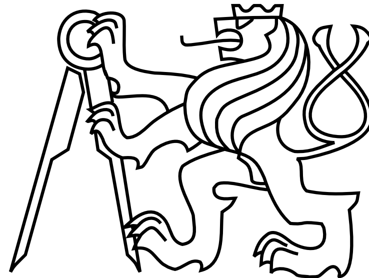
České vysoké učení technické v Praze
Fakulta elektrotechnická



Bakalářská práce

Syntéza asynchronních sekvenčních obvodů

Zbyněk Moler

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika strukturovaný bakalářský

Obor: Informatika a výpočetní technika

červenec 2008

Poděkování

Rád bych poděkoval rodině za podporu.

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).

V Praze dne 4.7.2008 ....

Abstract

The thesis deals with design and implementation of a tool for synthesis of asynchronous sequential circuits. The input of the tool is a file describing signal transition graph. The tool is a console application, which doesn't use any external applications. The output of this application is a description of circuit in VHDL code.

At the end of the thesis there is a made simulation of the selected circuit and C-element. The simulation of the circuit is executed on the basis of VHDL code, which is obtained from the created tool.

Abstrakt

Práce se zabývá návrhem a implementací programu pro syntézu asynchronních sekvenčních obvodů. Vstupem programu je soubor popisující signálově přechodový graf. Program je konzolová aplikace, která nevyužívá žádné externí programy. Výstupem programu je popis obvodu ve VHDL kódu.

Na konci práce je provedena simulace vybraného obvodu a C-elementu. Simulace pro vybraný obvod proběhne na základě VHDL kódu získaného z vytvořeného programu.

Obsah

Seznam obrázků xv

Seznam tabulek xvii

1 Úvod 1
1.1 Motivace . . . 1
1.2 Synchronní obvody a synchronizace . . . 1
1.3 Proč má smysl se zabývat asynchronními obvody . . . 1
1.4 Hlavní nevýhody asynchronních obvodů . . . 1
1.5 Přínos . . . 2

2 Popis problému, specifikace cíle 3
2.1 Úvod do návrhových vzorů . . . 3
2.2 Popis problému . . . 3
2.3 Vytýčení cílů . . . 3
2.3.1 Pojmové cíle . . . 3
2.3.2 Implementační cíle . . . 4
2.3.3 Simulační cíl . . . 4
2.4 Čím se zabývat nebudu . . . 4
2.5 Existující nástroje pro syntézu asynchronních obvodů . . . 4
2.5.1 ATACS online . . . 4
2.5.2 Petrify . . . 5

3 Uvedení do problematiky 7
3.1 Speed Independent obvody . . . 7
3.1.1 Hazardy . . . 7
3.1.2 Modely zpoždění (Delay model) . . . 7
3.1.3 Vstupně-výstupní mód (Input-Output mode) . . . 7
3.1.4 Syntéza Speed Independent obvodu . . . 8
3.1.5 Proč jsem použil C-element . . . 8
3.1.6 Signálově přechodový graf (Signal transition graph, STG) . . . 8
3.1.7 Stavový graf (State Graph, SG) . . . 9
3.1.8 Regiony . . . 10
3.1.9 Excitované regiony(Excistation region, ER) . . . 10
3.1.10 Ustálené regiony (Quiescent region, QR) . . . 10
3.1.11 Minimalizace . . . 10
3.1.12 Časový diagram (Timing diagram) . . . 11
3.1.13 Vztah signálově přechodového grafu a časového diagramu . . . 12
3.1.14 Matematické vyjádření algoritmu pro transformaci STG na Stavový Graf . 12
3.1.15 Muller C-element . . . 14
3.1.16 C-element ve VHDL . . . 16
3.2 Fundamentální mód . . . 17
3.2.1 Syntéza podle fundamentálního módu . . . 18
3.2.2 Nevýhody oproti Vstupně-výstupnímu módu . . . 18
3.3 Příklad postupu syntézy . . . 18

4 Analýza a návrh řešení 25
4.1 Shrnutí před analýzou a návrhem . . . 25

4.2 Implementační jazyk . . . 25
4.3 Vstup programu . . . 25
4.4 Reprezentace grafu . . . 25
4.5 Návrh reprezentace grafu . . . 25
4.6 Počáteční zakódování . . . 26
4.7 Návrh reprezentace počátečního zakódování . . . 26
4.8 Vstupy, výstupy a interní signály . . . 26
4.9 Návrh reprezentace vstupů, výstupů a interních signálů . . . 26
4.10 Marking . . . 26
4.11 Návrh přidělování tokenů . . . 26
4.12 Modul zpracování textu . . . 26
4.13 Návrh modulu pro zpracování textu . . . 27
4.14 Modul pro zpracování příkazů . . . 27
4.15 Návrh modulu pro zpracování příkazů . . . 27
4.16 Modul pro transformaci . . . 27
4.17 Návrh modulu pro transformaci . . . 27
4.18 Modul pro vytvoření regionů . . . 28
4.19 Návrh modulu pro vytváření regionů . . . 28
4.20 Modul pro minimalizaci . . . 28
4.21 Důvody použití vlastního minimalizačního algoritmu . . . 29
4.22 Algoritmus minimalizace . . . 29
4.23 Návrh pro modul minimalizace . . . 29
4.24 Návrh algoritmu . . . 30
4.25 Výstup programu . . . 30
4.26 Návrh výstupu programu . . . 30
4.27 Modul pro generování VHDL . . . 30
4.28 Návrh modulu pro generování VHDL . . . 30

5 Implementace **31**
5.1 Všeobecné informace k implementaci . . . 31
5.2 Vstup programu . . . 31
5.3 Třída FromFile . . . 31
5.4 Třída DoCommand . . . 31
5.5 Třída Prikaz . . . 31
5.6 STG v programu . . . 31
5.7 Třída MakeSG . . . 34
5.8 Stuktura stavového grafu . . . 35
5.9 Třída Region . . . 37
5.10 Třída KMap . . . 37
5.11 Třída MakeCorLog . . . 37

6 Testování **39**
6.1 Vstupní zdroje pro naimplementovaný program . . . 39
6.2 Výstup ATACS . . . 39
6.3 Vybrané grafy . . . 39
6.4 Průběh syntézy na ATACS . . . 40
6.5 Výstup z ATACS . . . 40
6.6 Průběh generování VHDL kódu v naimplementovaném programu . . . 40
6.7 Výsledné srovnání . . . 42
6.8 Časová složitost . . . 43
6.9 Testování vstupu . . . 44

6.10 Závěr testování 44

7 Simulace **45**
7.1 Pomůcky pro simulaci 45
7.2 Postup simulace, vytvoření simulace 45
7.3 Nastavení Xilinx ISE Simulátoru 45
7.4 Simulace C-elementu 47
7.5 Nastavení pro C-element 47
7.6 Výstup simulace pro C-element 47
7.7 C-element v FPGA 49
7.8 Simulace asynchronního obvodu 50
7.9 Závěr simulace 53

8 Závěr **55**

9 Literatura **57**

A Seznam použitých zkratek **59**

B Uživatelská příručka **61**
B.1 Vstup programu 61
B.2 Píšeme vstup 62
B.3 Spuštění programu a manipulace 62

C Obsah přiloženého CD **63**

Seznam obrázků

3.1 Signálově přechodový graf a časový diagram 11
3.2 Transformační algoritmus 12
3.3 Vnitřní reprezentace C-elementu 15
3.4 C-elementu typ 1 16
3.5 C-elementu typ 2 16
3.6 C-elementu typ 3 16
3.7 Huffmanuv mód 18
3.8 Časový diagram 19
3.9 Signálově přechodový graf 20
3.10 Stavový graf 21
3.11 Regiony naznačení 22
3.12 Karnaughova mapa 23
3.13 Výsledná logika 24

5.1 Struktura STG 33
5.2 Struktura SG 36

7.1 Nastavení Xilinx Ise Simulátoru 46
7.2 Vstupy pro C-element 47
7.3 Výstup pro C-element 48
7.4 Schéma C-elemntu 49
7.5 Časový diagram 50
7.6 Vstup pro obvod var. 1 51
7.7 Výstup pro obvod var. 1 51
7.8 Vstup pro obvod var. 2 52
7.9 Výstup pro obvod var. 2 52

Seznam tabulek

3.1 Ukázka tabulky 14

6.1 Vybrané grafy z ATACS 43
6.2 Parametry grafů popsané v souborech test1.g a test2.g. 44
6.3 Časové údaje pro úseky výpočtu pro test1.g a test2.g. a počet stavů v signálově přechodovém grafu 44

7.1 Ukázka tabulky 47
7.2 Pravdivostní tabulka pro LUT 49
7.3 Pravdivostní tabulka pro vnitřní rovnici popisující C-element 50

1 Úvod

1.1 Motivace

V dnešní době je výkon procesorů a mikrokontrolerů závislý na mnoha faktorech. Jeden z nich je hodinový takt a ve světě návrhářů sekvenčních obvodů je snaha jej neustále zvyšovat za účelem vyššího výkonu. Jednou z metod jak dosáhnout lepších hodnot hodinového taktu je např. pipelening, kdy dochází ke vkládání klopných obvodů do kombinační logiky. Já se však vydám jinou cestou jak zvýšit výkon, rozhodl jsem se úplně vypustit logiku související s hodinovým taktem. Na základě těchto souvislostí jsem došel až k asynchronním obvodům.

Asynchronní obvody nejsou převratnou novinkou ve světě obvodových návrhářů, avšak jejich rozšíření není takové jako u synchronních obvodů. Jak u synchronních obvodů, tak i u asynchronních obvodů musíme provést syntézu, abychom získali výslednou logiku. Proto jsem se rozhodl seznámit se s postupem syntézy u asynchronních obvodů a s pojmy popisující tyto obvody. A aby moje nabyté znalosti nebyly jen v myšlenkové formě, vytvořil jsem program, který bude automatizovat postup syntézy. Výsledkem syntézy bývá popis logického obvodu, proto výstup programu by měl tento fakt respektovat a nejlépe jej popsat v jazyce, který je pro návrháře typický a to je VHDL kód.

1.2 Synchronní obvody a synchronizace

Synchronizaci obvodu provádíme proto, protože některé části obvodu jsou rychlejší než jiné a je třeba zajistit ustálení všech signálů před dalším zpracováním. Synchronizace obvodu se dnes obvykle provádí pomocí hodinového signálu, který je rozveden po obvodu. Obvody s rozvedeným hodinovým signálem se nazývají synchronní a jejich převaha na trhu je enormní. Příčinou převahy je jejich jednoduchý a spolehlivý návrh, dobrá přenositelnost a jednoduché testování.

1.3 Proč má smysl se zabývat asynchronními obvody

Upozornil jsem na fakt, že limitujícím faktorem synchronního logického obvodu je hodinový takt zpomalující běh obvodu, který by potencionálně mohl běžet rychleji. Řešení, které navrhuji spočívá ve využití asynchronních obvodů, které však neskýtají pouze výhodu vyšší operační rychlosti, ale taky nižší spotřebu energie obvodu, menší emise elektro-magnetického šumu, robustnost vzhledem ke skokovým změnám dodávky napětí a teplotní změně.

1.4 Hlavní nevýhody asynchronních obvodů

Asynchronní obvody by mohly přinést mnohé výhody, které by synchronní obvody mohly jen stěží nabídnout, ale nic není tak jednoduché jak to na první pohled vypadá.

Předností synchronních obvodů je jejich výborná testovatelnost, která je umožněna testovatelností malých celků nezávislých na zbytku obvodu. V asynchronním světě spolu komunikuje celý obvod v jednom okamžiku, a tak jen stěží se dá vyjmout část obvodu, aniž bych se nemuselo ohlížet na zbytek.

Další nevýhodou asynchronních obvodů může být paradoxně odstranění hodin a vodičů, které rozvádí hodinový signál. Důvodem je nárůst logiky, která musí nahradit hodinový signál

sloužící k synchronizaci.

Posledním problémem, ne však nevýhodou, je zaručení bezhazardnosti v obvodu.

1.5 Přínos

Práce má přinést nejen programový nástroj, který by měl být schopen syntetizovat asynchronní obvod, ale taky by měla být informačním zdrojem na téma asynchronní obvody.

2 Popis problému, specifikace cíle

2.1 Úvod do návrhových vzorů

Pro návrh asynchronních obvodů se používají v současné době různé návrhové styly. Styly se od sebe liší modely zpoždění, o které se opírají. Základní dva modely zpoždění jsou model vázaného a nevázaného zpoždění. Modely zpoždění jsou podkladem pro módy. Módy jsou metodiky, jak si obvody signalizují povolení dalších vstupů a stabilitu obvodu.

Návrhový styl založený na modelu vázaného zpoždění je propagován Huffmanovým módem[10]. Na opačném pólu stojí vstupně-výstupní mód[10] s modelem nevázaného zpoždění podporující obvody nazývané Speed Independent [8]. Co se týče návrhového stylu zahrnujícího Speed Independent obvody, pak mohu konstatovat, že je velice populární a budu se mu věnovat v následující práci.

2.2 Popis problému

Každá syntéza obvodu je závislá na vstupní specifikaci. Vstupní specifikace pro syntézu asynchronních obvodů, dodržující podmínky stanovené Speed Independent obvody [8], mohu vyjádřit v grafové formě. Konkrétní graf, který pojme celou problematiku specifikace se nazývá signálově přechodový graf [9]. Signálově přechodový graf vychází z Petriho sítí [5] a dodržuje některé podmínky stanovené Petriho sítěmi. Navíc bude po signálově přechodovém grafu požadováno, aby byl omezený pouze na jednoduché cykly. Po nastínění specifikace vstupu, mohu přejít k dalšímu kroku a tím je transformace signálově přechodového grafu do stavového grafu [9] [3]. Účelem získání stavového grafu je jeho poměrně jednoduché převedení na logiku. Při transformaci signálově přechodového grafu na stavový graf může dojít k porušení pravidel pro realizaci stavového grafu, proto je třeba provádět kontrolu během transformace. Každý stav stavového grafu má vlastní binární kód, který je třeba přiřadit pro jednotlivé signály do jejich regionů [8]. Pokud jsou pro signály vytvořené regiony, tak na jejich základě následuje vytvoření funkcí charakterizující chování vstupů C-elementu [8]. Každý C-element v obvodu slouží jak k odstranění hazardů,tak i k uchování informační hodnoty po určitou dobu. Výstupem C-elementu je buď výstupní nebo interní signál. Na základě vytvořených funkcí lze již získat logiku popisující obvod. Mou snahou nyní bude výše zmíněný problém převést do programu, který bude schopen automatizovat syntézu. Vytvořené funkce, ze kterých se získává logický obvod, lze ještě minimalizovat, proto se program pokusí postupnou expanzí počet proměnných ve funkci minimalizovat a na závěr program převede funkce i s naportováním na C-element do VHDL kódu.

2.3 Vytýčení cílů

Jelikož syntéza asynchronních obvodů není příliš rozšířená, zaměřím se zprvu na podrobnější rozpracování potřebných pojmů. Proto bude vhodné rozdělit cíle na pojmové, implementační a simulační.

2.3.1 Pojmové cíle

Účelem této sekce bude zavést si pojmy jako je signálově přechodový graf, Speed Independent obvody a podmínky, které musí splňovat, abychom mohli přistoupit k transformaci. Popsat si regiony, stavový graf a C-elementem . Objasnit si matematické vyjádření transformačního algoritmu ze signálově přechodového grafu do stavového grafu. Jelikož Speed Independent obvod

je položen na Vstupně-výstupním módu, měl bych se obeznámit s jeho základy i se základy jiných módu a naznačit jejich rozdíly.

2.3.2 Implementační cíle

Nejprve získat signálově přechodový graf v textové formě. Textová forma by měla být vstupem programu. Program by měl být členěn do jednotlivých modulů, které by měly mít daný vstup a výstup. Cílem členění do modulu je zajistit možnost kterýkoliv modul přepracovat s tím, že zůstanou dodrženy vstupy a výstupy. Moduly by se měly starat o následující: převádění textového vyjádření signálově přechodového grafu do datové podoby, datový obraz signálově přechodového grafu přetransformovat do stavového grafu. Následně se postarat o regiony ze stavového grafu a vytvořit funkce a minimalizovat počet proměnných. Funkce a jejich vztah k C-elementu, vstupů, výstupů a interních signálů převést na VHDL kód.

2.3.3 Simulační cíl

Odsimulovat si vybraný asynchronní obvod, který bude splňovat podmínky z pojmové části a projde naprogramovaným programem. Simulace by měla být založena na VHDL kódu, získaného jako výstup programu. Taktéž samostatně si odsimulovat C-element.

2.4 Čím se zabývat nebudu

Nebudu se zabývat dekompozicí logiky, vkládání signálu, pokud dojde k redundantnímu zakódování a ani opravou špatně specifikovaného signálově přechodového grafu.

2.5 Existující nástroje pro syntézu asynchronních obvodů

2.5.1 ATACS online

ATACS je online program pro syntézu asynchronních obvodů, který lze nalézt na stránkách [4]. Tento program je velice schopný, může syntetizovat z různých vstupních popisů asynchronních obvodů. Výsledek syntézy jsou buď atomické hradla nebo kombinace s C-elementem . Výstup programu je textový popis v němž je naznačený vztah mezi signály a použitými hradly. Já se konkretně zaměřím na vstupní soubory s příponou *.g*, jelikož tyto soubory popisují signálově přechodový graf. Z tohoto textového popisu následně budu vycházet pro vstup programu.

Musím konstatovat, že tento nástroj pro syntézu bude na vyšší úrovní než můj program. Avšak naším cílem je si vyzkoušet postup syntézy asynchronních obvodů a seznámit se s ním. Jedinou výhodou od ATACSu je ,že můj nástroj pro syntézu vygeneruje přímo VHDL kód, který pak budu moci použít k simulaci.

Jelikož vstupem programu bude textová forma signálově přechodového grafu, tak budu vycházet ze stejného základu jako jsou soubory u ATACS. Navíc budu předpokládat kompatibilitu vstupních souboru mezi programem a ATACS. Účelem kompatibility je porovnat vygenerovaný VHDL kód s výstupem z ATACS.

2.5.2 Petrify

Dalším nástrojem určeným pro syntézu asynchronních obvodů je Petrify. Jedná se o jeden z nejznámějších nástrojů pro syntézu asynchronních obvodů. Dokáže transformovat různé vstupní popisy a převádí si je do své vlastní vnitřní formy. Velkou výhodou je optimalizace logiky na dostupné typy hradel. Pracuje jak se Speed Independent obvody, tak i obvody založenými na Huffmanovém módu. Seznámit se s tímto programem můžeme na stránkách [2].

Petrify je volně stažitelný program typu konzolové aplikace. Program je určen především pro operační systém UNIX, a proto se veškeré preference nastavují pomocí přepínačů. Vstup a výstup programu je opět v textové formě. Textová specifikace pro signálově přechodový graf se dá získat z grafického obrazu grafu. Program, který umí vygenerovat z grafického obrazu grafu textovou specifikaci pro Petrify se jmenuje VSTGL (Visual STG Lab) [6].

