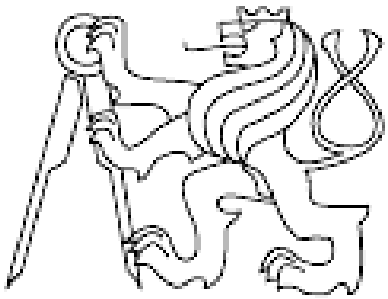
České vysoké učení technické v Praze
Fakulta elektrotechnická



Diplomová práce

Simulace logických obvodů v SystemC

Miroslav Maněna

Vedoucí práce: Ing. Petr Fišer

Studijní program: Informatika a výpočetní technika

leden 2008

Poděkování

Chtěl bych poděkovat všem, kteří mě podporovali během studia, zejména mé rodině a kamarádům. V neposlední řadě pak také ing. Petru Fišerovi za poskytnuté materiály k práci a cenné připomínky ke zkvalitnění práce.

Prohlášení

Prohlašuji, že jsem svou diplomovou 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 České Kamenici dne 18.1.2008 . . . . . .

Abstract

This work deals with integration of the SystemC library into the current EDA (Electronic Design Automation) systém EDuArd as a simulation kernel of this system and with implementation of interface between EDA systém and SystemC. This interface is a brand-new module in the EDA system. Objective of this module is a conversion of structures represented in the kernel of the EDA system to classes of the SystemC library. These classes are used for simulation of logic circuits. Next objective is support of the stuck-at fault simulation. Output of the simulation is generated in VCD format (Value Change Dump) from whom it is possible draw a time chart of the signals.

Anotace

Práce se zabývá začleněním knihovny SystemC do stávajícího EDA (Electronic Design Automation) systému EDuArd jakožto simulačního jádra tohoto systému a implementací rozhraní mezi EDA systémem a SystemC. Jedná se vlastně o nový modul v tomto systému. Modul má za úkol převádět struktury EDA systému do tříd SystemC. Tyto třídy se dál používají pro simulaci logických obvodů. Práce se také zabývá simulací trvalých poruch v obvodech. Výstup simulace je generován ve formátu VCD (Value Change Dump), z kterého je možné zobrazit časové průběhy na signálech.

Obsah

Seznam obrázků

Seznam tabulek

1. Úvod

Cílem práce je vytvoření modulu do vyvíjeného experimentálního systému EDuArd (Electronic Design Automation) pro simulaci elektronických obvodů. Modul má mít za úkol umět simulovat jak funkčnost obvodu (tedy závislost výstupních hodnot na vstupních), tak také simulaci poruch na vodičích. Součástí práce je také spustitelná konzolová aplikace, která provede simulaci a výsledek simulace zapíše do výstupního souboru.
Systém EDuArd je souborem algoritmů, např. pro konverzi mezi různými formáty popisu elektronických obvodů nebo minimalizací boolovských funkcí. Jedná se vlastně o výukový systém pro návrh elektronických obvodů, do kterého v současné době vznikají další moduly, např. pro kreslení elektronických schémat.
Pro simulaci je využita knihovna SystemC implementována v jazyce C++. SystemC je knihovna tříd v jazyce C++ a metologie pro efektivní modelování softwarových algoritmů a hardwarových architektur. SystemC se používá se standartními C++ vývojovými prostředími pro vytvoření modelu, k rychlému otestování správnosti modelu a optimalizaci návrhu pomocí simulace.
Výsledný modul je tedy spíše rozhraním mezi SystemC a jádrem EduArda, tzv.HUBem. Modul vlastně převede reprezentaci obvodu z hubu do reprezentace v SystemC, přivede na vstupy obvodu zadané signály, obvod odsimuluje a zapíše výsledky do výstupních souborů. Ty se pak mohou zobrazovat dalšími nástroji, např. pro zobrazování časových průběhů.
Diplomová práce je zorganizována do několika kapitol. Druhá kapitola se věnuje jádru systému EDuArd, nazývanému HUB. Ten je potřeba znát, neboť je nutné z něho převést reprezentaci obvodů do SystemC. Třetí kapitola stručně pojednává o konstruktech knihovny SystemC, jak se v ní implementují moduly (entity), jak se propojují a jak se řídí simulace. Čtvrtá kapitola popisuje přístupy pro modelování poruch v číslicových obvodech a o možnostech paralelizace pro zvýšení výkonu simulace. V páté kapitole je obsažen postup při návrhu a implementaci samotného simulačního modulu a spustitelné řádkové aplikace pro odsimulování obvodu. V šesté kapitole jsou naměřené hodnoty doby simulace pro některé obvody různých velikostí.

2. Systém EDuArd (EDA)

2.1. Úvod

V této kapitole se budu věnovat pouze částem EDA systému, které jsem využil pro napsání simulačního modulu obvodů. Nejprve je zmíněn pohled na celý systéma poté jsou detailněji popsány části systému, které se využívají pro převod reprezentace obvodu z EDA systému do reprezentace v SystemC.

Jádrem systému je datová základna modelující navrhovaný obvod a zajišťující unifikaci, integritu a konzistenci dat, jinak nazývaná HUB. Nad touto základnou se implementují další moduly pro import a export dat do a ze systému a další výkonné procesy. Výkonné procesy realizují syntetické a analytické kroky návrhu. Dělí se na automatické procesy, které nepotřebují interakci s uživatelem a na interaktivní, které naopak musí s uživatelem komunikovat. Příkladem automatického procesu může být modul realizující simulaci navrhovaného obvodu, interaktivním pak může být modul pro kreslení schémat. Nejvyšší úrovní je uživatelské rozhraní a řízení. Ta se stará o konkrétní průchod pracovním postupem, souběžnou práci více interaktivních procesů a podobně. Datová základna (HUB) a řízení se dohromady nazývá rámec nebo prázdný EDA systém.

obr. 1 – Pohled na systém EDuArd (převzato z dokumentace systému)

2.2. Datová základna (HUB)

Jak bylo řečeno výše, datová základna je jádrem celého systému a modeluje navrhovaný obvod. Její návrh vychází z jazyka VHDL (Very high speed integrated circuit Hardware Description Language). V tomto jazyce se každý popisovaný obvod (model) skládá alespoň

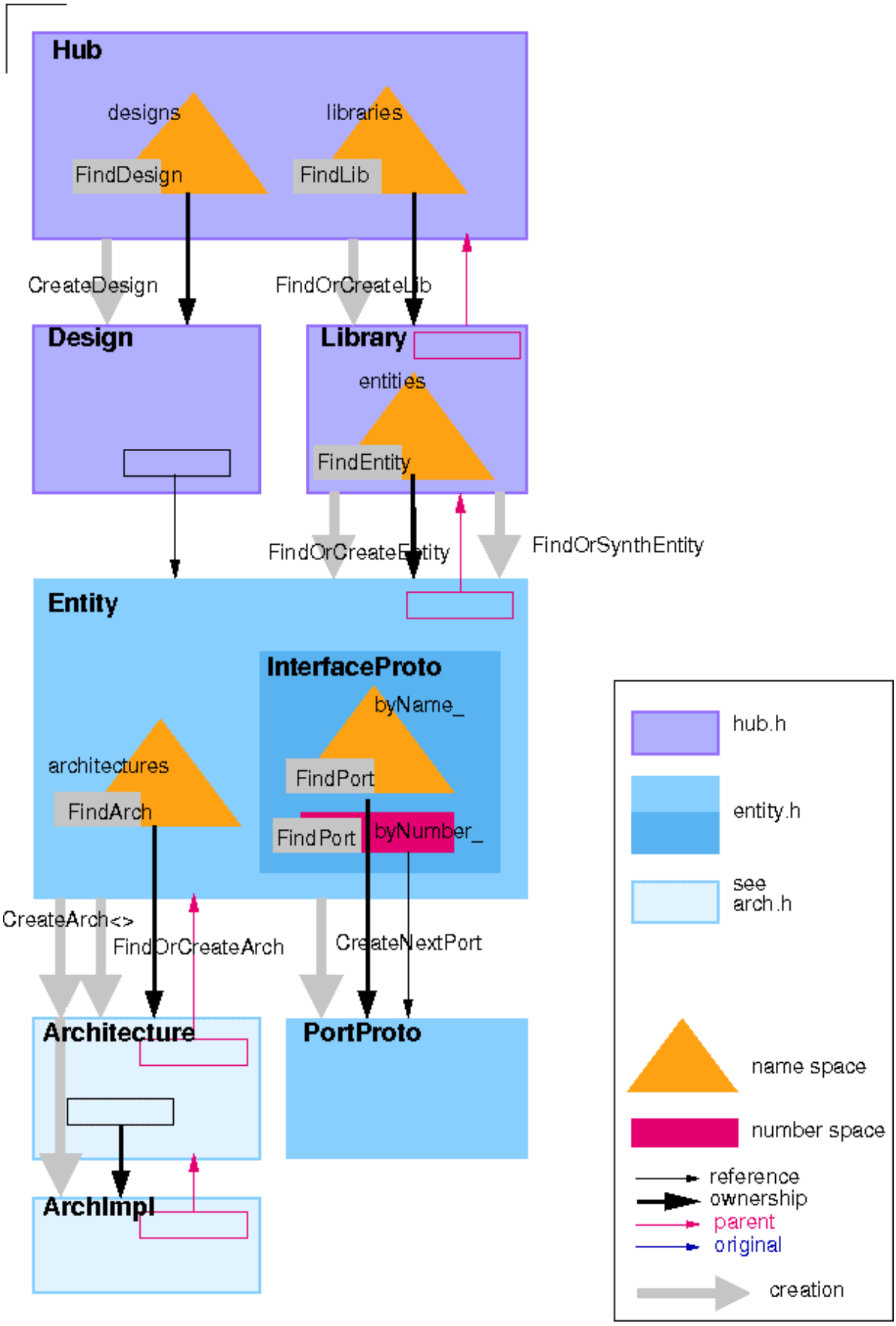
ze dvou samostatných jednotek, entity a architektury. Entita specifikuje hlavně vstupní a výstupní porty modelu (dá se přirovnat k deklaraci funkcí v programovacích jazycích). Pomocí těchto portů je zajištěna komunikace mezi více entitami vzájemně propojených vodiči (u přirovnání k programovacím jazykům se jedná o předávání parametrů funkcím – výstupní hodnoty jedné entity se předají jako vstupní hodnoty jiné entitě). Samotné chování entity se pak definuje pomocí architektury (přirovnáno k definici těla funkce v programovacích jazycích). Entita může mít více architektur, ale pouze jedna se může současně používat. VHDL popisuje dva druhy architektur. Jedná se strukturní popis a behaviorální popis. V prvním případě se jedná o hierarchické vkládání dalších entit do jedné nadřazené entity, zatímco v druhém případě se jedná o sled prováděných instrukcí (např. u čítače o zvyšování nějaké vnitřní proměnné o jedničku). V přirovnání k programovacím jazykům si u strukturního popisu můžeme představit, že chování funkce je popsáno pouze voláním nějakých jiných funkcí, zatímco u behaviorálního popisu se vyskytují např. podmínky, smyčky, aritmetické nebo logické výrazy. Oba druhy architekur jsou dále popsány podrobněji.

HUB se tedy dělí na tři části, které jsou popsány níže:
- část organizující entity (vrchní vrstva)
- část strukturního popisu entity
- část behaviorálního popisu entity (popis chování)

Pro převod reprezentace modelu z HUBu je nutné o každém z těchto částí něco vědět.

