednoduchých po složitější metody. Mým cílem bylo vytvořit jednoduchý, rychlý a přesto efektivní algoritmus pro zakódování FSM. Algoritmus se snaží o co nejnižší průměrnou bitovou (Hammingovu) vzdálenost mezi jednotlivými přechody v FSM, což by mělo snížit celkovou „switching activity“ na klopných obvode sekvenčního obvodu a tím i jeho spotřebu. Kromě tohoto algoritmu jsem do programu začlenil i některé jiné základní metody. Výsledek práce bude využit v EDA systému EDuArd vyvíjeném na katedře počítačů ČVUT.

# Seznam obrázků


| obrázek | strana | popis |
|---|---|---|
| obr. 3.1.1 | 15 | schéma FSM typu Moore, K=kombinační logika, FF=flip-flop |
| obr. 3.1.2 | 16 | schéma FSM typu Mealy, K=kombinační logika, FF=flip-flop |
| obr. 3.2.3 | 17 | Moorův FSM znázorněný stavovým stromem |
| obr. 3.2.4 | 17 | Mealyho FSM znázorněný stavovým stromem |
| obr. 3.2.5 | 18 | Moorův FSM znázorněný stavovým diagramem (STG) |
| obr. 3.2.6 | 18 | Mealyho FSM znázorněný stavovým diagramem (STG) |

# Seznam tabulek


| tabulka | strana | popis |
|---|---|---|
| tab. 3.2.1 | 17 | Moorův FSM zadaný tabulkou ($x_0$=vstup, sloupec pod ním přechod do vybraného stavu) |
| tab. 3.2.2 | 17 | Mealyho FSM zadaný tabulkou ($x_0$=vstup, sloupec pod ním přechod do vybraného stavu nebo výstup) |
| tab. 4.1 | 20 | tabulka stavů zakódovaná binárním kódem |
| tab. 4.2 | 21 | tabulka stavů zakódovaná Grayovým kódem (postupně -> čítač) |
| tab. 4.3 | 21 | tabulka stavů zakódovaná one-hot kódem |
| tab. 4.4 | 22 | tabulka stavů zakódovaná zero-hot kódem |
| tab. 4.5 | 22 | tabulka stavů zakódovaná Johnsonovým kódem (postupně -> čítač) |
| tab. 4.6 | 23 | tabulka stavů zakódovaná kódem m-n |
| tab. 4.7 | 23 | tabulka stavů zakódovaná two-hots kódem |
| tab. 7.1.1a | 34 | proměnné třídy MyState v mé metodě |
| tab. 7.1.1b | 35 | funkce třídy MyState v mé metodě |
| tab. 7.1.2a | 35 | proměnné třídy MyFSM v mé metodě |
| tab. 7.1.2b | 35 | funkce třídy MyFSM v mé metodě |
| tab. 7.1.3a | 36 | funkce třídy MyEncUtils v mé metodě |
| tab. 7.1.4b | 36 | funkce třídy fsmEncoder v mé metodě |
| tab. 8.1 až tab. 8.20 | 37 až 42 | výsledky testů na jednotlivých benchmarcích (*method*=použitá metoda, *bits*=počet bitů stavu, *avg. H*=prům. Hammingova vzdálenost přechodů, *nodes*=počet hradel, *lits*=počet literálů, *power*=spotřeba, *time*=doba kódování danou metodou |
| tab. 8.21 | 42 | celkový souhrn výsledků testů |

# Seznam zkratek

| zkratka | význam |
|---|---|
| ČVUT | České vysoké učení technické v Praze |
| FEL | Fakulta elektrotechnická |
| CTU | Czech Technical University in Prague |
| EDA | electronical design aid |
| FSM | finite state machine |
| STG | state transition graph |

# Obsah

Seznam obrázků 8

Seznam tabulek 9

Seznam použitých zkratek 10

Obsah 11

1. Úvod 13
2. Popis problému, specifikace cíle 14
3. Definice použitých pojmů 15
   3.1. FSM – finite state machine 15
      3.1.1. Reprezentace FSM 16
   3.2. Hammingova vzdálenost 18
   3.3. Pravděpodobnosti přechodů 18
   3.4. Cenová funkce 19
   3.5. Ztrátový výkon 19
4. Základní a jednoduché kódovací algoritmy 20
   4.1. Binární kódování 20
   4.2. Grayovo kódování 20
   4.3. One-hot kódování 21
   4.4. Zero-hot kódování 22
   4.5. Johnsonovo kódování 22
   4.6. Kódování m-n 23
   4.7. 2-hot kódování 23
   4.8. Náhodné kódování 24
   4.9. Metoda postupného procházení 24
5. Pokročilé metody 25
   5.1. Teorie dělení (Hartmanis, Stearns, Kohavi) 25
   5.2. Metoda vyhodnocování sloupců ( Dolotta, Mc Cluskey) 25
   5.3. Vypočítávaná metoda (Story) 26
   5.4. Metoda „větví a hranic“ (Perkowski, Lee, Zasowska) 26
   5.5. Metody kvadratického přiřazování (Armstrong, De Micheli, Perkowski) 27
   5.6. Metoda Morozova 27
   5.7. Metoda De Michelliho, Braytona, Sangiovanniho, Vincentelliho 27
   5.8. Metoda depth_first 28
   5.9. Metoda minimální vzdálenosti 28

5.10. Metoda 1_level 28
5.11. Metoda 1_level_tree 29
5.12. Metoda „weakly crossed edge cuts“ 29
5.13. POW3 30
5.14. Spanning tree algoritmy (Nöth-Kolla) 30
5.15. JEDI 31
5.16. Loop-based algoritmus 32
5.17. FEL-code 32
6. Vlastní algoritmus 33
7. Implementace 34
7.1. Datové struktury a metody 34
7.1.1. Třída MyState 34
7.1.2. Třída MyFSM 35
7.1.3. Třída MyEncUtils 36
7.1.4. Třída fsmEncoder 36
8. Výsledky testů 37
9. Závěr 43
10. Seznam literatury 44
A. Uživatelská příručka 45
B. Obsah přiloženého CD 46

# 1.0 Úvod

FSM alias finite state machines jsou každodenní součástí návrhů sekvenčních obvodů, ať už jsou to řadiče, procesory či jiná zařížení vyžadující nejen informaci o výstupu obvodu dle zadaného vstupu, ale také informaci o tom, ve kterém stavu se momentálně nacházejí.

Jednotlivé stavy je však třeba nějakým způsobem interpretovat v binární soustavě, aby s nimi dokázaly obvody pracovat. K tomuto účelu slouží různé metody kódování stavů, kterými se budu zabývat v této práci.

Nové algoritmy přibývají téměř pravidelně a jsou oborem zájmu mnoha technických univerzit po celém světě. Většina materiálů je však těžko dostupná, ať už se jedná o odborné časopisy nebo materiály na internetu. Tímto bych chtěl znovu poděkovat Ing. Petru Fišerovi za pomoc při jejich shánění.

V práci se budu zabývat jednotlivými algoritmy, od základních a jednoduchých po relativně složité heuristické algoritmy. V poslední době se k řešení problému přidávají i genetické algoritmy, autorem posledního a údajně zatím nejefektivnějšího je Dr. Santanu Chattopadhyay, kterého jsem kontaktoval za účelem získání podrobnějších informací, bohužel na 2. e-mail už neodpověděl, tudíž algoritmus v práci zmíněn není. Další (díky stále dostupnějším clusterům a víceprocesorových systémům) rozvíjející se metodou je paralelizace současných algoritmů poskytující jejich zrychlení.

# 2.0 **Popis problému, specifikace cíle**

Specifikace cíle dle zadání BP: Z dostupných zdrojů nastudujte známé algoritmy pro kódování stavů konečného automatu a některé z nich naimplementujte. Srovnejte jejich efektivitu z hlediska výpočetní složitosti i kvality výsledného řešení. Naimplementované algoritmy začleňte do stávajícího EDA systému.

Jak jsem již naznačil v úvodu, práce se zabývá jednotlivými metodami kódování stavů konečného automatu a pokud je to možné i jejich srovnání. Problémem jsou různé metodiky provádění testů samotných autorů algoritmů, proto nelze udělat globální srovnání „vše v jednom“. V současné době je hlavní důraz kladen na minimální plochu obvodu a/nebo s tím související menší spotřebu a vyzařované teplo obvodu.

Úlohou ideálního algoritmu je minimalizovat počet přepínání 0 – 1 (switching activity) zejména na klopných obvodech zajišťujících přepnutí automatu ze zdrojového do cílového stavu při zachování co nejmenšího počtu bitů nutných pro zakódování jednotlivých stavů. Ideální algoritmus je však pouze teorie, proto je třeba najít aspoň částečně ideální řešení – heuristiku.

Pro zakódování N stavů FSM je zapotřebí minimálně $\log_2 N$ bitů, v praxi tedy min. $\log_2 N$-bitový registr ( $\log_2 N$ klopných obvodů). „Neideální“ kódování stavů tak může mít velký vliv na celkovou velikost, zpoždění a spotřebu obvodu.

Dalším cílem je vytvoření utility umožňující zakódovat FSM pomocí vlastního i jednodušších algoritmů a její integrace do EDA systému vyvíjeného na katedře.

# 3.0 **Definice použitých pojmů**

Před samotným výčtem algoritmů je potřeba definovat hlavní pojmy tohoto tématu.