Petrify je opět velice důmyslný nástroj pro syntézu asynchronních obvodů, výhodou mého programu by opět měla být schopnost generování VHDL kódu.

3 Uvedení do problematiky

3.1 Speed Independent obvody

Jsou součástí návrhového stylu zabývající se syntézou asynchronních obvodů. Tyto obvody jsou bezhazardní v tzv. vstupně-výstupním módu a používají model nevázaného zpoždění. Speed Independent obvody spoléhají na model, který se staví pesimisticky ke zpoždění na hradlech a realisticky předpokládá zpoždění na vodičích. Obvody se neohlížejí na globální omezení vnějším prostředím. Každá vstupní změna je signalizována obvodu, že vnější prostředí je připraveno nastavit další vstupní vzor. Obvody jsou robustní vzhledem k nepravidelným změnám parametrů jako jsou procesní změny, změny napětí a teploty. Jsou jednodušeji verifikovatelnější než obvody, které jsou založeny na modelu vázaného zpoždění. Nejpopulárnější specifikační styl pro Speed Independent obvody jsou signálově přechodové grafy. Pro získání více informací je vhodné navštívit stránky [1].

3.1.1 Hazardy

Pro návrháře obvodu je hazard nechtěný glitch na signálu. Obvod, který je ve stabilním stavu, samovolně neprodukuje hazardy. Hazardy se vztahují k dynamickým operacím v obvodě. Je to tedy vztah mezi vstupními signály, které se dynamicky mění, hradly a vodiči. Výsledkem snažení návrháře je udělat obvod bezhazardní (hazard-free).

Když mluvíme o hazardech musíme vzít v úvahu jaký model zpoždění obvod praktikuje a jaké jsou vytvořeny předpoklady ohledně interakce mezi obvodem a vnějším prostředím. Pro asynchronní obvody je specifické, že předpokládáme jak zpoždění na hradlech, tak i zpoždění na vodičích. Na základě zpoždění na vodičích a hradlech, pak specifikujeme model zpoždění.

3.1.2 Modely zpoždění (Delay model)

Nejednoduší model zpoždění je fixní model zpoždění (fixed delay model), kde je zpoždění konstantní. Alternativou je min-max model zpoždění, kde zpoždění je neznámé, ale je stanovena dolní a horní hranice $t_{min} <= t_{delay} <= t_{max}$. Více pesimistický model je model nevázaného zpoždění (unbounded delay model), kde zpoždění je pozitivní. $0 < t_{delay} < \infty$.

3.1.3 Vstupně-výstupní mód (Input-Output mode)

Předpokládáme, že obvod se nachází ve stabilním stavu, pak je dovoleno vnějšímu prostředí měnit vstupy. Při změně obvod vyprodukuje korespondující výstup a opět je vnějšímu prostředí dovoleno měnit vstup. Neberou se v úvahu interní signály, protože další vstupní změna nastane až obvod bude stabilizován, což je odpověď na předchozí vstupní signál. Změna vstupního signálu není časově vázaná.

Omezení na vnější prostředí je formulováno jako příčina vztahu mezi vstupními signály přechodů a výstupními signály přechodů. Z tohoto důvodů jsou obvody často popisovány nákresy, které sledují všechny možné sekvence vstupních a výstupních signálů přechodů a které mohou být v obvodu pozorovány. Proto se pro tyto účely nejvíce hodí signálově přechodový graf.

3.1.4 Syntéza Speed Independent obvodu

Je založena na vstupní specifikaci popsaného pomocí signálově přechodového grafu, který splňuje požadavky níže uvedené nebo literatury [10]. Proto je nezbytné provést analýzu obvodu a zohlednit v ní požadavky Speed Independent obvodů jako jsou bezhazardnost obvodu, vstupní-výstupní mód a model zpoždění. Pokud je správně vytvořený signálově přechodový graf, mohu přejít k jeho transformaci na stavový graf. Výsledkem transformace je stavový graf, jež dodržuje kompletní zakódování stavů. Jinými slovy, ve stavovém grafu neexistují dva stavy se stejným zakódováním. Splním-li zakódování a vytvořím stavový graf, pak je třeba pro jednotlivé výstupy a interní signály přechodů vytvořit regiony. Po vytvoření regionů jsem připraven na tvorbu funkcí, které jsou vstupy C-elementu. C-element je předřazen před každý výstup nebo interní signál. Aby asynchronní obvody fungovaly správně, je třeba zajistit bezhazardnost celému obvodu. Pro synchronizaci signálu se používá C-element, bezhazardost C-elementu je zajištěna jeho strukturou. Pokud se ve funkcích Set nebo Reset vyskytuje více jak jeden minterm a je tedy nezbytné v obvodu použít hradlo OR, pak je třeba zjistit, zda proměnné v mintermu Reset nejsou taky obsaženy v mintermech Set, jestliže dojde k schodě je potřeba zajistit bezhazardnost.

3.1.5 Proč jsem použil C-element

Obvody lze samozřejmě sestavit i z atomických hradel, ale aby takový obvod byl bezhazardní, pak všechny zpoždění musí být formulovány na výstupu posledního hradla pro každý signál sítě. Ovšem, když takový obvod namapujeme do opravdových hradel a zpoždění těchto hradel jsou posuzovány individuálně, pak se v implementaci mohou začít šířit hazardy v důsledku rozdílných rychlostech na cestách. Pokud požaduji, aby se rychlosti synchronizovaly, a tak předešel vzniku šíření hazardů, musím použít prvek, který nějakou dobu drží stejnou hodnotu stavu. Stejnou hodnotu stavu drží tak dlouho, dokud mu nepřijde signál, že může přijmout další stav. Takovým prvkem v asynchronních obvodech je C-element. C-element přijímá dva vstupní signály, které prvku udávají zda má držet stav nebo nastavit nový stav na výstup . C-element, který budu používat má jeden vstup invertovaný a jedná se o tzv SET-RESET C-element. Struktura C-elementu zajišťuje, že v tomto komplexním prvku nevznikne hazard.

3.1.6 Signálově přechodový graf (Signal transition graph, STG)

Signálově přechodový graf je interpretací Petriho sítí, jehož přechody jsou interpretovány jako rostoucí a klesající signály. Rostoucí resp. padající hrany jsou označeny v grafu „+“ resp. „-“. STG má dva typy vrcholů: přechody a místa. Místa mohou být označkovány tokenem (obvykle značíme černým puntíkem). Token je označení aktivního místa. Množina všech míst (pozn. M_0), které jsou aktuálně označené tokenem, označujeme jako *marking*. Přechody jsou aktivní, jestliže všechny místa vstupující do přechodů jsou označeny tokenem. Každý aktivní přechod může být tzv. *„odpálen“*. *Odpálení* přechodu znamená odebrání tokenu ze všech vstupních míst daného přechodu a přemístit je na místa výstupní daného přechodu. Místa, jež jsou dosažitelná a označitelná mohou asociovat s určitým binárním kódem signálů. Na základě tohoto poznatku jsme schopni vytvořit určitou souvislost mezi STG a SG. Signál může být buď v jednoduchém cyklu, když má pouze jeden rostoucí a jeden padající přechod, jinak se jedná o více cyklický signál. Grafické vyjádření dovoluje návrháři uchopit chování asynchronního obvodu a navíc zde existuje jistá podobnost s časovým diagramem. Více v literatuře [9].

STG je šestice, kde $\langle P, T, F, M_0, N, s_0, \lambda_0 \rangle$

P je množina míst.
T je množina přechodů.
$F \subseteq (P \times T) \cup (T \times P)$ je množina hran, kde žádná hrana nemůže spojovat dva přechody nebo dvě místa.
$N = I \cup O$ je průnik množin signálů, kde I je množina vstupních signálů a O je množina výstupních signálů.
s_0 je počáteční hodnota pro každý signál v počátečním stavu.
$\lambda_0 \rightarrow N \times \{+, -\}$ je přechodově označená funkce.
M_0 je množina míst obsahující token.

Podmínky souvislosti Petriho sítě a STG:

1.Input free choice: Výběr mezi alternativy musí být kontrolovány navzájem se vylučujícími vstupy.

2.1-bounded: Na jednom místě se nesmí vyskytovat více než jeden token.

3.Liveness: V STG se nesmí vyskytovat deadlocky.

Podmínky STG:
STG je živý(live) a bezpečný(safe):

1.Základ je položen na 1-bounded Petriho sítě.

2.STG je silně souvislý graf

3.Klesající a rostoucí přechody jsou v grafu vystřídány.

Jestliže jsou splněny předchozí podmínky můžeme dojít k uspokojení následujících podmínek:
Kompletní zakódování (CSC)

1.Jestliže každý stav má rozdílný binární kód .

2.Nebo když dva nebo více stavů mají identický binární kód a nevstupní signály přechodu jsou aktivovány, pak se jedná o ty samé.

Podmínky pro STG popisující Speed Independet obvod:

1.Konzistence: přechody signálu se musí striktně střídat mezi + a -.

2.Musí být persistentní: Jestliže signál přechodu je aktivní, tak na vstupním místě přechodu je token a zároveň nemůže být deaktivován jiným signálem. (pozn. Obvod musí garantovat persistenci výstupních a interních signálů, zatímco pro vstupy musí být persistence garantována prostředím).

3.1.7 Stavový graf (State Graph, SG)

Je binární reprezentace stavů, kde událost je reprezentována signály přechodů.
Stavový graf je trojice $\langle S, \delta, \lambda_s \rangle$
S je množina stavů.
$\delta \subseteq S \times T \times S$ je množina přechodů.
$\lambda_s : S \rightarrow (N \rightarrow \{0, 1\})$ je funkce pro označení stavu.
T je množina přechodů Signal Trasition Grafu
N je množina průniku množin vstupních a výstupních množin signálů v Signal Trasition Grafu.

Každý stav s je označen vektorem $\langle s(0), s(1), ...s(n)\rangle$, kde s(i) je buď 1 nebo 0., hodnota je indikována funkcí λ_S.

3.1.8 Regiony

Jsou množiny binárních hodnot stavů stavového grafu pro daný výstupní nebo interní signál. Regiony se dělí do dvou základních regionů: excitovaný a ustálený. Stavy se do těchto množin přidělují na základě vztahu binárních hodnot stavů k danému signálu. Význam regionu spočívá ve zjištění stavů, které jsou pro daný signál ve stavu excitace a kolem těchto stavů vytvořit pokrytí za účelem získání funkcí pro vstupy C-elementu. Stavy, které nejsou excitované nemusí být pokryté, jelikož logika, která by vznikla pokrytím neexcitovaných stavů by neměla vliv na C-element, ale hlavně by mohla vést ke vznikům hazardů.

3.1.9 Excitované regiony(Excistation region, ER)

Stav SG se nachází v ER, jestliže dojde k přechodu z nynějšího stavu do nového stavu pro daný signál v binárním kódu a nastane změna z 0 na 1 resp. z 1 na 0, pak se jedná buď o rostoucí signál a+ resp. o klesající signál a-, kde a je signál. Excitovaný region značíme jako ER+(a) resp. ER-(a) pro daný signál a.

3.1.10 Ustálené regiony (Quiescent region, QR)

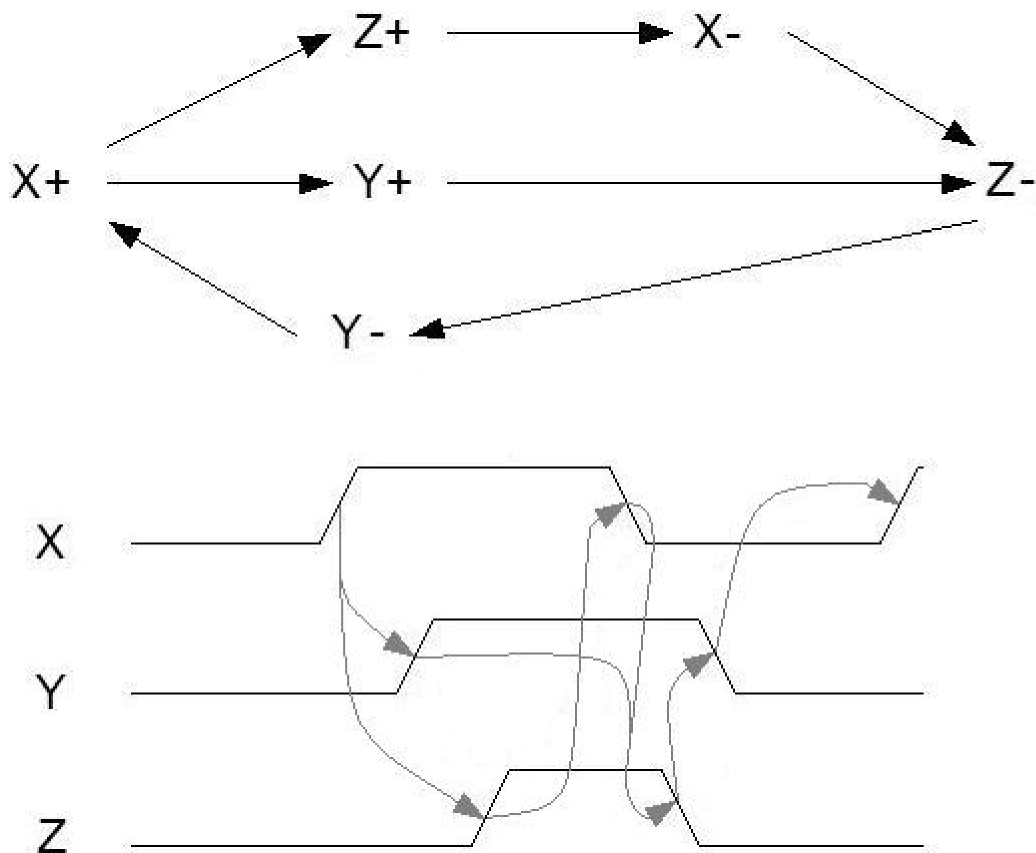
Stav SG se nachází v QR, jestliže po přechodu nynější stav na nový stav zůstane hodnota signálu v novém stavu nezměněna, pak se jedná buď o setrvání rostoucího signálu a+ nebo o setrvání klesajícího signálu a-, kde a je signál. Ustálený region se značí QR(a+) resp. QR(a-) pro daný signál a.

3.1.11 Minimalizace

Cílem minimalizace je vytvořit funkce s minimálním počtem proměnných pro vstupy C-elementu , kde invertovaný vstup budu nazývat Reset a neinvertovaný Set . Každá binární hodnota v regionu odpovídá jednomu termu a regiony udávají výstupní hodnoty stavů pro daný signál. Hodnotě 1 odpovídají stavy s binárním zakódováním v regionech ER+(a) a QR+(a). Hodnotě 0 odpovídají stavy v regionech ER-(a) a QR-(a). Don't care je pro ostatní stavy, které nejsou zahrnuty v regionech. Pro vstup Set se budu snažit najít co největší pokrytí kolem stavu v ER+(a). Pokud je v množině ER+(a) více stavů vytvořím z jejich pokrytí funkci v disjunktivní formě. Pro Set se hledají co největší pokrytí v mapě přes hodnoty 1 a don't care. Naproti tomu pro Reset se hledají co největší pokrytí 0 a don't care kolem ER-(a), popřípadě se opět vytváří funkce v disjunktivní formě. Pokud funkce Reset nebo Set obsahují více jak jeden minterm, pak je potřeba zkontrolovat, zda některý z minterů z funkce Reset není obsažen v mintermech z funkce Set. Pokud zjistím, že některý minterm z funkce Reset je obsažen v jednom z mintermu funkce Set, pak je potřeba tento problém řešit pomocí monotonního pokrytí, které omezuje pokrývání na stavy dosažitelné v obvodu. Dosažitelné stavy obvodu jsou stavy, pro které není výstup don't care. Monotonní pokrytí stačí vytvořit pro jeden z dvojice mintermů, pro které nastala shoda, obvykle se pokrývá minterm z funkce Reset.

3.1.12 Časový diagram (Timing diagram)

Reprezentuje množinu signálů v časové doméně. Časový diagram obsahuje řádky, ve kterých je průběh signálů nebo množiny signálů v čase zobrazen. Konvence v časovém diagramu: pokud je signál zakreslen vodorovnou čarou, která se vyskytuje ve vyšší hladině nákresu, představuje logickou jedničku. Pokud je signál v nižší hladině nákresu, přestavuje logickou nulu. Přechod mezi nulou a jedničkou resp. jedničkou a nulou je naznačen svislou nebo šikmou čarou. Časový diagram je reprezentován obrázkem 3.1.



Obrázek 3.1: Signálově přechodový graf a časový diagram

3.1.13 Vztah signálově přechodového grafu a časového diagramu

Přechody signálů z nula na jedna resp. z jedničky na nulu jsou naznačeny v signálově přechodovém grafu plusem resp. mínusem pro daný signál. V časovém diagramu jsou uvedeny odpovídající signály a k ním řádky na níž běží odpovídající průběh signálu. Změna hodnoty signálu v časovém diagramu nastává ve stejný okamžik, kdy dojde k odpálení přechodu v signálově přechodovém grafu. Odpálený přechod signálu ve signálově přechodovém grafu odpovídá změně u stejného signálu v časovém digramu. Vztah je naznačen v obrázku 3.1.