2.2.1. Vrchní vrstva

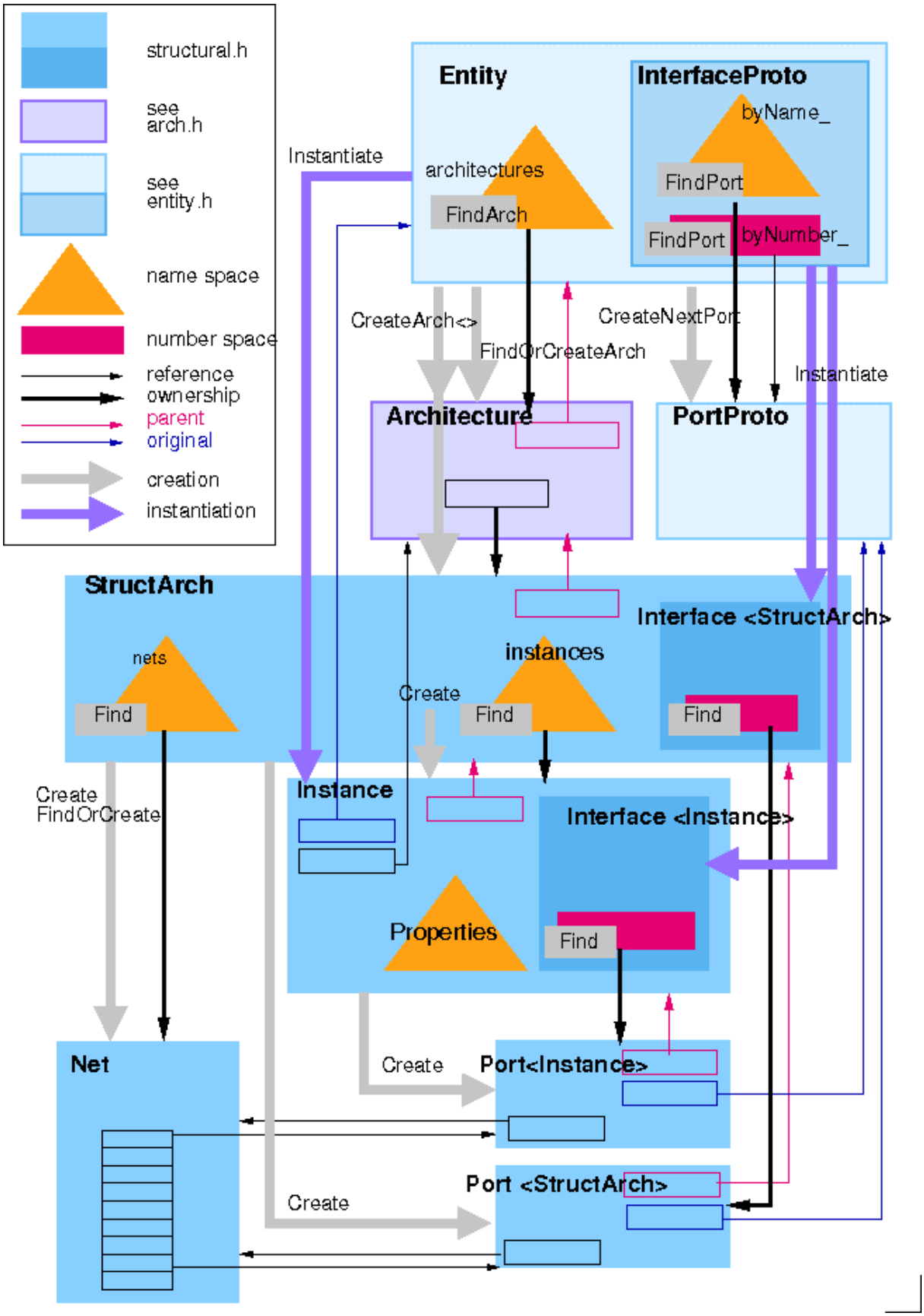
Každý program stavěný nad HUBem musí mít přístup k instanci třídy **`Hub`**. Ta obsahuje dva jmenné prostory (což jsou kontejnery implementované pomocí STL šablony **`Map`**, kde klíčem jsou názvy objektů, hodnotami jsou ukazatele na objekty), jeden pro návrhy (designs) a druhý pro knihovny. Dále poskytuje metody pro vytváření knihoven a návrhů, případně pro jejich hledání. Návrhy jsou implementované pomocí třídy **`Design`**, knihovny třídou **`Library`**.
Návrhem se rozumí nějaký pojmenovaný odkaz na existující entitu. Nemá žádné speciální metody, je to pouze odkaz. Knihovnou je soubor několika pojmenovaných entit, je tedy dalším jmenným prostorem. Obsahuje metody pro vytváření nových entit a jejich vyhledání podle jména. Každá knihovna nebo návrh má v prostoru jmen své unikátní jméno, stejně tak i každá entita v určité knihovně má své unikátní jméno.
Každá entita obsahuje jmenný prostor architektur popisující entitu a popis rozhraní. Implementuje ji třída **`Entity`**. Kontejner pro architektury je zde z důvodu, že každá entita může mít více architektur. Vždy má ale nějakou jednu konkrétní architekturu zvolenou, tedy současnou. Třída obsahuje metody pro vytváření, likvidování a hledání architektur podle jména. Uvnitř třídy **`Entity`** je instance třídy **`InterfaceProto`**, reprezentující rozhraní entity. Má prostor jmen pro porty a navíc má také číselný prostor pro odkazování na porty podle pořadí. Proto je možné porty entity hledat jak podle jména, tak podle pořadí, v jakém byly porty vytvářeny.
Porty jsou reprezentovány třídou **`PortProto`**. Obsahuje atributy portů, jako např. zda je vstupní nebo výstupní.
Architektura je reprezentována třídou **`Architecture`**, a protože je někdy potřeba měnit arhitekturu za provozu, je použit idiom *pimpl* (pointer to implementation). Obsahuje ukazatel na abstraktní třídu **ArchImpl**, kde je obsažena informace o typu popisu. Tento interface pak implementují třídy pro strukturní a behaviorální popis, viz dále.



obr. 2 – Vrchní vrstva HUBu (převzato z dokumentace EDA systému)

2.2.2. Část strukturního popisu entit

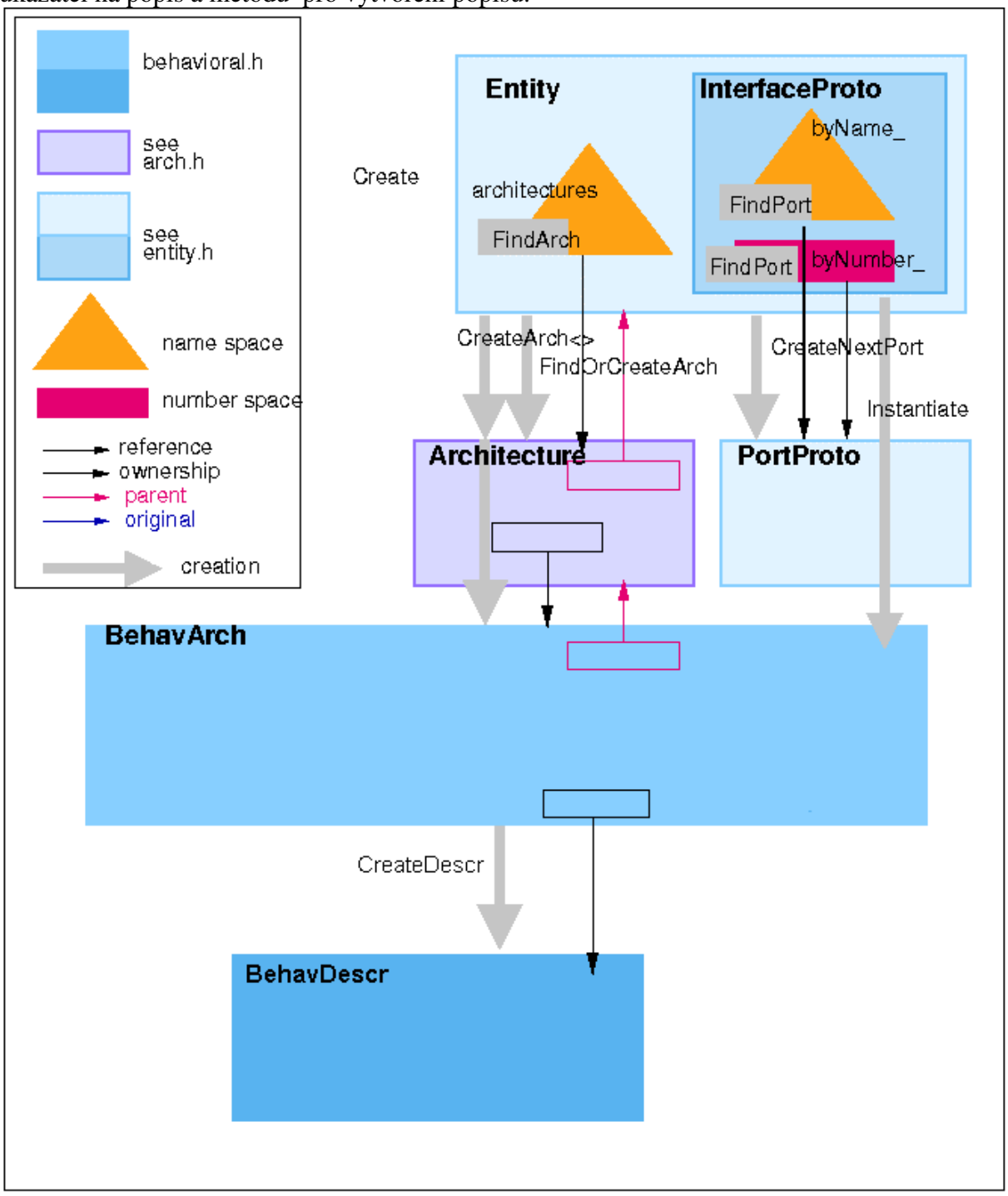
V této části je popsáno, jak vypadá strukturní implementace architektury. Strukturní implementace architektury se skládá z instancí entit a spojů. Spoje propojují porty instancí mezi sebou a také porty rozhraní popisované entity. Na diagramu dole je pokračování vrchní vrstvy HUBu. Z vrchní části jsou vidět třídy **`Entity`**, **`Architecture`**, **`InterfaceProto`** a **`PortProto`**. Abstraktní třídu **`ArchImpl`** zde zastupuje potomek této třídy **`StructArch`**. Ta obsahuje jmenný prostor spojů (nets) a instancí entit (instances), a také rozhraní (instance třídy **`Interface`**), což je číselný prostor portů ekvivalentní číselnému prostoru portů prototypu rozhraní entity s tím rozdílem, že porty nejsou instance třídy **`PortProto`**, ale třídy **`Port`**. Ten je zde pro zajištění propojení portů mezi popisovanou entitou a portů samotných instancí vnitřních entit.
Spoj je reprezentovaný třídou **`Net`**. Má své jméno a obsahuje seznam s odkazy na propojené porty. Porty v tomto případě nejsou instance třídy **`PortProto`**, ale třídy **`Port`**, viz dále.
Třída **`StructArch`** dále poskytuje metody pro vytváření a vyhledávání spojů, vytváření instancí entit a portů.
Každá instance je reprezentovaná třídou **`Instance`**. Stejně jako samotný strukturní popis obsahuje číselný prostor portů pro vzájemné propojování, odkaz na aktuálně konfigurovanou architekturu a jmenný prostor dvojic jméno – hodnota, které zobrazují speciální vlastnosti.
Jak bylo řečeno výše, při propojování instancí mezi sebou a mezi porty entity se používají jiné druhy portů, než jsou v definici rozhraní ve třídě InterfaceProto. Nejedná se tedy o instance třídy **`PortProto`**, ale o instance třídy **`Port`**. To jsou vlastně jen odkazy na své prototypy, na rozdíl od prototypů je zde navíc informace o vlastníkovi (které instanci patří). Protože spoje mohou odkazovat na instance různých tříd, dědí třída **`Port`** od třídy **`Joinable`**.



obr. 3 – Vazby u strukturního popisu entity (převzato z dokumentace EDA systému)

2.2.3. Část behaviorálního popisu entit

Tato část popisuje, jak je v HUBu řešen behaviorální popis entity. Na diagramu dole je opět jako na předchozím diagramu vidět část z vrchní části HUBu, tedy bloky **`Entity`**, **`Architecture`**, **`InterfaceProto`** a **`PortProto`**. Namísto abstraktní třídy **`ArchImpl`**, je zde její potomek, třída **`BehavArch`**. Ta opět podobně jako třída **`Architecture`** využívá idiomu *pimpl*, z toho důvodu, že behaviorálních popisů může být více druhů (např. symetrická funkce nebo popis automatu). **`BehavArch`** obsahuje tedy pouze ukazatel na popis a metodu pro vytvoření popisu.



obr. 4 – Vazby u behaviorálního popisu entity (upraveno z dokumentace EDA systému)

Popisy jsou reprezentované potomky bázové třídy **`BehavDescr`**. Následuje seznam behaviorálnách popisů a jejich krátké představení, které jsou v současné době v HUBu implementovány.

Plně symetrické funkce

Tento popis je implementovaný ve třídě **`SymFunc`**. Reprezentuje plně symetrickou funkci s jedním výstupem. Obsahuje informaci o typu funkce a počtu vstupů. V závislosti na počtu vstupů dělíme funkce do tří kategorií:

- Funkce s žádným vstupem: 0, 1
- Funkce s jedním vstupem: invertor, buffer
- Funkce s dvěma a více vstupy: and, nand, or, nor, xor, xnor

RTL klopný obvod

Tento popis je implementovaný ve třídě RTLReg. Reprezentuje RTL popis klopného obvodu bez hodinového a resetujícího vstupu.