## 3.1 **FSM (finite state machine)** [1]

FSM („stroj s omezeným počtem stavů“) je nástroj sloužící k popisu sekvenčních obvodů. Je definován jako $M = \langle X, Y, Q, q_0, \delta, \lambda \rangle$ , kde

X: množina možných kombinací hodnot vstupních proměnných

Y: množina možných kombinací hodnot výstupních proměnných

Q: množina stavů (rozlišné kombinace hodnot vnitřních proměnných)

$q_0$: počáteční stav (kombinace hodnot vnitřních proměnných v poč. stavu)

$\delta$: stavově přechodová funkce – X x Q -> Q (příští stav po přechodu)

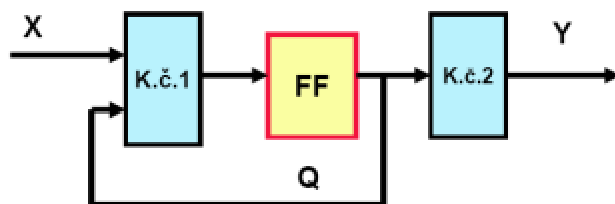
$\lambda$: výstupní funkce (závisí na typu FSM, viz dále)

a) X x Q -> Y .. typ Mealy
b) Q -> Y .. typ Moore

Typy FSM:

Moorův (stavově orientovaný)

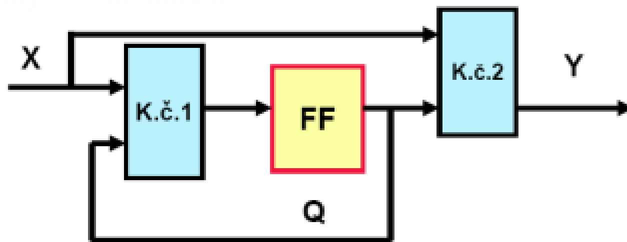
- výstup závisí pouze na daném stavu, hodnota výstupu se projeví až při přechodu do daného stavu (obr. 3.1*)




obr. 3.1.1

Mealyho (vstupově orientovaný)

- hodnota výstupu závisí na kombinaci stavu a vstupních hodnot, ke změně výstupu dochází okamžitě se změnou vstupu (obr. 3.2*)




obr. 3.1.2

* K.č.1 = kombinační logika č.1 (vstupní), K.č.2 = kombinační logika č. 2 (výstupní), FF=flip-flop (množina klopných obvodů)

## 3.2 **Reprezentace FSM** [2]

FSM může být reprezentován několika způsoby:

- výčtem – jsou definovány jednotlivé množiny a funkce z definice FSM (viz níže)
- tabulkou – do tabulky je zanesen daný stav, následující stav(y) dle daného vstupu a výstup (viz níže, tab. 3.2.1 a 3.2.2)
- stavovým stromem – strom větvený od počátečního stavu, každý nižší uzel představuje přechod (obr. 3.2.3 a 3.2.4)
- stavovým diagramem – graf, jehož uzly představují stavy a hrany jednotlivé přechody mezi stavy (obr. 3.2.5 a 3.2.6)

př. výčtem:

$Q = \{ A, B, C, D \}, X = \{ 0, 1 \}, Y = \{ 0, 1 \}, q_0 = A$

$\delta (A, 0) = A , \delta (A, 1) = B$
$\delta (B, 0) = B , \delta (B, 1) = C$
$\delta (C, 0) = C , \delta (C, 1) = D$
$\delta (D, 0) = D , \delta (D, 1) = A$
Moore:
$\lambda (A) = 0, \lambda (B) = 1, \lambda (C) = 0, \lambda (D) = 1$

Mealy:

$\lambda$ (A, 0) = 0 , $\lambda$ (A, 1) = 1

$\lambda$ (B, 0) = 1 , $\lambda$ (B, 1) = 0

$\lambda$ (C, 0) = 0 , $\lambda$ (C, 1) = 1

$\lambda$ (D, 0) = 1 , $\lambda$ (D, 1) = 0

př. tabulkou

Moore:

| stav | $x_0$=0 | $x_0$=1 | výstup |
|---|---|---|---|
| A | A | B | 0 |
| B | B | C | 1 |
| C | C | D | 0 |
| D | D | A | 1 |

tab. 3.2.1

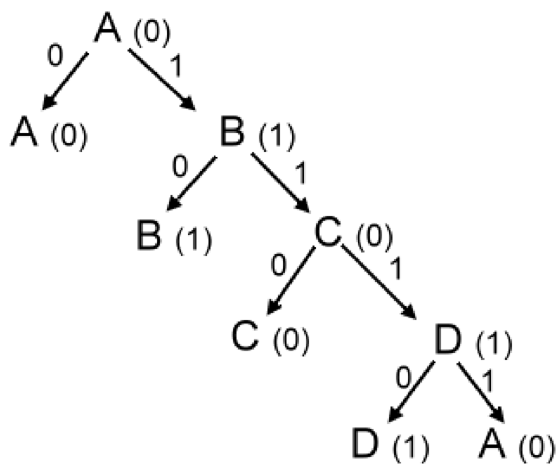
Mealy:

| stav | $x_0$=0 | $x_0$=1 | $x_0$=0 | $x_0$=1 |
|---|---|---|---|---|
| A | A | B | 0 | 1 |
| B | B | C | 1 | 0 |
| C | C | D | 0 | 1 |
| D | D | A | 1 | 0 |

tab. 3.2.2

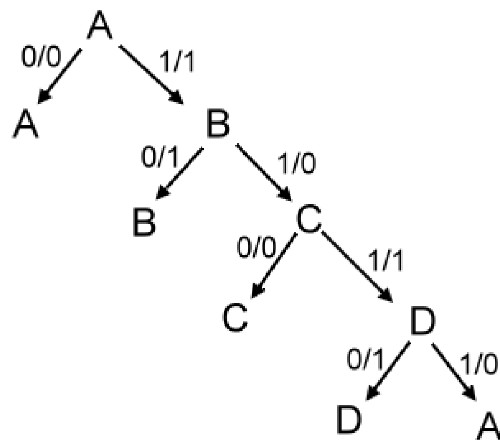
př. stavovým stromem

Moore:




obr. 3.2.3

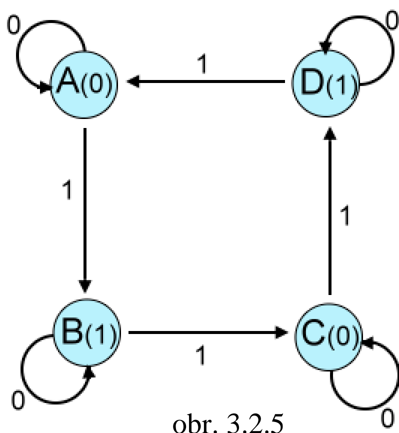
Mealy:



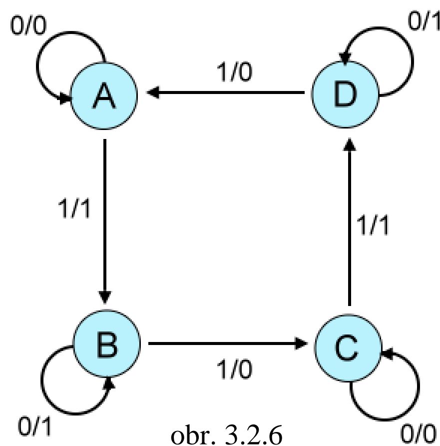

obr. 3.2.4

př. stavovým diagramem

Moore: Mealy:




obr. 3.2.5




obr. 3.2.6

## 3.2 **Hammingova vzdálenost**

Hammingova vzdálenost je počet rozdílných bitů mezi dvěma slovy, lze ji vyjádřit i jako počet jedniček po použití funkce XOR mezi dvěma slovy. V sekvenčních obvodech je zejména zajímavá vzdálenost mezi jednotlivými kódy stavů, kde nižší vzdálenost v praxi znamená menší počet nutných přepnutí klopných obvodů a tím i snížení spotřeby obvodu.

Např. H(0011, 0110)=2, protože se liší pouze 2. nejvyšší bit a nejnižší bit

## 3.3 **Pravděpodobnosti přechodů** [4]

Některé algoritmy využívají pro nejoptimálnější přiřazení kódů stavům (hlavně z hlediska spotřeby a snížení „přepínací aktivity“ (switching activity) klopných obvodů) pravděpodobnost, že v praxi k přechodu mezi danými stavy opravdu dojde. Nejpravděpodobnější hrany (přechody mezi 2ma vybranými stavy) by měly mít zároveň nejmenší Hammingovu vzálenost mezi sebou. Jednotlivé funkce k výpočtu používané jsou vysvětlené níže.