3.1.14 Matematické vyjádření algoritmu pro transformaci STG na Stavový Graf

Vstupem algoritmu je STG ve formě $\langle P, T, F, M_0, N, s_0, \lambda_0 \rangle$ a výstupem je SG ve formě $\langle S, \delta, \lambda_s \rangle$. Podstata algoritmu je procházení STG, tak dlouho dokud nejsou nalezeny všechny možné kombinace míst, které spolu tvoří stavy stavového grafu. Přechody mezi stavy stavového grafu odpovídají přechodům mezi místy STG, které shlukují jednotlivé stavy. Každý stav má samozřejmě svou binární hodnotu. Jak algoritmus vypadá vidíme níže 3.2, vycházím ze slajdu viz. [3].

```
1.  find_SG( P,T,F,M0,N,s0, λ0 )
2.          M = M0;
3.          s = s0;
4.          Te = {t ∈ T | M ⊆ •t};
5.          S = {M};
6.          λS(M) = s;
7.          done = false;
8.      while ( ¬ done) {
9.          t = select(Te);
10.         if (Te – {t} ≠ Ø) then push(M,s,Te - {t});
11.         if ((M - •t) ∩ t•≠ Ø) then return ("STG is not safe.");
12.         M' = (M - •t) ∪ t•;
13.         s' = s;
14.         if (λ0(t) = u+) then s'(u) = 1;
15.             else if (λ0(t) = u) then s'(u) = 0;
16.         if (M' ∉ S) then {
17.             S = S ∪ {M'};
18.             λS(M') = s';
19.             δ = δ ∪ {(M,t,M')};
20.             M = M'
21.             s = s';
22.             Te = {t ∈ T | M ⊆ •t};
23.         } else {
24.         if (λS(M') ≠ s') then return ("Inconsistent state assignment.");
25.         if (stack is not empty) then (M,s,Te) = pop();
26.             else done = true; } }
27. return( ); }
```

Obrázek 3.2: Transformační algoritmus

Rozbor jednotlivých řádků:

1. Inicializace vstupu grafem STG.
2. Kopírování množiny míst M_0 obsahující token do množiny M.
3. Kopírování počátečního zakódování stavu stavového grafu s_0 do s.
4. Vytvoření množiny Te, která bude obsahovat všechny přechody z množiny T, které splňují podmínku. Podmínka udává, že se bude jednat o přechody, které mají na místech, ze kterých se dostaneme do daného přechodu, token.
5. Do množiny S je přiřazena množina M, jejíž celek odpovídá jednomu prvku množiny S. Prvek množiny S je chápán jako jeden stav.
6. Danému prvku množiny S jsme přiřadili označení neboli binární zakódování stavu.
7. Proměnnou *done* jsme nastavili na false. Done slouží k ukončení cyklu. Pokud je *done* rovno true.
8. Kontrola podmínky cyklu.
9. Do proměnné t přiřadíme prvek z množiny Te.
10. Otestujeme zda množina Te, která je ochuzena o prvek t, obsahuje ještě nějaké prvky navíc. Pokud ano, uložíme do zásobníku současný stav množiny M, proměnou s se zakódováním aktuálního stavu stavového grafu a množinu Te bez prvku t.
11. V podmínce ověříme, zda množina míst M ochuzená o místa, které obsahují token a lze se z nich dostat do přechodu, jež je obsažen v proměnné t, dá prázdnou množinu po průniku s místy, které po přechodu tokenu přes přechod t budou obsahovat token. Pokud ne, algoritmus vrací informaci, že STG není bezpečný.
12. Do množiny M' překopírujeme sjednocení množiny M ochuzeného o místa , které obsahují token a lze se z nich dostat do přechodu, jež je obsažen v proměnné t a místy, které po přechodu tokenu přes přechod t budou obsahovat token.
13. Překopírování zakodování uložíme v s do s'.
14. Jestliže přechod roste, jinými slovy by měl jít z nuly na jedničku, jež bývá označeno +, pak v proměnné s', kde je uloženo zakódování nastavíme na danou pozici pro daný signál binární 1.
15. Jestliže přechod klesá, jinými slovy by měl jít z jedničky do nuly, jež bývá označeno -, pak v proměnné s', kde je uloženo zakódování nastavíme na danou pozici pro daný signál binární 0.
16. Jestliže množina M', je chápána jako jeden prvek a není obsažena v množině S, představující množinu stavů stavového grafu, pak dojde k vykonání následujících řádků (17. až 22.).
17. Do množiny S přidáme množinu M', která představuje jeden prvek množiny S.
18. Prvku množiny S, který odpovídá množině M', přidáme označení binární zakódování stavu.

19. Do množiny δ, která představuje množinu přechodu ve Stavovém Grafu, přidáme přechod mezi prvky množiny S, tedy M a M'.

20. Nastavení množiny M na M'.

21. Nastavení proměnné s na s'.

22. Viz. řádek 4..

23. Pokud podmínka na řádku 15. neprojde, budou se provádět následující řádky (24. až 26.).

24. Pokud se nebude shodovat zakódování v s' se zakódováním prvku množiny S odpovídajícímu prvku M', pak algoritmus skončí s nekonzistentním zakódováním.

25. Jestliže, zásobník není prázdný, pak se nastaví množina M , proměnná s a Te, podle konfigurace uložené na vrcholu zásobníku, a pak se tato konfigurace zahodí.

26. Jestliže podmínka v řádku 25. neprojde nastaví se *done* na true.

27. Konec algoritmu, návratová hodnota je stavový graf.

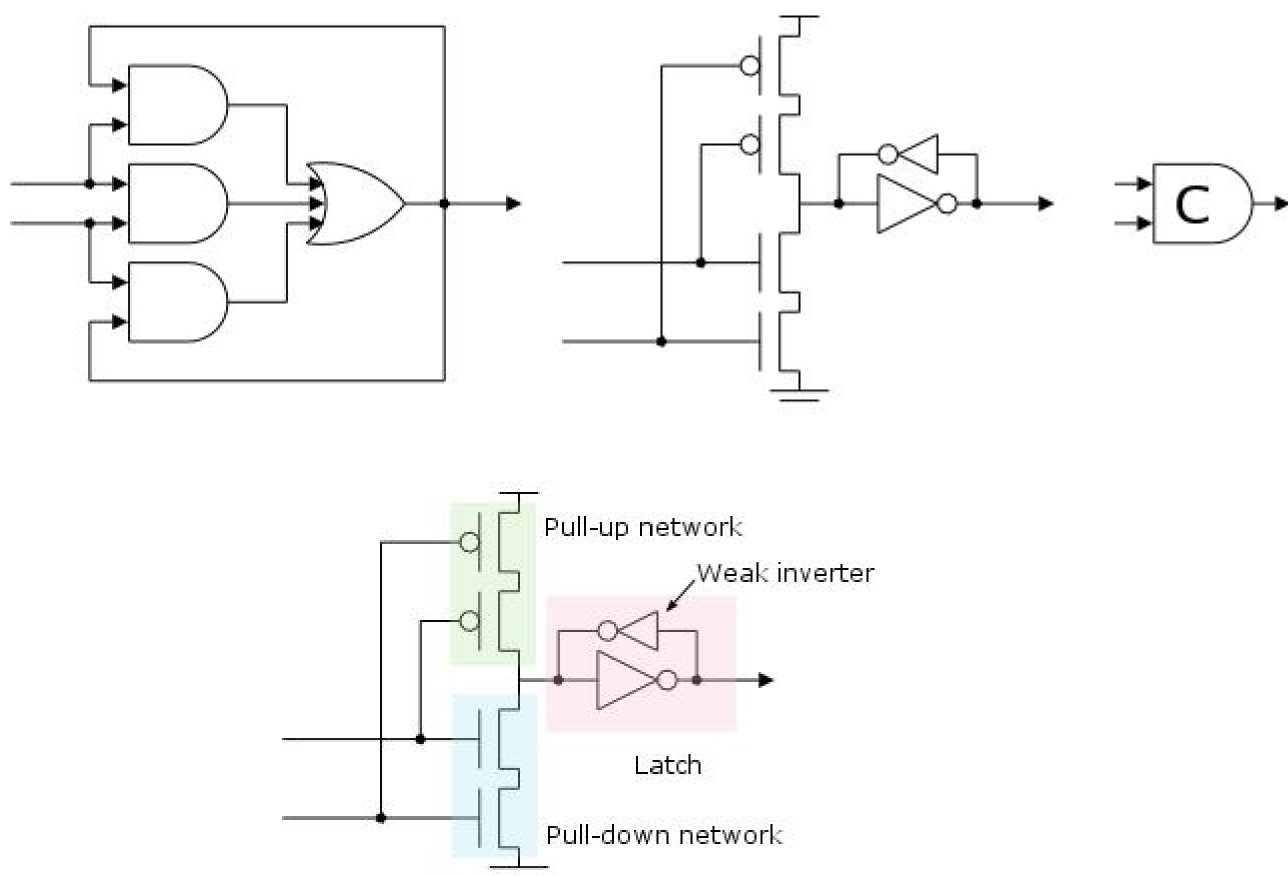
3.1.15 Muller C-element

Muller C-element, též často nazýván jako C-element nebo C-hradlo. Obvykle se vyskytuje v asynchronním návrhu, kde se používá při synchronizací událostí nebo jako prvek držící stav. C-element není atomické hradlo, ale jedná se o komplexní prvek, tvořen několika atomickými hradly. Chování C-elementu je popsáno pravdivostní tabulkou 3.1.

Set	**Reset**	**C**
0	0	0
0	1	C_{n-1}
1	0	C_{n-1}
1	1	1

Tabulka 3.1: Ukázka tabulky

Pokud jsou vstupy Set a Reset na 0, pak na výstupu je 0. Pokud jsou vstupy Set a Reset na 1, pak na výstupu je 1. Pokud jsou vstupy rozdílné, pak na výstupu je C_{n-1}, který odpovídá starému stavu. Starý stav je po nezbytnou dobu uložen v C-elementu.



Obrázek 3.3: Vnitřní reprezentace C-elementu

C-element

ukládá předchozí stav za pomocí dvou protichůdných invertorů. Jeden z invertorů je slabší, než zbytek obvodu a tak může být přetlačen spínací a rozpínací sítí (pull-up and pull-down networks). Jestliže oba vstupy jsou 0, pak se spínací síť (pull-up network) mění latch stav a výstup C-elementu je 0. Jestliže vstupy jsou 1, pak rozpínací síť (pull-down network) mění latch stav a na výstup přichází jedna. V ostatních případech není spojení mezi V_{dd} nebo zemí, slabý invertor převládá a latch stav výstupu kopíruje předchozí stav. Daný popis je znázorněn na obrázku 3.3.

Obrázek 3.4: C-elementu typ 1

Jestliže se vstupy schodují, pak se kopíruje na výstup, jinak se drží předchozí stav. Obrázek 3.4.

Obrázek 3.5: C-elementu typ 2

Jestliže se vstupy schodují, pak na výstup jde invertovaný výstup, jinak se drží předchozí stav. Obrázek 3.5.

Obrázek 3.6: C-elementu typ 3

Jestliže jsou vstupy rozdílné, pak se kopíruje neinvertovaný na výstup, jinak se drží předcházející. Jedná se o tzv. SET-RESET C-element. Tento C-element se nejčastěji využívá pro syntézu asynchronních obvodů. Obrázek 3.6.

3.1.16 C-element ve VHDL

C-element ve VHDL Implementoval jsem C-element jako komponentu ve VHDL. Komponenta bude sloužit jako komplexní hradlo v obvodu popsaném ve VHDL kódu. Na jeho vstupy pojmenované Set a Reset se namapuje vstupní logika vygenerovaná programem, výstup se přes portování spojí s patřičným signálem.

Naimportované knihovny kódu VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

V entitě jsou nadefinovány vstupy S a R , představující Reset a Set. Výstup C-elementu je pojmenován *out_c*.

```
entity cm is
Port ( S : in STD_LOGIC;
R : in STD_LOGIC;
out_c : out STD_LOGIC);
end cm;
```

Architektuře označuje, že v jeho těle bude charakterizováno chování obvodu.

```
architecture Behavioral of cm is
```

Předdefinované pomocné signály pro interní výstupy v C-elementu.

```
signal lev1a : std_logic;
signal lev1c : std_logic;

signal lev2a : std_logic;
signal lev2b : std_logic;
signal lev2c : std_logic;

signal lev3or : std_logic;
```

Begin označuje začátek těla architektury, ve které probíhá vlastní definice obvodu.

```
begin
```

Do pomocných proměnných *lev2a, lev2b, lev2c* přiřadíme výsledek z prvních třech ANDů.

```
lev2a <= lev1a and S;
lev2b <= S and not R;
lev2c <= not R and lev1c;
```

Výsledek ORů z pomocných signálů *lev2a, lev2b, lev2c* přiřadíme do lev3or, který se následně předá n výstup C-elementu *out_c*. Pomocí *lev1a a lev2c* vytvoříme zpětnou vazbu do ANDů.

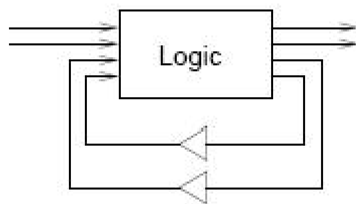
```
lev3or <= lev2a or lev2b or lev2c;
lev1a <= lev3or;
lev1c <= lev3or;
out_c <= lev3or;

end Behavioral;
```

3.2 Fundamentální mód

Většinou označovaný v literatuře jako Huffmanuv mód. U fundamentálního módu předpokládáme, že obvod se nachází ve stavu , kde jsou všechny vstupní, výstupní a interní signály

stabilní. V tomto stavu je dovoleno vnějšímu prostředí měnit jeden vstupní signál. Po změně signálu vnějším prostředím nesmí dojít ke změně vstupního signálu do té doby, dokud vstupy nejsou opět stabilní. Interní signály udávající stav proměnných jsou pro vnější prostředí neznámé a proto je třeba spočítat nejdelší zpoždění v obvodu. Po vnějším prostředí je pak požadováno, aby drželo vstupní signály stabilní přinejmenším po vypočítanou dobu zpoždění. Zpoždění na hradlech a vodičích je tedy v těchto obvodech vázané a omezení na vnější prostředí je formulováno jako absolutní časový požadavek. Obrázek 3.7 vyjadřuje představu o fundamentálním módu. Šipky na zpětné vazbě obvodu na obrázku představují vypočítané zpoždění, po jehož uplynutí bude obvodu opět dovoleno přijmout vstupní signál.



Obrázek 3.7: Huffmanuv mód

3.2.1 Syntéza podle fundamentálního módu

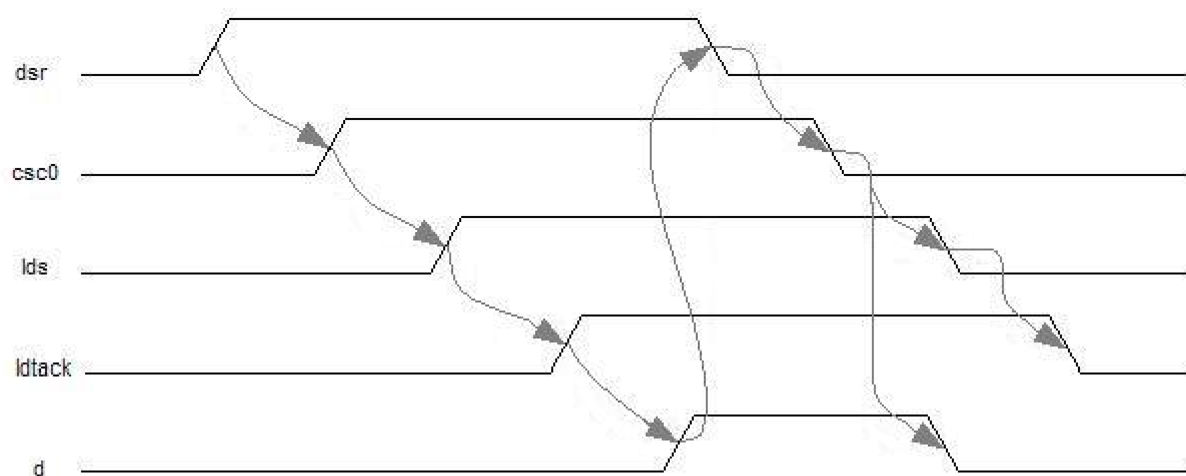
Je taky známa jako Huffmanuv návrhový styl. Vnějšímu prostředí je dovoleno změnit jeden vstup za vypočítaný čas. Jako odpověď na změnu vstupu, kombinační logika vyprodukuje nový výstup a taky zpětnou vazbu viz 3.7. Předpokládá se, že zpětnovazební signál je natolik zpožděn, že vstupy budou ve stabilizovaném stavu dříve než dorazí zpětnovazební signál. Důvodem zpoždění je kombinační logika, kterou musí signál projít než nastane nový výstup. Nakonec se přes sekvenci jednotlivých signálů přechodů v obvodu dosáhne stabilizovaného stavu a poté může vnější prostředí vyprodukovat další jeden signál. Vstup syntézy podle fundamentálního módu je specifikován asynchronním konečným automatem, který je velmi podobný vstupní specifikací u synchronních sekvenčních obvodů.

3.2.2 Nevýhody oproti Vstupně-výstupnímu módu

Může se měnit pouze jeden vstupní signál. Musíme brát v úvahu zpoždění na interních signálech. Zpoždění interních signálů ovlivňuje vnější prostředí.

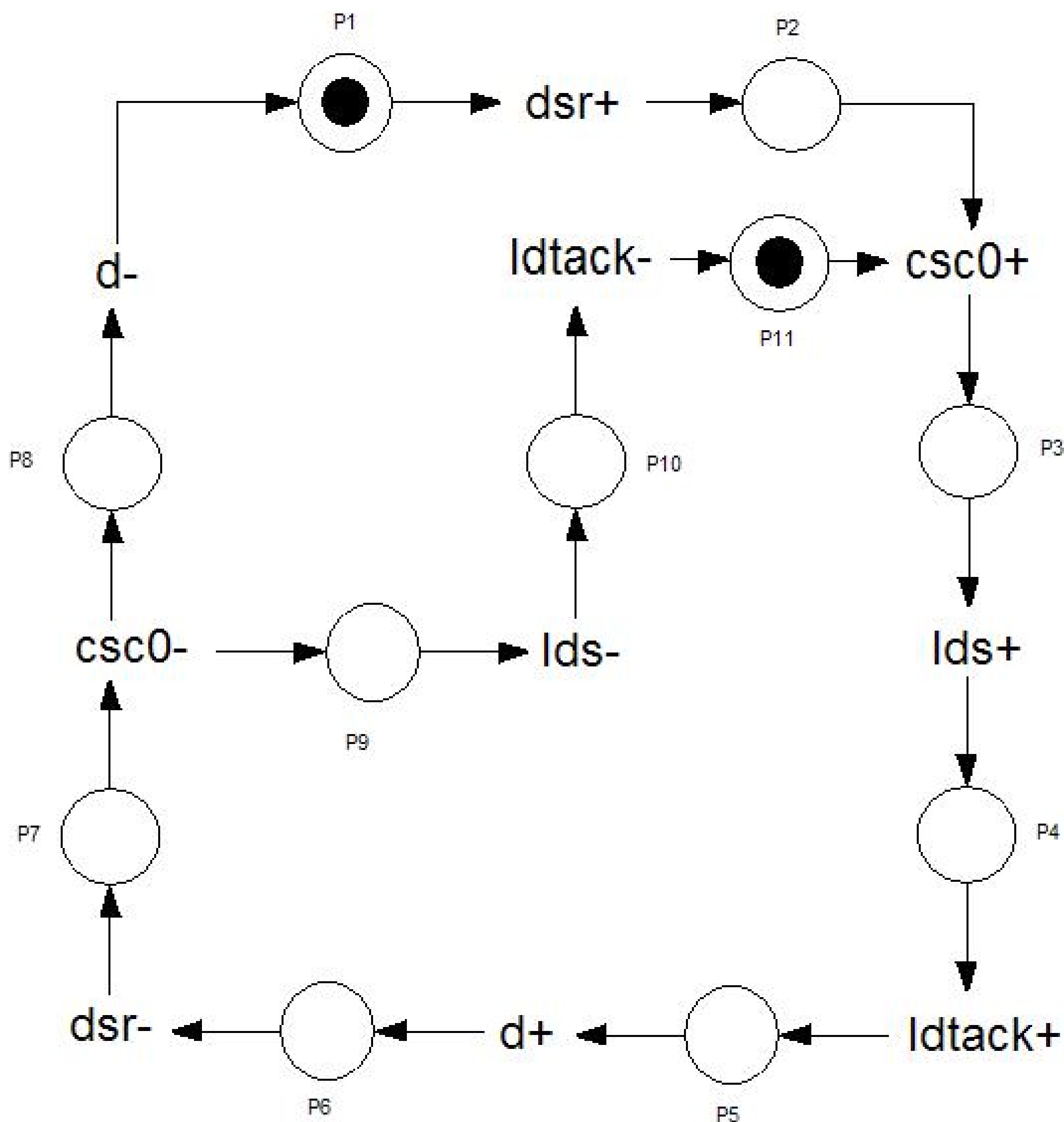
3.3 Příklad postupu syntézy

Pro příklad jsem zvolil syntézu asynchronního řadiče, na kterém chci objasnit postup syntézy v naimplementovaném programu. Na rozdíl od klasického synchronního sekvenčního řadiče zde nemusím pro uchování stavů používat klopné obvody typu D. Asynchronní řadič bude používat C-elementy. Vstupní signály jsou označeny *dsr* a *ldtack*, výstupní signály jsou *lds* a textitd. Interní signál značíme textitcsc0. Jednotlivé události, které v čase nastanou jsou popsány pomocí signálově přechodového grafu viz. 3.9. Chování řadiče je naznačeno v časovém diagramu viz. 3.8.