PLA popis

Tento popis je implementovaný ve třídě **`PlaDescr`**. Je založený na principu logické pravdivostní tabulky. Má několik vstupů i výstupů. Podrobný popis viz [2]. Příklad popisu obvodu pomocí formátu PLA je zde:

```
.i 12
.o 8
----1------0 10000000
----0-----0- 10000000
0---------- 10000000
-----1-----0 01000000
-----0----0- 01000000
-0--------- 01000000
------1----0 00100000
------0---0- 00100000
--0-------- 00100000
-------1---0 00010000
-------0--0- 00010000
---0------- 00010000
0---1---0--- 00001000
0---0----0-- 00001000
-0---1--0--- 00000100
-0---0---0-- 00000100
--0---1-0--- 00000010
--0---0--0-- 00000010
---0---10--- 00000001
```

V tomto příkladu obvod obsahuje 12 vstupů a 8 výstupů (to znázorňují první dva řádky). Na dalších řádcích je již samotná pravdivostní tabulka. Každý řádek má dvě části oddělené mezerou. První část je vektor ohodnocení vstupních proměnných v termu, druhou částí je pak ohodnocení výstupních funkcí. V první části „1“ znamená, že se proměnná v termu objevuje v přímé formě, „0“ v negované formě a „-“ (don’t care) znamená, že se proměnná v termu vůbec neobjevuje. Výstupní část pak může mít více významů, který definuje typ PLA. Např. v implicitním typu „fd“ jsou významy hodnot následující: při „1“ bude na výstupu „1“, při „0“ bude na výstupu „0“ a při „-“ je výstup nedefinovaný. Více o typech PLA viz [2].
`PlaDescr` uchovává všechny informace o PLA uvnitř sebe, vstupní a výstupní matice jsou reprezentovány jako seznamy řetězců.

FSM popis (popis konečného automatu)

Tento popis je implementovaný ve třídě **`FsmDescr`**. Je založen na formátu Kiss, viz [2], popisující přechodovou tabulku stavů Mealyho automatu. Ukázka v tomto popisu následuje zde:

```
.i 1
.o 2
.p 14
.s 7
0 START state6 00
0 state2 state5 00
0 state3 state5 00
0 state4 state6 00
0 state5 START 10
0 state6 START 01
0 state7 state5 00
1 state6 state2 01
1 state5 state2 10
1 state4 state6 10
1 state7 state6 10
1 START state4 00
1 state2 state3 00
1 state3 state7 00
```

Tento obvod obsahuje jeden vstup, dva výstupy, sedm stavů a 14 přechodů (tzv. produkty). To je popsáno na prvních čtyrech řádcích. Následující řádky jsou už řádky přechodové tabulky. Na každém řádku je vektor vstupních hodnot, následuje pro aktuální stav, ve kterém se automat nachází, třetí hodnotou řádku je stav, do kterého se při zadaných vstupních hodnotách přejde a poslední hodnotou je vektor výstupních hodnot.
Třída **`FsmDescr`** obsahuje všechny informace z Kiss formátu, navíc dovoluje také popsat Mooreův automat. Tabulka přechodů je implementována už jako graf přechodu.

3. Stručný přehled SystemC

3.1. Úvod

Tato kapitola je stručně věnována konstruktům SystemC, které je potřeba pochopit a které jsem pro vytvoření modulu potřeboval znát. Bližší popis o použití knihovny SystemC lze nalést v [3], případně v [5], kde jsou detailně popsány všechny třídy a šablony knihovny SystemC.

SystemC není formálním jazykem jako je např. VHDL. Je to knihovna tříd a šablon pro jazyk C++, která poskytuje potřebné konstrukty pro hardwarovou simulaci. Poskytuje simulační jádro, s kterým mohou být číslicové obvody simulovány, používaje přitom přístup tzv. událostně založená simulace. Tato metoda je obdobná simulačním mechanizmům využívaných ve většině VHDL prostředích. SystemC definuje čtyři základní konstrukty, které se používají při hardwarovém modelování. Jsou to:

- Modul: Reprezentuje jednu hardwarovou entitu.
- Port: Použitím portů jsou jednotlivé entity propojovány.
- Kanály: Poskytují událostmi řízenou komunikaci.
- Datové typy: Různé datové typy vhodné pro návrh hardwaru (např. bitový vektor)

SystemC byl navržen tak, aby byl vysoce rozšiřitelný, takže všechny třídy, které definuje mohou být rozšířeny a nové mohou být definovány. Příkladem může být kanál, který bude přenášet datový typ Packet, který obsahuje jak hlavičku, tak data.

3.2. Typický popis simulovaného systému v SystemC

Typický program v SystemC vypadá nějak následovně:

```
#include "systemc.h"
#include "entita1.h"
#include "entita2.h"

int sc_main(int argc, char* argv[])
{
      /*
        definice signalu propojujici jednotlive entity
      */

      /*
        definice hodinovych signalu
      */

      /*
        instancovani entit a propojeni jejich portu se signaly
      */

      /*
        simulace
      */

      return 0;
}
```

Toto je tzv. top-level funkce. V ní se vždy instancuje simulovaná entita a spouští simulace, případně se propojuje více entit, pokud jsou na stejné úrovni. Jednotlivé entity jsou popsány v samostatných hlavičkových souborech jako definice tříd dědící vlastnosti nějaké bázové třídy SystemC pro popis modulů (entit).
SystemC má svoji vlastní funkci **`main()`**, která je volána při spuštění programu a vypadá následovně:

```
int main( int argc, char* argv[] )
{
      return sc_core::sc_elab_and_sim( argc, argv );
}
```

Volaná funkce **`sc_elab_and_sim()`** pak zavolá v bloku `try` námi definovanou top-level funkci **`sc_main()`** a ošetřuje případné vyjímky. Její výpis je zde:

```
int sc_elab_and_sim( int argc, char* argv[] )
{
      int status = 0;
      argc_copy = argc;
      argv_copy = new char*[argc];
      for ( int i = 0; i < argc; i++ )
            argv_copy[i] = argv[i];

      try
      {
            pln();

            // Perform initialization here
            sc_in_action = true;

            status = sc_main( argc, argv );

            // Perform cleanup here
            sc_in_action = false;
      }
      catch( const sc_report& x )
      {
            message_function( x.what() );
      }
      catch( const char* s )
      {
            message_function( s );
      }
      catch( ... )
      {
            message_function( "UNKNOWN EXCEPTION OCCURED" );
      }

      delete [] argv_copy;
      return status;
}
```

3.3. Definice modulů (entit)

Při klasické tvorbě modelů v SystemC se jednotlivé entity definují v samostatných modulech. Do hlavičkového souboru .h se napíše definice třídy reprezentující entitu a do .cpp se pak

případně píší definice metod, které třída obsahuje podobně, jako je tomu v klasickém C++. Syntaxe definice modulu v SystemC vypadá následovně:

```
#include "systemc.h"
SC_MODULE (module_name)
{
      // deklarace portu modulu
      // deklarace signalovych promennych
      // deklarace datovych promennych
      // deklarace clenskych funkci
      // deklarace procesu

      // konstruktor modulu
      SC_CTOR (module_name)
      {
            // registrovani procesu
            // deklarace citlivostniho seznamu
      }
};
```

SC_MODULE je makro, které definuje třídu se jménem **module_name**, která dědí vlastnosti od třídy **sc_module**. Tato bázová třída obsahuje základní vlastnosti pro reprezentaci a simulaci entity v SystemC. Makro SC_MODULE je definováno následnovně:

```
#define SC_MODULE(user_module_name)                              \
    struct user_module_name : ::sc_core::sc_module
```

Konstruktor třídy je pak definován pomocí makra **SC_CTOR**. Makro pouze navíc definuje vnitřní typ, odkazující se na právě definovanou třídu.

```
#define SC_CTOR(user_module_name)                                \
    typedef user_module_name SC_CURRENT_USER_MODULE;             \
    user_module_name( ::sc_core::sc_module_name )
```

Moduly se pak instancují klasicky jako v C++, kde parametrem konstruktoru je vlastní název instance:

```
Filter f1("filter1");
```

Poté, co se modul instancoval, je možné propojovat porty se signály, viz. dále.

3.3.1. Definice portů modulu

Každý modul má několik vstupních a výstupních (případně vstupně/výstupních) portů, které definují rozhraní modulu. Vstupně-výstupní porty se definují, pokud chceme z výstupního portu také číst. Porty se propojují se signály, které dovolují mezi jednotlivými moduly komunikovat.Všechny tyto druhy portů se definují pomocí šablon C++ **sc_in**, **sc_out** a **sc_inout**. Parametrem šablony je datový typ, který reprezentuje data přenášená pomocí signálů. Datovým typem může být jakákoli C++ skalární proměnná, složený datový typ (např. reprezentace paketů) nebo některý z typů definovaných SystemC (např. **sc_int<n>**, **sc_bit**, **sc_logic**, …). Pro účely integrace do hubu EduArda je nejvýhodnější používat typ **sc_logic**, reprezentující čtyři logické hodnoty (0 – false, 1 – true, X – neznámé, Z –

stav vysoké impedance). Tyto šablony mají metody a přetížené operátory pro čtení, zápis, propojování se signály a několik dalším méně důležitých metod.

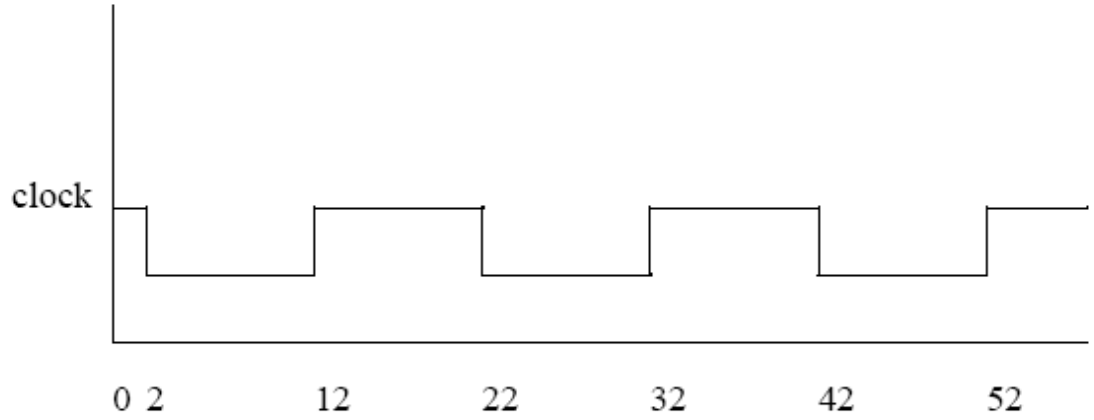
3.3.2. Definice signálů

Jak bylo řečeno výše, moduly používají porty pro komunikaci mezi sebou. V případě, že máme modul s hierarchickou strukturou (máme více modulů v jednom), musí se porty instancovaných modulů propojit signály. Dalším použitím signálů je komunikace mezi více paralelními procesy v modulu.
Signály se definují podobně jako porty, jen s tím rozdílem, že se používá C++ šablona **`sc_signal`**. Parametrem šablony je opět datový typ, reprezentující přenášená data po signálu. Pokud propojujeme porty se signály, musí mít jak port tak i signál stejný datový typ.
Příklad definice portů a signálů:

```
SC_MODULE (module_name)
{
      // deklarace portu modulu
      sc_in<port_type> port1;
      sc_out<port_type> port2;
      sc_inout<port_type> port3;

      // deklarace signalovych promennych
      sc_signal<signal_type> signal1;
      sc_signal<signal_type> signal2, signal3;

      // deklarace datovych promennych
      // deklarace clenskych funkci
      // deklarace procesu

      // konstruktor modulu
      SC_CTOR (module_name)
      {
            // registrovani procesu
            // deklarace citlivostniho seznamu
      }
};
```

Speciálním druhem signálů je hodinový signál. Generují časovací signály pro synchronizaci událostí během simulace. Hodiny řadí události v čase tak, že paralelní události v hardwaru jsou správně simulovány na sekvenčním počítači.
Hodiny se vytváří pomocí SystemC třídy **`sc_clock`**, například takto:

```
sc_clock clock1(“clock1”, 20, 0.5, 2, true);
```



obr. 5 – Nadefinovaný hodinový signál pomocí sc_clock

Tato definice vytvoří hodinový signál, který se jmenuje „clock1“, jeho perioda je 20 časových jednotek, střída 50%, první hrana nastane v době 2 časových jednotek a první hodnota je true. Hodiny jsou typicky vytvářeny v top-level funkci **`sc_main()`** a propouštěny zbytku obvodu přes celou hierarchii. To dovoluje synchronizaci celému návrhu pomocí jedněch hodin.
Hodiny se pak připojují k portu typu **bool**.

3.3.3. Propojování signálů a portů

Jsou dva způsoby, jak propojovat porty se signály. První způsob se nazývá poziční a druhý jmenný.
Jak vypadá poziční způsob, ukazuje následující příklad.

```
// includovane moduly
#include "mult.h"
#include "coeff.h"
#include "sample.h"

SC_MODULE(filter)
{
      sample *s1;
      coeff *c1;
      mult *m1;
      sc_signal<sc_uint<32> > q, s, c;

      SC_CTOR(filter)
      {
            // instancujeme jednotlive moduly
            s1 = new sample("s1");
            c1 = new coeff("c1");
            m1 = new mult("m1");

            // propojime porty a signaly
            (*s1)(q, s);
            (*c1)(c);
            (*m1)(s, c, q);
      }
)
```

Jak je vidět, pro poziční mapování signálů na porty se používá přetížený operátor () ve třídě modulu. Prvním parametrem je signál, který se připojí na prvně definovaný signál v modulu, druhým je signál, který se připojí na druhý definovaný port v pořadí, atd. Tento způsob však nené vhodný, pokud je v modulu hodně portů, protože je to pak málo přehledné a snadno se dělají chyby z důvodu kompatibility portů.

