ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická – Katedra počítačů

Diplomová práce

Převaděč formátů pro specifikaci logických obvodů

V Praze 2005 Vlastimil Kozák

Obsah

Obsah ... 2
Zadávací formulář ... 4
Anotace ... 5
Prohlášení autora o využití práce fakultou ... 6
Prohlášení o samostatném zpracování diplomové práce ... 7
Poděkování ... 8
1. Úvod ... 9
2. Struktura 1:1 ... 13
3. Architekturní struktura ... 17
4. Formát Bench ... 23
4.1 Základní formát ... 23
4.2 Rozšíření ... 24
4.3 Nedostatky ... 25
5. Formát Edif ... 26
6. Ostatní formáty ... 30
6.1 CIR ... 30
6.2 PLA ... 31
6.2.1 Typ fd ... 32
6.2.2 Typ fr ... 33
6.2.3 Zvolená implementace ... 33
6.3 BLIF ... 33
7. Simulátor ... 36
7.1 Implementace simulace kombinačních obvodů ... 36
7.2 Implementace simulace sekvenčních obvodů ... 37
7.3 Implementace nastavení vnitřního stavu DKO ... 38
8. Konvertory ... 39
9. Závěr ... 40
10. Seznam použité literatury ... 41
11. Přílohy ... 42
11.1 Gramatika formátu BENCH ... 42
11.2 Gramatika formátu EDIF ... 42
11.3 Příklad obvodu C17 z hradel AND a OR ve formátech EDIF, BENCH a CIR ... 44
11.3.1 EDIF ... 45
11.3.2 BENCH ... 49
11.3.3 CIR ... 49

11.4 Příklad převodu formátu EDIF obsahujícího součástky LUT do formátu BENCH. 51
11.4.1 Obsah vstupního souboru 51
11.4.2 Obsah výstupního souboru 52
11.5 Příklad převodu formátu BENCH do formátu CIR 54
11.5.1 Obsah vstupního souboru 54
11.5.2 Obsah výstupního souboru 54
11.6 Popis ovládání ukázkového programu 56
11.7 CD se zdrojovými kódy 56

Zadávací formulář

Anotace

Zadáním diplomové práce bylo navrhnout vhodné datové struktury vnitřní reprezentace různých standardizovaných i nestandardizovaných formátů pro popis struktury elektronických obvodů. Důležité bylo počítat s tím, že takovou univerzální strukturu bude fakulta používat v rozsáhlém EDA (Electronic Design Automation) systému. Dále pak implementovat načítání těchto formátů ze vstupních souborů, jejich vzájemnou konverzi a simulátor kombinačních a synchronních sekvenčních obvodů. Navrhnul jsem tedy dvě různé datové struktury. Jedna z nich je velmi jednoduchá, koresponduje z velké části s nestandardizovaným formátem BENCH. Zároveň se velmi dobře hodí pro simulaci. Druhá z nich je architekturní, strukturovaná, složitější, koresponduje z velké části s formátem EDIF. Hodí pro strukturované ukládání a využívaní komplexních informací o elektronickém obvodu. Pro účely simulace jsem implementoval převod architekturní struktury do té jednodušší. Alespoň do jedné z těchto struktur se podařil implementovat převod těchto formátů: BENCH, EDIF, PLA, BLIF. Podporován je ještě výstupní formát CIR. Simulovat je možné strukturu vzniklou načtením libovolného souboru obsahujícího jednu z výše zmíněných specifikací. Implementace byla provedena v prostředí jazyka Borland C++ 5.0 s tím, že převod na jiné platformy by vyžadoval pouze minimální množství úprav.

The objective of the diploma thesis was to design appropriate structures for internal representation of several different specifications of electronic circuits. The specification is intended to be further used in a wide electronic design automation system. Another objective was to implement a reading of those specifications from input files, their conversion and simulation of combinational and synchronnous sequentional circuits. I have designed two different structures. The first one is very simple, corresponds with unstandardised BENCH specification. It is very useful for simulation. The second one is architectural, structured, more complex, corresponds to the EDIF specification. It is useful for storing complex electronic circuit‘s information. There has been implemented a conversion of the second one to the first one, for the simulation purposes. The specifications implemented were: BENCH, EDIF, PLA, BLIF and the output specification CIR. The sourcecodes are designed for Borland C++ 5.0 language. It is also possible to compile them with another C++ compiler without major changes.

Prohlášení autora o využití práce fakultou

Autor Vlastimil Kozák, narozen 24.1.1979, prohlašuje, že souhlasí s využitím práce školou pro účely výuky a činností souvisejících s výukou.

………………………………..

Prohlášení o samostatném zpracování diplomové práce

Autor Vlastimil Kozák, narozen 24.1.1979, prohlašuje, že tuto diplomovou práci vypracoval sám a že citace použité v této diplomové práci jsou úplné.

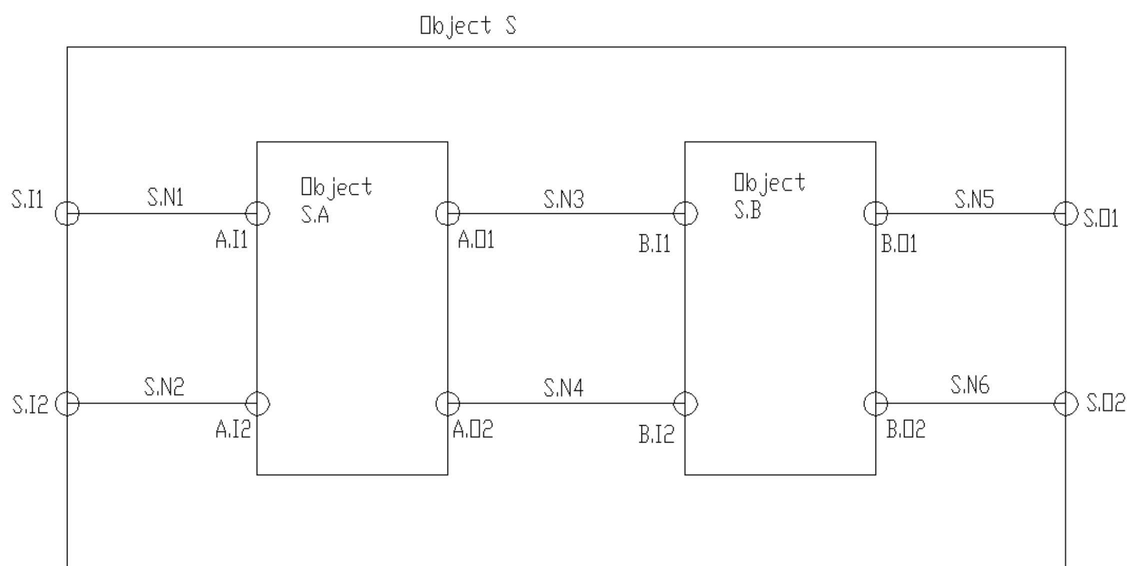
………………………………..

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu diplomové práce Ing. Petru Fišerovi za podnětné připomínky a návrhy k mé práci. Dále bych rád poděkoval všem svým blízkým a přátelům za morální podporu a cenné rady.

1. Úvod

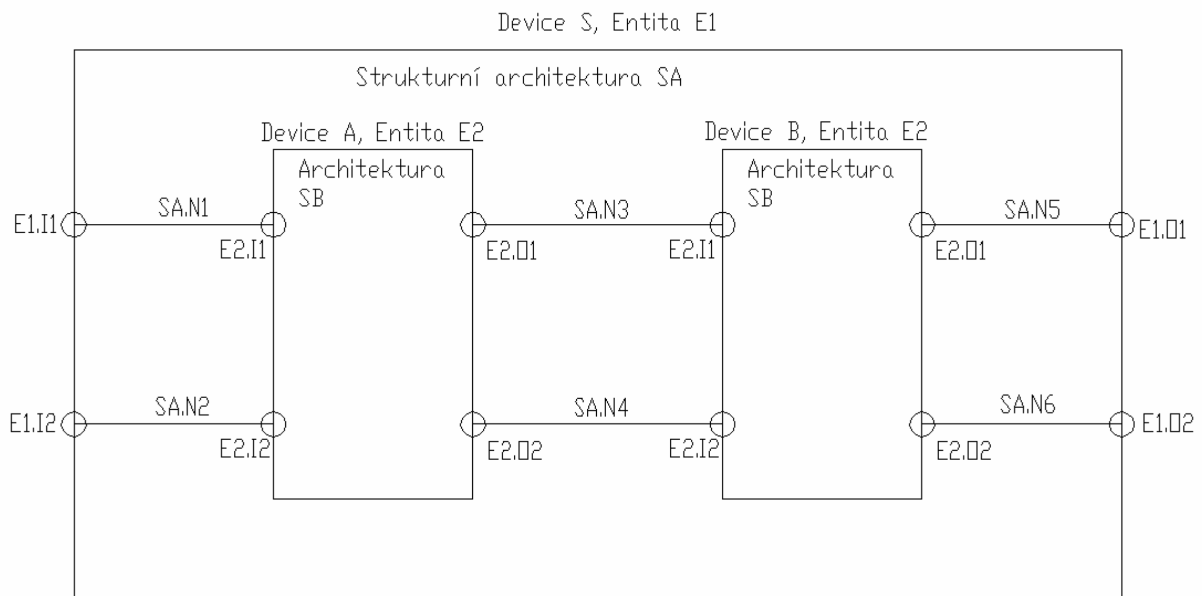
Na navrhované datové struktury jsem měl předem několik požadavků. Na jedné straně stála jejich snadná konstrukce a na druhé straně byla možnost efektivního provádění algoritmů na simulaci, převod do jiných forem a konverzi do výstupních formátů. Proto jsem nenavrhnul strukturu jednu ale dvě. Jednu ze struktur jsem navrhnul tak, aby co nejlépe fyzicky odpovídala skutečnému obvodu. Každá reálná součástka odpovídá jedné instanci dané třídy v navrhované struktuře. Každý vodič odpovídá jedné instanci třídy vodičů a konečně každý port koresponduje s instancí své třídy v navržené struktuře. Jednoduchý příklad je na obr. 1. Tato struktura má na první pohled několik předností a několik zjevných nevýhod. Její předností je zejména to, že každý myslitelný objekt je v ní fyzicky zastoupen a to ulehčuje simulaci obvodu reprezentovaného takovou strukturou. Pokud má fyzický elektronický obvod více úrovní, např. tištěný obvod, na kterém jsou integrované obvody a v nich ještě hradla, je snadné ho odsimulovat poměrně jednoduchým algoritmem popsaným v kapitole o simulaci. V něm hrají instance vodičů roli kontejnerů na hodnoty, které se na nich objevují v reálném provozu součástek. Taková víceúrovňová struktura musí mít poslední úroveň z hradel, u kterých dopředu známe jejich funkci. Fyzický obsah skutečných hradel nemá v naší struktuře svou reprezentaci a je nahrazen dohodnutým symbolem, který jeho funkci exaktně definuje. V konkrétním případě se jedná o konstatování, že jde např. o hradlo NAND. V obecném případě to ani musí být hradlo, ale libovolná součástka, u které si dohodneme její chování. Simulátor se pak řídí podle toho. Další požadavek je na snadné ukládání do formátu BENCH, který v podstatě s touto strukturou dobře koresponduje. Zde nebyly velké problémy, protože do navržené struktury se ukládájí i některé nadbytečné informace, které výstup do tohoto formátu usnadňují. U převodu do výstupního textového formátu CIR taktéž byly všechny informace v navržené struktuře snadno dostupné a nebyl tedy problém konverzi provést.