$p(S_i)$ = stálá pravděpodobnost stavu (pravděpodobnost, že FSM je ve stavu $S_i$ )

$p(S_i, S_j)$ = podmíněná pravděpodobnost přechodu ze stavu $S_i$ do $S_j$ (tj. pravděpodobnost, že FSM je ve stavu $S_i$ a přechází do stavu $S_j$)

$$p(S_i) = \sum_j p(S_j) * p(S_i, S_j) \quad , \quad \sum_i p(S_i) = 1$$

$P(S_i, S_j)$ = celková pravděpodobnost, že nastane přechod v grafu přechodů ze stavu $S_i$ do $S_j$ v cyklu hodin

$$P(S_i, S_j) = p(S_i) * p(S_i, S_j)$$

Pro určení „přepínací aktivity" (viz níže) je ještě potřeba definovat celkovou pravděpodobnost přechodu stavu $P_\Gamma(S_i) = \sum_{j,j\neq i} P(S_i, S_j) + \sum_{j,j\neq i} P(S_j, S_i)$ , kde $P_\Gamma(S_i)$ může být bráno jako měřítko aktivity stavu (určuje pravděpodobnost přechodu v grafu přechodů do nebo ze stavu kromě stavu samotného)

## 3.4 **Cenová funkce** [4]

Cenová funkce vyjadřuje celkový součet součinů pravděpodobnosti přechodů mezi jednotlivými stavy a jejich Hammingovou vzdáleností. Úkolem kódovacích algoritmů zaměřených na efektivní rozložení kódů stavů pro co nejnižší spotřebu obvodu je minimalizace hodnoty této funkce použitím kódů stavů s co nejnižší vzdáleností mezi nimi.

$$F = \sum_i \sum_j P(S_i, S_j) * H(kód(S_i), kód(S_j))$$

$P(S_i, S_j)$ = pravděpodobnost přechodu mezi stavy $S_i$ a $S_j$ , $H(kód(S_i), kód(S_j))$ = Hammingova vzdálenost mezi kódy stavů $S_i$ a $S_j$

## 3.5 **Ztrátový výkon**

Ztrátový výkon sekvenčního obvodu modelovaného pomocí FSM na technologii CMOS je přímo úměrný průměrné „přepínací aktivitě" na jednotlivých klopných obvodech.

$$P = 0.5V_{DD}^2 * f * \sum_n C_n E_n$$

kde $V_{DD}$ je napájecí napětí, $f$ je hodinová frekvence obvodu, $C_n$ je kapacitní odpor klopného obvodu udržujícího informaci o bitu $n$, $E_n$ je „přepínací aktivita" (switching activity) klopného obvodu. Při přechodu z jednoho stavu do jiného je třeba přepnout tolik klopných obvodů, v kolika bitech se kódy těchto stavů liší -> Hammingova vzdálenost. Snížením počtu přepínání mezi jednotlivými stavy je možné snížit celkovou „přepínací aktivitu" obvodu.

# 4.0 **Základní a jednoduché kódovací algoritmy**

Základnímy algoritmy jsem nazval algoritmy, ze kterých pokročilejší metody metody většinou vychází, ať už je to následnou modifikací nebo kombinováním. Pro algoritmy je až na výjimky společná jejich rychlost (lineární odpovídající počtu stavů), nevýhodou je často neoptimální rozložení kódů stavů bez ohledu na strukturu STG, jelikož kódování probíhá sekvenčně dle načtené tabulky.

## 4.1 **Binární kódování**

Binární kódování pracuje na principu očíslování jednotlivých stavů tak, jak jsou na číselné ose. Každý stav je reprezentován jedním číslem v binární podobě. Výhodou tohoto kódování je kromě rychlosti i minimální délka slova (názvu stavu) potřebná pro zakódování automatu a tím i minimální počet nutných klopných obvodů. Počet potřebných bitů odpovídá $\lceil \log_2 N \rceil$, kde N je počet stavů FSM. Nevýhodou je, že jednotlivá čísla v binárním kódu většinou nemají minimální možnou Hammingovu vzdálenost mezi sebou (např . 3 a 4, tj. H(0111, 1000)=4).

Př.

| Název stavu | Zakódovaný (dec) | Zakódovaný (bin) |
|---|---|---|
| st0 | 0 | 000 |
| st1 | 1 | 001 |
| st2 | 2 | 010 |
| st3 | 3 | 011 |
| st4 | 4 | 100 |
| st5 | 5 | 101 |

tab 4.1

## 4.2 **Grayovo kódování**

Grayův kód je specifický tím, že se dvě sousední slova liší pouze v jednom bitu (tj. mají H=1). Grayovo kódování využívá stejného principu k zakódování stavů. Takto pracující algoritmus by měl zakódovat všechny přechody FSM tak, aby jejich vzdálenost skutečně byla 1, to je ovšem u složitějších FSM (zejména obsahujících kružnice) problém složitosti NP-H. Rozšířená ač většinou nepravdivá teorie je, že stačí každý člen sekvence

binárních čísel upravit vztahem (n >> 1) XOR n, kde (n >>1) je binární číslo posunuté o 1 místo doprava a XOR je log. Funkce XOR, čímž se vytvoří sekvence čísel liších se od dalšho a předchozího o 1 bit (ve své podstatě permutace původních čísel), kterou se zakódují stavy tak, jak jsou v tabulce. Tento způsob však v praxi funguje pouze u čítačů pracujících v tomto kódu a vyjímečně jiných FSM, pokud mají vhodnou strukturu přechodů. Př.

| Název stavu | Zakódovaný (dec) | Zakódovaný (bin) | (n >> 1) XOR n (dec) | (n >> 1) XOR n (bin) |
|---|---|---|---|---|
| st0 | 0 | 000 | 0 | 000 |
| st1 | 1 | 001 | 1 | 001 |
| st2 | 2 | 010 | 3 | 011 |
| st3 | 3 | 011 | 2 | 010 |
| st4 | 4 | 100 | 6 | 110 |
| st5 | 5 | 101 | 7 | 111 |

tab. 4.2

## 4.3 **One-hot kódování**

One-hot kódování přiřazuje každému stavu právě jednu logickou 1 v binárním řetězu o délce odpovídající počtu stavů. Výhodou tohoto kódování je Hammingova vzdálenost rovna dvěma (H=2) mezi libovolnými dvěma stavy FSM. Díky této vlastnosti je tento kód vhodný pro rozsáhlá FSM s mnoha stavy (praxi s více než 8) a velkým větvením. Nevýhodou je (z délky zakódovaného řetězu vyplývající) počet nutných klopných obvodů, který odpovídá počtu stavů FSM. Někdy se pro zmenšení celkové vzdálenosti mezi stavy (tedy „přepínací aktivitou, switching activity) používá jako počáteční stav řetězec obsahující samé 0, což umožňuje nejen zkrátit řetěz o jeden bit, ale také sníží při přechodu ze stavu 0 do libovolného jiného Hammingovu vzdálenost na 1 místo 2.

Př.

| Název stavu | Zakódovaný | Zakódovaný (s poč. stavem 0) |
|---|---|---|
| st0 | 000001 | 00000 |
| st1 | 000010 | 00001 |
| st2 | 000100 | 00010 |
| st3 | 001000 | 00100 |
| st4 | 010000 | 01000 |
| st5 | 100000 | 10000 |

tab. 4.3

## 4.4 **Zero-hot kódování**

Jak již název napovídá, nejedná se o nic jiného, než opačné kódování k one-hot, tj. místo logické 1 určuje stav logická 0 mezi jedničkami. Stejně jako u one-hot je možné zmenšit délku slova použitím samých 1 jako startovního stavu.

Př.

| Název stavu | Zakódovaný | Zakódovaný (s max. poč. stavem) |
|---|---|---|
| st0 | 111110 | 11111 |
| st1 | 111101 | 11110 |
| st2 | 111011 | 11101 |
| st3 | 110111 | 11011 |
| st4 | 101111 | 10111 |
| st5 | 011111 | 01111 |

tab. 4.4

## 4.5 **Johnsonovo kódováním** [2]

Johnsonovo kódování probíhá následujícím způsobem. Pro zakódování je potřeba $q = \left\lceil \frac{N}{2} \right\rceil$ bitů, kde N je počet stavů FSM. První stav je zakódován pomocí samých 0. U další stavů se od konce slova postupně mění 0 na 1 do doby, než slovo obsahuje samé 1. U zbývajících stavů se následně postupně opět od konce mění 1 na 0 (viz tab. 5.2). Stavy následující v tabulce za sebou se opět liší 1 bitem, tj. mají H=1, jedná se tedy o variantu Grayova kódu vyžadujícího stejné předpoklady jako u 4.2, praktické použití je opět v čítačích v tomto kódu.

Př. N=6 => q=3

| Název stavu | Zakódovaný |
|---|---|
| st0 | 000 |
| st1 | 001 |
| st2 | 011 |
| st3 | 111 |
| st4 | 110 |
| st5 | 100 |

tab. 4.5

## 4.6 **Kódování m-n**

Při tomto kódování je zvolena délka slova (n) a počet logických 1, které ve slově mají být obsažené (m). Jednotlivé kódy jsou potom permutacemi slova obsahujícího n jedniček a (m-n) nul. Parametry m a n však musí být zvoleny tak, aby jimi bylo možné všechny stavy pokrýt. Maximální počet stavů, které lze zakódovat, odpovídá kombinačnímu číslu $\binom{n}{m}$, proto m a n musí být zvoleno tak, aby $\binom{n}{m} \geq N$ .

Př. m=3 n=5 (tedy max. 10 stavů)

| Název stavu | Zakódovaný |
|---|---|
| st0 | 00111 |
| st1 | 01110 |
| st2 | 11100 |
| st3 | 01011 |
| st4 | 10011 |
| st5 | 01101 |

tab. 4.6

## 4.7 **2-hot kódování**

2-hot kódování je ve své postatě specifická varianta m-n kódování. Je tedy možné zakódovat $\binom{n}{2}$ stavů, v optimálním případě je možné udržet Hammingovu vzdálenost mezi všema přechody na 2.

Př.

| Název stavu | Zakódovaný |
|---|---|
| st0 | 10000 |
| st1 | 10001 |
| st2 | 10010 |
| st3 | 10100 |
| st4 | 01000 |
| st5 | 01001 |
| st6 | 01010 |
| st7 | 01100 |

tab. 4.7

## 4.8 **Náhodné kódování**

Tato metoda vygeneruje n náhodných možných řešení zakódování stavů binárním kódem, průběžně se pro každé z nich spočítá cenová funkce a jako výsledné řešení je bráno řešení s nejmenší cenou. V praxi se ukázalo, že toto řešení se často příliš neliší od optimálního (u menších FSM) a přitom je velmi rychlé.