Jmenné mapování popisuje tento příklad:

```
SC_MODULE(filter)
{
      sample *s1;
      coeff *c1;
      mult *m1;
      sc_signal<sc_uint<32> > q, s, c;

      SC_CTOR(filter)
      {
            s1 = new sample("s1");
            s1->din(q);
            s1->dout(s);

            c1 = new coeff("c1");
            c1->out(c);

            m1->a(s);
            m1->b(c);
            m1->q(q);
      }
}
```

Zde se naopak používá přetížený operátor () šablon portů. Parametrem se signál, který se k portu připojí. Při tomto způsobu mapování nezávisí na pořadí definic portů v modulu.

3.3.4. Definice vnitřních proměnných

Stejně jako v klasických C++ třídách můžeme definovat vnitřní pomocné datové členy. Syntaxe je naprosto stejná. Jejich použití je vhodné jako vnitřní paměť obvodu, jako například čítač. Neměly by se ale používat mezi jednotlivými procesy, neboť pak hrozí, že dojde k jejich nedefinovanému stavu. Pro přenos dat mezi procesy je vhodné používat signály.

3.3.5. Definice procesů

SystemC procesy se deklarují v definici třídy modulu a registrují v konstruktoru modulu. Procesy se deklarují stejně jako metody třídy. Musí mít návratový typ **`void`** a žádné parametry. Deklarovaná metoda se pak musí v konstruktoru zaregistrovat pomocí makra **`SC_METHOD`**. Toto makro má jeden parametr a to název metody. Je reprezentováno takto:

```
#define SC_METHOD(func)                                          \
    declare_method_process( func ## _handle,                     \
                            #func,                               \
                            SC_CURRENT_USER_MODULE,              \
                            func )
```

SC_METHOD tedy volá další makro **declare_method_process**, které zaregistruje metodu pro SystemC plánovač procesů a automaticky tuto metodu zařadí do citlivostního seznamu, jak je patrné z následujícího výpisu.

```
#define declare_method_process(handle, name, host_tag, func)        \
    {                                                               \
        ::sc_core::sc_method_process* handle =                      \
            simcontext()->register_method_process( name,            \
                     SC_MAKE_FUNC_PTR( host_tag, func ), this );    \
        sensitive << handle;                                        \
        sensitive_pos << handle;                                    \
        sensitive_neg << handle;                                    \
    }
```

Příklad definice procesu:

```
SC_MODULE(my_module)
{
      // nejake porty
      // nejake signaly
      // ...

      // deklarace procesu
      void my_method_proc();

      // konstruktor modulu
      SC_CTOR(my_module)
      {
            // registrace procesu
            SC_METHOD(my_method_proc);

            // definice citlivostniho seznamu
      }
};
```

3.3.6. Definice citlivostního seznamu

Metody zaregistrované pomocí makra SC_METHOD reagují na množinu signálů zaregistrované v tzv. citlivostních seznamech. Každý modul má tři seznamy:
1. sensitive – do tohoto seznamu se registrují signály, na které je proces citlivý úrovní
2. sensitive_pos – signály, na které je proces citlivý náběžnou hranou
3. sensitive_neg – signály, na které je proces citlivý sestupnou hranou
Citlivostní seznam se musí vždy definovat ihned po zaregistrování procesu v konstruktoru pro každý proces zvlášť, tedy ihned za makro SC_METHOD.
Pro registraci signálů do citlivostních seznamů jsou k dispozici metody **sensitive()**, **sensitive_pos()**, **sensitive_neg()** nebo proudy **sensitive**, **sensitive_pos**, **sensitive_neg**.
Pro kombinační obvod se definují úrovní spouštěné procesy. K tomu slouží **sensitive** seznam. Do proudu je možné zapsat jakýkoli počet signálů, do metody pouze jeden. Ukazuje to následující příklad:

```
SC_MODULE(my_module)
{
      sc_in<int> a;
      sc_in<bool> b;
      sc_out<int> x;
      sc_out<int> y;

      sc_signal<bool> c;
      sc_signal<int> d;
      sc_signal<int> e;

      void my_method_proc();

      SC_CTOR(my_module)
      {
            SC_METHOD(my_method_proc);
            sensitive << a << c << d;
            sensitive(b);
            sensitive(e);
      }
};
```

Pro spouštění procesů sekvenčních obvodů je potřeba definovat seznam signálů citlivých na hranu. Pro to složí zbylé dva seznamy (**sensitive_pos** a **sensitive_neg**). Porty a signály, přidávané do těchto seznamů musí být typu **bool**. Syntaxe je stejná, jako u předchozího příkladu.

3.3.7. Čtení a zápis na porty, signály a vnitřních proměnných

Šablony pro porty a signály mají metody **`read()`** a **`write()`**. Pro porty a signály není vhodné používat operátor přiřazení. Hodnota je signálu nebo portu přiřazena až ve chvíli, kdy skončí metoda procesu.
Naproti tomu, pro vnitřní proměnné se používá pouze operátor přiřazení. Hodnota je vnitřní proměnné přiřazena okamžitě po provedení příkazu.

3.4. Simulace

3.4.1. Řízení simulace

Simulaci lze nastartovat pouze poté, co jsou všechny moduly instancované a správně propojené.
Simulace se startuje pomocí volání funkce **`sc_start()`** z top level funkce **`sc_main()`.**
Této funkci se předává parametr typu double, který vyjadřuje dobu v počtu časových jednotek, po kterou bude simulace trvat. Záporná hodnota nastavuje čas simulace na neurčitou dobu. Tato funkce generuje všechny hodinové signály v patřičných chvílích a volá SystemC scheduler (plánovač procesů).
Simulace může být kdykoli ukončena z kteréhokoli procesu voláním funkce **`sc_stop()`**.
Aktuální čas simulace lze zjistit voláním funkce **`sc_simulation_time()`**.
Pro generování hodinových signálů a řízení simulace je v SystemC ještě jedna metoda, než použití funkce **`sc_start()`**. V tomto případě je nutné nejprve zavolat funkci **`sc_initialize()`** k inicializaci SystemC scheduleru. Poté se mohou do signalů zapsat

hodnoty a voláním funkce **sc_cycle()** se simuluje výsledek nastavení signálů. Funkce **sc_cycle()** má stejně jako **sc_start()** parametr typu double, vyjadřující dobu, po kterou má simulace běžet. Pokud je např. výchozí časová jednotka 1 ns, pak volání **sc_cycle(10)** pokročí v simulaci o 10 ns.

Řekněme například, že máme hodinový signál definován takto:

```
sc_clock clk("my clock",20, 0.5);
```

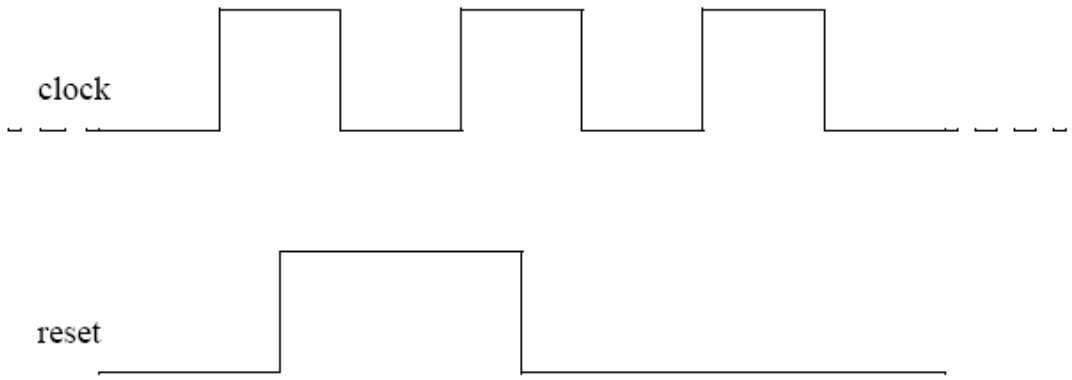
Simulaci nastartujeme na dobu 200 časových jednotek voláním

```
sc_start(200);
```

Tento hodinový signál ale můžeme také generovat takto:

```
sc_signal<bool> clock;

sc_initialize();
for( int i = 0; i <= 200; ++i )
{
      clock = 1;
      sc_cykle(10);
      clock = 0;
      sc_cycle(10);
}
```

Generování signálů tímto způsobem nám navíc dovoluje vyvolat asynchronní události při zachování hodinového signálu, jak je ukázáno na následujícím obrázku:



obr. 6 – Hodinový signál s asynchronním resetem

Pro implementaci takového průběhu, zapíšeme do top-level funkce sc_main() následující kód:

```
sc_initialize();

// nejprve bezi hodiny bez vyvolaneho resetu po dobu deseti cyklu
for( int i = 0; i <= 200; ++i )
{
      clock = 1;
      sc_cykle(10);
      clock = 0;
      sc_cycle(10);
```

```
}

// vyvolani asynchronniho resetu
clock = 1;
sc_cycle(5);
reset = 1;
sc_cycle(5);
clock = 0;
sc_cycle(10);
clock = 1;
sc_cycle(5);
reset = 0;
sc_cycle(5);
clock = 0;
sc_cycle(10);

// ted budou hodiny bezet do nekonecna
for( ;; )
{
      clock = 1;
      sc_cykle(10);
      clock = 0;
      sc_cycle(10);
}
```

3.4.2. Výstup časových průběhů

SystemC poskytuje funkce, které vytvářejí soubory formátu VCD (Value Change Dump), ASCII WIF (Waveform Intermediate Format), nebo ISDB (Integrated Signal Data Base), které obsahují hodnoty proměnných a signálů tak, jak se mění během simulace. Časové průběhy zaznamenané v těchto souborech se dají prohlížet standardními prohlížeči časových průběhů signálů podporující VCD, WIF nebo ISDB formáty, např. freewarovým produktem Dinotrace (**http://www.veripool.com/dinotrace/**).

Generování výstupních časových průběhů má několik pravidel:
- Sledovány mohou být pouze všechny signály a datové členy modulů (proměnné tříd reprezentující moduly). Lokální proměnné funkcí nemohou být sledovány.
- Sledovány mohou být proměnné a signály skalárních typů, polí nebo složených typů.
- Během simulace může být vytvořeno více různých výstupních souborů.
- Signál nebo proměnná může být sledována několikrát v různých výstupních formátech.

3.4.3. Vytvoření výstupního simulačního souboru

Pokud chceme nějaký výstup simulace, musíme nejprve vytvořit výstupní soubor. Výstupní soubor se nejčastěji vytváří z top-level funkce sc_main() poté, co jsou všechny moduly a signály instancovány. Pro každý výstupní formát existuje jedna funkce, která soubor vytváří (každá má parametr typu řetězec, kterému se předává název výstupního souboru bez koncovky, koncovka se doplní sama, podle zvoleného formátu):
- formát VCD - **`sc_create_vcd_trace_file(char *filename)`**
- formát WIF - **`sc_create_wif_trace_file(char *filename)`**
- formát ISDB - **`sc_create_isdb_trace_file(char *filename)`**

Příklad pro vytvoření výstupního souboru:

```
sc_trace_file *my_trace_file;
my_trace_file = sc_create_isdb_file("my_trace");
```

Na konci simulace je nutné všechny soubory ukončit. K tomu slouží následující funkce:
- `sc_close_isdb_trace_file(my_trace_file);`
- `sc_close_wif_trace_file(my_trace_file);`
- `sc_close_vcd_trace_file(my_trace_file);`

3.4.4. Sledování proměnných, signálů a portů

Pro sledování proměnných a signálů slouží funkce **`sc_trace()`**. Tato funkce má tři parametry:

1. ukazatel na otevřený soubor
2. odkaz nebo ukazatel na proměnnou, která se má sledovat
3. odkaz na řetězec, kterým bude proměnná nebo signál pojmenovaný v prohlížeči.

Příklad:

```
sc_signal<logic> a;
float b;

sc_trace(trace_file, a, "Signal_A");
sc_trace(trace_file, b, "Promenna_B");
```

Volání funkce **`sc_trace()`** registruje proměnnou nebo signál tak, že ho ukládá do seznamu. Sledování je během simulace zajištěno plánovačem procesů.
Sledování polí a složených typů je podobné, viz. uživatelská příručka SystemC.

3.4.5. Plánovač procesů

Úlohou plánovače procesů je rozlišit pořadí spouštěných procesů tak, aby na sekvenčním počítači bylo simulováno paralelní chování obvodů. Jak plánovač procesů pracuje je popsáno v dokumentu [4] **FUNCTIONAL SPECIFICATION FOR SYSTEMC 2.0**, včetně podrobnějšího pseudokódu.
Jednodušší popis je v příloze C.