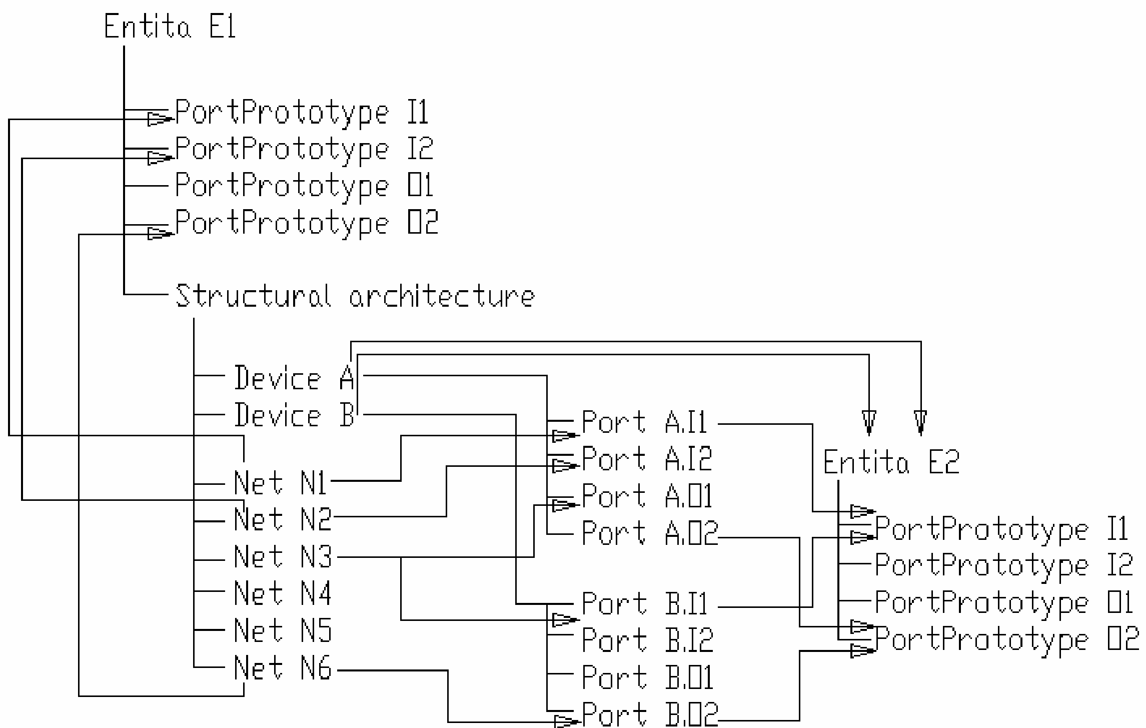
Obr. 1. Struktura 1:1 – pojmenování elementů

Její nevýhodou je skutečnost, že pokud máme v elektronickém obvodu několik stejných strukturovaných součástek, je jejich obsah v navržené struktuře přítomen vícekrát, aniž by musel být. Pro tu simulaci je to samozřejmě výhodné, ale pro uložení v některých strukturních formátech poměrně nevhodné. Navrhnul jsem proto ještě jednu vnitřní strukturu. Ta nově zná pojmy, jako je Entita a Architektura, známé např. z VHDL. Ty budou vysvětleny v kapitole o architekturní struktuře. Její výhoda spočívá v tom, že pokud máme nějakou opakující se součástku, řekněme např. sčítačku, a do vnitřní struktury jí umístíme vícekrát, její obsah (architektura) bude v paměti pouze jednou. Samozřejmě zde nejde jen o efektivní využití paměťového prostoru, ale zejména informace, že se jedná o totožné součástky, je velmi důležitá. Máme tím na mysli i takovou informaci, že stejná součástka (má stejnou entitu) může být vyrobena různými výrobci (obecně má jinou architekturu). I zde ovšem platí, jako v předešlém případě, že fyzický obsah skutečných hradel nemá ani v této struktuře svou reprezentaci a je nahrazen dohodnutým symbolem, v této struktuře mu říkáme Behavior.

Na obr. 2 vidíme příklad obvodu, do kterého jsou zapojeny dvě totožné součástky. Na obr. 3 pak nástin hierarchie instancí tříd architekturní struktury pro takový obvod. Třídy budou podrobně vysvětleny v kapitole o architekturní struktuře.



Obr. 2. Architekturní struktura – pojmenování elementů



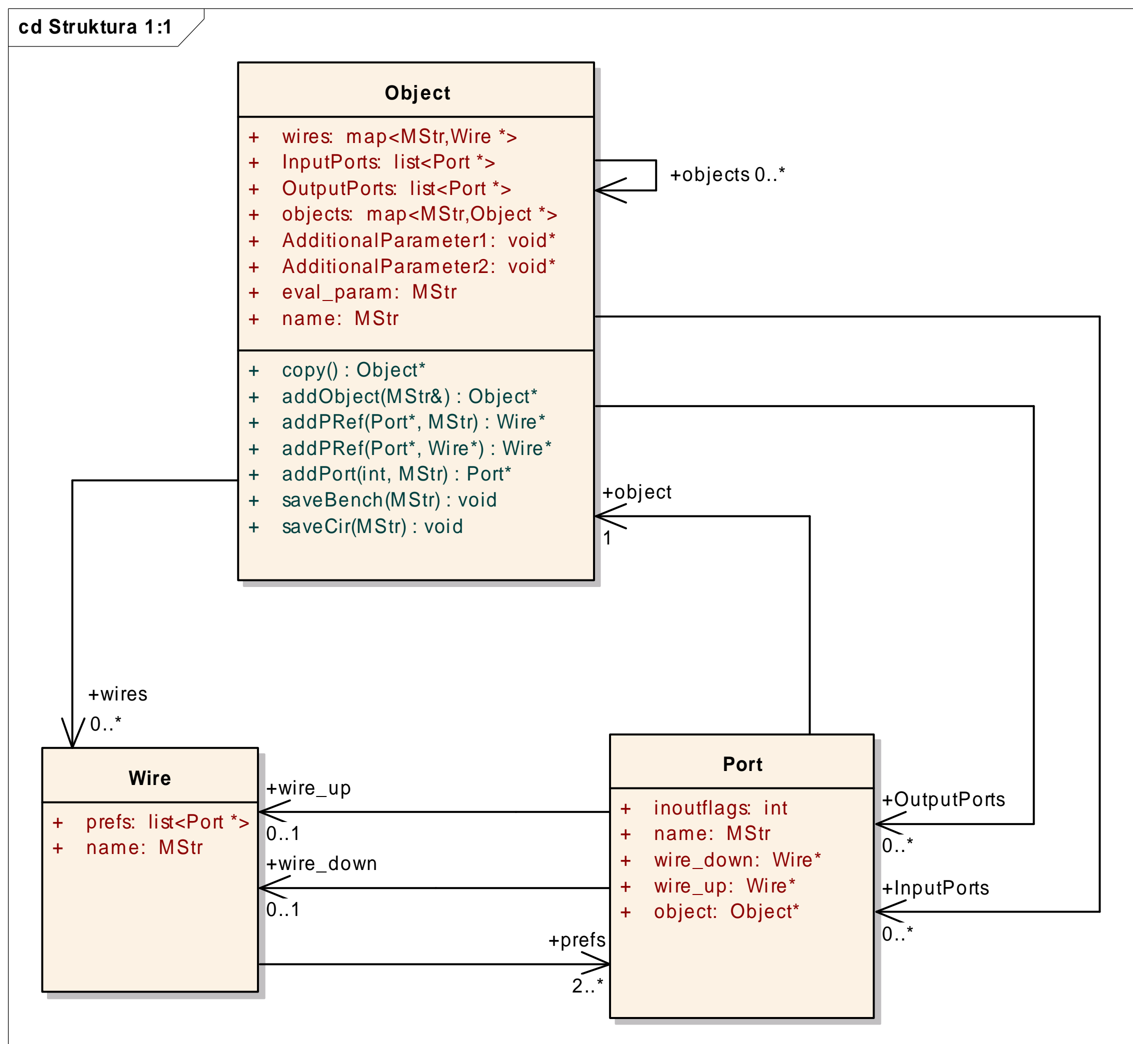
Obr. 3. Architekturní struktura – hierarchie instance

Taková struktura má ovšem i své nevýhody. Bez problémů nad ní můžeme provést simulaci kombinačních obvodů. U sekvenčních to ale nejde, protože nemáme kam zaznamenat informace o minulých stavech u součástek, které se v obvodu vyskytují vícekrát, ale mají společnou strukturní architekturu. Jako řešení problému jsem nezvolil kopírování

těchto architektur, ale implementoval jsem mechanismus převodu této struktury na strukturu předešlou, u které nám simulace nečiní žádný problém.

2. Struktura 1:1

Struktura, kterou jsem nazval 1:1, nejlépe odpovídá intuitivnímu vnímání jednoduché i složité struktury jakéhokoliv elektronického obvodu. Obsahuje proto jen tři třídy, jejichž důležité datové položky a metody včetně vzájemných asociací shrnuje obr. 4. U šipek je vždy uvedena položka, ke které se vztahuje, a kardinalita takového vztahu.



Obr. 4. Struktura 1:1 – class model

Popíšeme nyní, jak vypadá struktura, která svým obsahem kopíruje fyzický elektronický obvod. Třída *Object* je společná třída pro hradla a pro zařízení se svou definovanou strukturou. Pod pojmem zařízení budeme obvykle rozumět celý obvod, jehož struktura je např. načtena z nějakého souboru. Instance třídy *Wire* je reprezentací fyzického vodiče. Obdobně třída port je reprezentací fyzického portu. Do portu v naší struktuře vede vždy jeden vodič tzv. ven, tj. tak, že spojuje zařízení, kterému port patří, s nadřazeným obvodem. Ven vodič nevede jen tehdy, pokud takové zařízení není nikam zapojeno. Do portu

pak vede vždy jeden vodič tzv. dovnitř, tj tak, že spojuje obsah zařízení, kterému port patří, s tímto portem. Dovnitř vodič nevede jen tehdy, pokud takové zařízení nemá vnitřní strukturu a jeho obsah je definován jiným způsobem, jedná se tedy o hradla a jiné jednoduché součástky. Jeden vodič obecně spojuje dva a více portů. Implementace je taková, že ukazatel na instanci portu je zmíněn v seznamu *Wire::prefs*. U vstupních portů je nepřípustné propojit jich více jedním vodičem zevnitř. Důvod je vlastně ještě obecnější, a to ten, že jeden vodič nesmí mít více než jeden budič. Každá instance portu ví, který vodič, pokud takový existuje, do něj vede zevnitř a který zvenku. To má uloženo v datových položkách *Port::wire_down* a *Port::wire_up*. Dále si pamatuje, kterému Objektu patří, datová položka *Port::object*. Ve všech třech případech se jedná o ukazatele na instance těchto tříd. Každý port má ještě informaci o tom, zda je vstupní (1), výstupní (2) nebo vstupně výstupní (3). Čísla v závorkách jsou hodnoty, kterých nabývá datová položka *Port::inoutflags*. Další poslední nezbytná položka je *Port::name*, která má identifikační význam ve výstupech konverzních algoritmů, kde výstupní formát rozhraní zařízení identifikuje jmény těchto portů. Object, který má vnitřní strukturu, se skládá z kontejneru na Podobjekty (*Object::objects*), z kontejneru na vstupní porty (*Object::InputPorts*), z kontejneru na výstupní porty (*Object::OutputPorts*) a z kontejneru na vodiče (*Object::wires*). V případě objektů a vodičů jsou kontejnery instance třídy *map* ze standardní šablonové knihovny C++. Jejich prvky jsou v obou případech indexovány datovou položkou *name*, tzn., že všechny instance musejí mít hodnotu této položky unikátní. Důležitou datovou položkou třídy *Object* je *eval_param*. Tato položka je to jediné, co určuje, zda má součástka vnitřní strukturu. Pokud nemá, tedy tato položka je neprázdná, je to zároveň definice chování. Používá se např. při ukládání do výstupních forem a používá jí také simulátor. Tato položka je prázdná, tj. nabývá řetězcové hodnoty "", pokud součástka má vnitřní strukturu. Položka *name* je samozřejmě název součástky (ne typ součástky), pokud ji při konstrukci struktury neznáme, jako např. při načítání souboru ve formátu BENCH, je potřeba ho vygenerovat. Poznamenávám, že náš ukázkový program (na CD) je generuje ve tvaru "Gate_" + pořadové číslo. Metody těchto tříd, důležité pro vytvoření této struktury, jako jsou *Object::addPRef*, *Object::addPort*, a další, uvedeme v následujícím příkladu.

Popíšeme nyní typický způsob konstrukce takové struktury. Je samozřejmé, že ne všechny informace nutně musíme mít chronologicky v tomto pořadí, ale jako příklad to snad bude stačit.

V paměti si vytvoříme novou instanci třídy Object, která nám bude zastřešovat nejvyšší úroveň našeho zařízení.

*Object *o = new Object;*

Do rozhraní budeme přidávat porty, první parametr, určuje, zda jde o vstupní, výstupní nebo vstupněvýstupní port (viz výše datová položka *Port::inoutflags*), druhý parametr je jméno portu. Návratová hodnota je ukazatel na vytvořený port, kterou bychom si schovali.

```
Port *pI=o->addPort(1,”in”);
Port *pO=o->addPort(2,”out”);
```

Nyní máme vytvořený obal našeho zařízení a budeme přidávat jednotlivé podobjekty a spojovat jejich porty s rozhraním již vytvořeného objektu a také mezi sebou. Nejdříve obyčejné hradlo (NAND), do kterého vedou např. vodiče jménem W1 a W2 a ze kterého vychází vodič W10.

```
Object *to=o->addObject("unique name“);
Port *p1 = to->addPort(1,“I1“);
Port *p2 = to->addPort(1,“I2“);
Port *p10 = to->addPort(2,“O“);
to->eval_param = “NAND“;
```

Nyní je vytvořeno nové hradlo NAND s porty pojmenovanými I1,I2,O. Na tyto porty připojíme výše pojmenované vodiče. Metodě *addPRef*, kterou použijeme, je jedno, zda vodič toho jména již existuje nebo neexistuje. V případě, že neexistuje, sama ho vytvoří a přidá do kontejneru na vodiče v daném zařízení. Zároveň v portu, který připojuje, vyplní správnou položku *Port::wire_down* nebo (v našem případě) *Port::wire_up*. Návratová hodnota je ukazatel na instance vodiče specifikovaného druhým parametrem. Může to tedy být i ten vytvořený.

```
Wire *w1 = o->addPRef(p1,“W1“);
Wire *w2 = o->addPRef(p2,“W2“);
Wire *w10 = o->addPRef(p10,“W10“);
```

Tato metoda má ještě jednu přetíženou variantu, kde druhý parametr je přímo ukazatel na instanci vodiče, který se do portu připojuje. V tomto případě se ušetří čas na vyhledání vodiče v kontejneru vodičů (log n).