## 4.9 **Metoda postupného procházení** [2]

Tato metoda postupně vyzkouší všechny možné kombinace binárních kódů, pro každý případ počítá cenovou funkci, ale také pro jeho složitost a přitom si pamatuje nejoptimálnější řešení, kterým ve výsledku FSM zakóduje. Vzhledem k faktoriální časové složitosti je metoda určena pouze pro FSM s malým počtem stavů.

# 5.0 **Pokročilé algoritmy**

Pokročilé algoritmy, ať už se jedná o heuristiky, genetické či jiné algoritmy, většinou využívají základní algoritmy pro prvotní zakódování stavů a následně je dle rozlišných kritérií optimalizují. Nejčastějšími metodami jsou hledání nejnižší hodnoty cenové funkce nebo kódování dle pravděpodobnosti přechodu mezi stavy. Některé poskytují optimální řešení z hlediska rychlosti (snížení přepínací aktivity), jiná se zaměřují na velikost obvodu (počet nutných hradel) a tím i menší spotřebu.

## 5.1 **Teorie dělení** [7]

(Partition theory of Hartmanis, Stearns, Kohavi)

Teorie je založena na představě dělení stavů FSM do bloků. Oddíly jsou nalezeny pomocí tabulky stavů a použity k nalezení množin oddílů vytvářejících unikátní a optimální kódování. Tato teorie je velice elegantní z matematického úhlu pohledu, umožňuje pohled na přirozené vlastnosti struktury FSM, přitom je velmi obecná (umožňuje také dekompozici FSM, minimalizaci počtu stavů, realizaci s posuvnými registry apod.). Přesto publikované výsledky programů, které tuto teorii využívají, dokázaly, že obecně tento způsob není vhodný pro počítačové řešení automatů obsahujících víc jak 10 stavů, 10 vstupních a 10 výstupních proměnných.

## 5.2 **Metoda vyhodnocování sloupců** [7][2]

(Column evaluation approach of Dolotta and Mc Cluskey)

Sloupce tabulky stavů jsou ohodnoceny s ohledem na mnoho různých kritérií ovlivňujících kvalitu přiřazení kódů stavům. Hodnocení jsou využity k nalezení vhodných oddílů a následnému přiřazení kódů. Tato metoda může vést k velmi dobrým výsledkům (v současné době také pro různé druhy klopných obvodů), výsledky jsou ještě lepší než u výše zmíněné metody, ale stále počítačově těžko řešitelné pro FSM s více než 12ti stavy. Původní metoda, která byla uvedena již v roce 1964, obsahovala několik omezení a předpokladů – nebyla uvažována výstupní funkce (Mooreův model), vstupy do sekvenčního

systému a výstupy z paměťové části byly k dispozici v přímé i negované formě, kombinační část byla realizována pouze pomocí OR a AND bez použití invertorů, klopné obvody mohly být pouze typu D, vychází z tabulky přechodů s minimálním počtem vnitřních stavů, nebrala ohled na cílovou techonologii ale pouze počet hradel nutných k realizaci. Postupně byla většina omezení odstraněna.

## 5.3 **Vypočítávaná metoda** [7]

(Enumerative approach of Story)

Všechny možné oddíly (množiny stavů) jsou vyhodnoceny jako kandidáti na přiřazení kódu odděleně vypočtením složitosti realizace odpovídajících booleovských funkcí a předpokladem, že všechny ostatní oddíly byly optimálně vybrány. Podmnožina oddílů s nejlepšími ohodnoceními (nebo jiná shodnými) je vybrána pro přiřazení kódů. Tato metoda se ukázala jako ještě lepší než předchozí, ovšem je ještě pomalejší.

## 5.4 **Metoda „větví a hranic“** [7]

(Branch and bound approach of Perkowski, Lee and Zasowska)

Tato metoda má dvě varianty. První umožňuje zakódování FSM s 8mi vstupy, 8 vnitřními a 8mi výstupními stavy a vytváří optimální řešení pro libovolnou technologii a klopné obvody. Bohužel v této variantě musí být spočítány všechny možné oddíly. Tím tento algoritmus možná představuje nejoptimálnější řešení, nicméně je velmi pomalý. Druhá, aproximační varianta postavená na této metodě, nevyhodnocuje všechny oddíly, ale pouze heuristicky vybírá ty nejlepší. Dovoluje realizaci FSM se 12ti stavy, ale může být rozšířena až na 18-20 stavů. Obě tyto varianty umožňují kombinovat přiřazování stavů s jejich minimalizací.

## 5.5 **Metody kvadratického přiřazování** [7]

(Quadratic assignment approaches of Armstrong, De Micheli and Perkowski)

Všechny tyto metody jsou postaveny na vkládání grafů vytvořených z tabulky FSM do grafů hyperkrychle. Tato metoda umožňuje zakódovat FSM obsahující až 100 stavů, ale poskytuje řešení velmi vzdálené od optimálního. Byla vytvořena teoretická vylepšení algoritmů, avšak nebyla naprogramována. De Micheli vytvořil dva algoritmy pro kódování stavů během doby, co byl na technické univerzitě v Berkeley. První z nich byl na metodě kvadratického přiřazování postaven – přiřazování stavů je zjednodušeno na problém vkládání grafů. Druhý algoritmus byl postaven na jiném principu, jelikož nebyl spokojen s kvalitou prvního. Výsledky obou se jsou docela uspokojivé (podařilo se zakódovat FSM se 136 stavy), ale zatím nebylo provedeno detailní srovnání s ostatními metodami.

## 5.6 **Metoda Morozova** [7]

(Approach of Moroz)

Tento velmi rychlý konstruktivní vnořovací algoritmus byl velmi rozšířen v návrhu automatizovaných systémů v Rusku. Autor implementoval tento algoritmus a zjistil, že dokáže najít řešení pro FSM s více než 100 stavy, ale kvalita přiřazení pro malé FSM měla k optimálnímu daleko. Tento algoritmus je možná jedním z nejrychlejších dostupných s výjimkou paralelních algoritmů. Rychlost je způsobena tím, že algoritmus neřeší kvadratické přiřazování ale jednoduchý problém vnořování hran a zároveň graf pro vnořování je vytvořen přímo z STG.

## 5.7 **Metoda De Michelliho, Braytona a Sangiovanni-Vincentelliho**

(Approach of De Michelli, Brayton and Sangiovanni-Vincentelli) [7]

Tato metoda je postavena na minimalizaci vícekrát ohodnocených booleovských funkcích pro nalezení skupin stavů a jejich následné konstruktivní přiřazení vnořením těchto skupin do stěn hyperkrychle. Výsledky této metody jsou velmi dobré. V době vydání materiálů, ze kterých jsem čerpal, to byla pravděpodobně nejlepší metoda z pohledu poměru kvalita/čas. Byla otestována na FSM se 100 stavy, ale největší publikovaný výsledek byl pro 27 stavů.

Skupiny stavů jsou nalezeny využitím minimalizace vícehodnotové logiky programem Espresso použitého na FSM předtím, než dojde k přiřazení stavů. Tato metoda nalezení skupin stavů zkombinovaná s minimalizací booleovských funkcí byla zdrojem úspěchu programu Kiss. Ten je/byl využíván mnoha univerzitami a společnostmi k tvorbě komerčního softwaru.

## 5.8 **Metoda depth_first** [8]

(Depth_first method)

Metoda dept_first nejprve využije řadící funkci k objevení cesty mezi následujícími uzly v grafu s nejvyššími pravděpodobnostmi přechodu. Jakmile jsou cesty vytvořeny, metoda je postupně zakóduje Grayovým kódem.

## 5.9 **Metoda minimální vzdálenosti** [8]

(Minimum distance method)

V této metodě se postupně vybírají hrany grafu s největší vahou (pravděpodobností přechodu) ze seznamu hran náležících jednomu ze stavů předtím vybrané hrany. Následuje druhé řazení, potom je použito kódové slovo, které minimalizuje Hammingovu vzdálenost mezi párem stavů dané hrany stejnou metodou jako v prvním řazení.

## 5.10 **Metoda 1_level** [8]

(1_level_method)

Metoda 1_level využívá stejný způsob řazení stavů jako předchozí metoda, potom aplikuje jedno z možných kódových slov na nepřiřazené stavy $s_p(k)$ (nebo $s_q(k)$) spočítáním částečné ceny. Je zvolen binární kód $c_k$, který minimalizuje vážený součet Hammingových vzdáleností se zřetelem na binární kódy vybrané hrany a hrany s největší vahou, která zatím obsahuje nepřiřazení stav $s_p(k)$. Pokud je váha dvou hran stejná, preferuje se hrana s menším počtem rozlišných bitů od předtím vybrané hrany.

## 5.11 **Metoda 1_level_tree** [8]

(1_level_tree_method)

Metoda 1_level_tree je třetí variantou předchozích 2 algoritmů. Po provedení prvního kroku se vybírá hrana s nejvyšší vahou, která obsahuje alespoň jeden už předtím přiřazený stav. Pro přiřazení kódu se používá částečná cenová funkce $C_{p}$.