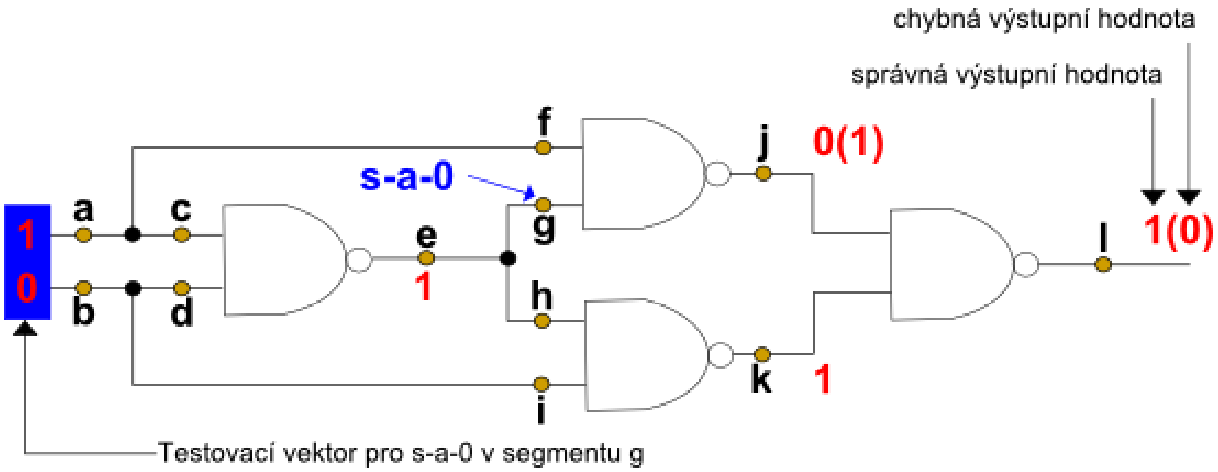
4. Simulace poruch

4.1. Úvod

Mimo simulace pro ověření funkčnosti hardwarového návrhu bývá často potřebná simulace poruch pro analýzu chování systému při poruše obvodu. Existuje několik modelů simulací poruch digitálních obvodů. Mezi základní modely patří:

- *model trvalých poruch (stuck-at fault model)* – signál nebo výstup hradla je uváznutý v jedné z logických hodnot (0 nebo 1) nezávisle na vstupech obvodu.
- *model poruch přemostění (bridging fault model)* – dva signály jsou spojené dohromady. Výsledek této poruchy je pak závislý na implementaci obvodu (funkce wired-AND nebo wired-OR). Protože v obvodu je potenciální počet těchto poruch n^2, tak se vymezují pouze na signály, které leží v sousedství ve fyzickém návrhu.
- *model přerušení (open fault model)* – zde se předpokládá, že signál je přetržen a tím pádem je jeden nebo více vstupů je odpojeno od výstupu, který je měl budit. Stejně jako u modelu poruch přemostění je chování závislé na implementaci obvodu.
- *model poruch zpoždění (delay fault model)* – v tomto modelu se předpokládají správné hodnoty na signálech, ale s větším zpožděním (nebo vzácně s menším zpožděním) než při normálním chování obvodu.

Tato práce se věnuje pouze modelu trvalých poruch. Pokud některý signál uvázne v logické hodnotě 0 resp. 1, značí se tato událost **s-a-0** (stuck-at 0) resp. **s-a-1** (stuck-at 1). Tyto poruchy se uvažují na všech primárních vstupech obvodu, na všech výstupech hradel a také v každém větvení signálu. Při simulaci se vždy předpokládá, že pro danou sadu vstupních testovacích vektorů je aktivní pouze jedna porucha.
Na následujícím obrázku je obvod logické funkce XOR postavený pomocí hradel NAND. Na něm je vidět, že je zde dvanáct míst, kde může dojít k poruše, z čehož vyplývá, že počet poruch, které mohou nastat, je dvacetčtyři. Na příkladě je také vidět, co se stane s obvodem, pokud na signálu **g** vznikne porucha s-a-0.
O simulaci poruch pojednává článek [7].



obr. 7 – obvod XOR s aktivní poruchou na signálu g

4.2. Metody simulace trvalých poruch

Pro simulaci poruch hardwaru byly stanoveny tři metody:

- *Sabotéři* - jedná se o obvod způsobující poruchu. Vkládá se do cesty signálu mezi dvěmi komponentami. Provedení poruchy je aktivováno řídícím vstupem tohoto obvodu.
- *Mutanti* - jedna komponenta je zaměněna jinou rozšířenou komponentou. Tato komponenta obsahuje parametry pro způsobení poruchy v obvodu.
- *Simulační příkaz* – pro zmanipulování hodnoty nějakého signálu se používá přímo příkaz simulačního nástroje.

V následujících podkapitolách jsou jednotlivé způsoby popsány podrobněji.

4.2.1. Sabotéři (Saboteurs)

Vkládání sabotérů do cest signálů je oblíbená metoda, neboť je relativně snadná na implementaci. Sabotéři jsou také vhodní pro modelování poruch zpoždění na signálech.
Sabotéra pro simulaci trvalých poruch si lze představit jako obvod se dvěma vstupy a jedním výstupem. Jeden vstup a výstup se připojí do cesty na signálu, druhý vstup pak řídí aktivitu poruchy. V následujícím kódu je ukázka implementace jednoduchého sériového sabotéra. Pokud je na řídícím vstupu “1” resp. “2“, aktivuje se porucha s-a-1 resp. s-a-0. Pokud je na řídícím vstupu jakákoli jiná hodnota, bude na výstupu sabotéra hodnota přenesená ze vstupu.

```
switch( fault_active.read() )
{
case 1:
      out.write(sc_logic_1);
      break;
case 2:
      out.write(sc_logic_0);
      break;
default:
      out.write(in.read());
      break;
}
```

4.2.2. Mutanti (Mutants)

Mutanti se dělí do dvou kategorií. Do první kategorie se řadí záměna správně pracující komponenty za špatně pracující komponentu. Například namísto hradla AND se zapojí hradlo NAND. Bohužel tyto poruchy nelze v SystemC provádět dynamicky (nelze za běhu simulace odpojit entitu a nahradit ji jinou). Bylo by totiž nutné kvůli každé změně obvodu znovu překompilovat kód, proto se tato metoda používá jen zřídka.
Do druhé kategorie se řadí záměna správně pracující komponenty za modifikovanou komponentu. Modifikovaná komponenta je zpravidla rozšířená pouze pro modelování poruch. Přidá se jí jeden řídící vstup pro aktivaci poruchy podobně jako u sabotérů. V následujícím kódu je takto modifikováno hradlo NAND. V implementaci se přečte vstupní hodnota řídícího portu CLT a v závislosti na ní se pak zapíše buď příslušná porucha nebo správná hodnota.

```
SC_MODULE( faultyNand2 )
{
      sc_out<sc_logic> Z0;
      sc_in<sc_logic> A0, A1;
      sc_in<int> CLT;

      void process();

      SC_CTOR( faultyNand2 )
      {
            SC_METHOD( process );
            sensitive << A0 << A1 << CTL;
      }
};

void faultyNand2::process()
{
      switch( CTL )
      {
      case 0:
            Z0.write(sc_logic_0);
            break;
      case 1:
            Z0.write(sc_logic_1);
            break;
      default:
            Z0.write(~(A0 & A1));
            break;
      }
}
```

4.2.3. Příkaz simulátoru (Simulator Command)

V některých HDL simulátorech existují příkazy, které dovolují manipulovat se signály a proměnnými. Tyto příkazy pro modelování poruch mají velkou výhodu v tom, že není potřeba měnit popis obvodu (není nutné vkládat další obvody nebo je modifikovat).
Představme si, že chceme v obvodu z obrázku 7 přikázat zakreslenou poruchu s-a-0 v signálu g. Pak takový příkaz může vypadat nějak takto:

```
XOR.g.write(0);
```

Bohužel SystemC zatím tyto příkazy neimplementuje, proto pokud chceme tuto metodu použít, je potřeba napsat si svoje vlastní implementační struktury.

4.2.4. Porovnání metod pro simulaci trvalých poruch

Nejvhodnější pro simulaci trvalých poruch je metoda příkazů simulátoru, neboť nepotřebuje měnit popis obvodu. Ostatní dvě metody vždy musí měnit popis obvodu. V praxi přidané porty u sabotérů a mutantů zesložiťují obvod.
Následuje tabulka naměřených výkonů převzatá z [7]. Zde je K1 nějaký malý obvod, K2 je nějaký větší obvod. Oba jsou popsány na úrovni hradel. Navíc jsou zde zmíněny hybridní metody (sabotéři a mutanti se simulačními příkazy), kde poruchy nejsou řízeny vstupními porty, ale nějakými simulačními příkazy.

	Čas (normalizovaný)	
Obvod	K1	K2
simulační příkazy	1,00	1,00
mutanti se simulačními příkazy	1,01	1,23
mutanti	1,05	1,96
sabotéři se simulačními příkazy	1,28	1,81
sabotéři	1,30	1,84
FIT (VHDL Fault Injection Tool)	3,71	5,21

Tab. 1 – porovnání různých metod simulace trvalých poruch

4.3. Zrychlení simulace

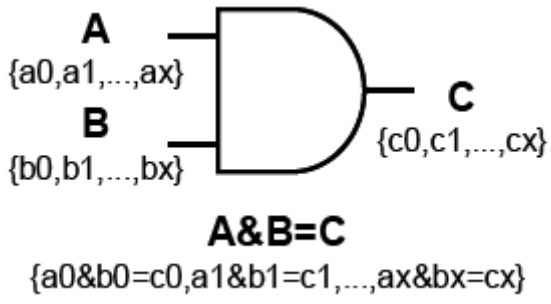
V praxi se často simulují složité obvody, které obsahují mnoho hradel nebo primárních vstupních portů. V případě simulace trvalých poruch je v těchto obvodech navíc větší riziko nějaké poruchy, neboť vzniká také více míst, kde může porucha nastat. Tím pádem musíme při simulaci poruch otestovat mnoho vstupních vektorů, abychom našli co nejvíce nedetekovatelných poruch. Při takto náročné simulaci roste simulační čas, a proto je nutné hledat nějaké řešení, které povede ke zrychlení výpočtu.
Dlouho existující způsob zrychlení simulace je paralelní výpočet. Zde je tím hlavně myšlen paralelní výpočet na počítači s jedním procesorem. Uplatňují se dva druhy paralelních výpočtů:

- paralelní simulace testovacích vektorů
- paralelní simulace poruch

Na oba přístupy se nyní podíváme blíže.

4.3.1. Paralelní simulace testovacích vektorů

Nejjednodušším způsobem, jak zrychlit simulaci, je odsimulovat více vstupních vektorů najednou. Pokud předpokládáme dvouhodnotovou logiku, můžeme odsimulovat v jednom průchodu třicetdva vektorů najednou, neboť tím využijeme celé 32 bitové slovo. Tohoto přístupu lze však využít pouze u jednoduchých obvodů, například u logických hradel. Na obrázku 8 je zobrazeno paralelní hradlo AND.

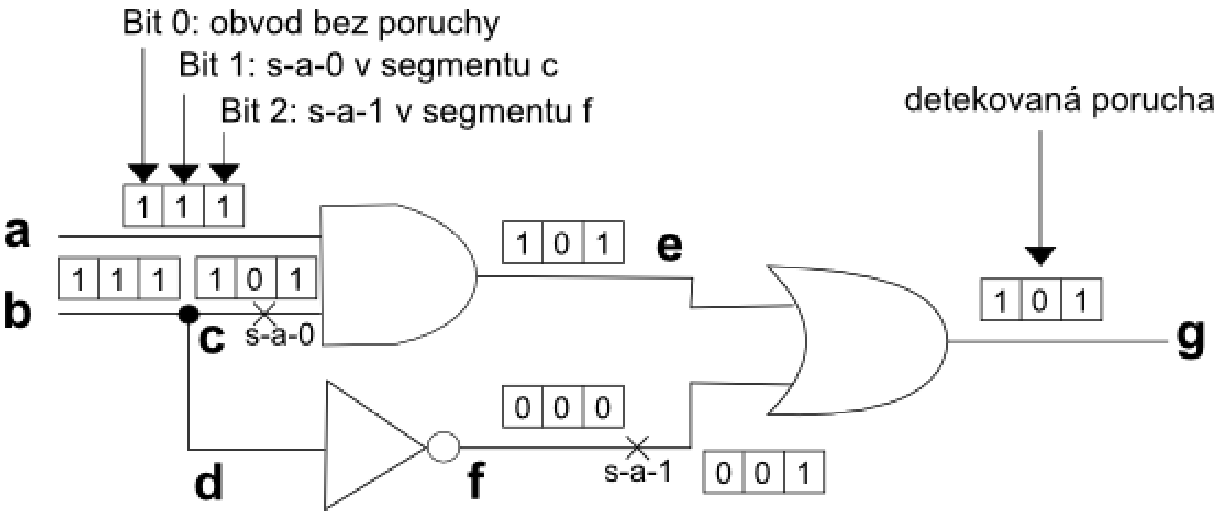


obr. 8 – Paralelní hradlo AND

Další nevýhodou je fakt, že je tento způsob paralelizace nepoužitelný při simulování sekvenčních obvodů, neboť výstupní hodnoty jsou závislé na pořadí vektorů.