Pokud přidáváme součástku, která bude mít vnitřní strukturu, nevyplňujeme položku eval_param. Pro definici jeho vnitřní struktury použijeme od začátku rekurzivně tento návod (vynecháme samozřejmě vytvoření objektu).

Na porty vytvářeného objektu musíme zevnitř připojit některý z vodičů. Jinak by pochopitelně struktura nebyla konzistentní. Použijeme opět metodu addPRef. Na port ‘in’ připojíme vodič ‘W1‘ a na port ‘out’ připojíme vodič ‘W10’.

o->addPRef(pI,w1);
o->addPRef(pO,w10);

Alternativně lze použít samozřejmě i předchozí variantu:

o->addPRef(pI,“W1“);
o->addPRef(pO,“W10“);

Tímto postupem získáme konzistentní strukturu, kterou lze uložit do výstupních formátů BENCH a CIR. Toho docílíme zavoláním metod *Object::saveBench* a *Object::saveCir*. Jejich jediný parametr je název souboru, do kterého má být výstupní formát uložen.

o->saveBench(“test.bench”);
o->saveCir(“test.cir”);

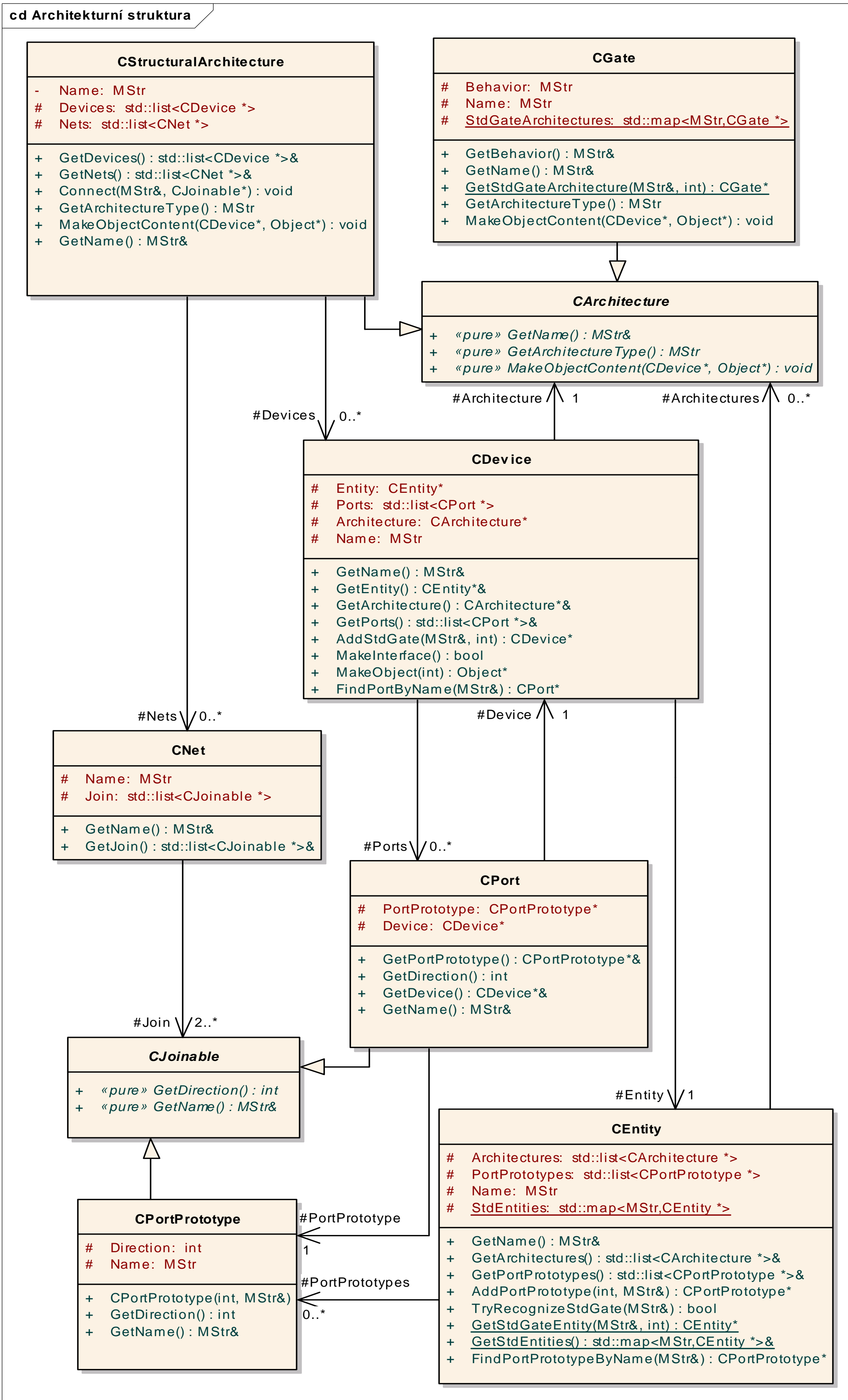
Třída *Object* má ještě další netypované ukazatelové položky, do kterých se ukládají různé informace v závislosti na hodnotě položky *eval_param*. Jmenují se *AdditionalParameter1* a *AdditionalParameter2*. Příkladem může být např. instance třídy *Object*, která reprezentuje součástku, jejíž chování máme zadáno PLA maticí. V položce *AdditionalParameter1* je uložen seznam párů řetězcových hodnot PLA matice. Tento parametr se použije při simulaci, když položka *eval_param* součástky začíná *“PLA”*. Popis formátu PLA je uveden v kapitole o formátu PLA.

V této chvíli zmiňme, že do této struktury lze ještě načíst formát BLIF, který obsahuje jednotlivé podsoučástky ve formátu PLA, dále definuje jejich propojení a své rozhraní.

Použité třídy mají ještě další položky, které v nezbytné míře zmíníme v kapitole věnující se simulaci.

3. Architekturní struktura

V této kapitole se zaměříme na sofistikovanější vnitřní strukturu, která má minimum redundantních informací. Inspirována je z velké části jazykem VHDL a také formátem pro popis vnitřní struktury EDIF. Svým obsahem nekopíruje fyzický elektronický obvod. Všímá si, které součástky do obvodů přidáváme vícekrát. Tyto pak mají shodnou architekturu. Pokud dvě součástky plní v obvodu stejnou funkci, mají stejné rozhraní, ale mají jinou vnitřní strukturu, mají jinou architekturu, ale mohou (nemusí) být stejné entity. Proto všechno obsahuje více tříd, jejichž důležité datové položky a metody včetně vzájemných asociací shrnuje obr. 5.



Obr. 5. Architekturní struktura – class model

Třída *CDevice* je reprezentantem každé součástky, kterou můžeme umístit do obvodu jako funkční celek. V typickém případě je součástí nějaké strukturní architektury (viz. níže), pokud není, znamená to, že dosud nikam nebyla tato součástka zapojena. Z hlediska implementace je v dané strukturní architektuře zmíněna jako ukazatel na instanci třídy *CDevice* v kontejneru na součástky, viz níže. Každá instance třídy *CDevice* je nějaké entity, která definuje jeho rozhraní. Implementace je taková, že metodou CDevice::GetEntity*()* získáme refenci na položku obsahující ukazatel na instanci třídy *CEntity*. Třída *CEntity* obsahuje kontejner prototypů portů. Referenci na něj získáme voláním metody *CEntity::GetPortPrototypes()*. Prototyp portu je instance třídy *CPortPrototype*. Tato třída má metodu *GetDirection()*, kterí vrací informaci, zda port vytvořený podle tohoto prototypu bude vstupní (1), výstupní (2) nebo vstupněvýstupní (3). Další jeho metoda je *GetName()*, která vrací referenci na jeho jméno. Entita má také seznam přípustných architektur. Referenci na tento seznam získáme voláním metody *CEntity::GetArchitectures*(). Po přiřazení entity našemu zařízení, nejčastěji zavoláme metodu *CDevice::MakeInterface*(), která zařídí vytvoření fyzických portů právě podle prototypů portů, získaných z entity. Takový port je instance třídy *CPort*. Její důležité metody jsou *CPort::GetPortPrototype()*, která vrací reference na prototyp portu, ze kterého byl vytvořen, a *CPort::GetDevice()*, která vrací referenci na datovou položku, obsahující ukazatel na instance třídy *CDevice*, které tento port patří. Metoda *CPort::GetName()*, vrací přímo referenci na datovou položku prototypu portu, ze kterého byl port vytvořen, obsahující název. Zařízení má datovou položku, ve které si uchovává svou architekturu, přičemž entita si sama uchovává seznam možných architektur pro zařízení mající tuto entitu. Referenci na tuto architekturu získáme voláním metody *CDevice::GetCurrentArchitecture()*. Architektura je instance některého z potomků třídy *CArchitecture* a specifikuje chování a vnitřní uspořádání zařízení. Na obrázku 5. vidíme, že z třídy *CArchitecture* dědí třída *CStructuralArchitecture* a třída *CGate*. To, že zařízení má za architekturu instanci třídy *CStructuralArchitecture*, znamená, že takové zařízení má vnitřní netriviální strukturu. Jinými slovy skládá se z jiných zařízení a vodičů, které propojují jejich rozhraní a rozhraní nadřazeného zařízení. To, že zařízení má za architekturu instanci třídy *CGate*, znamená, že nemá vnitřní strukturu a jeho chování je specifikováno jiným způsobem. Takže například při převodu do již popsané 1:1 struktury se použije informace získaná voláním metody *CGate::GetBehavior()*, která vrací referenci na řetězcovou hodnotu, která chování takové součástky definuje. Z názvy třídy plyne, že se bude s největší pravděpodobností jednat o hradla. Metoda *GetBehavior* by tedy vracela reference např. na hodnoty *"NOR"* nebo *"NAND"* apod. Strukturní architerura obsahuje kontejner na zařízení, instance třídy *CDevice*, a kontejner na vodiče, instance třídy *CNet*. Třída *CNet* tedy spojuje instance třídy *CPort* patřící k jednotlivým instancím třídy *CDevice* z kontejneru na zařízení. Pokud vodič propojuje i rozhraní zařízení mající tuto strukturní architekturu, nespojue fyzické

porty tohoto zařízení, protože architektura je společná obecně pro více instancí. V takovém případě v naší struktuře vodič do svého kontejneru na spoje přidá ukazatel na instanci prototypu portu, který připojuje. Datová položka, o kterou se v tomto případě jedná se jmenuje *CNet::Join* a je to seznam ukazatelů na instance potomků třídy *CJoinable*, viz. obr. 5.

V následujím příkladu si ještě vysvětlíme, jak fungují metody jednotlivých tříd, které používáme při konstrukci takové struktury, jako např. *CEntity::AddPortPrototype, CDevice::MakeInterface,Device::AddStdGate,CStructuralArchitecture::Connect* a jiné.

Nejprve si v paměti vytvoříme potřebné instance tříd zastřešujících naše zařízení. Jedná se o samotnou třídu *CDevice*. Entitu její instance a strukturní architekturu.

*Device *D = new CDevice;*
*CStructuralArchitecture *A = new CStructuralArchitecture;*
*CEntity *E = new CEntity;*

Vytvořenou architekturu přidáme do seznamu přípustných architektur dané entity a přiřadíme jí také našemu zařízení.
E->GetArchitectures().push_back(A);
D->GetArchitecture() = A;

Entitu pojmenujeme a budeme do ní přidávat prototypy portů, které bude naše zařízení obsahovat. Pro to použijeme metodu *CEntity::AddPortPrototype*, kde první parametr určuje, zda porty vzniklé podle tohoto prototypu budou vstupní, výstupní nebo vstupněvýstupní. Druhý parametr je pak název prototypu portu.

E->GetName() = “Naše entita”;
E->AddPortPrototype(1,“in1“);
E->AddPortPrototype(1,“in2“);
E->AddPortPrototype(2,“out“);

Po vytvoření entity jí přiřadíme do zařízení a zavoláme metodu *CDevice::MakeInterface()*, která vytvoří fyzické porty.