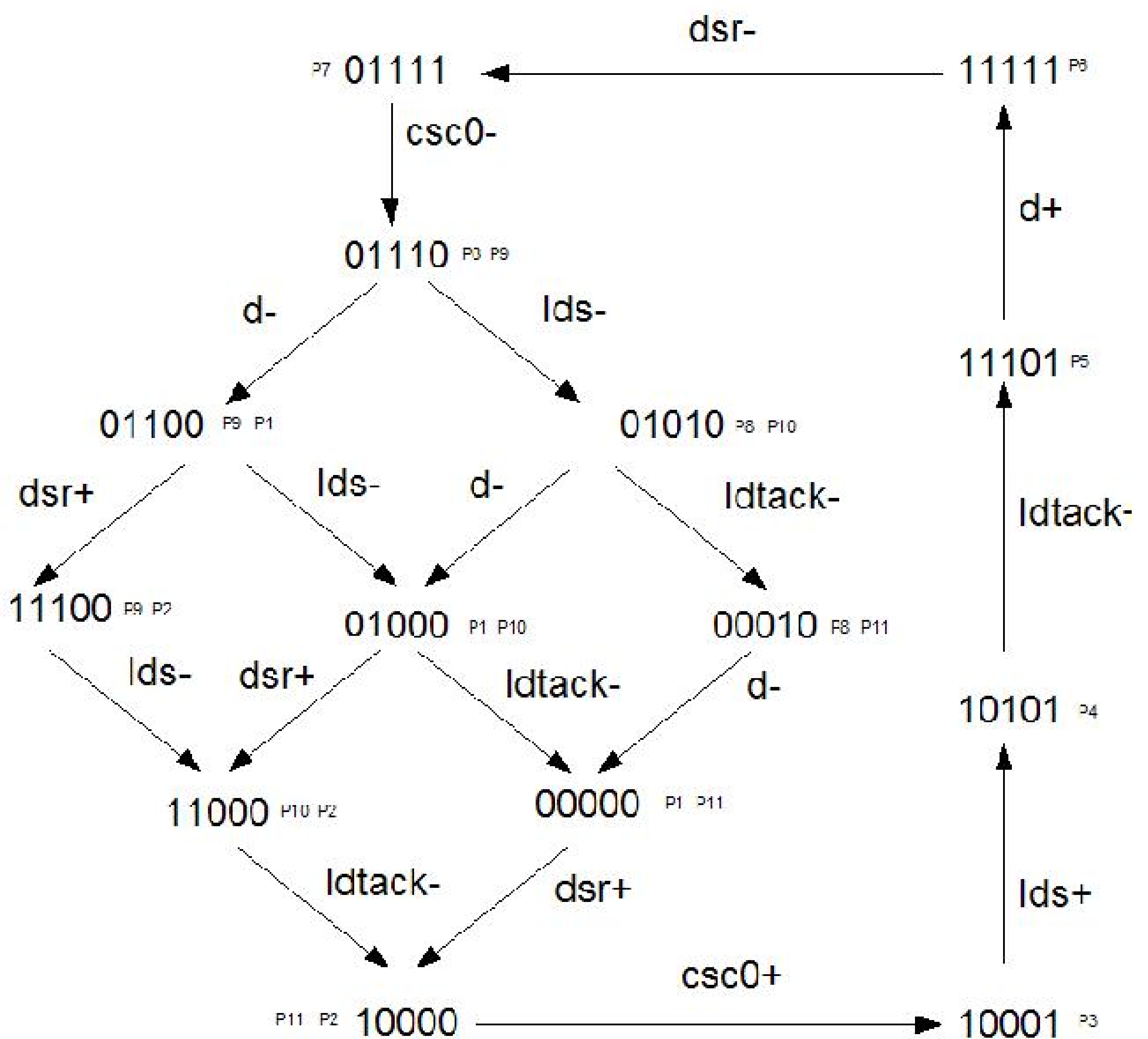
Obrázek 3.8: Časový diagram

Bílé kružnice na obrázku 3.9 znázorňují místa v signálově přechodovém grafu. Černé puntíky v těchto místech jsou tzv. tokeny. Místa obsahující token tvoří množinu míst tzv. *marking*. Nápisky s plusem nebo mínusem představují přechody signálů. Přes přechod v signálově přechodovém grafu lze přejít, pokud jsou ve všech místech, které vedou do přechodu, obsaženy tokeny (směr je naznačen šipkami). Přes přechod se prochází následovně. Vyjmou se všechny tokeny z příchozích míst a přemístí se na místa, které vycházejí z daného přechodu. Přecházení odpovídá šipkám v časovém diagramu. Místa v signálově přechodovém grafu jsou označeny P*, kde * je číslo.



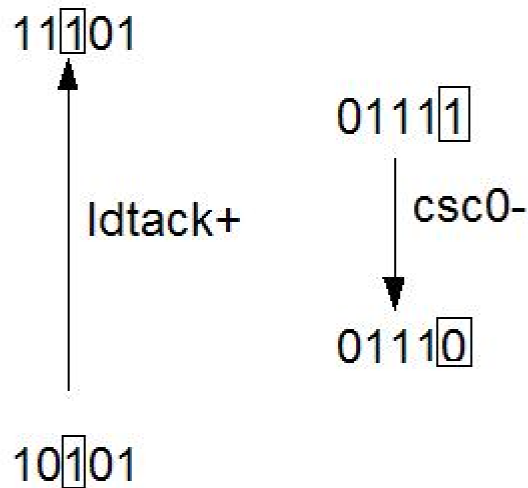
Obrázek 3.9: Signálově přechodový graf

Transformace spočívá ve seskupení míst ze signálově přechodového grafu do jednoho stavu stavového grafu. Seskupování míst probíhá následovně. Jeden stav stavového grafu odpovídá jednomu místu, ve kterém je právě token, v signálově přechodovém grafu. Pokud dojde k rozvětvení v STG, rozdělí se taky token a pak jeden stav SG odpovídá kombinaci místa, obsahující token, se všemi místy, kterými projdou ostatní tokeny. Kombinace míst trvá tak dlouho, dokud nedojde opět ke spojení tokenů v jeden. Přechody ve stavovém grafu jsou identické s přechody v STG, které jsou mezi místy, jež tvoří stavy SG. Stav má binární kódové označení, které vyjadřuje pro daný signál, zda se nachází v logické nule nebo jedničce. Při přechodu mezi stavy v SG, dochází ke změně pro daný signál v kódovém označení. Ve stavovém grafu jsou naznačeny kombinace míst ze signálového přechodového grafu pomocí P*, kde * je číslo. Výsledek transformace je vidět na obrázku 3.10.



Obrázek 3.10: Stavový graf

Po přetransformování STG je třeba vytvořit regiony pro každý výstupní a interní signál. Každý signál má čtyři regiony, do kterých se vkládají kódové označení stavů podle toho, zda signál ve stavu v SG je na nule resp. na jedničce a v následujícím stavu zůstane na nule resp. na jedničce, pak dojde k zařazení stavu do ustáleného regionu (QR), konkrétně do QR+(a) resp. QR-(a), kde *a* je daný signál. Pokud však při přechodu do budoucího stavu dojde ke změně pro daný signál z nuly na jedna resp.z jedné na nulu, pak se konkrétní stav zařadí do excitovaného regionu, konkrétně do ER+(a) resp. ER-(a). Pokud se stav vyskytuje zároveň v ER a QR, pak je třeba tento stav z QR oddělat.



Obrázek 3.11: Regiony naznačení

Popis obrázku 3.11. Pro signál vyznačený obdélníkem by stav s označením 10101 v první případě s přechodem ldtack+ patřil do region QR+. V druhém případě pro vyznačený signál by došlo k zařazení stavu s označením 01111 do ER-.

Příklad konkrétního přiřazení do jednotlivých regionů. Binární ohodnocení stavu je převedeno do dekadických hodnot v závorkách.

ER+(csc0) = 16
ER-(csc0) = 15
QR+(csc0) = 17, 21, 29, 31,
QR-(csc0) = 14, 12, 10, 28, 8, 2, 24, 0

ER+(lds) = 17
ER-(lds) = 14, 12, 28
QR+(lds) = 21, 29, 31, 15
QR-(lds) = 10, 8, 24, 2, 0, 16,

ER+(d) = 29
ER-(d) = 14, 10, 2
QR+(d) = 31, 15
QR-(d) = 12, 8, 0, 28, 24, 16, 17, 21

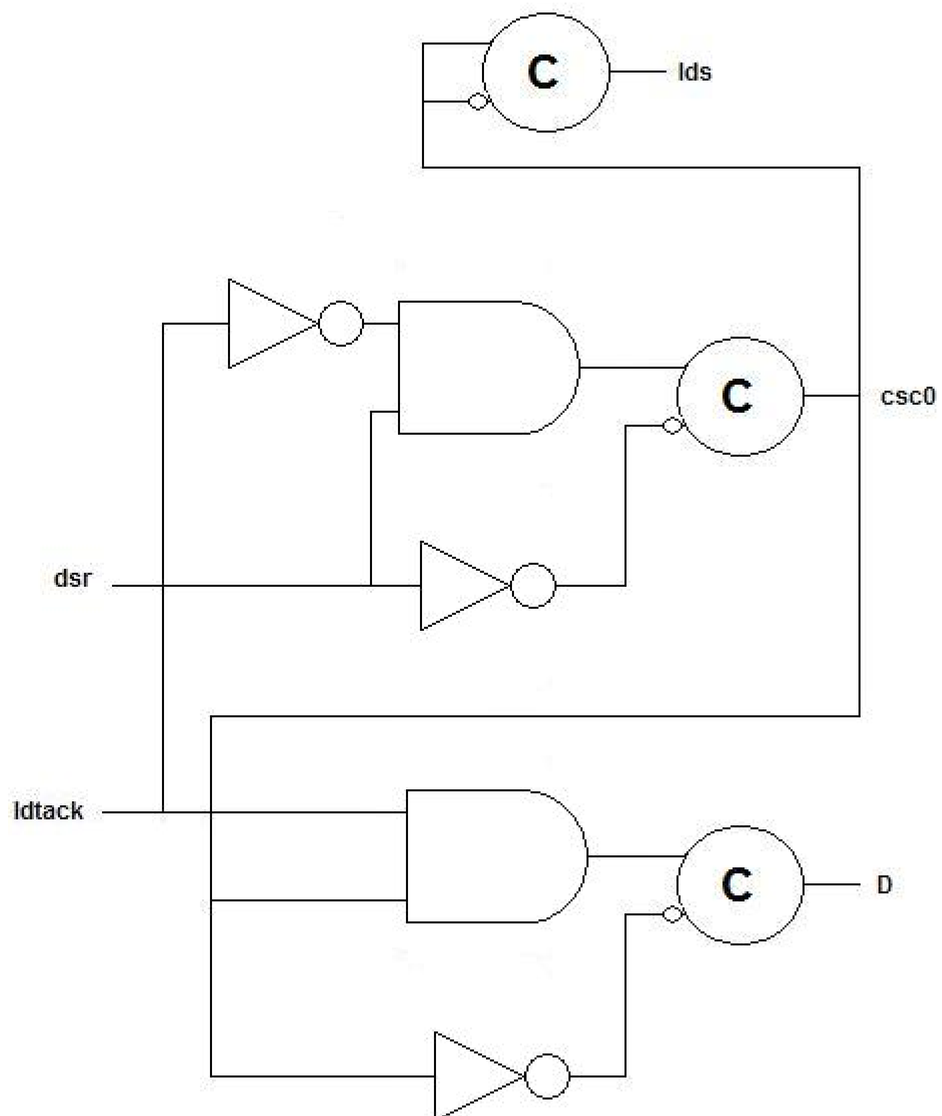
Po vytvoření regionu následuje část minimalizace, za účelem získání rovnic pro vstupy C-elementu

tedy pro vstup nazývaný Set resp. Reset. Rovnice se připravují pro každý výstupní a interní signál. Výstup C-elementu odpovídá danému výstupu nebo internímu signálu. Regiony udávají výstupní hodnotu. Binární ohodnocení stavu odpovídá úplnému mintermu. Minterm má hodnotu 1, jestliže patří do ER+(a) sjednoceno s QR+(a). Minterm má hodnotu 0, jestliže patří do ER-(a) sjednoceno s QR-(a). Pro názornost minimalizace použiji Karnaughovu mapu pro vybraný signál *csc0* viz 3.12. Pro ostatní zbývající signály dodám jen výsledky, postup by byl stejný. Mintermy, které se nevyskytují v regionech považuji za don't care. Hledám co největší pokrytí kolem mimtermu ze ER+(a) resp. kolem ER-(a). Pokrytí kolem ER+(a) bude odpovídat vstupní funkci pro Set, ER-(a) bude odpovídat Resetu. V našem konkrétním případě se snažím získat co největší pokrytí kolem okénka označeného 16 pro Set a pro Reset kolem okénka 15. Pokrytí je naznačeno na obrázku 3.12.

	$\overline{DSR}$				DSR				
	$\overline{CSC0}$	CSC0	CSC0	$\overline{CSC0}$	$\overline{CSC0}$	CSC0	CSC0	$\overline{CSC0}$	
$\overline{D}$	0 (0)	X (1)	X (5)	X (4)	X (20)	1 (21)	1 (17)	1 (16)	$\overline{LDTACK}$
D	0 (2)	X (3)	X (7)	X (6)	X (22)	X (23)	X (19)	X (18)	$\overline{LDTACK}$
D	0 (10)	X (11)	0 (15)	0 (14)	X (30)	1 (31)	X (27)	X (26)	LDTACK
$\overline{D}$	0 (8)	X (9)	X (13)	0 (12)	0 (28)	1 (29)	X (25)	0 (24)	LDTACK
	$\overline{LDS}$		LDS		LDS		$\overline{LDS}$		

Obrázek 3.12: Karnaughova mapa

Výstupní rovnice csc0 pro Set je f = dsr and not ldtack, pro Reset f = not dsr. Pro signál d jsou pro Set f = ldtack and csc0, pro Reset f = not csc0. Pro signál lds jsou pro Set f = csc0, pro Reset f = not csc0. Za povšimnutí stojí i fakt, že funkce obsahují pouze jeden minterm, proto není potřeba srovnání mintermů a provádění monotónního pokrytí.



Obrázek 3.13: Výsledná logika

4 Analýza a návrh řešení

4.1 Shrnutí před analýzou a návrhem

Shrnutí podstatných věcí před analýzou a před vytvořením programu, který bude generovat VHDL kód na základě vstupní specifikace STG. Syntéza bude založena na specifikaci signálově přechodovém grafu, jež popisuje Speed Independent obvody. Podstatou syntézy je přetransformovat signálově přechodový graf do stavového grafu a dodržet podmínky určující správnost grafů. Po transformaci je třeba stanovit regiony pro dané signály a na základě těchto regionů budou přiřazeny výstupní hodnoty. Z hodnot v regionech vytvořit funkce, jež zohledňují užití C-elementu. Na základě funkcí vytvoříme logiku zahrnující C-elementy. Některé C-elementy mohou být vypuštěny. Daná kombinační logika zahrnující C-element bude popsána pomocí VHDL kódu.

4.2 Implementační jazyk

Jako implementační jazyk jsem zvolil C++, který nám poskytuje možnost využít tříd a předpřipravených šablon knihovny STL. Navíc je daleko vhodnější pro programy, které nevyužívají grafické rozhraní, tedy pro konzolové aplikace. Můj program bude konzolová aplikace, jejímž vstupem bude textový soubor a výstupem soubor obsahující VHDL kód, který popisuje logiku. Výsledný program bude exe soubor.

4.3 Vstup programu

Vstupní specifikací pro syntézu je signálově přechodový graf. Graf lze obecně popsat vhodnou písemnou formou, která bude vyjadřovat patřičné informace a je srozumitelná pro uživatele i pro počítač.

4.4 Reprezentace grafu

Ze specifikace pro signálově přechodový graf můžu vyjádřit základní požadavky, které by popisná forma grafu měla mít. Jedná se konkrétně o dvě množiny. Množinu míst a množinu přechodů. Mezi těmito množinami existuje vztah, že pokud se chci dostat z jednoho z prvku množiny přechodů na jiný prvek téže množiny, musím nejprve přejít přes prvek množiny míst, který je v dané relaci s již zmíněnými prvky množiny přechodů. Stejné pravidlo platí i mezi prvky množiny míst a prvkem množiny přechodů. Z mého pohledu je lépe se držet představy dva přechody a jedno místo. Z toho důvodu se zaměřuji na signály přechodů, zatímco místa jsou pro mě implicitně dodané hodnoty potřebné pro transformaci. Výsledkem tedy je, že pro popsání vztahu přechodů a míst mi stačí znát vztahy mezi přechody, přičemž se předpokládá s implicitním dodáním míst.

4.5 Návrh reprezentace grafu

Vztah mezi jednotlivými přechody lze vyjádřit jako vztah jednoho přechodu k druhému přechodu nebo k přechodům, do kterých se lze dostat přes místo, do nějž výchozí přechod ústí. Tento vztah lze vyjádřit posloupností výchozího přechodu a dosažitelných přechodů z míst do

nichž vyústil výchozí přechod. Abych mohl říct, že tyto posloupnosti jsou opravdu vztahy grafu, je třeba nějakým identifikátorem označit začátek a konec, ve kterých je jejich platnost.

4.6 Počáteční zakódování

Další náležitost, která má charakterizovat signálově přechodový graf je počáteční stavové zakódování. Zakódování je důležité pro přiřazení binárního ohodnocení jednotlivých stavů stavového grafu při transformaci.

4.7 Návrh reprezentace počátečního zakódování

V textovém popisu by se měl vyskytnout identifikátor, podle něho budu moci určit, že se bude jednat o posloupnost znaků, vyjadřující binární zakódování počátečního stavu.

4.8 Vstupy, výstupy a interní signály

Pro VHDL kód, jež by měl být výstupem a pro generování přechodových rovnic je nezbytné, aby byly v textu rozlišeny vstupní, výstupní a interní signály. Měl bych poznamenat, že signál se může vyskytovat jen pod jednou identifikací, jinými slovy není možné, aby signál zároveň patřil mezi vstupy a výstupy nebo mezi jiné možné kombinace.