## 5.12 **Metoda „weakly crossed edge cuts“** [6]

(„weakly crossed edge cuts“ encoding algorithm)

Algoritmus nejprve vypočítání z STG matici sousednosti a incidence. K matici sousednosti je přidán sloupec obsahující stupně vrcholů $d(v_i)$. Potom začíná konstrukcí množiny oddílů pro kódování. Počet oddílů je roven $k = \lceil \log_2 N \rceil$, kde N je počet vrchol (stavů) STG a $k$ je délka kódu. Každý oddíl $\Pi_r$ se skládá ze dvou bloků – bloky jedniček a nul, které jsou reprezentovány proměnnými $B_r^1$ a $B_r^0$, kde $1 \leq r \leq k$. Oddíl produktů $\Pi_s$ je roven bloku jedniček $\Pi_s = \Pi_1$.

V prvním kroku jsou množiny $B_r^1$ a $B_r^0$ prázdné, vybere se první hrana ($v_i$, $v_j$) z matice incidence jako startovní bod. Vrcholy vybrané hrany se umístí do množiny $B_r^1$, takže bude obsahovat prvky { $v_i$, $v_j$ }. Váha hranových řezů se při proceduře zvyšuje přidáváním vrcholu $v_c$ do množiny $B_r^1$. Pro každý nerozdělený vrchol $v_c$ je tento přírůstek spočítán jako $\Upsilon^1(B_r^1, v_c) = d(v_c) - 2 * |N'(B_r^1, v_c)|$ , kde $d(v_c)$ je stupeň vrcholu $v_c$ a $N'(v_c)$ je množina sousedů vrcholu $v_c$ obsahující všechny vrcholy z množiny S sousedících uzlů $v_c$ kromě samotného $v_c$ s ohledem na $B_r^1$, takže se bude rovnat { $v_i$, $v_j$, $v_c$ }. Procedura počítání přírůstku přidáním vrcholu $v_c$ do množiny $B_r^1$ se opakuje, dokud kardinalita $B_r^1$ nedosáhne přibližně poloviny kardinality V. Poslední vrchol $v_I$ v množině $B_r^1$ je otestován příslušnost oběma množinám.

Potom se spočítá celkový součet přírůstku vrcholů $v_I$ v množinách $B_r^1$ a $B_r^0$. Pokud je součet v jednotkovém bloku menší nebo roven součtu v nulovém, je vrchol $v_I$ přidán do množiny $B_r^1$, jinak do $B_r^0$. Zbytek nerozdělených vrcholů je umístěn do $B_r^0$. Z incidenční matice jsou vymazány všechny řádky odpovídající vrcholům, které obsahuje množina $B_r^0$. Zbytek řádků je sečten

pomocí modulo 2. Množina hran označená jako 1 výsledného booleovského vektoru určuje jednotlivé hranové řezy kódovacího oddílu $\Pi_r$ .

V dalším kroku r=r+1. Nejprve $B_r^1 = \{ v_i, v_j \}$, kde $(v_i, v_j)$ je první hrana předcházejícího hranového řezu. Potom tato a předchozí procedura volá jedna druhou a opakuje se to do doby, než je r > k. Posledním krokem algoritmu je otestování $\Pi_s = \Pi_0$ a spočítání chyby navrhovaného kódu jako $\sum_I p(e_i) * H(e_i) / \sum_I p(e_i)$, kde $p(e_i)$ je pravděpodobnost přechodu mezi stavy hranou $e_i$ a $H(e_i)$ je Hammingova vzdálenost mezi kódy na vrcholech hrany $e_i$.

## 5.13 **POW3** [3]

POW3 je algoritmus pro kódování stavů zaměřený na nízkou spotřebu obvodu. Jeho autorem jsou Luca Benini a G. De Michelli. Zaměřuje se na snížení „přepínací aktivity“ (switching activity) ve stavovém registru během přechodů. Algoritmus používá pravděpodobnostní popis FSM a minimalizuje vzdálenost mezi kódy stavů s vysokou pravděpodobností přechodu. Hladový heuristický algoritmus přiřazuje kódy stavům bit po bitu, aby zajistil stavům s vysokou pravděpodobností přechodu stejné hodnoty bitů. Pokud je jeden bit nějakému z těchto stavů přiřazen, ostatním je přiřazena stejná hodnota bitu. Algoritmus počítá s omezením kódování způsobeným požadavkem na unikátnost kódů stavů. Heuristika používá cenovou funkci založenou na váženém součtu Hammingových vzdáleností mezi kódy stavů.

## 5.14 **Spanning tree algoritmy (Nöth-Kolla)** [3]

Metoda využívá algoritmus postavený na spanning tree kódování, které taktéž využívá pravděpodobnost přechodu mezi stavy. STG je přetransformováno na neorientovaný graf, každé hraně je přiřazena váha odpovídající pravděpodobnosti přechodu touto hranou. Maximálně se rozvíjející strom je z tohoto grafu vytvořen. Problém kódování stavů je formulován jako vnoření stromu do hyperkrychle. Pro toto byly navrženy 2 algoritmy – rychlý vnořovací algoritmus (fast embedding algorithm) a hladový vnořovací algoritmus (greedy embedding algorithm).

První z nich vybere hranu, kterou rozdělí strom na 2 stejně velké podstromy. Hrana je namapována na hranu hyperkrychle a stavy, které spojuje,

jsou mapovány do odpovídajících uzlů. Potom algoritmus rekurzivně zpracuje oba podstromy. Jakmile jsou všechny stavy přiřazené do uzlů, uzlům jsou přiřazeny kódy tak, že kódy dvou uzlů, které jsou spojené hranou, se liší přesně o 1 bit (H=1).

Hladová metoda rozšiřuje rychlou o hladový algoritmus pro výběr hrany hyperkrychle, na které se má graf namapovat. Pro tentokrát bere algoritmus na vědomí uzly, které již byly přiřazeny. Pokud už byl jeden ze stavů hrany přiřazen uzlu, algoritmus spočítá cenu pro přiřazení nepřiřazeného stavu do volného uzlu na opačné straně hrany. Tato cena je kombinací pravděpodobnosti přechodu a Hammingovi vzdálenosti mezi všemi hranami přiřazenými volnému uzlu. Kód je přiřazen uzlu s nejnižší cenou.

## 5.15 **JEDI** [5]

JEDI je kódovací algoritmus zaměřený na implementaci pro víceúrovňovou logiku. Obsahuje 2 hlavní fáze – přiřazení váhy a kódování stavů.

V první fázi JEDI přiřadí váhu všem hranám v STG, což je odhadovaný vztah stavů k ostatním stavům. Pro tento účel se vybírá nejlepší ze 4 heuristik– vstupně dominantní (input dominant), výstupně dominantní (output dominant), spojená (coupled) a střídavá (variation). Vstupně dominantní algoritmus přiřazuje vyšší váhy párům výchozích stavů, které produkují stejný výstup a stejnou množinu dalších vah. To ovlivňuje maximální velikost obecných krychlí při implementaci logických funkcí. Výstupně dominantní metoda přiřazuje vyšší váhy párům cílových stavů, které jsou generovány stejnou kombinací vstupů a stejnou množinou výchozích stavů. To maximalizuje počet obecných krychlí v logických funkcích. Spojená metoda přidává váhy generované vstupně i výstupně dominantním algoritmem.

Ve druhé fázi se JEDI pokouší zakódovat stavy náležící hraně s nejvyšší vahou, tedy se ve výsledku snaží o minimalizaci součtu $\sum_{i=1}^{N}\sum_{j=1}^{N} m_{ij} * H(s_i, s_j)$, kde $m_{ij}$ je váha hrany z matice vstupního přiřazení a $H(s_i,s_j)$ je Hammingova vzdálenost kódů $s_i$ a $s_j$, když jsou zakódovány minimálním počtem bitů. Samotné kódování je provedeno použitím „simulovaného žíhání“ (simulated annealing), což je „hill climbing“ heuristika, založená na pravděpodobnosti, pracující na následujícím principu.

Nejprve začne s výchozí teplotou T a náhodnou konfigurací stavů. Pro danou teplotu náhodně vybere 2 stavy a přiřadí jim kód nebo jejich kódování prohodí. Spočítá cenovou funkci po změně. Přijme lepší variantu pro snížení hodnoty cenové funkce. Zároveň ale dovoluje některým kódům, aby byly přijaty, i když to vede ke zvýšení hodnoty funkce, aby se předešlo lokálnímu minimu. Poté provádí opět vybírání 2 stavů atd., dokud není proveden určitý počet přesunů, potom sníží teplotu a proces opakuje. Zastaví se v okamžiku, kdy teplota dosáhne minima.

V současné době již byla vyvinuta Davidem A. Baderem (University of New Mexico) a Kameshem Maddurim (Indian Institute of Technologi Madras) paralelní verze JEDI, která přináší velké urychlení na víceprocesorových systémech.

## 5.16 **Loop-based algoritmus** [3]

Tento algoritmus je zejména určen pro velké FSM obsahující mnoho stavů, kde komplexní heuristiky přestávají být z časového hlediska zajímavé. Zaměřuje se na hledání kružnic v grafu, které se snaží optimálně zakódovat za účelem snížení přepínací aktivity. Algoritmus rozdělí FSM do oddílů odpovídajících kružnicím v grafu, čímž sníží váhy jeho jednotlivých hran. Během přiřazování kódů hranám jsou respektovány již přiřazené stavy, ale už bez zpětného vyhledávání, které kružnici patří. Tím je zaručeno optimální kódování pouze pro první kružnici. Základní algorimus nejprve projde grafem metodou depth-first a každému stavu přiřadí úroveň, každý uzel úrovně odpovídá nepřiřazenému kódu a každý větev k uzlu symbolizuje přiřazení stavu. Nejvyšší úroveň (0) představuje nepřiřazený FSM. Jakmile se strom projde od shora do konce, výsledná cesta odpovídá platnému kódování FSM. Na vyhledání nejlepšího řešení je použita cenová funkce.