D->GetEntity() = E;
D->MakeInterface();

Pokud do obvodu přidáváme standardní hradlo, můžeme použít následující postup. Standardním hradlem rozumíme součástku, která má architekturu typu *CGate* a entitu

s rozhraním, které má v prototypech jeden výstupní port pojmenovaný “O” a předem stanovený počet vstupních portů pojmenovaných I1,I2,I3,… Metoda *CDevice::AddStdGate* má dva parametry. První specifikuje typ hradla a druhý počet vstupů. Architektura takového hradla má v názvu i počet vstupů a v položce chování jen typ hradla. Typicky tedy položka *Name* má hodnotu “NAND_3” a Behavior “NAND”.

*CDevice *g = D->AddStdGate(“NAND”,3);*

Přidali jsme do obvodu hradlo NAND se třemi vstupy. Víme přitom, jak jsou pojmenovány porty tohoto hradla.

Pokud bychom přidávali strukturní součástku a nebo jen hradlo s jinými porty, museli bychom vytvořit instanci třídy *CDevice* a přiřadit jí entitu a architekturu. Entitu můžeme vytvořit výše popsaným způsobem a nebo použít již existující.

Pokud vytváříme novou entitu je vhodné, ale nikoliv nezbytné, přidat jí do kontejneru volných entit. Zabráníme tím vytváření duplicitní instancí se shodným obsahem zejména při postupném načítání většího množství obvodů. Metody, které při tom použijeme, nebudou mít žádné povědomí o existenci entity, kterou bychom jen přidali do jednoho z nich. V tomto kontejneru, implementovaného jako třída *map* ze standardní šablonové knihovny C++, ji pak můžeme najít podle jména. Připomeňme, že metoda *map::insert* bere parametr pár hodnot, kde první je index (v našem případě jméno entity, které musí být unikátní, a druhý je hodnota(v našem případě ukazatel na instanci třídy *CEntity*).

*CEntity::GetStdEntities().insert(pair<MStr,CEntity *>(“entity name“,lEntity));*

Strukturní architekturu vytvoříme postupem popsaným výše. Architekturu nestandardního hradla tímto způsobem:

*CGate *a = new CGate;*
a->GetName() = “naše architektura”;

Do položky *Behavior* musíme přiřadit hodnotu, které budou rozumět algoritmy pracující s touto strukturou. Např. při převodu do struktury 1:1 tato položka odpovídá v této struktuře položce Object::eval_param, viz. předchozí kapitola.

a->GetBehavior() = “NOR”;

Nyní použijeme metodu *CStructuralArchitecture::Connect* na propojení portů. Jako příklad vytvoříme ještě jedno hradlo NOT a spojíme jeho vstup I1 s výstupem již vytvořeného

hradla NAND. První parametr metody *Connect* určuje jméno vodiče, který na port připojujeme. Druhý je ukazatel na potomka třídy *CJoinable*.

*CDevice *g2 = D->AddStdGate("NOT",1);*

D->GetStructuralArchitecture()->Connect("náš vodič",g->FindPortByName("O"));
D->GetStructuralArchitecture()->Connect("náš vodič",g2->FindPortByName("I1"));

Pro připojení vodiče na port nadřazeného zařízení musíme vyhledat jeho prototyp. Připojíme např. vstup hradla NAND I1 na vstupní port 'in1' a výstup hradla NOT na port 'out'.

D->GetStructuralArchitecture()->Connect("vodič1" , D->FindPortByName("in1")->GetPortPrototype());
D->GetStructuralArchitecture()->Connect("vodič1",g->FindPortByName("I1"));

D->GetStructuralArchitecture()->Connect("vodič2" , D->FindPortByName("out")->GetPortPrototype());
D->GetStructuralArchitecture()->Connect("vodič2",g2->FindPortByName("O"));

Výše uvedeným postupem získáme konzistentní architekturní strukturu. V úvodu jsem slíbil možnost převodu takové struktury na strukturu 1:1, např. pro účely simulace. Tady je:

*Object *o = D->MakeObject();*

Architekturou A jsme vytvořili strukturu, kterou můžeme snadno opakovaně používat. Stačí nyní vytvořit další instanci třídy *CDevice*, přiřadit jí entitu, vytvořit rozhraní a přiřadit mu tuto architekturu.

4. Formát Bench

Proberme si nyní, na čem byla ověřena správnost návrhu a implementace struktur ze dvou předchozích kapitol. Tímto formátem jsem ověřování začínal, dokonce se dá říct, že byl pro návrh 1:1 struktury prvotní inspirací. Až na pár detailů jí svým obsahem dobře odpovídá. Je velmi jednoduchý a přehledný. Nemá žádnou normalizovanou specifikaci. Proto jeho gramatiku jsem musel před implementací vymyslet a doladit, aby generovala obsah všech souborů, které jsem s tímto formátem měl k dispozici.

4.1 Základní formát

Začněme malým příkladem. Zde je obvod C17, převzatý z [4], specifikovaný formátem BENCH:

```
#   total number of lines in the netlist ..............   17
#        lines from primary input  gates .......    5
#        lines from primary output gates .......    2
#        lines from interior gate outputs ......    4

INPUT(G1gat)
INPUT(G2gat)
INPUT(G3gat)
INPUT(G6gat)
INPUT(G7gat)
OUTPUT(G22gat)
OUTPUT(G23gat)

G10gat = nand(G1gat, G3gat)
G11gat = nand(G3gat, G6gat)
G16gat = nand(G2gat, G11gat)
G19gat = nand(G11gat, G7gat)
G22gat = nand(G10gat, G16gat)
G23gat = nand(G16gat, G19gat)
```

Na úvod řekněme, že ani složitější obvody nemají žádné jiné syntaktické elementy, proto, co vidíme, už nemůže být horší. Gramatiku pro tento formát najdeme v příloze. Nicméně učiníme i slovní intuitivní popis formátu a rozpoznávaných lexikálních elementů.

Oddělovače lexikálních elementů jsou mezera, tabelátor a konec řádku. Komentář začíná znakem '#' a končí koncem řádku. Bezkontextová gramatika, podle které byl realizován tzv. rekurzivní sestup, jak je navržena, rozeznává pouze dvě klíčová slova. A to 'INPUT' a OUTPUT, v gramatice označené jako kwINPUT a kwOUTPUT. Implementace lexikálního analyzátoru je nejprve označí za identifikátory a teprve potom pohledem do tabulky klíčových slov zjistí, že se jedná o klíčová slova. Dále rozpoznává symboly: čárku jako oddělovač identifikátorů, rovnítko jako výsledek funkce a levou a pravou závorku, jejichž význam si vysvětlíme. Ostatní platné lexikální elementy jsou identifikátory, v gramatice označené jako IDENT. Symbol 'e' v gramatice označuje prázdný řetěz.

Identifikátory jsou dvojího druhu. Jeden druh reprezentuje jméno hradla. V příkladu máme jen hradla 'nand', a jak vidíme jedná se o identifikátor rozpoznaný hned po rovnítku. V prvotní navržené specifikaci tento identifikátor může nabývat jedné z následujících řetězcových hodnot: AND, NAND, OR, NOR, XOR, XNOR, BUFF, NOT, DFF. Připomeňme, že takové omezení se nikdy neřešilo při syntaktické analýze.
Druhý druh je vždy reprezentace jména vodiče. V našem příkladu všechny tyto identifikátory začínají na 'G'.

Formát BENCH specifikuje, že obvod má vstupní a výstupní porty, klíčovými slovy INPUT a OUTPUT. Všimněme si v příkladu, že žádným způsobem nespecifikuje jména portů. Udává pouze, které vodiče jsou na ně připojeny.

Přiřazovací příkaz G16gat = nand(G2gat, G11gat) říká, že v obvodu se nachází hradlo NAND, které má dva vstupy, vodiče G2gat a G11gat, a jeden výstup, G16gat. Do celkového obvodu není možné (formátem BENCH) začlenit součástku, která má dva a více výstupů

4.2 Rozšíření

V tuto chvíli už můžu prozradit, že v rámci ověření navržených struktur jsem implementoval načtení a uložení jisté nadmnožiny formátu BENCH, tedy takového, jak jsem ho pochopil z dostupných příkladů. V podstatě se jedná o možnost do obvodu přidávat součástky neuvedené v předchozím seznamu.

Příkladem může být součástka LUT (lookup table) známá především jako elementární součást některých programovatelných obvodů. Taková součástka má obvykle dva až šest vstupů a jeden výstup. Výstupní hodnota je přečtena z paměti adresované hodnotami na vstupních vodičích. To znamená, že např. čtyřvstupový LUT má paměť o velikosti 16 bitů. Číslo v šestnáctkové soustavě používá pro jednotlivé cifry symboly 0-9 a A-F. Taková jedna cifra nese ekvivalent čtyřbitové informace a tedy čtyřciferné číslo má 16 bitů.

V přiřazovacím příkazu musíme přímo uvést obsah paměti LUTu. Identifikátor takové součástky bude mít tvar "LUTn_xxxx", kde n udává počet vsupů a kde xxxx udává obsah

paměti v šestnáctkové soustavě. Počet cifer je dán velikostí největšího šestnáctkového čísla pro daný počet vstupů.

Takže např. LUT4_FFFF má na výstupu vždy log.1 bez ohledu na vstupní hodnoty a např. LUT3_01 má chování třívstupového hradla NOR.

V následujícím příkladu je vidět celé přiřazení. A je vodič, který vede do výstupního portu. B,C,D,E jsou vodiče, které vedou do vstupů LUTu v pořadí od nejméně významného bitu k nejvýznamnějšímu.

A = LUT4_ED1C(B, C, D, E)

Implementace je taková, že tento řetězec (např. ‘LUT4_ED1C‘) se uloží do *Object::eval_param*. Metody třídy *Object*, v našem případě metoda *Object::eval*, která bude zmíněna v kapitole o simulaci, by s tím tedy měly umět pracovat.

4.3 Nedostatky

Chtěl bych také zmínit to, s čím si má implementace neporadí a zkusit navrhnout možná řešení. Týká se to součástek, které mají více než jeden výstupní port. Pokud zkusíme načtený obvod mající takové součástky (můžeme ho získat např. načtením nějakého EDIFu, viz. následující kapitola) uložit ve formátu BENCH, získáme pro každý výstupní port stejnou pravou stranu přiřazení, tedy např.

D = DEV(A, B, C)
E = DEV(A, B, C)

V takovém případě nevíme, zda jsou v obvodu dvě součástky s dohodnutým chováním ‘DEV’ a nebo pouze jedna s dvěma výstupy D a E.

Možná, že to ani není důležité, protože přece jen formát BENCH je používán tam, kde takové situace nenastávají. Nicméně je asi vhodné uvést, že ne všechno lze do formátu BENCH převést.

Nabízí se následující řešení, z důvodů nepotřebnosti ale nebylo implementováno.

D, E = DEV(A, B, C)

5. Formát Edif

I když formát BENCH byl programem přiloženým na CD načítán do obou struktur, stále bylo potřeba ověřit správnost návrhu načtením formátu, který využívá všech vlastností, na které jsem se při návrhu zaměřoval, beze zbytku. Takovým formátem je EDIF. Jeho gramatika je o mnoho složitější než v předchozím případě. Výhodou bylo, že jsem nic nevymýšlel a gramatiku v EBNF jsem převzal z [6]. Zároveň je tento soubor s originální gramatikou v EBNF je přiložen na CD. Pro snadnou implementaci jsem musel udělat dvě věci. První byla vybrat správnou podmnožinu jazyka (originálního EDIFu) tak, aby implementace nebylo zbytečně mnoho. K tomuto problému jsem přistoupil podobně, jako když jsem v případě BENCHe gramatiku vymýšlel. Za správnou podmnožinu úplné gramatiky jsem označil takovou minimální, která generovala všechny dostupné příklady již z jiných zdrojů vygenerovaných EDIFů, např. [7]. Druhý úkol byl převést EBNF na bezkontextovou LL gramatiku. K tomu dobře posloužil návod v [1]. Cílem přitom bylo, aby vzniklá gramatika měla co nejméně pravidel a byla pokud možno přehledná. Podle mého názoru se to podařilo a to i přesto, že nově vzniklé neterminání symboly jsem nepojmenovával. Ještě jedna důležitá věc je, že výsledná gramatika je LL2 a ne LL1, jak je obvyklejší. Víme, že LL2 jde jednoduchým způsobem na LL1 převést, to by ovšem počet neterminální symbolů ještě zvýšilo a pravé strany pravidel by nezůstaly v podobě, kde začínají levou závorkou a končí pravou závorkou, viz. níže. Naimplementovat LL2 syntaktický analyzátor není přitom o nic těžší, než LL1 syntaktický analyzátor. Přibude pouze fronta na dopředu čtené symboly, přičemž přístup k ní byl implementován jako metoda lexikálního analyzátoru. Výsledná gramatika je v příloze.

S malým příkladem to tentokrát bude o něco složitější. Příklad níže není kompletní obvod. Poslouží pouze k intuitivnímu vysvětlení významu nejdůležitějších syntaktických elementů, zejména těch použitých při konstrukci architekturní struktury.