4.9 Návrh reprezentace vstupů, výstupů a interních signálů

Opět by bylo vhodné do textu zařadit identifikátor, jež by oznamoval počátek posloupnosti pojmenování a identifikoval by sounáležitost signálu s danou sortou. A to konkrétně zařazení do sorty vstupních, výstupní nebo interních signálů.

4.10 Marking

Aby graf měl všechny náležitosti a počítač měl všechny údaje potřebné pro transformaci, je potřeba dodat tokeny na správná místa. Místa s tokeny jsou uloženy v množině *marking*. Místa jsou v programu implicitně generována. Označení místa tokenem se tedy musí provádět za pomocí dvojice přechodů, mezi nimiž se místo nachází.

4.11 Návrh přidělování tokenů

Začátek řádku bude identifikován identifikátorem za nímž bude následovat posloupnost znaků udávající místa obsahující token . V posloupnosti se budou vyskytovat dvojice přechodů, které budou označovat místa s tokenem. První přechod dvojice bude vstupní přechod místa, druhý přechod dvojice bude výstupní přechod daného místa.

4.12 Modul zpracování textu

Je třeba vytvořit třídu nebo skupinu tříd, které se budou starat o zpracovaní vstupního textu. Ve vstupním textu se obvykle vyskytuje identifikátor, jež označuje začátek posloupnosti

označení, obvykle oddělených mezerou. Vycházíme-li z předpokladu, že identifikátor je jistý druh příkazu, pak můžeme identifikátor označit za příkaz a posloupnost znaků označení za argumenty příkazu. Existuje jeden případ, kdy argumenty nejsou uvedeny příkazem. Jedná se o samotné vztahy přechodů, kterými se takto popisuje graf. Abych mohl samotné argumenty použít, je potřeba jej povolit samostatným příkazem.

4.13 Návrh modulu pro zpracování textu

Bylo by vhodné, aby třídy obsahovaly metody pro pársování textu, jež by byly schopny naparsovat příkaz i z jeho argumenty do nějaké STL šablony, nejlépe fronty. První položka by signalizovala příkaz a ostatní položky argumenty. Jednotlivé dílčí fronty, obsahující příkaz a argumenty, by na základě příkazu, byly schopny předat argumenty ke zpracování a vytvořily by datovou strukturu reprezentující signálově přechodový graf.

4.14 Modul pro zpracování příkazů

Mělo by se jednat o třídu, která bude zpracovávat jednotlivé příkazy přicházející z modulu pro zpracování textu. Metody, které by odpovídaly jednotlivým příkazům, by zpracovávaly jednotlivé argumenty podle požadavku příkazu. Cílem tohoto modulu by mělo být vytvoření datového obrazu signálově přechodového grafu. Metody by se měly starat o ukládání informací, o jaké signály jde, jestli vstupní, výstupní nebo interní, do dané datové struktury. Dále by měly ukládat do struktury data o přechodech a místech a vztahy mezi nimi. Měly by označovat místa tokenama.

4.15 Návrh modulu pro zpracování příkazů

Struktura, která by měla reprezentovat signálově přechodový graf v počítači, by měla obsahovat dvě pole STL vektory. V jednom by se měly ukládat ukazatele na struktury obsahující informace o místech a v druhém poli ukazatele nesoucí informace o přechodech. Vzájemné vztahy přechodů a míst by měly být vyjádřeny přes indexy polí, ukazující z pole přechodů do pole míst a naopak. Nosná struktura by měla dále nést informaci o zakódování počátečního stavu a název signálu. Metody, které budou v daném modulu, se budou starat o naplňování jednotlivých proměnných a jejich inicializaci. Konkrétně hovořím o vytváření struktur místa a přechodů a jejich začlenění do nosné konstrukce a vyjádření jejich vztahu přes pole nosné konstrukce.

4.16 Modul pro transformaci

Vstupem modulu je struktura popisující signálově přechodový graf. Cílem transformace by mělo být naplnění struktury daty, které reprezentují stavový graf. Modul by měl obsahovat algoritmus transformace signálově přechodového grafu na stavový graf. Algoritmus je založen na matematickém vyjádření, o kterém byla již řeč. V matematickém modelu, není implicitně naznačeno posouvání tokenu, proto bude třeba se o tento problém postarat.

4.17 Návrh modulu pro transformaci

Je třeba stanovit reprezentaci množin vyskytující se v algoritmu pro transformaci. Pro většinu množin bude vhodné použít STL šablonu a to konkrétně frontu. Pro zásobník opět

použijeme STL šablonu, ale tentokrát by se mělo jednat o šablonu pro zásobník tzv. Stack. Do zásobníku by se měla umísťovat struktura. Tato struktura by měla nést data o binárním zakódování stavu, množině míst obsahující tokeny a množinu vybraných přechodů. Struktury v zásobníku nesou informace, které jsou důležité pro větvení grafu a dávání jednotlivých míst dohromady. Bude potřeba metoda pro posouvání tokenu ve struktuře popisující signálově přechodový graf. Dále bude potřeba vybírat aktivní přechod, vytvořit množinu obsahující přechody, které na vstupních místech mají tokeny. Dále inicializovat množinu míst s tokeny. Výsledkem transformace má být stavový graf, proto je potřeba vytvořit struktury, ve kterých se budou nést příslušné data o grafu. Stavový graf by se měl skládat ze dvou základních struktur. Jedné nosné struktury, která by nesla struktury reprezentující jednotlivé stavy. Druhé struktury nesoucí informace o jednom stavu stavového grafu. Stav by měl minimálně obsahovat následující informace: zakódování stavu, vztah k ostatním stavům a informace o přechodech. Důležitou metodou třídy by měla být metoda pro porovnávání stavů stavového grafu se stavem, který má byt přidán. Pokud se porovnávaný stav ve stavech vyskytuje, měla by metoda tuto skutečnost oznámit.

4.18 Modul pro vytvoření regionů

Cílem modulu regionů je vytvořit čtyři základní regiony pro každý vstupní a interní signál. Do regionů budou vkládány binární zakódovaní stavů stavového grafu. Binární zakódování stavu představuje jeden term. Jedna z proměnných v termu představuje hodnotu vybraného signálu. Pokud se se hodnota signálu v následujícím stavu změní, pak binární zakódování výchozího stavu vložíme do ER, jinak binární zakódování vložíme do QR-.

4.19 Návrh modulu pro vytváření regionů

Je třeba projít všechny stavy stavového grafu pro daný signál. Informaci o stavech nese struktura reprezentující stavový graf. Při procházení grafu bude docházet k přidávání binárních zakódování do příslušných regionů . Dále je potřeba naimplementovat metodu starající se o jednoznačnost přiřazení stavů do regionu, jinými slovy, aby ve všech čtyřech regionech bylo zakódování stavu pouze jednou. Struktura, která bude nést informace o regionech, by se měla dělit do dvou částí. Jedna struktura by měla být nosná a druhá struktura by měla nést informace pro daný signál. Struktura pro daný signál by měla obsahovat minimálně 4 pole odpovídající jednotlivým regionům, v nichž by mělo být umístěno zakódovaní jednotlivých stavů. Nosná struktura by měla nést pole struktur pro jednotlivé signály. Pokud hovořím o jednotlivých signálech, pak se jedná o signály výstupní a interní. Výsledkem tohoto modulu bude struktura nesoucí informace o signálech a regionech s nimiž jsou spojené.

4.20 Modul pro minimalizaci

Cílem modulu je nalézt nejjednodušší vyjádření logické funkce. Aby byla funkce co nejvíce minimální musí mít minimální počet proměnných v každém termu. Funkce, které chci získat by měly být v disjunktivní formě. Vstupem modulu jsou regiony pro dané signály. Pro daný signál se snažím najít funkci, která pokrývá jedničky, tak i funkci, která pokrývá nuly. Logika získaná z těchto funkcí bude předřazena před vstupy C-elementu. Konkrétně pro vstup Set C-elementu bude použita logika získaná z funkce pokrývající jedničkové stavy a don't care a pro vstup Reset C-element bude použita logika získaná z funkce pokrývající nuly a don't care. Pokud po minimalizaci budou funkce obsahovat jednu proměnou v termu a proměnná se pro

funkci Reset bude lišit negací od proměnné ve funkci Set, pak mohu C-element vynechat . Term se vždy minimalizuje kolem stavů z excitovaných regionů pro daný signál. Funkci pro Set resp. Reset, pak tvoří součet termů získaných po minimalizaci z ER+(a) resp ER-(a). Pokud funkce Set a Reset obsahují více jak jeden minterm, je potřeba prověřit, zda mintermy z jedné funkce nejsou obsaženy v mintermech druhé vstupní funkce. Jestliže dojde ke shodě, musím tento problém odstranit, jinak by mohlo dojít k neočekávanému nastavení nového stavu na C-elementu. Odstranění provedu pomocí monotonního pokrytí. Výsledkem modulu by měly být dvě množiny, ve kterých jsou uchovány mintermy pro Reset a pro Set pro daný signál.

4.21 Důvody použití vlastního minimalizačního algoritmu

Pro minimalizaci budu používat vlastní algoritmus, jelikož program je implementovaný v jazyce C++ a nebylo by vhodné, abych musel připravovat pro každý term v excitovaném regionu výstupní soubor jako vstup pro externí minimalizátor. Navíc bych musel získávat data zase zpět do programu, přičemž bych musel rozdělit celistvost implementace programu na dvě části. Nezbytnou součástí programu by musel být externí minimalizátor, čímž by se porušila kompaktnost programu a přenositelnost. Posledním důvodem, proč se nevyplatí používat externí minimalizátor je, že pro zhruba 20 vstupních proměnných jsem schopen postupnou expanzí dosáhnout přibližně stejných výsledků jako u externích minimalizátorů. Vycházím-li z grafů, které jsou na ATACS, pak průměrný počet vstupních proměnných se pohybuje okolo 10.

4.22 Algoritmus minimalizace

Algoritmus je založen na porovnání termů z ER+(a) resp. ER-(a) s termy v ER-(a) a QR-(a) resp. ER+(a) a QR+(a). Nejprve vyberu term z ER+, který budu porovnávat s termy z ER- a QR-. Vybraný term porovnávám s porovnávanými termy následovně. Postupně vynechám jednu proměnnou ve vybraném termu a zbývající proměnné porovnám s odpovídajícími proměnnými v porovnávaných termech. Vynechávání se provádí od levé proměnné termu. Pokud pro vybraný term s vynechanou proměnnou narazím na shodu všech zbývajících proměnných v některém termu, pak vynechanou proměnou nemohu vynechat. V opačném případě mohu proměnou vypustit pro další porovnání. Nevynechané proměnné na konci algoritmu tvoří minterm. Porovnávací proces provádím pro všechny termy z ER+. Stejný postup aplikuji pro prvky z ER-, které porovnávám s prvky z ER+ a QR+.

4.23 Návrh pro modul minimalizace

Je třeba vytvořit metody, které budou zpracovávat strukturu nesoucí data o signálech a jejích regionech, a taky metody, které budou realizovat algoritmus minimalizace termů. Je potřeba realizovat procházení celé struktury nesoucí informace o regionech signálu. Pro každý signál, vzít excitované regiony a jejich termy nechat projít algoritmem pro minimalizaci termů. Výsledek pak přidat do struktury nesoucí informace o Reset a Set množině pro dané výstupní nebo interní signály. Proto potřebuji zavést dvě struktury, kde jedna bude nést informace o jednom signálu a druhá struktura bude nosná. Struktura, která bude nést data pro jeden signál musí obsahovat minimálně množiny Reset a Set, kde se budou ukládat minimalizované termy. Všechny termy v dané množině, budou tvořit funkci disjunktivní formy. Další metoda, která by se měla v modulu vyskytovat, je metoda pro porovnání mintermů a pro vytvoření monotonního pokrytí.

4.24 Návrh algoritmu

Vytvořím metodu, která bude provádět výše zmíněný algoritmus pro minimalizaci. Metoda by měla umět signalizovat vynechání proměnných v termu. Pro nejpravější proměnnou je potřeba projít zbytek termu v opačném směru expanze.

4.25 Výstup programu

Výstupem programu by měl být soubor obsahující VHDL kód. Kód, který mám vygenerovat by měl zohledňovat vstupy a výstupy, které by se měly objevit na entitě VHDL, jež popisuje vstupy. Interní signály by měly být předefinované ještě před popisem celého těla architektury VHDL. Tělo architektury ve VHDL kódu by mělo obsahovat vztahy mezi vstupy, výstupy a interními signály. Tyto vztahy mohou obsahovat popis přiřazení signálu, popis logiky mezi, kterou mají signály projít a naportování signálu na komponentu popisující chování C-elementu.

4.26 Návrh výstupu programu

Pro entitu VHDL popisující signály, které patří mezi vstupní či výstupní, budu přidávat před dané signály *in* nebo *out* jako součást názvu. Všechny signály vstupní, výstupní nebo interní nadefinuji před samotné tělo architektury uvozeným begin a navíc přidám ještě pomocné signály, které budou vstupní hodnotou pro Set a Reset C-elementu pro daný signál. V samotném těle architektury ve VHDL kódu budou popsány vztahy jednotlivých hradel , instance C-elementu, portace.

4.27 Modul pro generování VHDL

Vstupem modulu bude struktura nesoucí informace o signále a s nimi spjaté množiny Set a Reset pro daný signál. Modul by měl nejprve vygenerovat vstupní a výstupní signály pro entitu. Měl by vygenerovat pomocné signály a následně přejít ke generování popisu logiky a vztahu signálů, mělo by dojít k naportování na C-element. Výsledkem bude výstupní soubor obsahující VHDL kód, jež je popsán výše.

4.28 Návrh modulu pro generování VHDL

Modul by měl obsahovat metody tříd, které jsou schopny vygenerovat kód pro popis vstupu a výstupu. Metody by měly generovat pomocné a interní signály. Musí obsahovat metody pro procházení struktur nesoucí informace o signálech a regionech a generovat kód pro případné naportování na C-element.C-elementy budou vytvářeny jako instance ve VHDL kódu, instance bude vycházet z předlohy, jež je uložena v souboru cm.vhd. Musí obsahovat metodu pro zápis kódu do souboru.

5 Implementace

5.1 Všeobecné informace k implementaci

Modulární systém, který byl naznačen v analýze a návrhu, budu řešit třídami implementovanými v jazyce C++. O provádění jednotlivých úkonů syntézy se budou starat metody daných tříd. Veškeré grafové reprezentace a regiony jsou řešeny přes datové struktury. Z knihovny STL budu používat šablonu vector pro uchovávání množin grafů ve strukturách. Metody tříd, které budou pracovat s množinami pro jejich uchovaní, využívají šablonu fronta. Veškeré metody tříd jsou nainlajnované v hlavičkách tříd, proto je jen jeden soubor *.cpp*, ve které je main funkce.

5.2 Vstup programu

Vstupem programu je textový soubor. Soubor obsahuje textově popsaný signálově přechodový graf. Vstupní popis grafu je popsán v příloze.

5.3 Třída FromFile

Slouží k načtení souboru, čtení jednotlivých řádků, dekódování příkazů a odesílání argumentů příkazů ke zpracování. Příkaz a argumenty jsou po pársováním vkládány do fronty. Prvky fronty se postupně odebírají a interpretují.

5.4 Třída DoCommand

Třída obsahuje metody, které převedou informace naparsovaného vstupního textu do datových struktur, reprezentujících STG v programu. Vstupem této třídy je datová struktura fronta, ve které jsou informace v textových řetězcích. Výstupem třídy je struktura reprezentující STG.

5.5 Třída Prikaz

Účelem této třídy je, podle zadaného příkazu, rozeznat příkaz a přiřadit mu číslo, které při výběru bude volat metodu při jeho zpracování.

5.6 STG v programu

Místa v STG jsou tvořena pomocí struktury *Place*. Ve struktuře *Place* se uchovávají informace, zda je místo označené tokenem a na jaký přechod ukazuje. Schematické naznačení 5.1

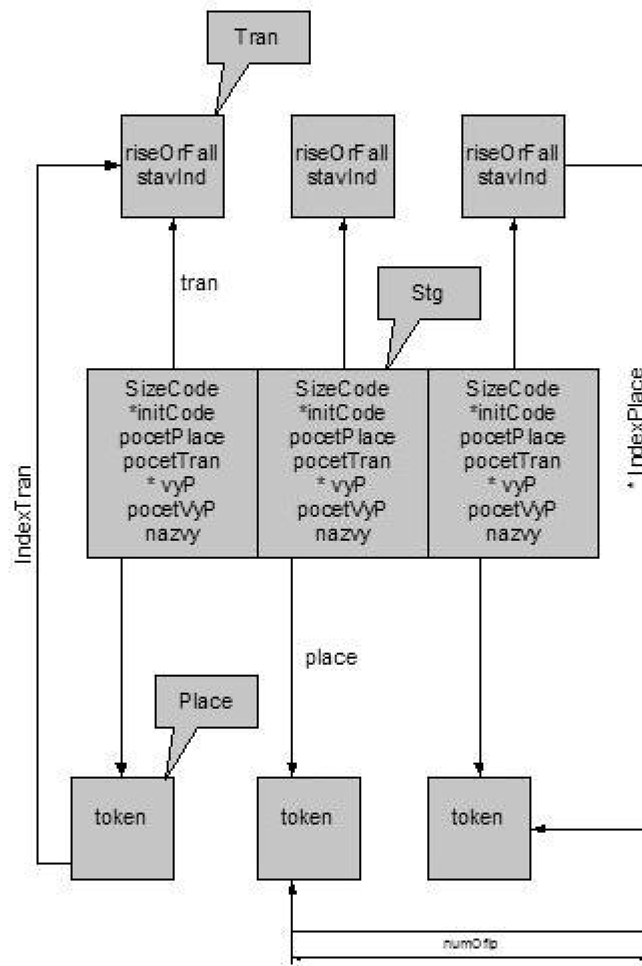
```
struct Place{
char token;
int indexTran;
};
```

Data o přechodu jsou uchovávány ve struktuře *Tran*, jež nese informace o ukazatelích na místa *indexPlace* a počet ukazatelů na místa *numOfip*. Jestli je přechod rostoucí nebo klesající udává proměnná *riseOrFall*, do které se nastaví + nebo – podle přechodu.

```
struct Tran{
int stavInd;
char riseOrFall;
int* indexPlace;
int numOfip;
};
```

Celkový graf STG je uložen ve struktuře pojmenované *Stg*.

```
struct Stg {
      int sizeCode;              //velikost binárního kódu
      char *initCode ;           // počáteční kod
      vector<Place*>  place;     //vektor míst
      vector<Tran*>   tran;      //vektor přechodu
      int pocetPlace;            //počet míst ve vektoru
      int pocetTran;             //počet přechodu ve vektoru
      int* vyP;                  // index výstupu
      int pocetVyP;              // počet výstupu
      vector<string>  nazvy; //názvy signálu
      vector<int>  nazvyp; //identifikator signálu
      string nameFile; //jméno souboru
};
```



Obrázek 5.1: Struktura STG

5.7 Třída MakeSG

Základním kamenem celého programu je tato třída, protože je zde realizována transformace signálově přechodového grafu na stavový graf. Transformační algoritmus je založen na matematickém algoritmu pro transformaci STG na SG popsanou výše. Parametry, které tato třída přijímá jsou struktury *Stg*a textitSG. Ve struktuře textitStg je uložen datový obraz STG.
Metody třídy MakeSG:

find_SG
Základní metoda pro hledání stavového grafu ze signálově přechodového grafu. Postupuje se v ní podle popsaného algoritmu.