## 5.17 **FEL-code** [2]

FEL-code je algoritmus vyvinutý Michalem Prokešem v rámci diplomové práce na FEL ČVUT pod vedením Doc. Ing. Hany Kubátové, CSc. Metoda je založena na sloučení one-hot a binárního kódování. Stavy FSM jsou rozděleny do skupin podle počtu přechodů mezi nimi. Tím vznikají oddělené menší husté

grafy přechodů, které jsou následně chápány jako samostatné stavy a dle pořadového čísla zakódovány one-hot kódem. Výsledný kód se tak skládá ze dvou částí – binárního kódu stavu ve skupině a one-hot kódu identifikujícího skupinu. Výhodou metody je, že husté STG budou v důsledku předchozí myšlenky zakódovány pouze binárním kódem a naopak řídké pouze one-hot kódem.

# 6.0 **Vlastní algoritmus**

Moje metoda kódování je založena na nalezení kódů stavů s co nejnižší průměrnou Hammingovou vzdáleností mezi nimi. Vlastní algoritmus vypadá následovně:

1. Po načtení tabulky přechodů je vytvořen neorientovaný graf přechodů neobsahující smyčky ani násobné hrany.
2. Je nalezen uzel s největším počtem sousedů a dostane přidělen kód 0
3. Všichni jeho sousedi získají kódy s H=1 od kódu stavu (v případě prvního uzlu v podstatě one-hot kódování)
4. Sousedi vybraného uzlu jsou seřazeni sestupně podle počtu jejich sousedů
5. Postupně jsou sousedi rekurzivně vybíráni a opakuje se bod 3
6. Jakmile jsou všechny uzly zakódované, spočítá se průměrná Hammingova vzdálenost mezi nimi a celková průměrná vzdálenost
7. Uzly jsou seřazeny sestupně podle průměrné H
8. Druhá fáze algoritmu takto seřazené uzly projde a každý ze sousedů stavu vygeneruje alternativy s H=1 od svého kódu a zároveň s maximálním počtem bitů, jakým je zakódovaný stav samotný (počet bitů stavu s nejvyšší hodnotou kódu).
9. Algoritmus vybere nejlepší alternativu a přepíše kód stavu
10. Opakuje se bod 6 do doby, než se průměrná vzdálenost ustálí (přitom si pamatuje hodnotu nejnižší a kódy stavů jí odpovídající) nebo čítač nedosáhne určenou hodnotu v případě kolísání vzdálenosti (velká FSM).

Teoreticky by algoritmus měl být vhodný pro rozvětvené FSM obsahující málo kružnic, vzhledem k uvažování kódu stavu v číselné podobě je jejich počet limitován operacemi nad datovými typy daného programovacího jazyka (v případě C++ je to typ unsigned int, nad kterým jsou definovány bitové posuny, unsigned long se při těchto operacích choval podivně, ale může to být způsobeno překladačem).

# 7.0 **Implementace**

Třída *fsmEncoder* načte při své iniciaci v konstruktoru popis FSM ze třídy *fsmDescr* (obsažené souboru fsm.h v adresáři hub EDA systému EDuArd). Následně vytvoří vlastní zjednodušený popis FSM ve třídě *MyFSM* (viz bod 7.0, krok 1). Jednotlivé stavy jsou poté reprezentovány třídou *MyState*, třída už neobsahuje informace o vstupech, výstupech stavu, pouze přechody k unikátním sousedům v STG (tj. žádné násobné přechody ani přechody do stejného stavu).

Jakmile je popis načten do tříd, zavolá se metoda třídy fsmEncoder *Encode* s parametrem typu string určujícícího metodu kódování (viz příloha A – uživatelská příručka). FSM je následně zakódován zvoleným algoritmem.

Nakonec je zavolána metoda třídy fsmEncoder *addCodes* s parametrem třídy fsmDescr obsahující původní popis FSM. Jednotlivým stavům jsou následně zapsány kódy do proměnné *encoded_* ve třídě FsmState EDA systému.

V době vzniku této práce EDuArd ještě neobsahoval podporu exportu zakódovaných stavů FSM, proto jsem jsem kromě integrace do aplikace *stubs* v EDA systému vytvořil i nezávislou aplikaci umožňující import z formátu KISS a export do zjednodušeného formátu BLIF, aby bylo možné provést testy. Obě varianty jsou přiloženy na CD.

## 7.1 **Datové struktury a metody**

### 7.1.1 **Třída MyState**

Třída reprezentuje zjednodušený stav FSM jako uzel v prostém grafu. Obsahuje základní metody pro práci s ním.

Proměnné:

| proměnná | typ | význam |
|---|---|---|
| name_ | string | jméno stavu |
| codeStr_ | string | zakódované jméno stavu v řetězci |
| code_ | SizeType | kód stavu v číselné podobě |
| neighbours_ | set <MyState *> | sousedi stavu v STG |
| NeighSortedNeighs | map<int, set<MyState *> > | sousedi seřazení podle počtu jejich sousedů v STG |
| neighboursNo_ | int | počet sousedů stavu |
| encoded_ | bool | je stav už zakódovaný? |

tab. 7.1.1a

Důležité funkce:

| název funkce | parametry | výstup | význam |
|---|---|---|---|
| insertNeighbour | MyState *state | void | přidá souseda stavu |
| avgDist | void | float | vrátí průměrnou Hammingovu vzdálenost od sousedů |
| sortNeighbours | void | void | seřadí sousedy dle počtu jejich sousedů do NeighSortedNeighs |
| getCandidates | int length | set <SizeType> | vrátí množinu jiných možností kódu stavu s H=1 od každého souseda |

tab. 7.1.1b

### 7.1.2 **Třída MyFSM**

Reprezentuje zjednodušený FSM ve formě prostého grafu a poskytuje metody pro počítání a získávání údajů o stavech.

Proměnné:

| proměnná | typ | význam |
|---|---|---|
| states_ | MyState * | pole stavů FSM |
| stateNo_ | int | počet stavů |
| numBits_ | int | počet bitů kódu stavu |
| maxNum_ | SizeType | maximální číselná hodnota kódu stavu |
| IndexFromStr | map<string, int> | mapa pro převod názvu stavu na index v poli stavů |
| StrFromIndex | map<int, string> | mapa pro rychlé získání názvu stavu dle jeho indexu v poli stavů |

tab. 7.1.2a

Důležité funkce:

| název funkce | parametry | výstup | význam |
|---|---|---|---|
| calcNumBits | void | void | spočítá počet bitů stavu dle hodnoty maxNum_ |
| avgDist | void | float | vrátí průměrnou Hammingovu vzdálenost všech stavů |
| codeToCodeStr | void | void | převádí číselnou podobu kódů stavů do řetězcové |
| mostNeighs | void | MyState * | vrátí stav (příp. jeden ze stavů) s nejvyšším počtem sousedů |

tab. 7.1.2b

### 7.1.3 **Třída MyEncUtils**

Třída poskytuje podpůrné funkce ostatním metodám. Má jedinou proměnnou a to množinu *H1Used*, do které se ukládají již použité kódy stavů, což následně zaručuje unikátnost jednotlivých kódů.

Důležité funkce:

| název funkce | parametry | výstup | význam |
|---|---|---|---|
| binLength | SizeType | int | vrátí počet bitů čísla |
| IntToBinStr | SizeType<br>SizeType, int | string | převede binární číslo na řetězec, ve druhém případě s nastavenou délkou slova |
| HammingDistance | SizeType, SizeType<br>string, string | int | spočítá Hammingovu vzdálenost dvou binárních čísel nebo řetězců |
| generateH1 | SizeType, int | set<SizeType> | vygeneruje množinu n čísel s H=1 od zadaného čísla |
| generateLimitedH1 | SizeType, int | set<SizeType> | vygeneruje množinu čísel s H=1 od zadaného čísla a maximálním počtem bitů n |

tab. 7.1.3a

### 7.1.4 **Třída fsmEncoder**

Třída reprezentuje samotný nástroj na kódování FSM. Její proměnné jsou pouze ukazatel na třídu MyFSM a mapa pro převod metody v textové reprezentaci na číslo.

Důležité funkce:

| název funkce | parametry | výstup | význam |
|---|---|---|---|
| getStateCodes | SizeType | map <string, string> | vrátí mapu obsahující název stavu a jeho kód v textové podobě |
| addCodes | FsmDescr * | void | přidá kódy stavů do vloženého popisu FSM |
| Encode | string | bool | spouští jednotlivé kódovací algoritmy (viz příloha A) |
| EncodeOwn_step1 | MyState * | void | první krok při kódování mým algoritmem (viz sekce 6.0, body 3 až 5 |
| EncodeOwn_step2 | void | void | druhý krok při kódování mým algoritmem (viz sekce 6.0, body 8 a 9 |

tab. 7.1.4a

# 8.0 **Výsledky testů**

Pro testování algoritmu jsem zvolil standardní benchmarky ze série LGSynth93. Testy probíhaly následovně – všechny benchmarky ve formátu KISS po minimalizaci stavů programem STAMINA byly nejprve zakódovány jednotlivými nástroji a metodami (metoda jedi, random a onehot programem JEDI z balíku SIS, metoda nova programem NOVA z balíku SIS, metoda my a binary mým kodérem), změřeny časy provádění, výsledky kódování následně uloženy do formátu BLIF, poté načteny programem SIS a podrobeny minimalizaci pomocí programu SIMPLIFY (viz soubor script.rugged). Pro získání výsledků byly použity příkazy SISu print_stats a power_estimate.