Pro podrobné vysvětlení bych odkázal na [2]. V příloze jsou pak v tomto formátu dvě různé definice obvodu C17.

```
(edif Synopsys_edif (edifVersion 2 0 0) (edifLevel 0)
 (keywordMap (keywordLevel 0))
 (status
  (written (timeStamp 1999 6 30 20 47 10)
   (program "Synopsys Design Compiler" (Version "1998.08"))
   (dataOrigin "company") (author "designer")
  )
 )
 (external pdt2 (edifLevel 0) (technology (numberDefinition))
```

```
 (cell NOR3_GATE (cellType GENERIC)
  (view Netlist_representation (viewType NETLIST)
   (interface (port I1 (direction INPUT)) (port I2 (direction INPUT))
    (port I3 (direction INPUT)) (port O (direction OUTPUT))
   )
  )
 )
)
(library DESIGNS (edifLevel 0) (technology (numberDefinition))
 (cell b02 (cellType GENERIC)
  (view Netlist_representation (viewType NETLIST)
   (interface (port reset (direction INPUT)) (port clock (direction INPUT))
    (port linea (direction INPUT)) (port u (direction OUTPUT))
   )
   (contents
    (instance U31
     (viewRef Netlist_representation (cellRef NOR3_GATE (libraryRef pdt2)))
    )
    (net (rename stato_2_ "stato[2]")
     (joined (portRef I1 (instanceRef U34)) (portRef I2 (instanceRef U39))
      (portRef I1 (instanceRef U43)) (portRef I3 (instanceRef U49))
      (portRef I2 (instanceRef U52)) (portRef Q (instanceRef stato_reg_2_))
     )
    )
   )
  )
 )
)
(design Synopsys_edif (cellRef b02 (libraryRef DESIGNS)))
)
```

Předem je třeba říct, že zdaleka ne všechny informace obsažené v takovém souboru jsem využil pro konstrukci datových struktur. Syntaktický analyzátor mnohé z nich jen zkontroluje, ale neznamenají žádný výstup. To je případ hned osmi prvních řádků v našem příkladu. Devátý řádek s elementem *external* je vlastně odkaz na externí definici knihovních součástek. Pod pojmem element budeme rozumět syntaktickou strukturu začínající levou závorkou následovanou jménem elementu s případnými dalšími symboly a končící pravou závorkou. Načtení knihovního souboru však zatím pro dostupné příklady EDIFů nebylo třeba

implementovat. Syntaktický element *external* je jinak v podstatě shodný s elementem *library*. *Library* je definice knihovny. V gramatice vidíme, že EDIF může obsahovat definici libovolného množství knihoven. Jméno knihovny (hned za klíčovým slovem *library*) se následně používá pro odkazování na konkrétní součástky. V definici knihovny nalézáme dále element *cell* následovaný řetězcovým identifikátorem. Pod ním se ještě nachází klíčové slovo *view* a ještě pod ním *interface*. Výskyt elementu *cell* říká, že v dané knihovně se nachází součástka (identifikovaná následovaným řetězcem) s architekturou definovanou elementem *view*. V EDIFu je definice rozhraní součástí architektury. Proto když při načítání do architekturní struktury vytváříme entitu, dáváme jí název ve tvaru: 'jméno knihovny' tečka 'jméno *cell*u'. EDIF může pro součástku rovnou definovat několik architektur. Tyto architektury mohou podle syntaxe mít dokonce jiné rozhraní, ale to naše struktura nepodporuje. V dostupných EDIFech se ovšem vyskytuje pouze jedna definice architektury, takže žádný problém nevzniká. Jméno architektury vytváříme ve tvaru: 'jméno *view*' tečka 'jméno entity'. Definice rozhraní následuje za klíčovým slovem *interface*. Jedná se o vyjmenování vstupních, výstupních a vstupněvýstupních portů. Směr je udán elementem *direction*. Jméno portu (jedná se vlastně o jméno prototypu portu) je uvedeno hned za klíčovým slovem *port*.

Pokud element *view* obsahuje element *contents*, znamená to že příslušné *view* je strukturní architektura. Samozřejmě hovoříme o případu načítání do architekturní struktury. V našem případě bude tedy architektura typu *CStructuralArchitecture*. V opačném případě jsou ještě dvě možnosti. Buď je chování součástky odhadnuto z jeho názvu (je volána metoda *CEntity::TryRecognizeStdGate*) a nebo je její struktura načtena z jiného EDIFu. Naše implementace se dívá po souboru s názvem, jakou má součástka, s příponou '.edf'. Alternativa s rozpoznáváním podle názvu je důležitá, protože pro následné konverze potřebujeme vědět, že se jedná o nějakou standardní součástku. Popis pouze jejího chování by nestačil např. pro převod do formátu BENCH, který má restrikce pro pojmenování hradel. Identifikace proběhne správně, pokud název začíná dohodnutým názvem hradla (jako ve specifikaci BENCHe) a pak ještě v případě v následujících jmen: LUTx (LUTx), INV (NOT), GND, FALSE, logic_0 (vše GND), logic_1 (DC), Flip_Flop_D_reset (DFF). V závorce je vždy uveden název součástky, kterému musí rozumět ostatní algoritmy. Pod elementem *contens* jsou obvykle elementy *instance* a *net*. Element *instance* umísťuje do obvodu součástku se jménem, které následuje po klíčovém slovu *instance* . Jeho entita je definovaná elementem *cellRef* a *libraryRef*. Jeho architektura elementem *viewRef* a jménem entity. Přítomnost elementu *net* umísťuje do obvodu vodič. Hned za klíčovým slovem *net* nalezneme jednu z možných definicí jména. V gramatice se jedná o neterminální symbol 'netNameDef'. Následuje element *joined*, pod kterým jsou již elementy *portRef*, kde každý identifikuje jedno místo, kam je vodič připojen. Pokud je součástí i element *instanceRef*, jedná se o označení

součástky, kterou jsme do obvodu umístily elementem *instance*. Pokud tento element není obsažen, jedná se přímo o port entity definované elementem *cellRef* a *libraryRef*.

Důležitým elementem je *design*, který říká jakou součástku vlastně soubor definuje. Po klíčovém slovu *design* následuje její jméno a pak již jednou zmíněná definice umístění součástky v knihovně. V EDIFech, kde tento element chybí, jsme předpokládali, v nesouladu se specifikací formátu, že existuje právě jedna knihovna s právě jedním elementem *cell* s právě jedním elementem *contents*. A právě tento element definuje naší součástku, její strukturní architekturu pak definuje příslušné *view*.

6. Ostatní formáty

V této kapitole ještě zmíníme další formáty, se kterými umí naše 1:1 spolupracovat. Formát CIR je implementován jako pouze výstupní a až na malé odchylky je podobně stručný a přehledný jako formát BENCH. Je potřeba jako vstupní formát do MILEFu, viz. [3]. PLA je naopak vstupní formát a umožňuje jednotlivé součástky zadávat pomocí speciálních matic, které budou podrobně rozebrány. Implementovat výstup do PLA obecného obvodu je velmi složité. BLIF je vstupní i výstupní formát, který definuje propojení jednoho z typů součástek PLA a D-klopných obvodů. Výstup do něj je možný pouze pro obvod splňující specifikaci standardního BENCHe.

6.1 CIR

Vysvětlíme si konstrukci formátu CIR, takového, jaký je generován naším ukázkovým programem a vytvořeným překladačem z formátu BENCH (viz. kapitola o konvertorech). Gramatika formátu CIR není k dispozici hlavně proto, že z něj nebylo potřeba načítat žádné obvody. V příloze najdeme dva navzájem podobné příklady obvodu v tomto formátu.
Jako první je klíčové slovo *Cicruit* následované jménem obvodu, který definujeme, a zakončené středníkem.

Jako druhé je klíčové slovo *Node* následované na dalším řádku seznamem jmen vodičů, které se v obvodu vyskytují, oddělených mezi sebou čárkou a mezerou. Seznam je zakončen řetězcem *“ : bit;“*, což specifikuje, že se jedná o spoje nesoucí pouze dvouhodnotovou informaci. Zmiňuji to také proto, že s jinými typy spojů pracovat neumíme.

Třetí sekce začíná klíčovým slovem C*on* následovaný řetězcem *“ delay:10”*. Údaj specifikuje zpoždění součástek, které ovšem nemáme v naší vnitřní struktuře k dispozici. Na dalších řádcích je seznam součástek našeho zařízení. Podporována jsou všechna jako v základním formátu BENCH. Na každém řádku je vždy jedna. Po jejím názvu následuje *“ : “* a typ součástky. Uváděn je s velkým počátečním písmenem následovaným podtržítkem a číslicí udávající počet vstupů a celý řádek je zakončen středníkem. Použité názvy jsou shodné s těmi uloženými ve struktuře v položce *Object::eval_param*, až na hradlo NOT, kde je jeho název nahrazen řetězcem *“Inverter”*, hradlo BUFF, nahrazeno řetězcem *“Driver”* a DFF, nahrazeno řetězcem *“Dflipflop”*. V těchto výjimečných případech se podtržíko a počet vstupů neuvádí.

Čtvrtá sekce definuje vstupní porty, resp. definuje vodiče, které jsou do nich zapojeny. Po klíčovém slovu *Input* je na dalším řádku seznam jmen takových vodičů, oddělených mezi sebou čárkou a mezerou. Řádka je zakončena středníkem.

Pátá sekce je stejná jako čtvrtá až na to, že definuje výstupní porty. Úvodní klíčové slovo je *Output*.

Šestá sekce začíná klíčovým slovem *Strobe* následované na dalším řádku řetězcem "*STROBEALL :* " a na posledním řádku je seznam vodičů vedoucích na výstupní porty jako v páté sekci.

Konečně poslední sekce začíná klíčovým slovem *begin* a končí klíčovým slovem *end* a tečkou. Mezi nimi jsou na samostatných řádcích přiřazovací příkazy v následujícím tvaru:

A(b, c; d);

Takový příkaz říká, že na vstupní porty součástky *A* jsou přivedeny vodiče *b a c*(odděleny čárkou) a za středníkem je uveden vodič připojený na výstupní port. Počet vstupních vodičů nutně musí souhlasit s typem součástky (třetí sekce).

6.2 PLA

Začneme jednoduchým příkladem obvodu zadaného PLA maticí.

```
.model simpledevice
.i 7
.o 2
.ilb f b c d a h g
.ob f0 f1
.p 9
.type fd
-1--1-- 10
1-11--- 10
-001--- 10
01---1- 10
-0--0-- 01
1---0-- 01
0-----0 01
01--1-- 01
10-0--- 01
.e
```

Vidíme, že jednotlivé sekce této specifikace začínají tečkou a jsou umístěny každá na svém řádku. Komentáře, pokud se vyskytují, začínají znakem '#' a končí koncem řádku. Obecně v PLA specifikaci rozeznáváme tyto sekce:

.N definice názvu obvodu
.model definice názvu obvodu
.type definice typu PLA matice

.I specifikace názvu jednoho vstupního portu
.O specifikace názvu jednoho výstupního portu
.ilb pojmenování všech vstupních portů
.ob pojmenování všech výstupních portů
.i konstatování počtu vstupních portů
.o konstatování počtu výstupních portů
.p konstatování počtu řádek matice
.e ukončení specifikace

Zde jsou ještě příklady sekcí, které neobsahuje standard PLA a ani náš příklad. Tyto sekce jsou tedy rozšířením, které je vynuceno existencí nestandardních PLA souborů. Příklady takových jsou na CD.

```
.N new_alu4_comb
.I na="A3"
.I na="A2"
.O na="P0"
.O na="F0"
```

Sekce .N není povinná, pokud ale takový obvod přidáváme jako součástku do jiného obvodu, je nutné jméno specifikovat, viz. kapitola o 1:1 struktuře.

Sekce .type má implicitní hodnotu *fd*. Matice mohou být buď ve formátu *fd* nebo ve formátu *fr*. Formát *f* je podmnožinou formátu *fd*.

Sekce pro specifikaci jmen portů jsou taktéž nepovinné. Pokud ale použijeme např. sekci .ilb, nemůžeme už použít sekci .I a naopak. Totéž platí i pro sekce .ob a .O. Pokud je neuvedeme, předpokládá se použití sekcí .i a .o, přičemž názvy portů je třeba zvolit.

Ostatní řádky jsou obsah PLA matice. Matice specifikuje hodnoty na výstupních portech v závislosti na hodnotách na vstupních portech. Má dva sloupce. Každý řádek prvního sloupce definuje, zda se se na výstupu uplatní příslušný řádek druhého. Jinými slovy testuje určitým způsobem shodu vstupů s hodnotami prvního sloupce. V matici rozeznáváme kromě mezery, která odděluje sloupce, tyto symboly: 0,1,2,-,~. Symboly ‘2’ a ‘-‘ mají význam neurčené hodnoty (don’t care). Symbol ‘~‘ znamená totéž jako ‘0’. Nuly a jedničky odpovídají při při simulaci (viz. kapitola 13) hodnotám log. 0, resp. log. 1. Objasníme tedy, jak se PLA vyhodnocuje.