Metody, které využívá metoda find_SG pro hledání stavového grafu:

selectTe
Metoda slouží k vybrání přechodu, přes který se bude následně přecházet. Přechod musí splňovat požadavek ,že na jeho vstupních místech se nachází token nebo tokeny.
Asymptotická složitost metody: $O(nx)$
n - počet prvků v množině Te,
x - počet míst v STG

doM
Metoda slouží k naplnění množiny *M*. *M* obsahuje prvky ze structury *Place*, ve kterých se nacházejí tokeny. Množina *M* je tvořena C++ šablonou fronta.
Asymptotická složitost metody: $O(n)$
n - je počet míst v STG

makeTe
Metoda naplní množinu *Te* přechody, které jsou reprezentovány strukturou *Tran* v *Stg*. Metoda vybírá pouze ty přechody, které mají na místech, ze kterých se dostaneme do daného přechodu, token.
Asymptotická složitost metody: $\Theta(n)$
n - je počet prvků v množině M

doS
Slouží k přidání množiny M do množiny S. Celá množina M představuje v množině S jeden prvek. Prvek množiny S odpovídá stavu ve stavovém grafu.
Asymptotická složitost metody: $\Theta(n)$
n - je počet míst k sjednocení

changToken
Metoda provádí přemístění tokenů ve struktuře *Stg*. Přemísťují se pouze tokeny, které jsou na vstupních místech vybraného přechodu *t*. Přes vybraný přechod se dá vždy přejít.
Asymptotická složitost metody: $\Theta(n + x)$
n - počet míst v STG
x - počet míst vycházejících z vybraného přechodu

comperM
Vytvoří průnik mezi množinou míst vycházejí z vybraného přechodu t a množinou *M*, která je ochuzena o místa vstupující do přechodu *t*. Pokud výsledek průniku není prázdná množina, pak metoda vrátí true.
Asymptotická složitost metody: $\Theta(n) + O(x)$
n - počet prvků množiny M
x- počet míst vycházejících z vybraného přechodu

isIndcludeS
Porovnává obsah prvku množiny S s množinou M. Prvek množiny S je tvořen množinou míst STG. Pokud se obsah prvku množiny S schoduje se všemi prvky množiny M, pak metoda vrátí false.
Asymptotická složitost metody: $\Theta(n) + O(xnn) + O(n)$
n - počet prvků množiny M
x - počet stavů v SG

sjednoceni
Do množiny M se překopíruje množina M ochuzená o vstupní místa vybraného přechodu t sjednocenou s výstupními místy vybraného přechodu t.
Asymptotická složitost metody: $\Theta(n)$
n - počet míst vycházejících z vybraného přechodu

changCode
Nastaví proměnnou obsahující binární kód aktuálního stavu stavového grafu. Dalším jejím účelem je přenastavit zakódovaní podle vybraného přechodu, buď na jedna pokud aktuální přechod je rostoucí nebo na nula pokud je klesající.
Asymptotická složitost metody: $O(1)$

transition
Vytváří přechod mezi stavy stavového grafu, jinými slovy vytváří vztah mezi prvky množiny S. Přechod je fyzicky umístěn v prvku z nějž vychází.
Asymptotická složitost metody: $O(n + 1)$
n - aktuální počet přechodů stavu
1 - přidáni nového přechodu

Celková asymptotická složitost transformace je součet všech asymptotických složitostí metod, z čehož plyne, že výsledná složitost bude $O(xn^2)$. Výsledek vychází ze složitosti metody isIncludes, ve které je potřeba použít tři cykly, abych byl schopen porovnat všechny množiny míst ve stavech stavového grafu s množinou míst, která by mohla tvořit nový stav.

5.8 Stuktura stavového grafu

Struktura se skládá ze dvou částí: *SGelem* a *SG*. Kde *SGelem* představuje informace o samotném stavu. *SG* je nosná konstrukce pro prvky typu *SGelem*. Schéma struktury je na obrázku 5.2.

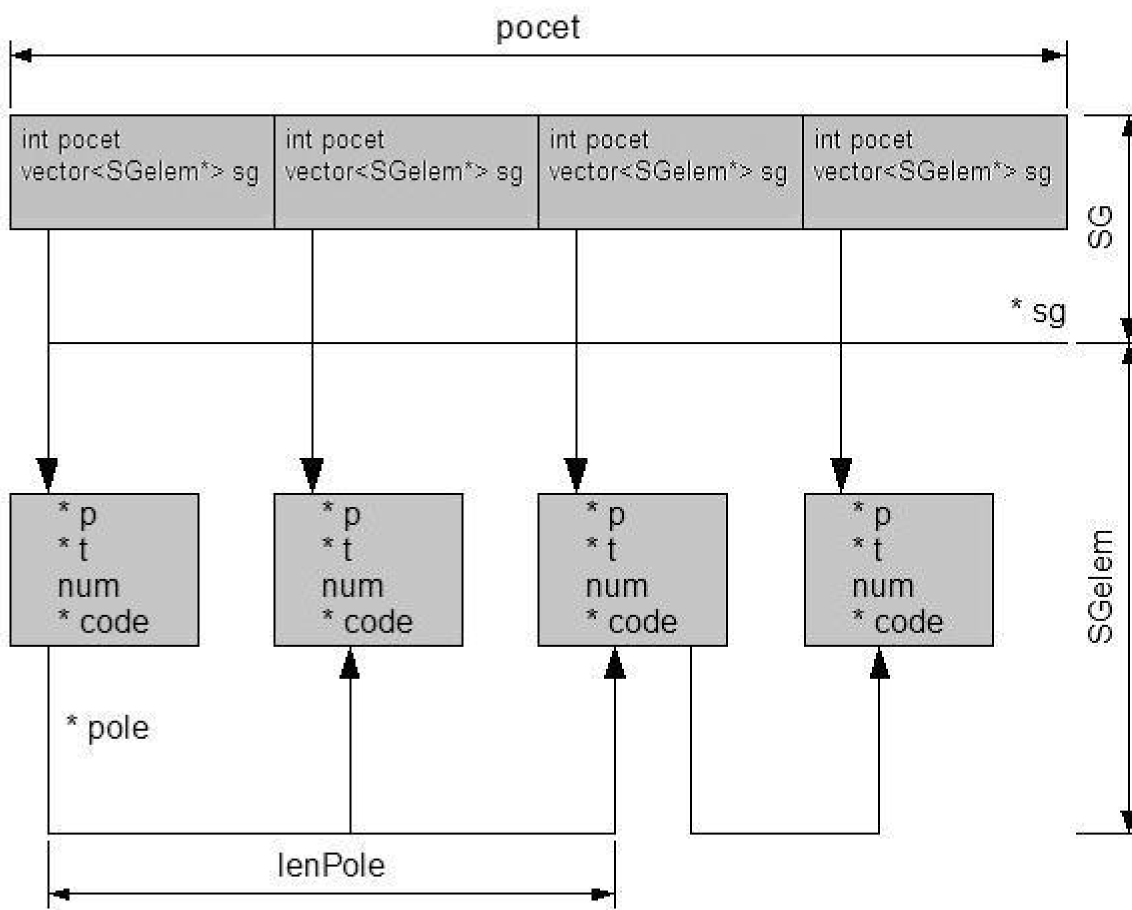
SGelem
Nese informace o zakódování stavu v proměnné *code*. V ukazateli *pole* jsou uloženy indexy, které udávají vztah k danému prvku v SG poli *sg*. *lenPole* udává počet odkazů. Informace o přechodu jsou v t a p.

```
struct SGelem{
int* pole;
int lenPole;
char* code;
int num;
int* t;
int* p;
};
```

SG

Nosná konstrukce obsahuje šablonu vector, ve které jsou uloženy jednotlivé elementy typu *SGelem*, dále obsahuje proměnnou udávající počet prvků ve vectoru.

```
struct SG {
int pocet;
vector<SGelem*> sg;
};
```



Obrázek 5.2: Struktura SG

5.9 Třída Region

Cílem třídy je vytvoření regionů, které se budou uchovávat ve struktuře *RegStavBuf*. Vstupem třídy je stavový graf, který je reprezentován strukturou *SG*. Třída má metody, které postupně projdou celou strukturu *SG* a pro jednotlivé vstupy, výstupy a interní signály rozřadí zakódování stavů do jednotlivých regionů. Cílem třídy je naplnit *RegStavBuf* vybranými daty z *SG*.

5.10 Třída KMap

Třída zpracovává strukturu *RegStavBuf*, struktura obsahuje ER a QR pro jednotlivé signály. ER obsahuje kódy stavů, které nabývají výstupní hodnotu jedna. QR obsahuje kódy stavů, které nabývají hodnoty nula. Pokud se nějaký kód nenachází v QR nebo v ER, pak se nachází v don't care a pro nás je nedůležitý. Na základě regionu poté probíhá minimalizace úplných mintermů. Mintermy jsou výchozí body pro vytvoření funkcí pro vstupy C-elementu. Minimalizace se provádí metodou fceExpanze.

fceExpanze

Metoda porovnává řetězec z ER- resp. ER+ s regiony z QR + a ER+ resp. QR- a ER-. Postup jak dochází k porovnávání a k minimalizaci je popsán v kapitole 4. v části Algoritmus minimalizace.

Asymptotická složitost metody, která se stará o minimalizaci je $O(xn)$. Kde n je počet proměnných v termu a x je počet termů v ER+ ∪ QR+ resp. ER- ∪ QR-.

5.11 Třída MakeCorLog

Cílem této třídy je převést získané funkce, ze třídy KMap proC-elementy, do VHDL kódu. Třída vkládá do *entitiní* části výstupní a vstupní signály a přidá ke jménům signálů *in* nebo *out* podle typu signálu. Dále třída předdefinovává pomocné signály, které mají stejná jména jako signály ve vstupním souboru , přidává pomocné signály pro Set a Reset daného C-elementu a provádí naportování signálu na C-element.

6 Testování

6.1 Vstupní zdroje pro naimplementovaný program

Textové popisy signálově přechodového grafu, dle specifikací na text a na samotný graf, lze nalézt na stránkách [4]. Konkrétně se jedná o soubory označené g/název_souboru.g. Ne však všechny tyto soubory jsou použitelné, použitelné soubory musí splňovat podmínky viz. Příloha.

6.2 Výstup ATACS

Na výstup přijdou jak informace o čase provádění jednotlivých úloh, tak rovnice popisující obvod. Pro příklad jsem nastavil generátor tak, aby zohledňoval C-element. Výstup pro ATACS může mít následující podobu:

[+Rx: (¬ Ax & Ao)]
[-Rx: (Ax & ¬Ao)]
[+Ax: (A1i & A2i)]
[-Ax: (¬A1i & ¬A2i)]
[+Ao: (Ri & ¬Rx)]
[-Ao: (¬Ri & Rx)]
[R1o: (Rx)] Combinational
[R2o: (Rx)] Combinational

Kde vstupní signály programu jsou *Ri, A1i, A2i* a výstupy nebo interní signály jsou *Rx Ax Ao R1o R2o* . Rovnice se generují pro výstupy a interní signály. Pokud je na konci řádku napsáno Combinational znamená to, že signál pro vstupy Reset a Set C-elementu je stejný, jen pro Reset je tento daný signál invertovaný. Výsledkem je vypuštění C-elementu. Pokud bych C-element nechal, vypadalo by to pro R1o následovně.

[+R1o: (Rx)]
[-R1o: (¬Rx)]

Znak ¬ znamená negaci signálu, amprsand & představuje logický AND. Textový řetězec, který je na levé straně od dvojtečky představuje jméno výstupu nebo interního signálu. Na pravé straně od dvojtečky jsou rovnice charakterizující výstup nebo interní signál Pokud je před výstupem plus nebo mínus jedná se o vstup pro Set, označeno + resp. pro Reset, označeno -. Pokud se ve výpisu vyskytuje více rovnic pro identický výstup nebo interní signál, pak mezi těmito rovnicemi je logický OR.

6.3 Vybrané grafy

Pro testování jsem zvolil grafy, které odpovídají specifikaci signálově přechodového grafu. Konkrétně se jedná o soubory g/roberto.g a g/vmeP.g. Příklady představují syntézu asynchronních kontrolérů. Grafy jsou specifikovány tak, že nedojde k redundantnímu zakódování při transformaci nebo k porušení výše zmíněných podmínek pro vstupní soubor. ATACS redundanci dokáže odstranit, například algoritmem vkládáním signálu a podat odpovídající výsledek. Podobný algoritmus však není součástí mého programu, proto pokud by došlo ke vložení takového grafu, pak by došlo k hlášení o redundantním zakódování.

6.4 Průběh syntézy na ATACS

Vybral jsem nejprve soubor g/roberto.g a poté jsme zvolil g/vmeP.g . ATACS jsem nastavil tak, aby jeho vstupem byly soubory s příponou .g. Nastavení zahrnuje parametr udávající, že program bude syntetizovat obvod s ohledem na C-elementy.

6.5 Výstup z ATACS

Pro vstupní soubor g/roberto.g jsem dostal následující funkce, které popisují chování obvodu.

[+done: (¬pc)]
[-done: (ev)]
[ev: (¬ done_in_1 & ¬ done_in_2 & done_out_1 & done_out_2 & done & pc)] Combinational
[¬ pc: (done_in_1 & done_in_2 & ¬ done_out_1 & ¬ done_out_2 & ¬ done & ¬ ev)] Combinational

Výstup pro vstupní soubor g/vmeP.g.

[lds: (csc0)] Combinational
[d: (ldtack & csc0)] Combinational
[+csc0: (dsr & ¬ ldtack)]
[-csc0: (¬ dsr)]

6.6 Průběh generování VHDL kódu v naimplementovaném programu

Jako vstup jsem použil stejné soubory jako pro ATACS, tedy roberto.g a vmeP.g. Po předložení prvního souboru vmeP.g naimplementovanému programu, jsem získal následující VHDL kód.
Hlavička kódu je, vždy stejná.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

Vstupům resp. výstupům se přidá před jméno signálu 'in' resp. 'out' .
entity vme is
Port (indsr: in STD_LOGIC;
inldtack: in STD_LOGIC;
outlds: out STD_LOGIC;
outd: out STD_LOGIC
);
end vme;

Začátek popisu chování obvodu ve VHDL se skrývá v architektuře.
architecture Behavioral of vme is Komponenta charakterizující C-element.
component cm
Port (S : in STD_LOGIC;

R : in STD_LOGIC;
out'_c : out STD_LOGIC);
end component;

Interní signály a ostatní pomocné signály jsou nadefinovány níže.
signal dsr : STD_LOGIC;
signal ldtack : STD_LOGIC;
signal lds : STD_LOGIC;
signal d : STD_LOGIC;
signal csc0 : STD_LOGIC;
signal csc0S : STD_LOGIC;
signal csc0R : STD_LOGIC;

Začátek od něhož dochází k vykonání chování obvodu.
begin
Přiřazení vstupních, výstupních signálů a pomocných signálů. Pomocné signály odpovídají jménům signálů ve vstupním souboru.
dsr <= indsr;
ldtack <= inldtack;
outlds <= lds;
outd <= d;

Samotné funkce vygenerovány naimplementovaným programem. Pokud je použit C-element, je k signálu přidáno písmeno S resp. R znázorňující Set a Reset.
lds<= csc0 ;
d<= ldtack and csc0 ;
csc0CM : cm
port map (csc0S , csc0R , csc0);
csc0S <= dsr and (not ldtack) ;
csc0R <= (not dsr) ;
Konec architektury
end Behavioral; Pro mě jsou nejdůležitější výstupní rovnice ty mohu porovnat s ATACS výstupními rovnicemi.

V našem případě se jedná o tyto řádky:
d<= ldtack and csc0 ;
csc0CM : cm
port map (csc0S , csc0R , csc0);
csc0S <= dsr and (not ldtack) ;
csc0R <= (not dsr) ;

Pro předložení druhého vstupního souboru roberto.g, získávám následující rovnice, přičemž vynechávám zbytek VHDL kódu, který není potřeba pro srovnání.
doneCM : cm
port map (doneS , doneR , done);
doneS <= (not pc) ;
doneR <= ev ;
ev<= (not done_in_1) and (not done_in_2) and done_out_1 and done_out_2 and done and pc ;
pcCM : cm

port map (pcS , pcR , pc);
pcS <= done ;
pcR <= done_in_1 and done_in_2 and (not done_out_1) and (not done_out_2) and (not done) and (not ev) ;

6.7 Výsledné srovnání

Pro soubor vmeP.g mohu přiřadit funkce VHDL kódu k jednotlivým rovnicím z ATACS. Vycházím z předpokladu, že na levé straně od '<=' ve VHDL kódu jsou výstupy a na pravé straně jsou odpovídající rovnice. Jména signálů jsou generovány naimplementovaným programem tak, aby bylo možné porovnat shodu. V hranatých závorkách jsou výsledky z ATACS.
lds<= csc0 ;
[lds: (csc0)] Combinational
d<= ldtack and csc0 ;
[d: (ldtack & csc0)] Combinational

csc0CM : cm
port map (csc0S , csc0R , csc0);
csc0S <= dsr and (not ldtack) ;
[+csc0: (dsr & ¬ ldtack)]
csc0R <= (not dsr) ;
[-csc0: (¬ dsr)]

Pro první vstupní soubor mohu konstatovat, že výsledky se shodují.

Srovnání pro druhý vstupní soubor :
doneCM : cm
port map (doneS , doneR , done);
doneS <= (not pc) ;
[+done: (¬ pc)]
doneR <= ev ;
[-done: (ev)]

ev<= (not done_in_1) and (not done_in_2) and done_out_1 and done_out_2 and done and pc ;
[ev: (¬ done_in_1 & ¬ done_in_2 & done_out_1 & done_out_2 & done & pc)] Combinational

pcCM : cm
port map (pcS , pcR , pc);
pcS <= done ;
???
pcR <= done_in_1 and done_in_2 and (not done_out_1) and (not done_out_2) and (not done) and (not ev) ;
[¬ pc: (done_in_1 & done_in_2 & ¬ done_out_1 & ¬ done_out_2 & ¬ done & ¬ ev)] Combinational

V druhém případě mohu konstatovat, že pro všechny rovnice ve VHDL se našly odpovídající

rovnice z ATACS, kromě jedné označené *???*. Ovšem vzápětí podotknu, že se nejedná o chybu v logice. Problém je způsoben tím, že naimplementovaný program nedokáže rozeznat, že pro signál v Setu, existuje invertovaný signál v Resetu a tudíž schopnost vypuštění. Program dokáže odstranit pouze prví výskyt, pokud v Resetu a Setu je jeden stejný signál, přičemž v Resetu je signál invertovaný. Proč to není z pohledu logiky chyba? Protože signálem vždy dojde zároveň k nastavení Setu a Resetu, tudíž C-element ztrácí váhu, jelikož se dá nahradit vodičem na kterém bude jistě menší zpoždění, než na hradlech tvořící C-element a obvod pak může běžet rychleji.