4.3.2. Paralelní simulace poruch

Dalším přístupem pro paralelizaci je simulovat více poruch najednou. Opět se využívá celé 32 bitové slovo, ale s tím rozdílem, že se na primární vstupy zapisuje hodnota 0x00000000 resp. 0xFFFFFFFF, pokud je ve vstupním testovacím vektoru na příslušném místě logická nula resp. logická jednička. V tomto případě se dá paralelizace využít opět pouze v jednoduchých obvodech tvořenými logickými hradly. Na rozdíl od minulého přístupu zde ale funguje simulace sekvenčních obvodů, neboť vstupní vektory jsou zapisovány na primární porty sériově.
Na následujícím obrázku je vysvětleno, jak tato simulace funguje.



obr. 9 – Paralelní simulace trvalých poruch

Všechny signály a porty jsou vlastně sběrnice, kde každý signál přenáší hodnotu pro jednu určitou poruchu. Na obrázku je vysvětleno, že první hodnota zleva platí pro případ, kdy je obvod bez poruchy, prostřední hodnota platí pro poruchu s-a-0 v signálu c a poslední hodnota platí pro poruchu s-a-1 v signálu f. Na výstupu obvodu je pak na prostřední pozici hodnota log. nuly, která se liší od hodnoty obvodu, který je bez poruchy. Tím pádem se porucha detekovala.

Pro implementaci paralelní simulace trvalých poruch je vhodné vkládat sabotéry. Takovýto sabotér pak bude mít jeden datový vstup, dva vstupy pro řízení poruch a jeden datový výstup. Následující kód je implementací sabotéra, který vyhodnotí výstupní hodnotu v závislosti na vstupních a řídících hodnotách. Protože signály tvoří jakousi sběrnici, přenášené hodnoty jsou vektory logických hodnot.

```
void ParallelSaboteur::Process()
{
      var_in = in.read();
      var_sa0 = sa0.read();
      var_sa1 = sa1.read();
      var_out = (var_in & (~var_sa0));
      var_out = (var_out | var_sa1);
      out.write(var_out);
}
```

Port `in` je datovým vstupním portem, `sa0` resp. `sa1` je port pro nastavení případné poruchy s-a-0 resp. s-a-1 a `out` je výstupní port. Pokud má sabotér provést nějakou poruchu, musí mít na vstupu sa0 nebo sa1 nastaven příslušný bit na logickou jedničku. Pokud budeme například uvažovat obvod z obr. 9 a poruchu s-a-0 v signálu c, bude na vstup sa0 přivedena hodnota "010" a na vstup sa1 hodnota "000".

4.3.3. Čtyřhodnotová logika při paralelní simulaci

V předchozích dvou odstavcích jsme uvažovali pouze dvouhodnotovou logiku (0 nebo 1). Ve většině případů je však potřebné mít alespoň tříhodnotovou logiku, kvůli neznámé hodnotě "X" (don't care).
SystemC má implementovaný typ `sc_logic`, který reprezentuje čtyřhodnotovou logiku (kromě neznámé hodnoty "X" také hodnotu "Z", která vyznačuje stav vysoké impedance).
Navíc SystemC implementuje šablonu sc_lv<> (logic vector), což je šablona pro definici sběrnicového signálu, kde parametrem je počet signálů na sběrnici. Tento typ je vhodný pro definici portů a signálů pro paralelní simulaci. Na následující tabulce jsou ukázané zrychlení při použití tohoto typu při paralelním zpracování. Tabulka je vyjmuta z [7]. FCT a MSAB jsou opět nějaké dva obvody o různých velikostech.

Obvod	sc_logic(1)		sc_lv<32>	
	čas (s)	zrychlení	čas (s)	zrychlení
FCT	32,62	1	8,3	3,65
MSAB	12,9	1	15,4	0,84

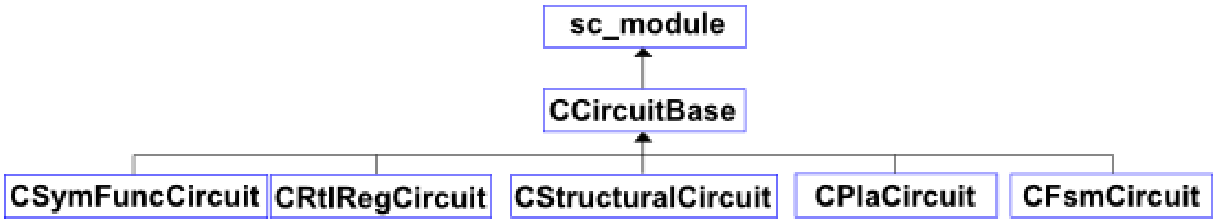
Tab. 2 – porovnání času simulace při sériovém a paralelním zpracování

5. Návrh a implementace

V této kapitole je popsán postup při návrhu a implementaci simulačního modulu a aplikace spustitelné z příkazové řádky pro odsimulování obvodu. Simulační modul tvoří rozhraní mezi HUBem a SystemC. Samotná aplikace pro simulaci obsahuje existující modul pro import různých formátů souborů a modulem pro simulaci. Protože požadavkem bylo implementovat paralelní simulaci trvalých poruch, jsou všechny porty kromě hodinových a resetových, sběrnicové o 32 signálech a typu `sc_logic` reprezentující čtyřhodnotovou logiku. Protože ne vždy je potřeba simulovat poruchy, tak jsou i při normální simulaci hodnoty v typu sc_lv<32>. V tomto případě je logická nula reprezentována jako "00000000", logická jednička hodnotou "FFFFFFFF" a nedefinovaný stav "XXXXXXXX". Na prvním signálu vždy probíhá simulace bez poruchy.
Simulátor bude pro výstup časových diagramů používat formát VCD, který lze zobrazit i některými volně šiřitelnými prohlížeči (na CD je freewarový program GtkWave Viewer pro operační systém Windows). Bohužel budou i při normální simulaci na výstupech 32bitové hodnoty, neboť SystemC nedovoluje sledovat samostatné signály sběrnice.

5.1. Reprezentace modulů v SystemC

První otázkou, kterou jsem se zabýval při tvorbě modulu pro simulaci, jak navrhnout převod popisu obvodu z HUBu do SystemC. Protože HUB popisuje pět různých popisů (strukturní, symbolické funkce, RTL registr, PLA popis a FSM popis), je nutné připravit pro každý tento popis třídu implementující funkčnost popisu. Protože každá z těchto tříd bude mít některé společné vlastnosti, vyplatí se využívat polymorfizmu a navrhnout nějakou bázovou třídu. Od ní budeme požadovat, aby dědila vlastnosti třídy SystemC **`sc_module`** pro přímé použití v simulaci. Výsledný graf dědičnosti tedy bude vypadat takto:



obr. 10 – Graf dědičnosti tříd pro popis obvodu v SystemC

5.1.1. Bázová třída CCircuitBase

Jak bylo řečeno výše, od této třídy dědí všechny třídy implementující konkrétní popis obvodu. Protože bázová třída nesmí žádný konkrétní popis implementovat, je po ní požadováno, aby byla abstraktní třídou.
Každý obvod má nějaké vstupní a výstupní porty. Z důvodu možnosti paralelní simulace poruch musí být typu `sc_lv<32>`, což reprezentuje sběrnici o 32 signálech čtyřhodnotové logiky. Protože při propojování portů signály je potřeba znát jména portů, byla jako STL kontejner zvolena šablona mapa. Ve třídě `CCircuitBase` jsou pak instance těchto šablon pojmenovány `m_InPorts` a `m_OutPorts`.
Další společnou vlastností všech obvodů je zpoždění. HUB sice v současné době zpoždění nepodporuje, nicméně tím bude modul do budoucna připraven pro rozšíření HUBu. Zpoždění je zadáváno v počtu nanosekund a reprezentuje ho členská proměnná `m_CircuitDelay`.

```
class CCircuitBase: public ::sc_core::sc_module
{
protected:
      std::map< std::string, sc_in<sc_lv<32> >* > m_InPorts;
      std::map< std::string, sc_out<sc_lv<32> >* > m_OutPorts;
      double m_CircuitDelay;

      ...
public:
      virtual void Process() = 0;
      virtual sc_in_clk* GetClock() = 0;
      virtual sc_in<sc_logic>* GetReset() = 0;
      virtual void SetResetState() = 0;

      ...
};
```

Následují společné metody, které každá třída implementuje jinak, proto jsou deklarovány jako abstraktní. Metoda Process() provádí funkci obvodu, např. u RTL registru zapisuje hodnotu do paměti. Pokud má obvod hodinový vstup, metoda GetClock() vrací ukazatel na tento port. Podobný účel má také metoda GetReset(), která však pracuje s portem pro reset sekvenčního obvodu. Metoda SetResetState() pak slouží pro nastavení počátečního stavu sekvenčního obvodu.
V přiloženém výpisu nejsou vypsnány některé metody např. pro získání ukazatele na nějaký vstupní port, neboť jejich význam je jasný.

5.1.2. Třída CSymFuncCircuit

Třída `CSymFuncCircuit` implementuje funkčnost behaviorálního popisu plně symetrických funkcí s jedním výstupem. Obsahuje ukazatel na popis obvodu `m_Descr`, který nese informaci typu funkce a počtu vstupů obvodu. Pro každý typ funkce obsahuje jednu privátní metodu vykonávající její funkci. Konstruktor pak přebírá jméno entity nebo instance obvodu, ukazatel na popis a seznamy jmen portů.
Zděděné abstrakní metody jsou přepsány tak, aby obvodu vyhovovaly. Metoda `Process()` podle typu funkce zavolá příslušnou soukronou metodu pro vykonání funkce, metody `GetClock()` a `GetReset()` vrací `NULL` a metoda `SetResetState()` neprovádí nic. Zkrácená deklarace třídy je zde, poté následuje ukázka implementace funkce and.

```
class CSymFuncCircuit: public CCircuitBase
{
private:
      hub::SymFunc* m_Descr;

      sc_lv<32> t0();
      sc_lv<32> t1();

      ...
public:
      typedef CSymFuncCircuit SC_CURRENT_USER_MODULE;

      CSymFuncCircuit(::sc_core::sc_module_name, hub::SymFunc* descr,
            std::list<std::string>& inPorts,
            std::list<std::string>& outPorts);
```

```
      ...
};

sc_lv<32> CSymFuncCircuit::and()
{
      sc_lv<32> out;
      std::map<std::string, sc_in<sc_lv<32> >* >::iterator it =
            m_InPorts.begin();

      out = it->second->read();
      ++it;
      for( it; it != m_InPorts.end(); ++it )
      {
            out &= it->second->read();
      }

      return out;
}
```

5.1.3. Třída CRtlRegCircuit

Tato třída implementuje klopný obvod typu D se synchronním resetem a citlivostí na náběžnou hranu hodinového signálu. V deklaraci třídy jsou explicitně přidány definice portů pro hodiny (`m_Clk`) a reset (`m_Reset`), neboť tato informace v HUBu není, ale předpokládá se, že obvod tyto porty má. Datový vstup a výstupy pak vytváří konstruktor. Počátečním vnitřním stavem je hodnota "X". Metoda `Process()` definuje chování obvodu při náběžné hraně hodin a synchronní reset. `SetResetState()` zapíše na negovaný výstup hodnoty logické nuly a na negovaný výstup logické jedničky. `GetClock()` resp. `GetReset()` vrací členské proměnné `m_Clk` resp. `m_Reset`.

```
class CRtlRegCircuit: public CCircuitBase
{
private:
      hub::RTLReg* m_Descr;
      sc_in_clk m_Clk;
      sc_in<sc_logic> m_Reset;

      ...
public:
      typedef CRtlRegCircuit SC_CURRENT_USER_MODULE;

      CRtlRegCircuit(::sc_core::sc_module_name, hub::RTLReg* descr);

      ...
};
```

5.1.4. Třída CPlaCircuit

Implementace této třídy zajišťuje simulaci obvodů s PLA popisem. Ukazatel `m_Descr` ukazuje na datovou strukturu s popisem PLA. Ten obsahuje počet vstupních a výstupních portů, typ PLA a hlavně pravdivostní tabulku potřebnou pro implementaci. Protože je při implementaci funkčnosti potřeba znát pozici portu podle čísla (z důvodu čtení a zapisovaní hodnot podle pozice v pravdivostní tabulce), obsahuje tato třída navíc vektory řetězců. Řetězec na první pozici určuje jméno prvního portu, řetězec na druhé pozici ve vektoru port na druhém místě atd. Tyto vektory se nazývají `m_InPortsMap` a `m_OutPortsMap`.

`SetResetState()` neprovádí nic a `GetClock()` a `GetReset()` vrací `NULL`.

Metoda `Process()` prochází pravdivostní tabulku a na výstupy zapíše "00000000" nebo "FFFFFFF" nebo nedefinovanou hodnotu.