6.2.1 Typ fd

U tohoto typu je defaultní hodnota každého výstupu ‘0’. To znamená, že na výstupu budou samé 0, pokud nebude nalezena žádná shoda. Tuto hodnotu tedy předvyplníme pro všechny výstupy.

Začněme nejdříve tím, jak se tedy hledá shoda konkrétního řádku levého sloupce PLA matice s hodnotami na vstupních portech. Nejprve zjistíme hodnoty vstupů. Použijeme přitom seznam vstupních portů (*Object::InputPorts)*. Dále procházíme odpředu jednotlivé řádky matice a porovnáváme je se zjištěnými vstupními hodnotami. Shoda je nalezena tehdy, když vstupní jedničky a nuly přesně odpovídají korespondujícím symbolům v daném řádku matice a nebo je v matici na této pozici znak '-'. V takovém případě upravíme hodnoty výstupů následujícím způsobem:

Pokud je v pravém sloupci matice na nějaké pozici znak '-' nebo znak '2', na výstup na této pozici přiřadíme '2'. Pokud je v pravém sloupci matice na nějaké pozici znak '1' a na výstupu na této pozici není znak '2', přiřadíme na tuto pozici znak '1'.
Připomeňme,že výstupní porty ve spávném pořadí máme v datové položce *Object::OutputPorts*.

6.2.2 Typ fr

U tohoto typu je defaultní hodnota každého výstupu '2'. To znamená, že na výstupu budou samé '2', pokud nebude nalezena žádná shoda.

Způsob nalezení shody se od předchozího typu neliší, rozdíl je jen v tom, co se v takovém případě děje s výstupem.

Pokud je v pravém sloupci matice na nějaké pozici znak '0' a na výstupu na této pozici není znak '1', přiřadíme na tuto pozici znak '0'. Pokud je v pravém sloupci matice na nějaké pozici znak '1' a na výstupu na této pozici není znak '0', přiřadíme na tuto pozici znak '1'.

6.2.3 Zvolená implementace

V kapitole věnované 1:1 struktuře jsme již zmínili, že nosičem hodnot při vyhodnocování jsou sami vodiče. Zbývá nám tedy dodat, že tomu je tak i v případě PLA součástek. Abychom mohli během simulace složitějších obvodů použít zvolený simulační algoritmus, viz. kapitola 13, bylo nutné při konstrukci PLA obvodu na každý port připojit známými metodami jeden vodič, který slouží právě a jen k uchování a výměně informací o své aktuální hodnotě.

6.3 BLIF

Pokud známe vše předchozí, popsat tento formát bude poměrně jednoduché. Neobsahuje totiž žádné nové typy součástek, ale právě naopak, skládá se pouze ze dvou nám již známých typů. Jedná se o D-klopné obvody a součástky PLA typu fd. Uveďme tedy jednoduchý příklad sekvenčního obvodu zadaného formátem BLIF (s27 [5]).

```
.model s27.bench
.inputs G0 G1 G2 G3
.outputs G17
```

```
.wire_load_slope 0.00
.latch     G10 G5  0
.latch     G11 G6  0
.latch     G13 G7  0
.names G11 G17
0 1
.names G14 G11 G10
00 1
.names G5 G9 G11
00 1
.names G2 G12 G13
00 1
.names G0 G14
0 1
.names G14 G6 G8
11 1
.names G1 G7 G12
00 1
.names G12 G8 G15
1- 1
-1 1
.names G3 G8 G16
1- 1
-1 1
.names G16 G15 G9
0- 1
-0 1
.end
```

A nyní si popišme jednotlivé části. Zprvu zmíním, že komentáře, pokud se vyskytují, začínají znakem '#' a končí koncem řádku. Sekce .model specifikuje jméno obvodu (podobně jako .N u PLA). Sekce .inputs a .outputs uvádějí, které vodiče jsou připojeny na vstupní a výstupní porty (podobně jako .ilb a .ob u PLA). Sekce .latch definuje přítomnost DKO, kde první identifikátor označuje vodič připojený na vstup D a druhý označuje vodič připojený na výstup.

Dále už jen následují sekce .names, kde každá definuje přítomnost jedné součástky PLA typu fd. Identifikátorů vodičů, které následují, je vždy stejný počet jako sloupců matice PLA, která je uvedena vždy na dalších řádcích. Vodiče jsou uvedeny za sebou v pořadí tak, že

každý koresponduje se svým sloupcem v PLA matici. Specifikace BLIFu připouští pouze jeden výstupní port každé součástky definované sekcí .names.
Definice formátu končí sekcí .end.

7. Simulátor

Simulátor je vhodný prostředek k tomu zjistit, zda je vnitřní struktura dobře navržená. Jedná se o ověření funkční, tedy z trochu jiné strany než to, že algoritmy typické danému problému jsou s touto strukturou snadno implementovatelné. Implementoval jsem tedy jednoduchý krokový simulátor načteného obvodu. Pod pojmem krokový si představuji jednorázové vyhodnocení hodnot na vodičích přivedených na vstupní porty obvodu a promítnutí výsledných hodnot na vodiče připojené na výstupní porty. Důležité bude, aby dvě specifikace téhož obvodu, u kterých máme jiným způsobem zaručeno, že jsou funkčně naprosto stejné, byly také stejně odsimulovány. Abychom tedy nenašli žádné rozdíly ve výstupních hodnotách pro tytéž vstupní hodnoty. Takových obvodů máme připraveno více. Nejjednodušší z nich je již jednou zmíněný obvod C17, kde máme k dispozici mimo jiné jak jeho specifikaci formátem BENCH pomocí hradel NAND, tak jeho specifikaci formátem EDIF, která obsahuje čtyřvstupové součástky LUT.

Pokusím se nyní popsat, jak naše implementace simulátoru funguje a navrhnout jeho vylepšení pro jeho novou verzi.

7.1 Implementace simulace kombinačních obvodů

Pro začátek mějme obvod nezapojený do obvodu, tedy takový, jak ho dostaneme načtením některým výše zmíněným způsobem. Už víme, že kontejnery na hodnoty, které se ve vodičích obvodu mění během simulačního kroku jsou sami instance třídy *Wire,* jmenovitě se jedná o položku *value*. S nezapojeným obvodem tedy především musíme udělat to, že hodnoty vstupů nakopírujeme na vodiče připojené do vstupních portů. Logicky posledním krokem bude tedy vykopírování hodnot z vodičů, které vedou do výstupních portů. Mezitím proběhne postupné vyhodnocování jednotlivých podobjektů obvodu. Postupné proto, na rozdíl od reálného obvodu, že máme k dispozici pouze jeden procesor. Vzniká tedy otázka, v jakém pořadí podobjekty vyhodnocovat. Na samotné vyhodnocení klademe požadavek, aby bylo stejně implementováno jako vyhodnocení nadobjektu i jakéhokoliv jiného podobjektu. Tedy abychom naší rekurzivní strukturu mohli vyhodnotit jednoduchým rekurzivním algoritmem. Z toho už plyne, že počáteční i konečné výše popsané kopírovaní hodnot se děje ještě před (resp. po) samotným zavoláním vyhodnocovací metody.

Metoda, kterou voláme se jmenuje *Object::eval()* bez parametrů. Důvod proč vstupní parametry nejsou hodnoty na vstupních portech a výstupní paramentry nejsou hodnoty na výstupních portech je ten, že nevíme odkud je tato volána a docházelo by tak ke zbytečnému kopírování hodnot, které by vyhodnocovací proces ještě více zpomalilo. On je už tak pomalý vzhledem k použitému poněkud naivnímu ale intuitivnímu algoritmu. To je zároveň jeden z důvodů, proč v 1:1 struktuře, která není to nejdůležitější v této práci, jsou všechny datové položky veřejné (public). Jak jsme již zmínili v předchozích kapitolách, to jediné, co v 1:1

struktuře určuje celkové chování obvodu, je jedna datová položka, *eval_param*. Implementovány tedy byly další metody, které vyhodnocují součástky s dohodnutým chováním. Jmenovitě jde o *Object::evalGate* a *Object::evalPLA*. První z nich vyhodnocuje hradla a má také schopnost rozpoznat, že se o hradlo nejedná. Druhá z nich vyhodnocuje součástky typu PLA, a to nejpoužívanější typy 'fd', 'f' a 'fr'. U součástek s vnitřní stukturou zavoláme metodu *getOrdinalNumbers*, která zjistí pořadí vyhodnocování jednotlivých podobjektů. Naplní položku *Object::SortedObjects*, což je seznam ukazatelů na jejich instance, setříděnými podobjekty podle pořadí vyhodnocování. Pro zjištění pořadí používá intuitivní algoritmus, který postupuje od vstupních portů po vodičích a zvyšuje hodnotu datové položky *Object::ordinal_number* o jedničku podle hodnoty v obvodu předřazeného objektu. Pokud jsou v obvodu D-klopné obvody (připomeňme, že např. formát BENCH je značí DFF), je situace složitější. V takovém případě každý D-klopný obvod se vyhodnocuje mezi prvními v pořadí a ve vyhodnocovacím algoritmu 'staví zeď' pro další zvyšování *ordinal_number*. To bude popsáno v další kapitole.Na všechny prvky v setříděném seznamu zavoláme *evalGate()*. Tato metoda, jak jsme již napsali, umí zjistit, zda se jedná o hradlo, a pokud ano, umí ho jednoduchým algoritmem pro všechna známá hradla vyhodnotit. Přečte si hodnoty z vodičů vedoucích do vstupních portů a nastaví hodnoty vodičů vedoucích z výstupních portů. Umí vyhodnotit i součástky LUT. Na objekty, které nejsou hradly(a ani jinými jednoduchými součástkami) znovu zavolá metodu *eval*.

7.2 Implementace simulace sekvenčních obvodů

Až dosud to bylo jednoduché. Nyní ale zbývá objasnit, jak je to se simulací sekvenčních obvodů. Na úvod uveďme, že umíme simulovat pouze synchronní obvody a to takové, které mohou vzniknout načtením z formátu BENCH. Načtením formátu EDIF může vzniknout jakýkoliv obvod, naštěstí v podstatě všechny soubory, které máme k dispozici v tomto formátu, jdou převést do formátu BENCH a tedy hlavně proto jdou simulovat. Pokud ovšem z EDIFu načtu obvod, který obsahuje např. D-klopné obvody se vstupy do CLK, a z vnějšího obvodu do něj nevede vodič přímo, ale přes nějakou logiku, pak takový obvod nepůjde uložit ve formátu BENCH a simulovat v tomto případě nepůjde proto, že nebude možné vyhodnotit chování D-klopných obvodů (stanovit hodnotu výstupu v časový okamžik viz. výše).

Již jsem uvedl, že jako první v obvodu budeme vyhodnocovat mimo jiné D-klopné obvody. V tuto chvíli poznamenám, že metoda, jak si pamatovat vnitřní stav D-klopného obvodu možná v této verzi simulační metody nebyla zvolena úplně vhodně a různí analytici a implementátoři na ni mohou mít různé názory. Nicméně pro naše obvody stoprocentně správně funguje. Celá věc bude jasná, když si uvědomíme, v jakém okamžiku zastavujeme náš krokový simulátor. Tento okamžik jsem zvolil jako dobu těsně před tím, než přijde

hodinový puls. V takovém okamžiku se totiž s jistotou vnitřní stav projevuje na výstupu DKO. V okamžiku, kdy přijde hodinový puls, se přečte stav vodiče připojeného na vstup DKO a za nějakou dobu (menší než perioda) se tato přečtená hodnota stane vnitřním stavem DKO. To pro implementaci znamená, že až do doby, než přijde na řadu vyhodnocení nějakého DKO, musíme tuto hodnotu pro toto vyhodnocení uchovat. Jinými slovy musíme zazálohovat minulý stav vodiče, který vede z výstupního portu součástky s nejvyšší prioritou pro vyhodnocování. Je to ta s *ordinal_number* rovné nule. Je jasné, že vyhodnocování jiné součástky, s nižší prioritou, tuto hodnotu nemůže přemazat. Zálohu provádíme do položky *Wire::OldValue*. Zároveň nastavujeme příznak *Wire::BackedUp*, kterým říkáme, ze které položky máme při vyhodnocování DKO vzít vstupní hodnotu. Vše ostatní je stejné jako v případě simulace kombinačních obvodů.