Z výsledků byly vygenerovány tabulky pro porovnání jednotlivých metod, kde *method* značí použitou metodu kódování, *bits* počet bitů využitých k zakódování stavu, *avg. H* je průměrnou Hammingovu vzdálenost mezi všemi přechody v přechodové tabulce, *nodes* počet nutných hradel, *lits* počet literálů v logických funkcích, *power* odhadovanou spotřebu v µW a *time* dobu kódování. Červené pozadí značí nejhorší hodnoty, zelené nejlepší, Některé benchmarky nebylo možné otestovat nejčastěji kvůli nepřiměřené časové náročnosti script.rugged nebo selhání power_estimate v SISu.

| **bbara** | states: 7, inputs: 4, outputs: 2, products: 42 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 3 | 0.738 | 9 | 56 | 322.4 | 0.004 |
| onehot | 7 | 0.857 | 13 | 63 | 480.4 | 0.002 |
| random | 3 | 0.738 | 7 | 61 | 326.6 | 0.002 |
| jedi | 3 | 0.643 | 6 | 53 | 305.1 | 0.004 |
| nova | 3 | 0.643 | 8 | 49 | 278.4 | 0.227 |
| my | 3 | 0.619 | 7 | 48 | 284.8 | 0.005 |

tab 8.1

| **bbsse** | states: 13, inputs: 7, outputs: 7, products: 208 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 4 | 1.683 | 19 | 123 | 542.6 | 0.009 |
| onehot | 13 | 1.692 | 27 | 127 | 828.1 | 0.003 |
| random | 4 | l.49 | 21 | 131 | 592.4 | 0.003 |
| jedi | 4 | 1.966 | 18 | 121 | 546.2 | 0.022 |
| nova | 4 | 1.721 | 18 | 109 | 505.7 | 0.105 |
| my | 4 | 1.409 | 17 | 123 | 511.6 | 0.01 |

tab 8.2

| **bbtas** | states: 6, inputs: 2, outputs: 2, products: 24 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 3 | 0.875 | 6 | 32 | 219.9 | 0.004 |
| onehot | 6 | 1.167 | 9 | 29 | 284.7 | 0.001 |
| random | 3 | 1 | 7 | 28 | 201.0 | 0.001 |
| jedi | 3 | 0.792 | 7 | 24 | 172.9 | 0.003 |
| nova | 3 | 1.167 | 6 | 26 | 197.1 | 0.078 |
| my | 3 | 0.583 | 7 | 25 | 187.7 | 0.004 |

tab 8.3

| **beecount** | states: 4, inputs: 3, outputs: 4, products: 20 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 2 | 0.95 | 8 | 31 | 180.6 | 0.004 |
| onehot | 4 | 1.2 | 8 | 30 | 235.5 | 0.001 |
| random | 2 | 0.65 | 7 | 30 | 177.0 | 0.001 |
| jedi | 2 | 0.65 | 6 | 26 | 168.3 | 0.002 |
| nova | 2 | 0.65 | 6 | 29 | 185.6 | 0.079 |
| my | 2 | 0.8 | 7 | 25 | 170.4 | 0.004 |

tab 8.4

| **cse** | states: 16, inputs: 7, outputs: 7, products: 91 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 4 | 1.44 | 27 | 215 | 767.3 | 0.006 |
| onehot | 16 | 1.495 | 48 | 221 | 1098.8 | 0.002 |
| random | 4 | 1.571 | 24 | 245 | 879.3 | 0.002 |
| jedi | 4 | 1.44 | 25 | 212 | 764.1 | 0.022 |
| nova | 4 | 1.44 | 26 | 189 | 682.9 | 0.094 |
| my | 5 | 1.231 | 33 | 214 | 762.5 | 0.006 |

tab 8.5

| **dk27** | states: 7, inputs: 1, outputs: 2, products: 14 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 3 | 1.714 | 5 | 23 | 196.8 | 0.004 |
| onehot | 7 | 2 | 9 | 29 | 320.6 | 0.001 |
| random | 3 | 1.571 | 6 | 26 | 196.6 | 0.001 |
| jedi | 3 | 1.714 | 6 | 22 | 183.2 | 0.003 |
| nova | 3 | 1.571 | 6 | 26 | 196.5 | 0.078 |
| my | 4 | 1.286 | 6 | 30 | 245.9 | 0.004 |

tab 8.6

| **lion9** | states: 4, inputs: 2, outputs: 1, products: 16 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 2 | 0.625 | 3 | 16 | 131.8 | 0.004 |
| onehot | 4 | 0.875 | 6 | 20 | 203.0 | 0.001 |
| random | 2 | 0.625 | 3 | 15 | 131.8 | 0.001 |
| jedi | 2 | 0.625 | 4 | 15 | 122.8 | 0.002 |
| nova | 2 | 0.438 | 3 | 16 | 130.7 | 0.077 |
| my | 2 | 0.438 | 3 | 13 | 119.6 | 0.004 |

tab 8.7

| **lion** | states: 4, inputs: 2, outputs: 1, products: 11 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 2 | 0.727 | 3 | 15 | 120.7 | 0.004 |
| onehot | 4 | 1.091 | 5 | 23 | 216.3 | 0.001 |
| random | 2 | 0.727 | 3 | 14 | 127.0 | 0.001 |
| jedi | 2 | 0.727 | 3 | 14 | 127.0 | 0.002 |
| nova | 2 | 0.545 | 3 | 13 | 121.9 | 0.078 |
| my | 2 | 0.545 | 4 | 15 | 126.9 | 0.004 |

tab 8.8

| **mc** | states: 4, inputs: 3, outputs: 5, products: 10 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 2 | 0.7 | 6 | 23 | 154.4 | 0.004 |
| onehot | 4 | 1 | 9 | 32 | 269.5 | 0.001 |
| random | 2 | 0.7 | 7 | 22 | 149.4 | 0.001 |
| jedi | 2 | 0.5 | 6 | 25 | 173.5 | 0.002 |
| nova | 2 | 0.8 | 7 | 27 | 167.0 | 0.078 |
| my | 2 | 0.5 | 6 | 25 | 173.5 | 0.004 |

tab 8.9

| **s27** | states: 5, inputs: 4, outputs: 1, products: 30 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 3 | 1.033 | 7 | 30 | 205.1 | 0.004 |
| onehot | 5 | 1.333 | 7 | 54 | 343.1 | 0.001 |
| random | 3 | 1.133 | 7 | 36 | 237.4 | 0.002 |
| jedi | 3 | 1.033 | 7 | 28 | 197.4 | 0.003 |
| nova | 3 | 1.1 | 4 | 23 | 164.2 | 0.079 |
| my | 3 | 0.867 | 6 | 31 | 206.0 | 0.004 |

tab 8.10

| s386 | states: 13, inputs: 7, outputs: 7, products: 64 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 4 | 1.641 | 18 | 114 | 513.4 | 0.005 |
| onehot | 13 | 1.656 | 31 | 126 | 798.2 | 0.002 |
| random | 4 | 1.656 | 20 | 138 | 564.7 | 0.002 |
| jedi | 4 | 1.563 | 16 | 119 | 543.4 | 0.013 |
| nova | 4 | 1.734 | 18 | 115 | 511.2 | 0.085 |
| my | 4 | 1.375 | 19 | 128 | 560.0 | 0.006 |

tab 8.11

| s510 | states: 47, inputs: 19, outputs: 7, products: 77 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 6 | 1.429 | 25 | 281 | 1241.2 | 0.007 |
| random | 6 | 2.13 | 61 | 507 | 1978.2 | 0.003 |
| nova | 6 | 3.377 | 43 | 353 | 1506.9 | 0.109 |
| my | 6 | 0.831 | 35 | 302 | 1333.6 | 0.008 |

tab 8.12

| s820 | states: 24, inputs: 18, outputs: 19, products: 230 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 5 | 1.961 | 52 | 364 | 1246.4 | 0.011 |
| onehot | 24 | 1.661 | 60 | 278 | 1476.4 | 0.008 |
| random | 5 | 2.013 | 52 | 402 | 1383.6 | 0.008 |
| jedi | 5 | 1.757 | 47 | 325 | 1174.7 | 0.121 |
| nova | 5 | 2.217 | 43 | 283 | 1012.1 | 0.172 |
| my | 7 | 1.587 | 52 | 343 | 1247.1 | 0.015 |

tab 8.13

| s832 | states: 24, inputs: 18, outputs: 19, products: 243 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 5 | 1.922 | 42 | 340 | 1180.6 | 0.012 |
| onehot | 24 | 1.605 | 63 | 287 | 1471.9 | 0.009 |
| random | 5 | 1.984 | 55 | 438 | 1461.1 | 0.009 |
| jedi | 5 | 1.605 | 49 | 347 | 1145.2 | 0.124 |
| nova | 5 | 2.152 | 43 | 294 | 1059.9 | 0.165 |
| my | 7 | 1.654 | 49 | 362 | 1240.5 | 0.016 |

tab 8.14

| **sse** | states: 13, inputs: 7, outputs: 7, products: 208 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 4 | 1.683 | 19 | 123 | 542.6 | 0.009 |
| onehot | 13 | 1.692 | 27 | 127 | 828.1 | 0.003 |
| random | 4 | 1.49 | 18 | 135 | 638.8 | 0.003 |
| jedi | 4 | 1.966 | 18 | 121 | 546.2 | 0.022 |
| nova | 4 | 1.721 | 18 | 109 | 505.7 | 0.105 |
| my | 4 | 1.409 | 17 | 123 | 511.6 | 0.01 |