6.8 Časová složitost

Pro vytvoření představy o době výpočtů jednotlivých částí programu jsem vytvořil dva grafy s 50 vstupními proměnnými, na kterých by testováni mělo proběhnout. Hodnota 50 je zvolena záměrně, jelikož se jedná o mezní hodnotou počtů vstupních proměnných programu. Mezní hodnota odpovídá více jak dvojnásobku největší hodnoty u příkladů z ATACSu. Vstupní proměnné zahrnují jak vstupní signály, tak i výstupní a interní signály. Počet míst v STG je omezen na 200 a počet stavů grafu SG na 2000. Pro stanovení hodnoty míst pro STG jsem vycházel ze skutečnosti, že grafy uvedené na stránkách ATACS, málokdy překročí počet míst o dva a půl násobek vstupních proměnných. Hodnoty pro vybrané grafy z ATACS jsou v tabulce 6.1, grafy jsou součástí obsahu CD. Hodnota stanovená pro SG je záměrně naddimenzována, jelikož stanovení hodnoty pro počet stavů je předem velice obtížné.

Graf	**Vstupní signály**	**Výstupní signály**	**Interní signály**
$G1.g$	5	7	0
$G2.g$	3	3	0
$G3.g$	2	2	1
$G4.g$	3	2	0
$G5.g$	4	3	0
Graf	**Počet přechodů v STG**	**Počet větveni v STG**	**Počet míst v STG**
$G1.g$	24	2	29
$G2.g$	12	3	15
$G3.g$	10	1	11
$G4.g$	10	2	12
$G5.g$	14	2	22
Graf	**Počet stavů v SG**	**Čas výpočtu**	
$G1.g$	64	$0.563s$	
$G2.g$	24	$0.531s$	
$G3.g$	14	$0.453s$	
$G4.g$	17	$0.485s$	
$G5.g$	66	$0.562s$	

Tabulka 6.1: Vybrané grafy z ATACS

Grafy, které jsem připravil splňují stanovené podmínky pro STG. Správnost a průchodnost grafů byla prověřena na ATACS. Názvy souboru s popisem grafů jsou test1.g a test2.g, parametry grafů jsou uvedeny v tabulce 6.2. U grafu test2.g je druhé rozvětvení v STG vnořené v prvním rozvětvení. Soubory popisující grafy jsou součástí obsahu CD.

V tabulce 6.3 jsou uvedeny časy výpočtu pro vybrané úseky programu. Úseky se zaměřují na převedení vstupního souboru do struktury reprezentující STG, na transformaci STG na SG a

Název	*test1.g*	*test2.g*
Vstupní signály	17	20
Výstupní signály	18	17
Interní signály	15	13
Počet přechodů v STG	100	100
Počet větveni v STG	1	2
Počet míst v STG	101	102

Tabulka 6.2: Parametry grafů popsané v souborech test1.g a test2.g.

na vytvoření regionů a minimalizaci. Dále je v tabulce parametr udávající počet vygenerovaných stavů v SG. Časy uvedené v tabulce jsou získány za pomocí programu Cygwin a příkazu *time*.

Název	*test1.g*	*test2.g*
Počet stavů v SG	700	1888
Převod vstupního souboru [s]	0.453	0.485
Transformace [s]	0.234	1.437
Tvorba regionů [s]	1.568	13.162
Minimalizace[s]	1.737	$35,023$
Celkový čas výpočtu[s]	4.383	50.530

Tabulka 6.3: Časové údaje pro úseky výpočtu pro test1.g a test2.g. a počet stavů v signálově přechodovém grafu

Z hodnot, které jsem získal pro grafy popsané v souborech test1.g a test2.g, mohu konstatovat, že použitý algoritmus pro transformaci grafů je poměrně rychlý, ikdyž jeho asymptotická složitost je $O(xn^2)$. Kde n je počet míst obsahující tokeny a x je aktuální počet stavů v SG. Největší časové kvantum, pak zabírají úseky pro vytvoření regionů a minimalizaci. Asymptotická složitost pro vytvoření regionů je $\Theta(abc)$, kde a je počet vstupních proměnných, b je počet stavů SG a c počet přechodů mezi stavy SG. Pro úsek minimalizace je asymptotická složitost $O(defg)$, kde samotný algoritmus minimalizace má $O(fg)$. Proměnná d je počet výstupních a interních signálů, e je počet stavů v ER+ a ER-, f je počet vstupních proměnných a g je počet termů v ER+ $\cup$ QR+ resp. ER- $\cup$ QR-.Urychlit minimalizaci by možná mohl externí minimalizátor za cenu narušení kompaktnosti programu. Pro úplnost asymptotická složitost úseku pro převod vstupního souboru je $O(h)$, kde h je počet argumentů ve vstupním souboru.

6.9 Testování vstupu

Zde musím konstatovat, že nebylo v mých silách otestovat všechny možné varianty nesmyslných vstupů. Program samozřejmě funguje pro správně zadané vstupní soubory. Obsah souboru tedy musí dodržovat stanovené podmínky pro textové vyjádření STG.

6.10 Závěr testování

Pro správné vstupní soubory, které dodržují pravidla specifikace signálově přechodového grafu a vymezenou množinu popisného textu, je program schopen vygenerovat správný VHDL kód, který odpovídá skutečnosti.

7 Simulace

7.1 Pomůcky pro simulaci

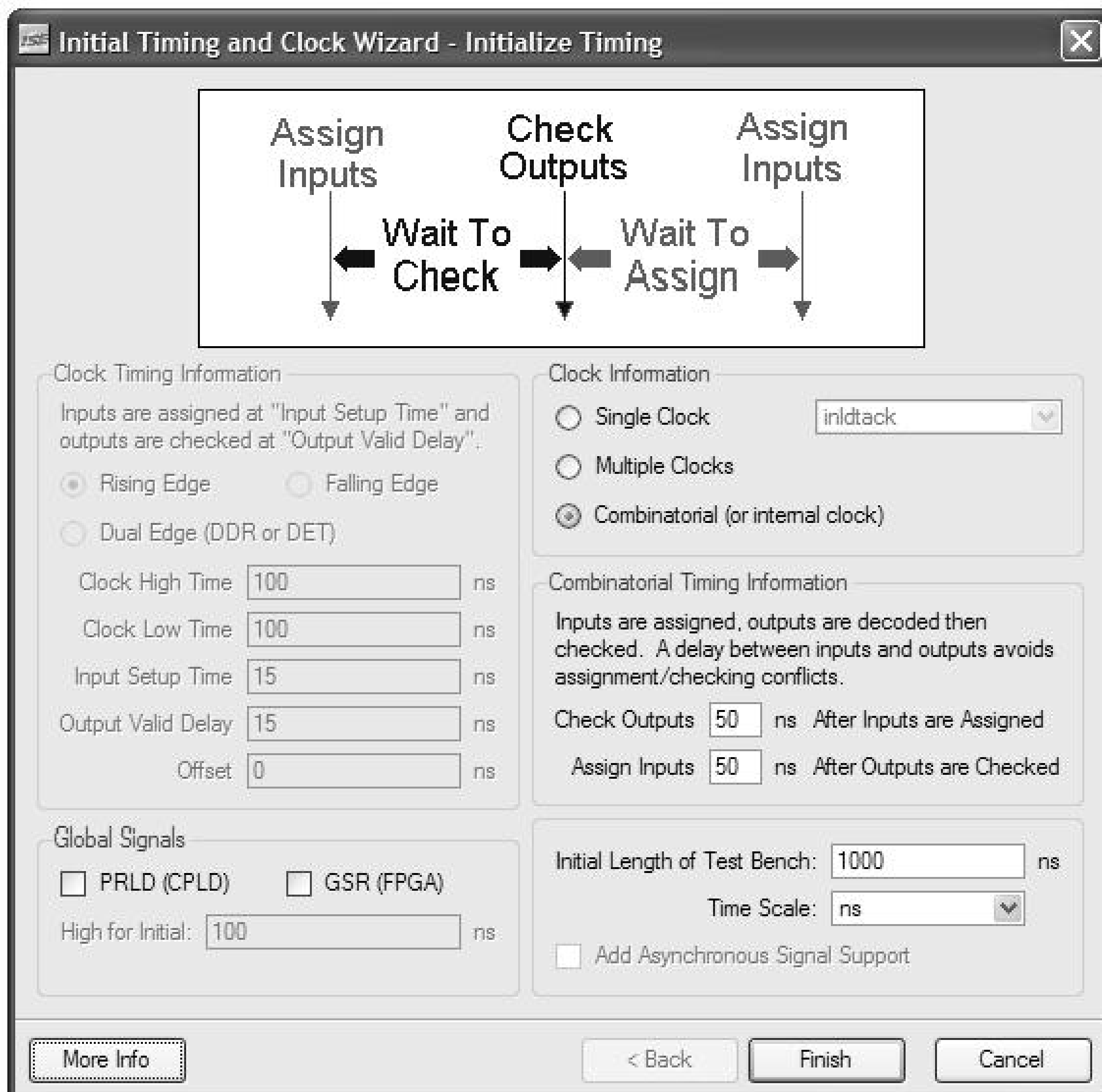
Xilinx ISE Simulátor je součástí Xilinx ISE 10.1. Veškeré informace o programu ISE, lze získat ze stránek [7].

7.2 Postup simulace, vytvoření simulace

Nejprve jsem vytvořil projekt v ISE 10.1. Přidal jsem do projektu VHDL soubory s koncovkou .vhd. Pro VHDL soubory jsem vytvořil Test Bench Waveform, ve kterém jsem nastavil vstupní signály na hodnoty jedna nebo nula, podle potřeby specifikace grafu nebo tabulky. Test Bench Waveform je šablona pro testování v programu ISE. Výsledkem simulace bude průběh signálů v čase a reakce na vstupní signály.

7.3 Nastavení Xilinx ISE Simulátoru

Simulace probíhala při následujících parametrech viz. 7.1.



Obrázek 7.1: Nastavení Xilinx Ise Simulátoru

7.4 Simulace C-elementu

Vstupem je popis C-elementu ve VHDL kódu, kód byl popsán v kapitole 3. Pro daný VHDL kód je vytvořen Test Bench Waveform, ve kterém jsem nastavil průběhy pro vstupy R aS, představující vstupy C-elementu Reset a Set.

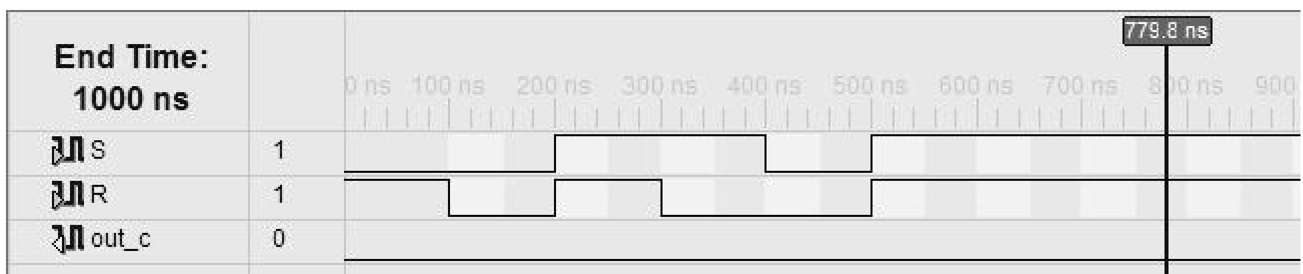
7.5 Nastavení pro C-element

Nastavení Test Benche jsem zvolil tak, abych prošel všechny možné vstupy, které pro C-element mohou nastat. Možnosti vstupů a jejich odpovídající výsledky jsou uvedeny v tabulce 7.1.

Set	**Reset**	**C**
0	1	0
0	0	C_{n-1}
1	1	C_{n-1}
1	0	1

Tabulka 7.1: Ukázka tabulky

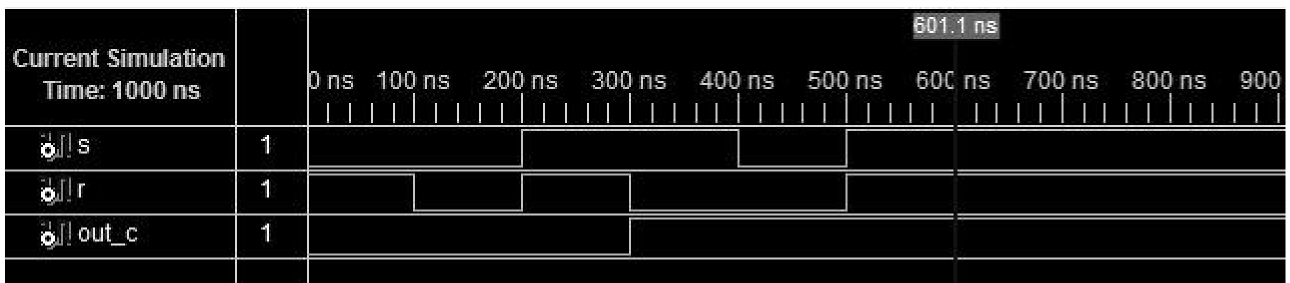
Nastavení Setu a Resetu jsem zvolil tak, abych nejprve vyzkoušel výstup na nulu a kopírování předchozího stavu a následně přechod na jedničku a kopírování předchozího stavu. Tento postup je uplatněn při nastavení 7.2.



Obrázek 7.2: Vstupy pro C-element

7.6 Výstup simulace pro C-element

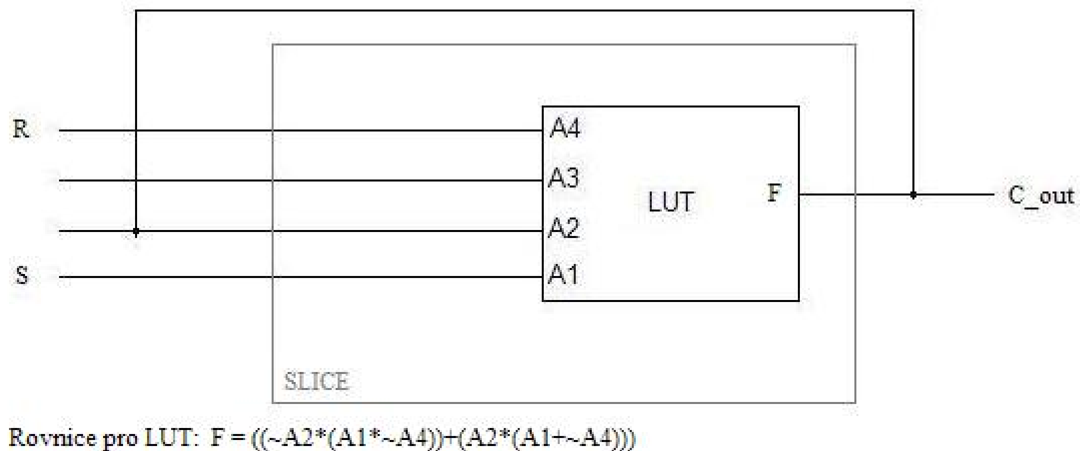
Na obrázku 7.3 je vidět průběh simulace C-elementu, reakce na vstupy, nastavení odpovídajících výstupních stavů korespondující s tabulkou.



Obrázek 7.3: Výstup pro C-element

7.7 C-element v FPGA

Po odsimulováni C-elementu na behavioralní úrovni, jsem nechal program ISE, aby obvod vysyntetizoval a umístil do FPGA. Program ISE vytvořil propojení mezi jednotlivými buňkami FPGA (slice). Vnitřek buňky především tvoří LUT, ve kterém je nahraná pravdivostní tabulka. Schematicky jsem se pokusil nakreslit propojení použitých prvků, tak jak je rozmístil program ISE.



Obrázek 7.4: Schéma C-elemntu

Obrázek 7.4 zachycuje schematický nákres vnitřku FPGA. Kombinační logika, kterou je C-element popsán, je umístěna v LUTu v podobě pravdivostní tabulky. V tabulce 7.2 jsem pro názornost vypsal pravdivostní tabulku LUTu . Pravdivostní tabulku pro LUT lze popsat rovnicí $F = (\neg A2 * (A1 * \neg A4) + (A2 * (A1 + \neg A4)))$. Na vstup A2 přichází signál ze zpětné vazby jak u C-elementu. A1 je vstup odpovídající vstupu S a A4 odpovídá vstupu R u C-elementu. Pro srovnání je vypsaná i pravdivostní tabulku popisující vnitřek C-elementu 7.3. Vnitřek C-elementu je popsán rovnicí: $f = Z * S + S * \neg R + Z * \neg R$, kde Z představuje zpětnovazební signál pro vstupy ANDů, R je negovaná vstup a S je vstup C-elmentu.

A2	**A1**	**A4**	**¬A2*(A1*¬A4)+(A2*(A1+¬A4))**
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Tabulka 7.2: Pravdivostní tabulka pro LUT

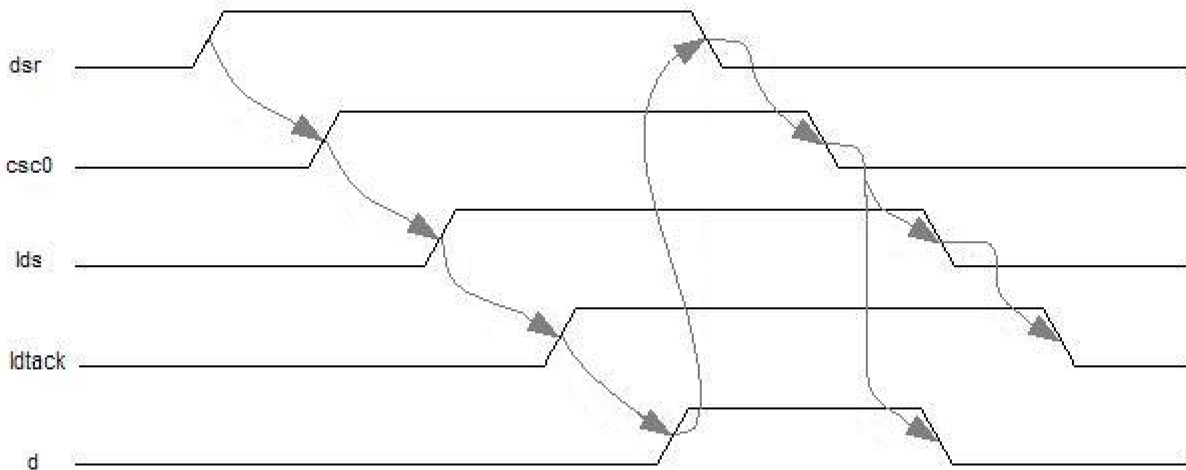
Z	R	S	Z*S+S*¬R+Z*¬R
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Tabulka 7.3: Pravdivostní tabulka pro vnitřní rovnici popisující C-element

Při porovnání hodnot v tabulkách jsem dospěl k závěru, že obvod vytvořený v programu ISE je správně sestaven.