7.3 Implementace nastavení vnitřního stavu DKO

Pro účely snadné analýzy chování sekvenčního obvodu jsem implementoval možnost nastavit vnitřní stav DKO kdykoliv, kdy je simulátor zastaven. Myšlenka je taková, že v okamžik, který jsem popsal výše, změníme vnitřní stav DKO. V ten okamžik simulátor stále stojí, tzn., že vnitřní stav se v tomto časovém okamžiku nerovná výstupu DKO. Když simulaci spustíme dále, vlastně v době, kdy by za jiných okolností měla být ze vstupu kopírována nová hodnota, se toto neděje, ale na výstupu se projeví nastavený stav. Nastavování vnitřního stavu tedy probíhá synchronně a nový stav se projeví až při dalším vyhodnocovacím kroku.

K tomu byla použita datová položka *Object::AdditionalParameter1*, který když obsahuje ukazatel na vytvořený *pair<bool,int>* operátorem *new*, nekopíruje v kritickém okamžiku hodnoty ze vstupu DKO, ale v případě, že první položka páru je *true*, zpropaguje na výstup druhou položku páru a první zase shodí.

8. Konvertory

Ve chvíli, kdy máme implementaci načtení výše zmíněných formátů, můžeme snadno vytvořit program, který po jejich načtení provede uložení do některého z výstupních formátů. Ukázkový program, dostupný na přiloženém CD, umí ručně udělat totéž. Jednoduše se mu zadá jméno vstupního souboru a stiskne tlačítko ‘načíst’. V tom okamžiku program řekne, že načtení bylo provedeno (nebo také ne) a čeká se na další pokyn, co s vnitřní strukturou provést. Kromě možnosti simulace máme tedy i možnost do příslušného políčka zadat jméno souboru a stisknout některé jiné tlačítko, což provede uložení do vybraného výstupního formátu. Tato funkčnost byla mnohokrát otestována a vcelku sní nebyly žádné větší problémy. Pro snazší použití byly na fakultě nejčastěji používané konvertory jednoho formátu do druhého navíc přeloženy do podoby konzolové aplikace. Ta samozřejmě nedělá nic jiného, než to, co se skrývalo pod stisky jednotlivých tlačítek v ukázkovém programu. Intuitivně program má dva parametry, první z nich je jméno zdrojového souboru, druhý z nich je jméno cílového souboru. Programy se jmenují ‘Bench2Cir‘ a ‘Edif2Bench‘ a jak plyne z jejich názvu převádějí formát BENCH do formátu CIR, resp. Formát EDIF do formátu BENCH.

9. Závěr

V rámci diplomové práce se zcela podařilo navrhnout datové struktury a ověřit správnost jejich návrhu. Bylo implementováno načítání množství formátů do těchto struktur a ukládání načtených zařízení do výstupních formátů. U jedné z nich byl dokonce implementován simulátor načteného zařízení. Celkem tedy nějakým způsobem umíme pracovat s formáty BENCH, EDIF, PLA, BLIF a CIR. Podařilo se zpracovat i nejsložitější obvody načtené z EDIFu a to takovým způsobem, že náš program rozumí chování všech jeho komponent.

Při ověřování správnosti návrhu nebyly nalezeny žádné nedostatky, které by např. mohly ohrozit použití navržených struktur jako základ pro budované EDA systémy. Naopak struktury se ukázaly jako velmi vhodné pro tento účel.

Byl detailně programátorsky popsán postup implentace načtení dalšího libovolného formátu, takže pro kohokoliv nebude sebemenší problém tyto struktury použít ve svých programech.

Formátů, které se dnes na fakultě používají, je více. Ačkoliv nebyly zpracovány úplně všechny, není nám známo žádné omezení navržených struktur nebo nekompatibilní funkce, které by v budoucnu jejich načtení znemožnilo nebo znesnadnilo.

Naproti tomu je vhodné, aby implementovaný simulátor v budoucnu doznal změn. Jedná se totiž o jakousi první naivní verzi, která pro velmi komplikované a složité obvody nebude stačit. Vyhodnocení jednoho kroku nám známých složitějších obvodů trvá řádově sekundy. Odhaduji, že se zřejmě bude jednat o inteligentní vyhodnocování pouze těch komponent, u nichž se hodnoty na vodičích, které vedou do jejich vstupů, oproti předchozímu kroku změnily.

Program byl implementován vývojovým prostředím Borland C++ Builder 5.0 tak , aby nebyly využívány specifika zabudovaného překladače firmy Borland, což nesporně umožní snadné přenesení zdrojových kódů na jiné platformy. Ukázkový program je Win32 okenní aplikace, převaděče jsou pak aplikace ovládané z příkazové řádky. Oboje se dnes zdá jako nejpoužívanější standard.

10. Seznam použité literatury

[1] Doc. Ing. Karel Müller, CSc., PROGRAMOVACÍ JAZYKY, ČVUT Praha, 2002

[2] EDIF Reference Manual Version 2.0.0. Washington, DC: Electronic Industries Association. ISBN 0-7908-0000-4

[3] U. Gläser and H. T. Vierhaus: MILEF: An Efficient Approach to Mixed Level Test Generation, Proc. IEEE EURO-DAC, Hamburg, September 1992

[4] F. Brglez, H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortan". Proc. of International Symposium on Circuits and Systems, pp. 663-698, 1985

[5] F. Brglez, D. Bryan, K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits". Proc. of International Symposium of Circuits and Systems, pp. 1929-1934, 1989

[6] http://www.edif.org/documentation/BNF_GRAMMAR/estruct0.d

[7] http://service.felk.cvut.cz/vlsi/prj/Benchmarks

11. Přílohy

11.1 Gramatika formátu BENCH

Start -> Inputs Outputs Assigns
Inputs -> e
Inputs -> kwINPUT (IDENT) Inputs
Outputs -> e
Outputs -> kwOUTPUT (IDENT) Outputs
Assigns -> e
Assigns -> IDENT = Gate Assigns
Gate -> IDENT (Idents)
Idents -> IDENT R_O_Idents
R_O_Idents -> e
R_O_Idents -> , IDENT R_O_Idents

11.2 Gramatika formátu EDIF

S -> (kwEDIF IDENT nameDef edifVersion edifLevel keywordMap a3)
nameDef -> IDENT
nameDef -> rename
nameDef -> name
name -> (kwNAME IDENT)
rename -> (kwRename a1 a2)
edifVersion -> (kwEDIFVERSION INTEGER INTEGER INTEGER)
edifLevel -> (kwEDIFLEVEL INTEGER)
keywordMap -> (kwKEYWORDMAP keywordLevel)
keywordLevel -> (kwKEYWORDLEVEL INTEGER)
status -> (kwSTATUS a4)
written -> (kwWRITTEN timestamp a5)
timestamp -> (kwTIMESTAMP INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER)
author -> (kwAUTHOR STRING)
program -> (kwPROGRAM STRING a28)
dataOrigin -> (kwDATAORIGIN STRING a28)
version -> (kwVERSION STRING)
library -> (kwLIBRARY nameDef edifLevel technology a6)
external -> (kwEXTERNAL nameDef edifLevel technology a6)
technology -> (kwTECHNOLOGY numberDefinition simulationInfo)
numberDefinition -> (kwNUMBERDEFINITION)
simulationInfo -> (kwSIMULATIONINFO a26)
simulationInfo -> e
logicValue -> (kwLOGICVALUE nameDef a27)
booleanMap -> (kwBOOLEANMAP booleanValue)
booleanValue -> (BOOLEAN)

cell -> (kwCELL nameDef cellType a8)
cellType -> (kwCELLTYPE a7)
view -> (kwVIEW nameDef viewType interface a13)
viewType -> (kwVIEWTYPE a9)
interface -> (kwINTERFACE a10)
port -> (kwPORT PortNameDef a12)
portNameDef -> nameDef
portNameDef -> array
array -> (kwARRAY nameDef INTEGER a11)
contents -> (kwCONTENTS a14)
instance -> (kwINSTANCE instanceNameDef a15 a16)
instanceNameDef -> nameDef
instanceNameDef -> array
viewRef -> (kwVIEWREF nameRef cellRef)
nameRef -> IDENT
nameRef -> name
cellRef -> (kwCELLREF nameRef a17)
libraryRef -> (kwLIBRARYREF nameRef)
net -> (kwNET netNameDef joined a20)
netNameDef -> nameDef
netNameDef -> array
joined -> (kwJOINED a18)
portRef -> (kwPORTREF aNameRef a19)
aNameRef -> nameRef
design -> (kwDESIGN nameDef cellRef a21)
property -> (kwPROPERTY nameDef typedValue a22)
typedValue -> string
string -> (kwSTRING STRING)
owner -> (kwOwner STRING)
direction -> (kwDIRECTION a23)
instanceRef -> (kwINSTANCEREF aNameRef a25)
viewList -> (kwVIEWLIST a24)

a1 -> identifier
a1 -> name
a2 -> stringToken
a3 -> e
a3 -> status a3
a3 -> library a3
a3 -> external a3
a3 -> design a3
a4 -> written
a5 -> e
a5 -> author a5
a5 -> program a5
a5 -> dataOrigin a5
a6 -> e
a6 -> cell a6
a7 -> kwTIE
a7 -> kwRIPPER

a7 -> kwGENERIC
a8 -> e
a8 -> view a8
a9 -> kwNETLIST
a10 -> e
a10 -> port a10
a11 -> e
a11 -> INTEGER a11
a12 -> e
a12 -> direction a12
a13 -> e
a13 -> property a13
a13 -> contents a13
a14 -> e
a14 -> instance a14
a14 -> net a14
a15 -> viewRef
a15 -> viewList
a16 -> e
a16 -> property
a17 -> e
a17 -> libraryRef
a18 -> e
a18 -> portRef
a19 -> e
a19 -> instanceRef
a20 -> e
a21 -> e
a21 -> property a21
a22 -> e
a22 -> owner
a23 -> kwINPUT
a23 -> kwOUTPUT
a23 -> kwINOUT
a24 -> e
a24 -> a15
a25 -> e
a25 -> instanceRef
a25 -> viewRef
a26 -> logicValue a26
a26 -> e
a27 -> booleanMap
a27 -> e
a28 -> version
a28 -> e

11.3 Příklad obvodu C17 z hradel AND a OR ve formátech EDIF, BENCH a CIR

Příklady vznikly načtením původního souboru ve formátu EDIF ukázkovým programem a vyvoláním funkce, které je uloží do ostatních formátů.

11.3.1 EDIF

```
(edif netlist
 (edifVersion 2 0 0)
 (edifLevel 0)
 (keywordMap (keywordLevel 0))
 (status
  (written
   (timeStamp 1993 2 0 16 45 4)
   (author "David Rickel")
   (program "autologic")
  )
 )
 (external PRIMLIB
  (edifLevel 0)
  (technology
   (numberDefinition
   )
   (simulationInfo
    (logicValue H (booleanMap (true)))
    (logicValue L (booleanMap (false)))
   )
  )
  (cell AND2
   (cellType GENERIC)
   (view INTERFACE
    (viewType NETLIST)
    (interface
     (port out (direction output))
     (port in0 (direction input))
     (port in1 (direction input))
    )
   )
  )
  (cell INV
   (cellType GENERIC)
   (view INTERFACE
    (viewType NETLIST)
    (interface
     (port out (direction output))
     (port in (direction input))
    )
   )
  )
 )
 (library USER_LIB
  (edifLevel 0)
  (technology
```

```
  (numberDefinition
  )
  (simulationInfo
    (logicValue H (booleanMap (true)))
    (logicValue L (booleanMap (false)))
  )
)
(cell TOP
  (cellType GENERIC)
  (view NETLIST
    (viewType NETLIST)
    (interface
      (port P_23GAT_9_ (direction output))
      (port P_22GAT_10_ (direction output))
      (port P_7GAT_4_ (direction input))
      (port P_6GAT_3_ (direction input))
      (port P_3GAT_2_ (direction input))
      (port P_2GAT_1_ (direction input))
      (port P_1GAT_0_ (direction input))
    )
    (contents
      (instance G_G0
        (viewRef INTERFACE (cellRef INV (libraryRef PRIMLIB))))
      (instance G_G1
        (viewRef INTERFACE (cellRef INV (libraryRef PRIMLIB))))
      (instance G_G2
        (viewRef INTERFACE (cellRef INV (libraryRef PRIMLIB))))
      (instance G_G3
        (viewRef INTERFACE (cellRef INV (libraryRef PRIMLIB))))
      (instance G_G4
        (viewRef INTERFACE (cellRef INV (libraryRef PRIMLIB))))
      (instance G_G5
        (viewRef INTERFACE (cellRef INV (libraryRef PRIMLIB))))
      (instance G_G6
        (viewRef INTERFACE (cellRef AND2 (libraryRef PRIMLIB))))
      (instance G_G7
        (viewRef INTERFACE (cellRef AND2 (libraryRef PRIMLIB))))
      (instance G_G8
        (viewRef INTERFACE (cellRef AND2 (libraryRef PRIMLIB))))
      (instance G_G9
        (viewRef INTERFACE (cellRef AND2 (libraryRef PRIMLIB))))
      (instance G_G10
        (viewRef INTERFACE (cellRef AND2 (libraryRef PRIMLIB))))
      (instance G_G11
        (viewRef INTERFACE (cellRef AND2 (libraryRef PRIMLIB))))
      (net N_N0
        (joined
          (portRef in (instanceRef G_G5))
          (portRef out (instanceRef G_G6))
        )
```

```
)
(net N_N1
  (joined
    (portRef in (instanceRef G_G4))
    (portRef out (instanceRef G_G7))
  )
)
(net N_N2
  (joined
    (portRef in (instanceRef G_G3))
    (portRef out (instanceRef G_G8))
  )
)
(net N_N6
  (joined
    (portRef out (instanceRef G_G3))
    (portRef in1 (instanceRef G_G6))
    (portRef in0 (instanceRef G_G7))
  )
)
(net N_N3
  (joined
    (portRef in (instanceRef G_G2))
    (portRef out (instanceRef G_G9))
  )
)
(net N_N7
  (joined
    (portRef out (instanceRef G_G2))
    (portRef in1 (instanceRef G_G7))
  )
)
(net N_N4
  (joined
    (portRef in (instanceRef G_G1))
    (portRef out (instanceRef G_G10))
  )
)
(net N_N8
  (joined
    (portRef out (instanceRef G_G1))
    (portRef in0 (instanceRef G_G6))
  )
)
(net N_N5
  (joined
    (portRef in (instanceRef G_G0))
    (portRef out (instanceRef G_G11))
  )
)
```

```
(net N_N9
  (joined
    (portRef out (instanceRef G_G0))
    (portRef in1 (instanceRef G_G8))
    (portRef in0 (instanceRef G_G9))
  )
)
(net N_N10
  (joined
    (portRef out (instanceRef G_G4))
    (portRef P_23GAT_9_)
  )
)
(net N_N11
  (joined
    (portRef out (instanceRef G_G5))
    (portRef P_22GAT_10_)
  )
)
(net N_N12
  (joined
    (portRef in1 (instanceRef G_G9))
    (portRef P_7GAT_4_)
  )
)
(net N_N13
  (joined
    (portRef in1 (instanceRef G_G11))
    (portRef P_6GAT_3_)
  )
)
(net N_N14
  (joined
    (portRef in1 (instanceRef G_G10))
    (portRef in0 (instanceRef G_G11))
    (portRef P_3GAT_2_)
  )
)
(net N_N15
  (joined
    (portRef in0 (instanceRef G_G8))
    (portRef P_2GAT_1_)
  )
)
(net N_N16
  (joined
    (portRef in0 (instanceRef G_G10))
    (portRef P_1GAT_0_)
  )
)
```

```
        )
      )
    )
  )
)
```