tab 8.15

| **tav** | states: 4, inputs: 4, outputs: 4, products: 49 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 2 | 1.551 | 8 | 27 | 164.0 | 0.004 |
| onehot | 4 | 2 | 5 | 21 | 209.8 | 0.002 |
| random | 2 | 1.551 | 8 | 28 | 164.9 | 0.002 |
| jedi | 2 | 1.449 | 8 | 27 | 164.0 | 0.002 |
| nova | 2 | 1.449 | 8 | 27 | 164.0 | 0.079 |
| my | 2 | 1 | 7 | 24 | 152.4 | 0.004 |

tab 8.16

| **tbk** | states: 16, inputs: 6, outputs: 3, products: 1024 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 4 | 1.884 | 21 | 260 | 883.3 | 0.03 |
| onehot | 16 | 1.764 | 52 | 223 | 985.7 | 0.01 |
| random | 4 | 1.883 | 23 | 273 | 930.0 | 0.01 |
| jedi | 4 | 1.878 | 28 | 247 | 841.7 | 0.148 |
| nova | 4 | 1.881 | 29 | 256 | 841.6 | 0.285 |
| my | 15 | 0.991 | 51 | 224 | 995.1 | 0.035 |

tab 8.17

| **train11** | states: 4, inputs: 2, outputs: 1, products: 15 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 2 | 0.533 | 3 | 11 | 108.1 | 0.004 |
| random | 2 | 0.667 | 4 | 16 | 126.2 | 0.001 |
| jedi | 2 | 0.533 | 3 | 11 | 108.1 | 0.002 |
| nova | 2 | 0.667 | 3 | 17 | 132.1 | 0.078 |
| my | 2 | 0.533 | 3 | 12 | 107.7 | 0.004 |

tab 8.18

| **train4** | states: 4, inputs: 2, outputs: 1, products: 14 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| binary | 2 | 0.714 | 3 | 13 | 104.7 | 0.004 |
| onehot | 4 | 1 | 6 | 19 | 174.5 | 0.001 |
| random | 2 | 0.714 | 3 | 12 | 110.9 | 0.001 |
| jedi | 2 | 0.5 | 4 | 17 | 116.5 | 0.002 |
| nova | 2 | 0.5 | 4 | 14 | 110.1 | 0.078 |
| my | 2 | 0.5 | 4 | 14 | 110.1 | 0.004 |

tab 8.19

| **shiftreg** | states: 8, inputs: 1, outputs: 1, products: 16 | | | | | |
|---|---|---|---|---|---|---|
| method | bits | avg. H | nodes | lits | power(uW) | time(s) |
| onehot | 8 | I.75 | 15 | 40 | 313.5 | 0.001 |
| random | 3 | 1.5 | 5 | 36 | 260.9 | 0.001 |
| jedi | 3 | 1.5 | 4 | 10 | 117.5 | 0.004 |
| nova | 3 | 1.5 | 2 | 8 | 115.0 | 0.079 |
| my | 4 | 1.125 | 5 | 37 | 284.8 | 0.004 |

tab 8.20

| **SOUHRN** | | | | | | |
|---|---|---|---|---|---|---|
| statistika | bits | avg. H | nodes | lits | power | time* |
| Nejlepší (nebo stejný jako nejlepší) ze všech | 14/20 **(70%)** | 18/20 **(90%)** | 5/20 **(25%)** | 3/20 **(15%)** | 3/20 **(15%)** | 0/46 **(0%)** |
| Nejhorší (nebo stejný jako nejhorší) ze všech | 0/20 **(0%)** | 0/20 **(0%)** | 0/20 **(0%)** | 1/20 **(5%)** | 1/20 **(5%)** | 0/46 **(0%)** |
| Stejný nebo lepší než NOVA | 14/20 **(70%)** | 19/20 **(95%)** | 10/20 **(50%)** | 10/20 **(50%)** | 7/20 **(35%)** | 45/46 **(98%)** |
| Stejný nebo lepší než JEDI | 13/19 **(69%)** | 17/19 **(90%)** | 11/19 **(58%)** | 7/19 **(37%)** | 10/19 **(53%)** | 26/46 **(57%)** |

tab. 8.21

*statistika doby kódování je brána ze všech zakódovaných souborů

# 9.0 **Závěr**

V práci jsem se pokusil vytvořit přehled o dohledaných metodách kódování stavů FSM od jednoduchých po složitější. Následně jsem vymyslel vlastní algoritmus pro zakódování FSM a ten kromě jiných implementoval ve dvou aplikacích.

První z nich je jednoduchá utilita, která načte popis FSM ze souboru formátu KISS, zakóduje zvolenou metodou a uloží do formátu BLIF. Druhou aplikací je upravená aplikace systému EDuArd – stubs (autorem rozhraní je Ing. Jan Schmidt, Ph.D.), do které jsem kodér zabudoval. Bohužel v době psání této práce ještě EDuArd nepodporuje v exportních modulech kódy stavů, proto byla pro testy využita první utilita.

Dle naměřených výsledků je algoritmus velmi efektivní (nejlepší nebo jeden z nejepších v 90% testů a nikdy nejhorší) z hlediska minimální Hammingovi vzdálenosti mezi jednotlivými stavy, tedy i minimální switching activity při zachování nízkého počtu bitů.

Co se počtu hradel, literálů a ztrátového výkonu týče, je můj algoritmus zhruba v 50% případů srovnatelný s JEDI (resp. NOVA). V 57% (resp. 98%) případů byl můj algoritmus rychlejší než jmenované, u složitých FSM velmi výrazně (např. benchmark s298 – doba kódování pomocí JEDI: cca 30s, mým algoritmem 1,5s). Celkově u jednoduchých FSM s málo stavy byla doba kódování relativně zanedbatelná pro všechny algoritmy – pohybovala se v řádu mikro až milisekund.

Na největší problém jsem kromě špatné dostupnosti materiálů narazil při integraci algoritmu do EDA systému zejména kvůli jeho rozsáhlosti a velké abstraktnosti. Problém jsme nakonec vyřešili s Ing. Fišerem a Ing. Schmidtem.

# 10.0 **Seznam literatury**

[1] Doc. Ing. Haně Kubátové, CSc.: přednášky X36LOB
[2] Michal Prokeš: diplomová práce (FEL code)
[3] Robert Eggermont: PROSA – Profiling-based State Assignment for Low Power Dissipation
[4] Luis Mengibar, Luis Entrena, Michael G. Lorenz, Raúl Sánchez-Reillo: State Encoding for Low-Power FSMs in FPGA
[5] David A. Bader, Kamesh Madduri: A Parallel State Assignment Algorithm
[6] Elena Fomina: State encoding technique for low power FSM
[7] Marek Perkowski: Digital design automation – Finite state machine design
[8] State Encoding Techniques for Low-Power FSMs
[9] Alberto Sangiovanni-Vincentelli a kolektiv: SIS - A System for Sequential Circuit Synthesis

# A. Uživatelská příručka

Jednoduchá utilita *fsmEncoder.exe*:

spuštění: *fsmEncoder.exe <cislo_metody> vstup.kiss [vystup.blif]*

nebo *fsmEncoder.exe <cislo_metody> < vstup.kiss [>][vystup.blif]*

kde: <cislo_metody> značí použitou metodu kódování, kde

1 = moje metoda kódování
2 = binární kódování
3 = onehot kódování
4 = brute-force kódování

př. *fsmEncoder.exe 1 bbara.kiss2 > bbara.blif*

Aplikace *stubs* systému EDuArd:

spuštění: *./stubs –I <vstup> -F <metoda> -l <knihovna> -e <entita> -a <architektura> -O <vystup> -A <vystupni_architektura>*

kde: parametry –l, -e, -a, -A jsou popsány v dokumentaci k systému EDuArd, -I značí vstupní soubor (nejčastěji KISS), -O značí výstup (v současné době ještě není export zakódovaného FSM implementován), -F je metoda kódování, kde <metoda> může být jedna z následujících:

own = moje metoda kódování
binary = binární kódování
onehot = one-hot kódování
s_onehot = one-hot kódování s počátkem v 000...
zerohot = zero-hot kódování
s_zerohot = zero-hot kódování s počátkem v 111...
random = náhodné kódování (min. H z max. 20ti permutací)
brute = postupné procházení všech možností (hledá min. H)
twohots = two-hots kódování
johnson = neoptimalizované Johnsonovo kódování

př.
*./stubs -I bbara.kiss2 –F own -l kiss_library -e bbara -a kiss -O bbara.dbg –A KISS*

# B. Obsah přiloženého CD

bakalarka.docx - bakalářská práce ve formát doc
bakalarka.pdf - bakalářská práce ve formát pdf
adresář benchmarky – použité benchmarky

benchmarky.zip - benchmarky v původním stavu
minimized.zip - benchmarky po minimalizaci stavů
encoded.zip - zakódované testy ve formátu BLIF
stats - statistiky získané při testování v SISu

adresář EDuArd - EDA systém EDuArd

upravene.txt - seznam změn a přidaných souborů do archivů

adresář encoder_final – jednoduchá verze kodéru (viz příloha A)

adresář materialy - materiály, ze kterých jsem čerpal (viz 10.0)

-číslo v hranatých závorkách v názvu souboru značí referenci na položku v seznamu použité literatury

adresář sis – balík SIS verze 1.3.6

adresář web - skripty použité pro vygenerování statistiky a tabulek testů