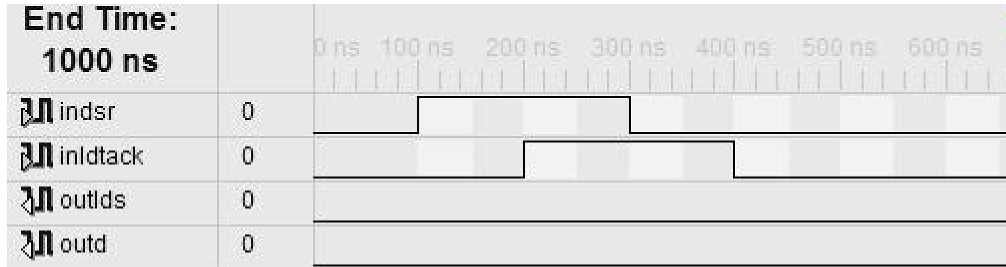
7.8 Simulace asynchronního obvodu

Pro syntézu jsem zvolil VHDL kód, který prošel při testu. Jedná se konkrétně o vstupní soubor vmeP.g. Pro tento soubor byl vygenerován soubor vme.vhd. Navíc je nutné do projektu přidat cm.vhd soubor, ve kterém se vyskytuje implementace C-elementu. Pro tento soubor jsem vytvořil odpovídající Test Bench Waveform. Dále jsem si připravil, ze signálově přechodového grafu, odpovídající časový diagram viz. 7.5. Test Bench Waveform jsem nastavil podle odpovídajícího časového diagramu. Časový diagram taktéž ukazuje předpokládané výstupy pro jednotlivé signály. Celý projekt je uložen ve složce vmeP.

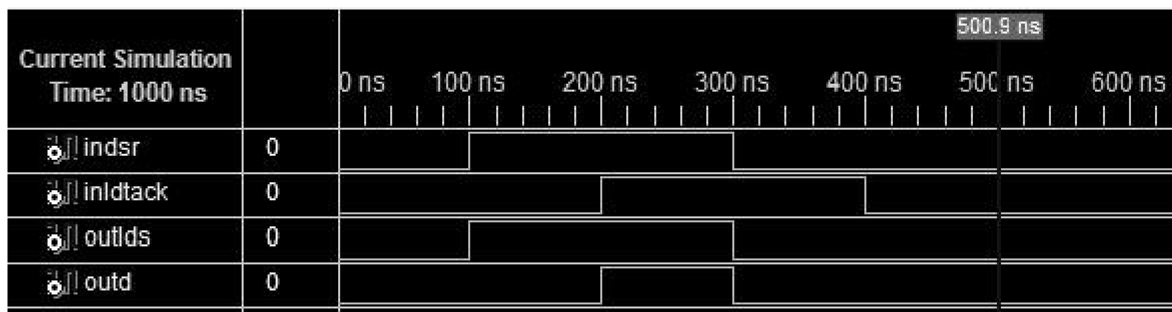


Obrázek 7.5: Časový diagram

Jména signálů v časovém diagramu odpovídají původním názvům ze souboru vmeP.g. Jména signálů v Test Bench Waveformu jsou rozšířeny o *in* resp. *out* pokud se jedná o vstupní signál resp. výstupní signál. Toto rozšíření odpovídá pojmenování vstupních a výstupních signálů ve vygenerovaném VHDL kódu.

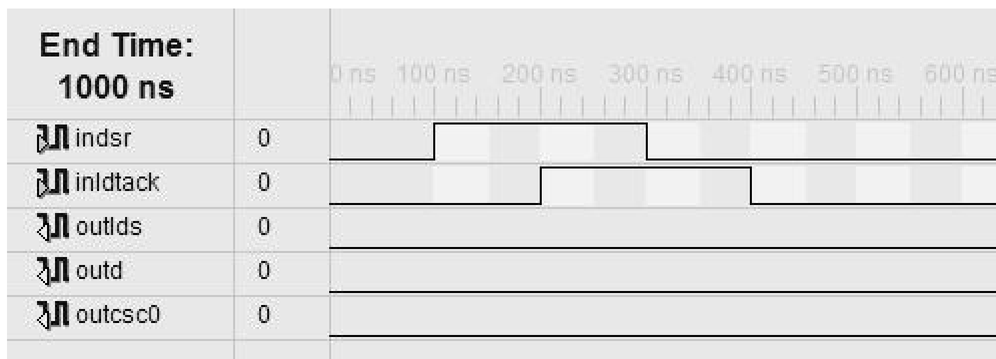


Obrázek 7.6: Vstup pro obvod var. 1

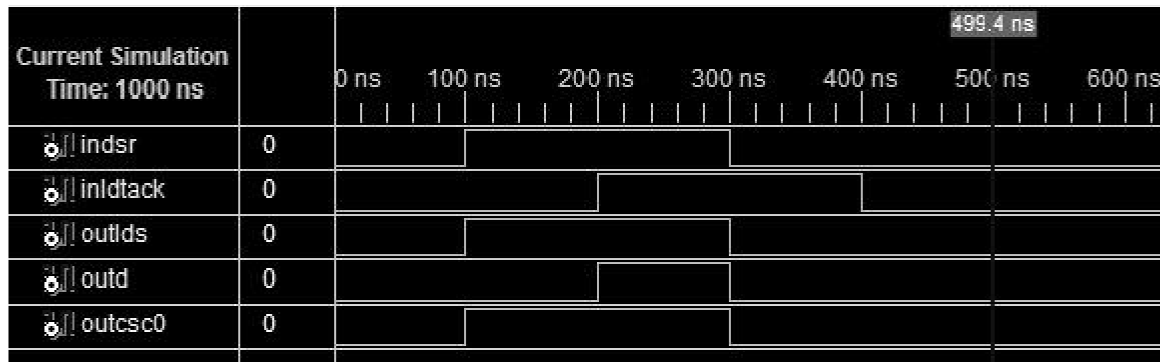


Obrázek 7.7: Výstup pro obvod var. 1

Na výsledku simulace lze pozorovat, že reakce na vstupní signály přichází okamžitě. Za povšimnutí stojí, že se zde nevyskytuje žádný hodinový signál. Na začátku průběhu se čeká na přechod signálu *indsr* z nula na jedna. Po příchodu signálu dojde ke změně signálu *outlds* a následnému čekání na vstupní signál *inldtack*. Při srovnání s časovým diagramem lze pozorovat změnu *dsr* signálu, následně změnu signálu *csc0* a *lds*. Jelikož signál *csc0* je interní není na simulaci vidět. Proto si signál *csc0* vyvedu a ukáži na dalším snímku výsledku simulace. Prozatím můžu konstatovat, že průběh odpovídá časovému diagramu. Po změně *outlds* obvod čeká na vstupní signál *ldtack*. V časovém diagramu vidím, že při změně *ldtack* dojde následně ke změně signálu *d*. Toto tvrzení je potvrzeno po změně signálu *inldtack* výsledkem simulace. Opět dojde k čekání na *dsr*. Změnu vyvolá snížení interního signálu a poté dojde zároveň ke změně výstupních signálu *d* a *lds* resp. v simulaci *outd* a *outlds*.
Původní Test Bench Waveform nezahrnuje interní signál, proto jsem jej vyvedl přidáním signálu *outcsc0*, abych mohl porovnat všechny signály s časovým diagramem.



Obrázek 7.8: Vstup pro obvod var. 2



Obrázek 7.9: Výstup pro obvod var. 2

Ve výsledné simulaci vidím, že interní signál *outcsc0* roste ve stejném časovém okamžiku jako signál *inlds*. Ve skutečnosti je změna signálu *inlds* vyvolaná signálem *outcsc0* a stejně tak i změna signálů *outd* a *outlds* vyvolaná signálem *outcsc0*, jak lze pozorovat na časovém diagramu.

7.9 Závěr simulace

Reakce na vstupní signály byly simulovány v Xilinx ISE Simulátoru. Porovnání změn jsem provedl vůči tabulce a časovému diagramu. Tabulka byla referenci pro C-element, časový diagram zase pro vybraný vygenerovaný VHDL soubor. Časový diagram byl připraven ze signálově přechodového grafu, který popisuje obvod, pro který byl vygenerován VHDL kód. Při porovnání výsledků, jak pro C-element s tabulkou, tak pro vybraný VHDL kód s časovým diagramem, jsem dospěl ke shodě.

8 Závěr

Práci jsem si na začátku rozdělil na tři základní cíle. Prvním cílem bylo se seznámit se světem asynchronní syntézy sekvenčních obvodů. Následujícím cílem bylo naimplementovat program, který bude automatizovat syntézu asynchronních sekvenčních obvodů, stavějící na poznatcích z první části. Konečným cílem bylo si odsimulovat asynchronní obvod, kde zdrojem pro simulaci byl VHDL kód získaný z naimplentováného programu.

Cílem pojmové části bylo získat základní informace o syntéze asynchronních sekvenčních obvodů a seznámit se s pojmy, které problematika obnáší. Blíže jsem se seznámil s návrhovým stylem založeným na modelu nevázaného zpoždění. Objasnil jsem si pojem Speed Independent obvod. Zadefinoval grafické formy specifikující asynchronní obvod a jejich nezbytné podmínky. Konkrétně se jednalo o graf signálově přechodový a stavový graf. Popsal jsem algoritmus, který transformuje signálově přechodový graf do stavového grafu. Objasnil si pojmy jako regiony a C-element, které jsem nezbytně potřeboval znát, abych mohl dosáhnou výsledku bezhazardní logiky.

V implementační časti jsem se pokusil implementovat program pro syntézu asynchronních sekvenčních obvodů popsaných signálově přechodovým grafem. Naimplementoval jsem jednotlivé moduly, které v celku tvoří syntetizační nástroj. Program dokáže úspěšně transformovat, minimalizovat počet proměnných ve funkcích a vytvářet výsledné VHDL soubory popisující chování obvodu. Účelem programu bylo si vyzkoušet automatizaci syntézy a získat nástroj pro syntézu asynchronních obvodů, čehož jsem úspěšně dosáhl.

V simulační části jsem se zabýval simulací C-elementu popsaného ve VHDL a vybraného obvodu popsaného ve VHDL kódu, jež byl vygenerován naimplementovaným programem. Mohu konstatovat, že jsem byl schopen odsimulovat veškeré možné vstupy podle tabulky popisující chování C-elementu. Kladný výsledek simulace mě opravňuje k použití VHDL kód C-elementu jako komponenty v jiných VHDL kódech. Následně jsem simuloval obvod, jehož VHDL kód byl vygenerován programem. K nastavení a porovnání simulace sloužil časový diagram, vycházející ze signálově přechodového grafu vybraného obvodu . Na základě výsledků simulace jsem mohl konstatovat shodu porovnávaných hodnot a prohlásit simulaci za úspěšnou.

9 Literatura

[1] Osobní stránky Luciana Lavagna - Osoba zabývající se asynchronními obvody. http://polimage.polito.it/~lavagno/.

[2] Petrify: a tool for synthesis of Petri Nets and asynchronous circuits. http://www.lsi.upc.edu/~jordicf/petrify/distrib/home.html.

[3] Algorithm to Find SG. http://www.es.isy.liu.se/courses/TSTE71/download/Lecture_4_2006.pdf.

[4] Myers Research Group - University of Utah, ATACS online. http://www.async.ece.utah.edu/atacs-bin/demo.

[5] Petri Nets World. http://www.informatik.uni-hamburg.de/TGI/PetriNets/.

[6] VSTGL (Visual STG Lab). http://vstgl.sourceforge.net/.

[7] Xilinx. http://www.xilinx.com/.

[8] Chris J. Myers. *Asynchronous Circuit Design.* John Wiley and Sons, Inc., 2001. http://www.async.ece.utah.edu/book/.

[9] J. Cortadella. Logic Synthesis of Asynchronous Controllers and Interfaces. *Springer*, pages 62–64.

[10] Jens Sparsø. *Asynchronous Circuit Design - A Tutorial.* 2006. http://www.imm.dtu.dk/ jsp/.

A Seznam použitých zkratek

VHDL VHSIC hardware description language

STG Signal Transition Graph

SG State Graph

ER Excistation region

QR Quiescent region

LUT Lookup table

FPGA Field-programmable gate array

B Uživatelská příručka

B.1 Vstup programu

Vstupem programu je textový soubor. Soubor obsahuje textově popsaný signálově přechodový graf. Popis grafu se skládá z části udávající daný příkaz, jež budu uvádět v apostrofech a argumentu, který budu uveden v uvozovkách. Příkaz je myšlený identifikátor, který uvádí argumenty. Řádek představuje jeden příkaz s argumenty, argumenty jsou oddělené mezerou. Textový popis vychází ze vstupního popisu programu ATACS.

'.name' „jméno“
Kde '.name' je příkazem , který bude určovat jméno výstupního souboru. Jméno výstupního souboru je uvedeno jako argument v uvozovkách. Ke jménu je přidán .vhd označující soubor s VHDL kód.

'.inputs' „arg“
Příkaz '.inputs' udává jména vstupů obvodu. Počet argumentu je libovolný. Pokud není, žádný vstup, příkaz '.inputs' se vynechává.

'.outputs' „arg“
Příkaz '.inputs' udává jména výstupů obvodu. Počet argumentu je libovolný. Pokud není žádný výstup, příkaz '.outputs' se vynechává. Pro dané výstupy bude vygenerován VHDL kód popisující vztahy vstupů, výstupů a interních signálu na výstup.

'.internal' „arg“
Příkaz '.internal' udává jména interních signálů, které se vyskytují v obvodu. Počet argumentu je libovolný. Pokud není žádný interní signál, příkaz '.internal' se vynechává. Pro daný interní signál bude vygenerován VHDL kód popisující vztahy vstupů, výstupů a interních signálu k internímu signálu.

'#@.init_state' „[*]“
Zakódování počátečního stavu stavového grafu je uvedeno v argumentu příkazu '#@.init_state'. Kde v argumentu * je uveden kód v binárním vyjádření, který musí být uvozen hranatými závorkami.

'.graph'
Tento příkaz je bez argumentu a udává, že následující řádky představují popis grafu. Kde jednotlivé argumenty na řádce jsou odděleny mezerami. První argument řádky představuje přechod z něhož se dostaneme do ostatních přechodů a ostatní přechody představují zbývající argumenty, přičemž je implicitně dané, že mezi přechody jsou místa. U přechodu označení + nebo – představuje, zda přechod roste nebo klesá.

'.marking' „ <*,*> “
Příkaz '.marking' označí místa tokenem, mezi přechody, označeny *, které jsou uvedeny v menšítku a většítku a jsou oddělenou čárkou. Pokud je více míst označeno tokeny, potom je popis oddělen mezerami. Celý popis je uzavřen mezi složené závorky. Příkaz '.marking' se uvádí, až po popsání grafu.

'.end'
Ukončení grafu oznamuje příkaz '.end' . Po jeho zadání se začne graf transformovat.

'# ' „arg“
Jedná se o komentář a jakýkoliv argument není brán v potaz.

Příklad vstupního souboru:

```
.name vme
.inputs  dsr ldtack
.outputs  lds d
.internal  csc0
#@.init_state [00000]
.graph
dsr+ csc0+
lds+ ldtack+
dsr- csc0-
lds- ldtack-
d- dsr+
ldtack+ d+
d+ dsr-
ldtack- csc0+
csc0+ lds+
csc0- d- lds-
.marking { <d-,dsr+> <ldtack-,csc0+> }
.end
```

B.2 Píšeme vstup

Vstup programu je textový soubor, který popisuje STG a musí splňovat dané podmínky. Začneme příkazem *.name*, za nimž bude následovat mezera a jméno výstupního souboru. Příkazy začínají vždy na novém řádku. Mezi řádky nejsou volné mezery. Příkazy *.inputs*, *.outputs*, *.internal* udávají ke jménům signálu jejich význam. Jména signálu jsou odděleny pomocí mezer. Pokud některý typ signálu neexistuje příkaz se neuvádí. Počáteční zakódování příkaz *#@.init_state*, následuje mezera a hranaté závorky. Uvnitř závorek se binární zakódování. Počet míst zakódování odpovídá součtu počtu vstupů, výstupů a interních signálů. Příkaz *.graf*, říká , že následující řetězce jsou vztahy mezi přechody. Vztahy byly, již popsány. Po popsání grafu přechody následuje .marking, který označuje místa, která mají být označena tokenem. Konec grafu *.end*.

B.3 Spuštění programu a manipulace

Jelikož je program exe soubor, tak stačí jej jen spustit. Konzole Vás poté vyzve k zadání názvu souboru, který obsahuje popis STG. Obsah je demonstrován výše. Pokud graf splňuje všechny požadavky dojde k přetransformování a k vygenerování VHDL kódu. Název souboru obsahující VHDL kód bude vypsán na konzoli.

C Obsah přiloženého CD

- Program - složka obsahuje program s příklady
  - program.exe - exe soubor program
  - vmeP.g - vstupní soubor, příklad
  - roberto.g - vstupní soubor, příklad
  - G1.g - vstupní soubor, příklad
  - G2.g - vstupní soubor, příklad
  - G3.g - vstupní soubor, příklad
  - G4.g - vstupní soubor, příklad
  - G5.g - vstupní soubor, příklad
  - test1.g - vstupní soubor, příklad
  - test2.g - vstupní soubor, příklad
- Zdrojove_kody - složka obsahuje zdrojové kódy programu
  - celem.h - obsahuje třídu pro vytvoření VHDL kódu
  - deleteStruct.h - obsahuje třídu pro dealokaci struktur
  - docommand.h - obsahuje třídu pro převod vstupních arg do STG struktury
  - expanse.h - obsahuje třídu pro a pro minimalizaci termů
  - fromfile.h - obsahuje třídu pro čtení a zpracování souboru
  - main.cpp - main funkce
  - makesg.h - obsahuje třídu pro transformaci STG na SG
  - pri.h - obsahuje třídu pro přidělování čísel příkazům
  - region.h - obsahuje třídu pro vytváření regionů
  - sg.h - obsahuje struktury SG
  - stg.h - obsahuje struktury STG
  - structregion.h - obsahuje struktury Regionu
  - *.* - zbývající soubory jsou projekt Dev-C++
- VHDL_projekt
  - c_muller - složka obsahující VHDL kód C-elementu a testebench C-elementu
    - c_muller.ise - spouštěcí soubor projektu v ISE
    - cm.vhd - VHDL kód C-elementu
    - *.* - zbývající soubory jsou součástí projektu v ISE
  - vmeP - soužka obsahující VHDL kódy příkladu, dva testbenche, kde u jednoho je vyveden interní signál
    - cm.vhd - VHDL kód C-elementu, nezbytná součást
    - vmeP.ise - spouštěcí soubor projektu v ISE
    - vme.vhd - VHDL kód příkladu, obsahuje portování C-elementu
    - *.* - zbývající soubory jsou součástí projektu v ISE
- BP_Zbyněk_Moler.pdf - Text bakalářské práce ve formatu pdf
- Text - složka obsahuje zdrojové soubory pro TeX