Pozn:
I když u obvodů s behaviorálním popisem není moc užitečné provádět simulaci poruch, má tato třída pro obecnost navíc metodu `ParallelProcess()`, která se používá v případech, kdy by uživatel chtěl připojit na vstupní porty tohoto obvodu sabotéry. V tom případě je nutné projít tabulku dvaatřicetkrát zvlášt pro každý signál na sběrnici, neboť injektovaná porucha může na výstupu vyvolat jinou výstupní hodnotu, než v případě bez poruchy. To, zda se jako proces modulu zaregistruje `Process()` nebo `ParallelProcess()`, definuje poslední parametr konstruktoru.

```
class CPlaCircuit: public CCircuitBase
{
private:
      hub::PlaDescr *m_Descr;
      std::vector<std::string> m_InPortsMap;
      std::vector<std::string> m_OutPortsMap;

      ...
protected:
      void AddInPort(std::string name, ESensitivity sens);
      void AddOutPort(std::string name);
public:
      typedef CPlaCircuit SC_CURRENT_USER_MODULE;

      CPlaCircuit(::sc_core::sc_module_name, hub::PlaDescr* descr,
            std::list<std::string>&, std::list<std::string>&, bool );

      ...
};
```

5.1.5. Třída CFsmCircuit

Třída `CFsmCircuit` implementuje chování konečných automatů. Stejně jako implementace D klopného obvodu má i tento obvod hodinový a resetový vstup. Ukazatel `m_Descr` opět ukazuje na popis obvodu, v tomto případě obsahuje počet vstupů a výstupů a hlavně přechodovou tabulku automatu, kterou prochází metoda `Process()`. Společnou vlastností s `CPlaCircuit` je potřeba znát názvy portů tak, jak jdou za sebou v pořadí, proto jsou zde opět vektory `m_InPortsMap` a `m_OutPortsMap`.
`GetClock()` resp. `GetReset()` vrací členské proměnné `m_Clk` resp. `m_Reset`. `SetResetState()` pak nastavuje počáteční stav, který je obsažen v popisu `m_Descr`.

Pozn:
Stejně jako `CPlaCircuit` má tato třída pro obecnost metodu `ParallelProcess()`, i když i zde je její využití minimální. Pokud bude na vstupu obvodu injektována porucha sabotérem, musí se opět přechodová tabulka projít dvaatřicetkrát pro každý vodič sběrnice. Proto také třída uchovává třicetdva vnitřních stavů. Aktuální stavy obvodu jsou obsaženy v poli `m_State`.

```
class CFsmCircuit: public CCircuitBase
{
private:
      hub::FsmDescr* m_Descr;
      sc_in_clk m_Clk;
      sc_in<sc_logic> m_Reset;
      sc_signal<hub::FsmState*> m_State[32];
      std::vector<std::string> m_InPortsMap;
      std::vector<std::string> m_OutPortsMap;
protected:
      void AddInPort(std::string name, ESensitivity sens);
      void AddOutPort(std::string name);
      void Process();
      void ParallelProcess();
public:
      typedef CFsmCircuit SC_CURRENT_USER_MODULE;

      CFsmCircuit(::sc_core::sc_module_name, hub::FsmDescr* descr,
            std::list<std::string>&, std::list<std::string>&, bool);

};
```

5.1.6. Třída CStructuralCircuit

Strukturní popis implementuje třída `CStructuralCircuit`. Mezi členské proměnné patří mapy uchovávající instancované komponenty (`m_Instances`) a signály, které propojují komponenty (`m_Signals`).
Protože SystemC požaduje, aby každý port byl připojen k nějakému signálu, má třída navíc seznamy připojených vstupních (m_ConnectedInPorts) a výstupních (m_ConnectedOutPorts) portů. Metoda `CreateMissingSignals()` pak pomocí těchto seznamů vyhledá porty, ke kterým nejsou připojeny žádné signály, vytvoří je a připojí je k těmto signálům. Příkladem může být obvod popsaný ve formátu BENCH obsahující D klopné obvody, které nemají nikam připojený negovaný výstup.
Přepsaná virtuální metoda `Process()` v tomto obvodu nedělá nic, stejně tak jako `SetResetState()`. `GetClock()` a `GetReset()` vrací `NULL`.

```
class CStructuralCircuit: public CCircuitBase
{
private:
      std::map<std::string, CCircuitBase*> m_Instances;
      std::map<std::string, sc_signal<sc_lv<32> > *> m_Signals;
      std::list<sc_in<sc_lv<32> > *> m_ConnectedInPorts;
      std::list<sc_out<sc_lv<32> > *> m_ConnectedOutPorts;
public:
      typedef CStructuralCircuit SC_CURRENT_USER_MODULE;

      CStructuralCircuit(::sc_core::sc_module_name,
            std::list<std::string>&, std::list<std::string>&);
      void CreateMissingSignals();

      ...
};
```

5.2. Paralelní simulace trvalých poruch

Mezi další potřebné moduly patří modul sabotéra a modul pro detekci poruch v obvodu. Ty se využívají při paralelní simulaci trvalých poruch. Oba tyto moduly dědí vlastnosti třídy sc_module, aby je bylo možné připojit k ostatním modulům.
Význam sabotérů již byl vysvětlen v kapitole 4, zde jen předvedeme definici tohoto modulu.
Modul pro detekci poruch je vlastně jakýsi obvod, který má pouze vstupy a žádný výstup. Na tyto vstupy se připojují všechny výstupy simulovaného obvodu. Jak funguje je popsáno v této kapitole.

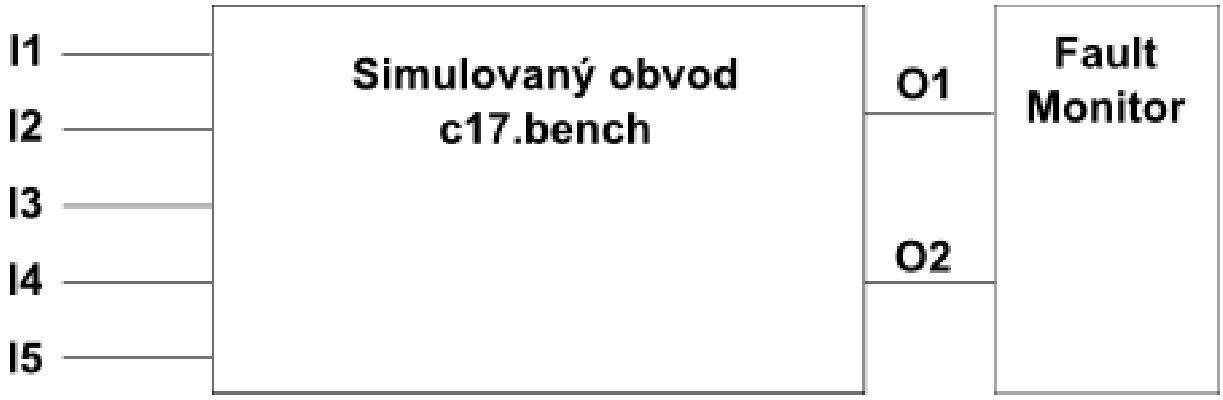
5.2.1. Třída CSaboteur

Třída CSaboteur implementuje funkci trvalých poruch tak, jak je popsáno v oddíle 4.3.2. Zde je pouze definice modulu, implementace metody Process() vykonávající požadovanou poruchu již byla zmíněna.

```
class CSaboteur: public ::sc_core::sc_module
{
private:
      sc_in<sc_lv<32> > m_In;
      sc_out<sc_lv<32> > m_Out;
      sc_in<sc_lv<32> > m_S0;
      sc_in<sc_lv<32> > m_S1;

      sc_lv<32> m_VarIn;
      sc_lv<32> m_VarOut;
      sc_lv<32> m_VarS0;
      sc_lv<32> m_VarS1;
public:
      typedef CSaboteur SC_CURRENT_USER_MODULE;
      void Process();

      CSaboteur(::sc_core::sc_module_name);
};
```

5.2.2. Třída CFaultMonitor

Třída `CFaultMonitor` monitoruje průběh simulace trvalých poruch a zapisuje výsledky do výstupního textového souboru. Má hlavně za úkol detekovat poruchy. To dělá tak, že porovná výstupní hodnotu bezporuchového provozu s výstupní hodnotou při zapnuté poruše. Pokud jsou obě hodnoty stejné, tak porucha nebyla detekována, pokud jsou různé, porucha detekována byla. Protože při simulaci používáme dvaatřiceti bitové slovo, můžeme pro tuto detekci použít operaci XOR a to tak, že vezmeme první bit zleva (který definuje výstupní hodnotu bezporuchového obvodu), vytvoříme hodnotu typu sc_lv<32>, které nastavíme všechny bity právě podle bezporuchové hodnoty a tuto hodnotu pak „XORujeme“ s výstupní hodnotou. Protože se testuje sekvence více vektorů, je potřeba na každém výstupním portu tyto výsledky logicky sčítat. Detekci poruch ilustruje následující příklad:



obr. 11 – Připojení fault monitoru k obvodu

Simulovaný obvod má pět vstupních a dva výstupní porty. Simulovalo se dvacetdva různých poruch v obvodu, simulace se tedy dala provést v jednom průchodu. Výstupní hodnoty na prvním portu byly: 0xFEBDABFF, 0x00818000, 0xFFAFFFFF, 0x00A18400, 0x00890000, 0xFFAFFDFF, na druhém byly hodnoty: 0xAFFDABFF, 0x22018000, 0x20052200, 0xEBFFEFFF, 0x28012200, 0xEBFFE7FF.

FEBDABFF	00818000	FFAFFFFF	00A18400	00890000	FFAFFDFF
AFFDABFF	22018000	20052200	EBFFEFFF	28012200	EBFFE7FF

obr. 12 – Časový průběh výstupních hodnot paralelní simulace poruch

Pokud se podíváme například na první hodnotu z prvního výstupního portu (0xFEBDABFF), vidíme, že v provozu bez poruchy má být výstupní hodnota v logické jedničce (první bit zleva je nastaven). Proto s touto hodnotou provedeme XOR operaci (0xFEBDABFF xor 0xFFFFFFFF) a vyjde 0x01425400. U druhé hodnoty na prvním výstupním portu je zobrazeno, že bez poruchy má být na výstupu logická nula. V tomto případě je 0x00818000 xor 0x0000000 rovno původní hodnotě (0x00818000). Takto xorujeme všechny výstupní hodnoty. Výsledné hodnoty pak budou následující:
Port O1: 0x01425400, 0x00818000, 0x00500000, 0x00A18400, 0x00890000, 0x00500200.
Port O2: 0x50025400, 0x22018000, 0x20052200, 0x14001000, 0x28012200, 0x14001800.
Všechny tyto hodnoty nakonec logicky sečteme operací OR a vyjde hodnota 0x7FFFFE00 (binárně 0111 1111 1111 1111 1111 1110 0000 0000). Vidíme, že u každé poruchy je nastaven bit, tím pádem byly všechny poruchy detekované.

```
class CFaultMonitor: public ::sc_core::sc_module
{
private:
      sc_in<sc_lv<32> > *m_Inputs;
      sc_lv<32> *m_DetectedFaults;

      ...
public:
      typedef CFaultMonitor SC_CURRENT_USER_MODULE;
      CFaultMonitor(::sc_core::sc_module_name, int inputNo, int vectorNo,
            int faultNo, std::string outputFn);
      void ConnectOutput(int index, sc_signal<sc_lv<32> >* signal);

      ...
};
```

5.3. Rozhraní mezi HUBem a SystemC

Nyní, když už můžeme reprezentovat obvody v SystemC musíme navrhnout třídu, která bude poskytovat rozhraní mezi HUBem a třídami implementující moduly v SystemC. Třída bude mít za úkol přečíst popis obvodu z HUBu a zkonstruovat obvod v SystemC reprezentaci pro následnou simulaci.
Protože pro převod obvodu z HUBu do SystemC i pro následnou simulaci potřebujeme znát mnoho informací, je potřeba definovat, jak tyto informace tomuto rozhraní předat. Protože parametrů simulace může být mnoho, bylo by nepohodlné zadávat je všechny z příkazové řádky (představte si zadávat např sto testovacích vektorů). Proto navrhneme nějaký textový formát, kde tyto informace budou.

Třída poskytující toto rozhraní se bude jmenovat `CSimulator`. Bude obsahovat všechny informace potřebné pro simulaci. Následující výpis představuje nejdůležitější metody této třídy.

```
class CSimulator
{
private:
      CCircuitBase* CreateSystemCModule(std::string name, hub::Entity* ent,
            bool);

      ...
public:
      CSimulator(hub::Hub*);

      void SetLibraryName(std::string name)
            { m_LibraryName = name; }
      void SetModelName(std::string name)
            { m_ModelName = name; }
      void SetParallelComputation(bool v)
            { m_ParallelComputation = v; }
      void SetHaveFaultResult(bool v)
            { m_HaveFaultResult = v; }
      void SetFaultResFn(std::string filename)
            { m_FaultResFn = filename; }

      bool LoadInput(std::string filename);
      void LoadFaultList(std::string filename);
      void LoadPattern(std::string filename);

      void CreateSystemCRepresentation();
      void Start(std::string output);
};
```

Konstruktor inicializuje členské proměnné a hlavně předává ukazatel na HUB, z kterého se bude převádět obvod. Dalších pět veřejných metod nastavují simulaci. `SetLibraryName()` nastavuje jméno knihovny, z které se bude obvod simulovat. Jméno obvodu se pak nastaví pomocí `SetModelName()`. `SetParallelComputation()` nastaví flag pro paralelní simulaci trvalých poruch, `SetHaveFaultResult()` nastaví flag pro připojení Fault Monitoru a `SetFaultResFn()` nastaví název výstupního souboru, do kterého bude FaultMonitor zapisovat.
Další tři metody načítají data pro simulaci. `LoadInput()` načítá konfigurační soubor pro obecnou simulaci. Jeho formát a příkazy jsou popsány v následující podkapitole. LoadFaultList() načítá seznam možných poruch v obvodu. Formát je převzat z generátoru poruch Atalanta-M. Je popsán v příloze A. LoadPattern() načítá seznam testovacích vektorů.