11.3.2 BENCH

```
INPUT(P_7GAT_4_)
INPUT(P_6GAT_3_)
INPUT(P_3GAT_2_)
INPUT(P_2GAT_1_)
INPUT(P_1GAT_0_)
OUTPUT(P_23GAT_9_)
OUTPUT(P_22GAT_10_)
N_N0 = AND(N_N8,N_N6)
N_N1 = AND(N_N6,N_N7)
N_N2 = AND(P_2GAT_1_,N_N9)
N_N3 = AND(N_N9,P_7GAT_4_)
N_N4 = AND(P_1GAT_0_,P_3GAT_2_)
N_N5 = AND(P_3GAT_2_,P_6GAT_3_)
N_N6 = NOT(N_N2)
N_N7 = NOT(N_N3)
N_N8 = NOT(N_N4)
N_N9 = NOT(N_N5)
P_22GAT_10_ = NOT(N_N0)
P_23GAT_9_ = NOT(N_N1)
```

11.3.3 CIR

```
Circuit TOP;

Node
  N_N0, N_N1, N_N2, N_N3, N_N4, N_N5, N_N6, N_N7, N_N8, N_N9,
P_1GAT_0_, P_22GAT_10_, P_23GAT_9_, P_2GAT_1_, P_3GAT_2_, P_6GAT_3_,
P_7GAT_4_ : bit;

Con (delay:10)
  G_G0 : Inverter;
  G_G1 : Inverter;
  G_G10 : and_2;
  G_G11 : and_2;
  G_G2 : Inverter;
  G_G3 : Inverter;
  G_G4 : Inverter;
  G_G5 : Inverter;
  G_G6 : and_2;
  G_G7 : and_2;
  G_G8 : and_2;
  G_G9 : and_2;
```

```
Input
  P_7GAT_4_, P_6GAT_3_, P_3GAT_2_, P_2GAT_1_, P_1GAT_0_;

Output
  P_23GAT_9_, P_22GAT_10_;

Strobe
 STROBEALL :
  P_23GAT_9_, P_22GAT_10_;

begin
  G_G0(N_N5; N_N9);
  G_G1(N_N4; N_N8);
  G_G10(P_1GAT_0_, P_3GAT_2_; N_N4);
  G_G11(P_3GAT_2_, P_6GAT_3_; N_N5);
  G_G2(N_N3; N_N7);
  G_G3(N_N2; N_N6);
  G_G4(N_N1; P_23GAT_9_);
  G_G5(N_N0; P_22GAT_10_);
  G_G6(N_N8, N_N6; N_N0);
  G_G7(N_N6, N_N7; N_N1);
  G_G8(P_2GAT_1_, N_N9; N_N2);
  G_G9(N_N9, P_7GAT_4_; N_N3);
end.
```

11.4 Příklad převodu formátu EDIF obsahujícího součástky LUT do formátu BENCH

11.4.1 Obsah vstupního souboru

```
(edif (rename c17 "komb")
 (edifVersion 2 0 0)
 (edifLevel 0)
 (keywordMap (keywordLevel 0))
 (status
  (written
   (timeStamp 2004 3 23 14 24 25)
   (author "Synplicity, Inc.")
   (program "Synplify Pro" (version "7.1, Build 152R"))
  )
 )
 (library VIRTEX
  (edifLevel 0)
  (technology (numberDefinition ))
  (cell LUT4 (cellType GENERIC)
   (view PRIM (viewType NETLIST)
    (interface
     (port I0 (direction INPUT))
     (port I1 (direction INPUT))
     (port I2 (direction INPUT))
     (port I3 (direction INPUT))
     (port O (direction OUTPUT))
    )
   )
  )
 )
 (library work
  (edifLevel 0)
  (technology (numberDefinition ))
  (cell (rename c17 "komb") (cellType GENERIC)
   (view arch (viewType NETLIST)
    (interface
     (port G1gat (direction INPUT)
)
     (port G2gat (direction INPUT)
)
     (port G3gat (direction INPUT)
)
     (port G6gat (direction INPUT)
)
     (port G7gat (direction INPUT)
)
     (port G22gat (direction OUTPUT))
     (port G23gat (direction OUTPUT))
```

```
    )
    (contents
     (instance g23gat_i (viewRef PRIM (cellRef LUT4 (libraryRef VIRTEX)))
      (property init (string "0EEE"))
     )
     (instance g22gat_i (viewRef PRIM (cellRef LUT4 (libraryRef VIRTEX)))
      (property init (string "E2EA"))
     )
     (net (rename g1gat "G1gat") (joined
      (portRef G1gat)
      (portRef I2 (instanceRef g22gat_i))
     ))
     (net (rename g2gat "G2gat") (joined
      (portRef G2gat)
      (portRef I0 (instanceRef g22gat_i))
      (portRef I1 (instanceRef g23gat_i))
     ))
     (net (rename g3gat "G3gat") (joined
      (portRef G3gat)
      (portRef I1 (instanceRef g22gat_i))
      (portRef I3 (instanceRef g23gat_i))
     ))
     (net (rename g6gat "G6gat") (joined
      (portRef G6gat)
      (portRef I3 (instanceRef g22gat_i))
      (portRef I2 (instanceRef g23gat_i))
     ))
     (net (rename g7gat "G7gat") (joined
      (portRef G7gat)
      (portRef I0 (instanceRef g23gat_i))
     ))
     (net (rename g22gat "G22gat") (joined
      (portRef O (instanceRef g22gat_i))
      (portRef G22gat)
     ))
     (net (rename g23gat "G23gat") (joined
      (portRef O (instanceRef g23gat_i))
      (portRef G23gat)
     ))
    )
   )
  )
 )
 (design (rename c17 "komb") (cellRef c17 (libraryRef work))
        (property PART (string "xcv800hq240-4") (owner "Xilinx")))
)
```

11.4.2 Obsah výstupního souboru

```
INPUT(G1gat)
```

```
INPUT(G2gat)
INPUT(G3gat)
INPUT(G6gat)
INPUT(G7gat)
OUTPUT(G22gat)
OUTPUT(G23gat)
G22gat = LUT4_E2EA(G2gat,G3gat,G1gat,G6gat)
G23gat = LUT4_0EEE(G7gat,G2gat,G6gat,G3gat)
```

11.5 Příklad převodu formátu BENCH do formátu CIR

Příklad vzniknul převodem souboru c17.bench programem Bench2Cir.

11.5.1 Obsah vstupního souboru

```
INPUT(G1gat)
INPUT(G2gat)
INPUT(G3gat)
INPUT(G6gat)
INPUT(G7gat)
OUTPUT(G22gat)
OUTPUT(G23gat)

G10gat = nand(G1gat, G3gat)
G11gat = nand(G3gat, G6gat)
G16gat = nand(G2gat, G11gat)
G19gat = nand(G11gat, G7gat)
G22gat = nand(G10gat, G16gat)
G23gat = nand(G16gat, G19gat)
```

11.5.2 Obsah výstupního souboru

```
Circuit c17;

Node
  G10gat, G11gat, G16gat, G19gat, G1gat, G22gat, G23gat, G2gat, G3gat, G6gat,
G7gat : bit;

Con (delay:10)
  Gate_1 : nand_2;
  Gate_2 : nand_2;
  Gate_3 : nand_2;
  Gate_4 : nand_2;
  Gate_5 : nand_2;
  Gate_6 : nand_2;

Input
  G1gat, G2gat, G3gat, G6gat, G7gat;

Output
  G22gat, G23gat;

Strobe
 STROBEALL :
  G22gat, G23gat;

begin
  Gate_1(G1gat, G3gat; G10gat);
  Gate_2(G3gat, G6gat; G11gat);
  Gate_3(G2gat, G11gat; G16gat);
```

```
  Gate_4(G11gat, G7gat; G19gat);
  Gate_5(G10gat, G16gat; G22gat);
  Gate_6(G16gat, G19gat; G23gat);
end.
```

11.6 Popis ovládání ukázkového programu

Ukázkovým programem můžeme docílit všech výše uvedených překladů. Navíc s jeho pomocí můžeme vizualizovat funkci simulátoru. Popíšeme tedy stručně jak se ovládá. Při načítání souborů je nutné dbát všech omezení, uvedených v celém předchozím textu, protože pouze tak lze zaručit jeho funkčnost.

Když tedy spustíme translator.exe z přiloženého CD, vidíme několik skupin ovládacích prvků. Začneme tou skupinou vpravo uprostřed. Jak napovídají názvy tlačítek slouží pro načtení souborů do vnitřních struktur. Do příslušného editačního pole zadáme název souboru a stiskneme tlačítko (např. Load from Edif pro načtení EDIFu). Po načtení už máme několik možností. Pro uložení do některého výstupního formátu použijeme skupinu ovládacích prvků vpravo nahoře. Opět zadáme jméno souboru a stiskneme příslušné tlačítko.

Simulaci zahájíme stiskem tlačítka 'Analyze'. V levé části se objeví editační pole reprezentující vstupní porty, výstupní porty a vnitřní stavy D-klopných obvodů. Zadáme tedy např. požadovanou kombinaci na vstupních portech a pro provedení simulačního kroku stiskneme 'Simulation step'. Výstup se ihned projeví v příslušných editačních polích. Pro jednorázovou změnu vnitřního stavu některého klopného obvodu zadáme příslušnou hodnotu do příslušných polí (jsou to ty úplně dole) a stisneme 'Set internal states'. Změny na výstupu se projeví po provedení dalšího simulačního kroku.

Konečně poslední skupina ovládacích prvků (vpravo dole) slouží pro souborově orientované provádění simulace. Vyrobíme si soubor s požadovanými kombinacemi hodnot na vstupních portech. Pro obvod se třemi vstupy by mohl vypadat např. takto:

```
000
100
010
110
001
101
011
111
```

V editačním poli Zadání můžeme opravit jeho jméno (zad.txt). Můžeme si i zadat jméno souboru s výsledky (pokud bychom chtěli jiné než vys.txt). Pak stiskneme zpracovat.
Soubor se zadáním ve tvaru naší ukázky může být vyroben i automaticky stiskem tlačítka Generovat Zad.Txt. Do editačního pole vedle tohoto tlačítka zadáváme číslo, které specifikuje, kolikrát je toto zadání do souboru nakopírováno.

11.7 CD se zdrojovými kódy