Jeden vektor je řetězec obsahující “0“, “1“ nebo “x“. Příklad ukazuje seznam tří vektorů pro obvod s pěti vstupy:

```
001xx
11x01
x00x1
```

O převod popisu obvodu z HUBu se starají dvě metody. `CreateSystemCModule()` vytváří jednotlivé instance entit. V případě, že se jedná o entitu se strukturní architekturou, je volána rekurzivně. `CreateSystemCRepresentation()` vybere tzv. top-level entitu a vytvoří ji pomocí `CreateSystemCModule()`.
Metoda `Start()` spouští a řídí celou simulaci.

5.3.1. Konfigurační soubor pro obecnou simulaci

Konfigurační soubor obsahuje všechny informace potřebné pro simulaci obvodu. Je v textovém formátu a jednotlivé informace jsou zadávány pomocí příkazů. Seznam a popis příkazů je popsán v příloze A včetně jednoduchého příkladu.

5.3.2. Převod reprezentace z HUBu do SystemC

Převod reprezentace zajišťuje metoda `CreateSystemCRepresentation()`. Ta vytáhne z HUBu ukazatel na simulovanou entitu a zavolá `CreateSystemCModule()`. Ta nejprve zjistí architekturu entity a podle toho pak vytvoří příslušnou instanci jednoho z potomků třídy `CCircuitBase`. Pokud se jedná o strukturní architekturu, je tato metoda volána rekurzivně a poté, co se vrátí do místa odkud byla volána, se všechny instance propojí pomocí signálů.

obr. 13 – Proces převodu reprezentace obvodu z HUBu do SystemC

Vkládání sabotérů má na starost také metoda `CreateSystemCModule()`. Tedy pokud se nejedná o připojování sabotérů na primární vstupy a výstupy obvodu, ale o připojování mezi instancemi entit ve strukturní architektuře. Sabotéři jsou vytvářeni v době čtení faultlistu, při konstrukci obvodu v SystemC jsou pouze připojováni. Tento postup byl nutný, neboť v SystemC není možné obvody rozpojovat.

5.3.3. Řízení simulace

Průběh simulace je řízen z metody `Start()`. Ta nejprve připojí k primárním vstupům a výstupům signály a případně také sabotéry. Dále připraví seznam hodnot pro zápis na vstupní porty. K tomuto účelu byla zvolena multisada z knihovny STL.

```
std::multiset<TWriteValue, CWriteValueCmp> writeList;
```

`TWriteValue` je struktura nesoucí informace o vstupní hodnotě.

`Time` je čas, kdy má být vstupní hodnota zapsána na port. Podle této hodnoty jsou také záznamy v multisadě setříděny. SaboteurSignal je název sabotéra, který má aktivovat poruchu.
`Port` definuje port, na který se má zapisovat. Pokud je to číslo větší nebo rovné nule, jedná se o primární vstup obvodu. Pokud má port hodnotu -1, jedná se o zápis na globální hodinový signál, -2 podobně na globální resetový signál. -3 odpovídá řízení sabotéra -4 se používá pro resetování aktivních sabotérů, tedy při každém odsimulování jednatřiceti poruch.
`Value` je hodnota zapisována na porty. V případě primárních vstupů je to přímo hodnota, která se na tyto porty zapisuje, v případě sabotérů odpovídá hodnota "0" resp. "1" poruše s-a-0 resp. s-a-1.
`SabValue` je hodnota přiváděná buď na port sa0 nebo sa1 a je to vektor obsahující jednu 1 na místě ve vektoru, kde se porucha simuluje.

```
struct TWriteValue
{
      double time;
      std::string saboteurSignal;
      int port;
      char value;
      sc_lv<32> sabValue;
};
```

CWriteValueCmp je funktor zajišťující řazení multisady tak, aby hodnoty v sadě byly řazeny podle času vzestupně.

```
class CWriteValueCmp
{
public:
      bool operator() (const TWriteValue& t1, const TWriteValue& t2) const
            { return t1.time < t2.time; }
};
```

Metoda Start() pak od začátku do konce prochází tento seznam hodnot a postupně zapisuje hodnoty na porty. Simulace končí po odsimulování všech vstupních hodnot.

5.4. Konzolová aplikace cirsim.exe

Aplikace cirsim.exe je program pro obecnou simulaci obvodů (dovoluje obyčejnou simulaci bez poruch i s poruchami) . Je řízena konfiguračním souborem popsaným v odstavci 5.3.1 a několika parametry z příkazové řádky.
Aplikace je tvořena HUBem, modulem pro import, knihovnou SystemC a modulem pro rozhraní mezi SystemC a HUBem.
Program nejprve naimportuje data, poté načte konfigurační soubor, převede reprezentaci z HUBu do SystemC a spustí simulaci. Výstupem aplikace je vždy časový průběh signálů ve formátu VCD, sekvence výstupních vektorů při normální simulaci a volitelně report o simulaci poruch v textovém formátu.

Popis parametrů předávaných programu je v příloze A.

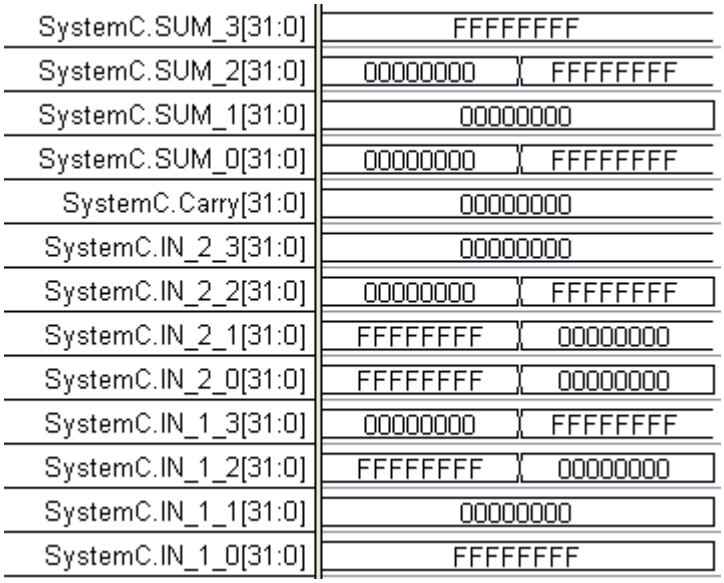
5.5. Konzolová aplikace faultsim.exe

Aplikace faultsim.exe je primárně zaměřena na simulaci poruch obvodů popsaných v BENCH formátu. Je tvořena stejnými moduly jako aplikace cirsim.exe. Na rozdíl od ní ale není řízena konfiguračním souborem. Stejně jako tato aplikace nejprve naimportuje obvod ze vstupního souboru, pak nastaví parametry simulace pro simulaci poruch, načte testovací vektory z dalšího vstupního souboru, načte seznam poruch (faultlist), převede reprezentaci z HUBu do SystemC a spustí simulaci. Výstupem aplikace je opět časový průběh signálů a report o testu trvalých poruch.

6. Testování, měření výkonu simulátoru

6.1. Demonstrace funkčnosti vlastní simulace

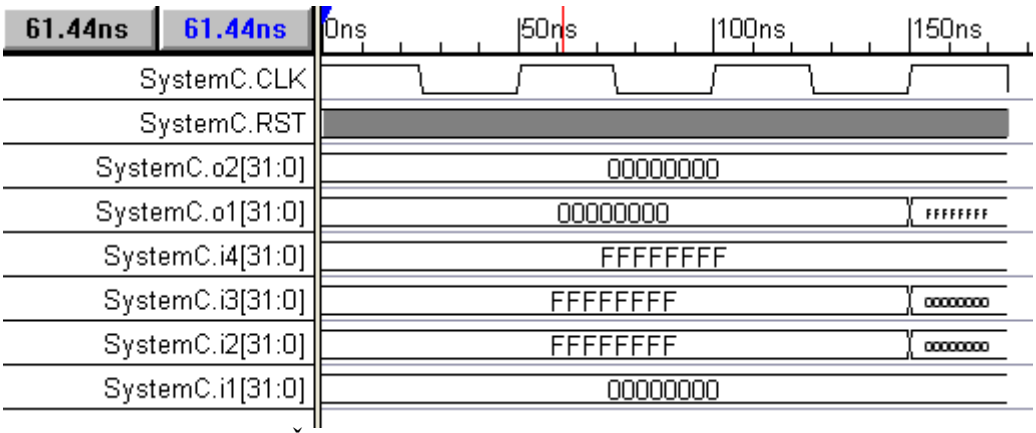
V tomto odstavci předvedu funkčnost simulátoru na simulaci čtyřbitové sčítačky popsané ve VHDL. Zdrojové kódy této sčítačky jsou v příloze E. Celý průběh simulace popisuje následující časový průběh signálů:



obr. 14 – Časový průběh signálů čtyřbitové sčítačky

Pro vstup prvního čtyřbitového čísla má obvod vstupy IN_1_3 až IN_1_0, pro vstup druhého čísla má vstupy IN_2_3 až IN_2_0. Výstupem obvodu je pětibitové číslo definované porty Carry a SUM_3 až SUM_0.
V průběhu je vidět, že simulace probíhala pro dva součty. Prvním součtem byla dvojková čísla 0101 a 0011.Výsledkem je správná hodnota 01000. Druhým součtem byla dvojková čísla 1001 a 0100. Jejich součtem je hodnota 1101. Připomínám, že hodnota “00000000“ odpovídá log. nule a hodnota “FFFFFFFF“ log. jedničce.

Dalším příkladem je funkčnost popisu konečných stavových automatů. Jedná se o obvod (bbara.kiss2) popsaný ve formátu KISS a jeho popis lze nalézt v příloze F.



obr. 15 – Časový průběh simulace konečného automatu

Zde má obvod čtyři primární vstupy i1 až i4 a dva výstupy o1 a o2. Navíc má obvod hodinový a resetový vstup. Z diagramu je vidět, že reset je po celou dobu simulace nedefinovaný a hodiny tikají s periodou 50ns. Na primární vstupy jsou hodnoty přiváděny také po 50ns. Na vstup jsou za sebou přiváděny tyto čtyři vstupní vektory:

0111
0111
0111
0001

Obvod je na začátku simulace v počátečním stavu st0. Po přivedení prvního vstupního vektoru (0111) se vyhledá v přechodové tabulce následující řádek:

```
-111 st0 st1 00
```

Obvod tedy přejde do stavu st1 a zapíše na výstup vektor 00. Po přivedení druhého vstupního vektoru se vyhledá tento řádek:

```
-111 st1 st2 00
```

Obvod přejde do stavu st2 a zapíše na výstup vektor 00. Po přivedení třetího vstupního vektoru se vyhledá tento řádek:

```
-111 st2 st3 00
```

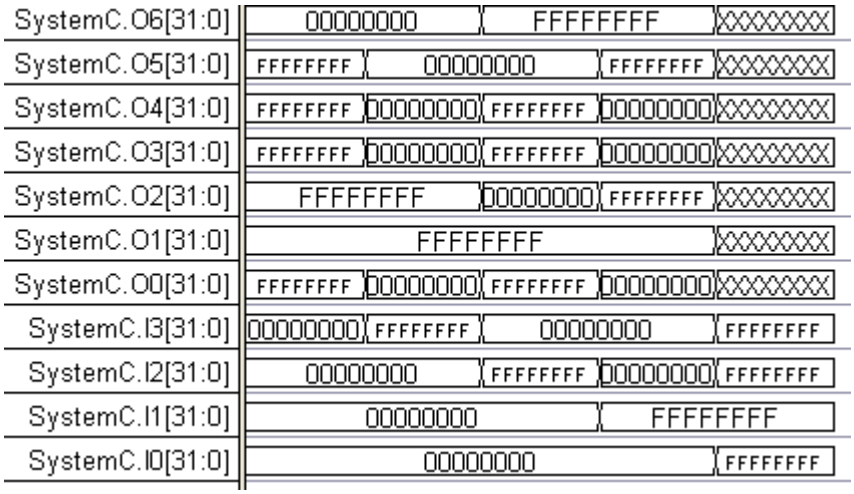
Obvod přejde do stavu st2 a zapíše na výstup vektor 00. Po přivedení posledního vstupního vektoru se vyhledá tento řádek:

```
--01 st3 st3 10
```

Konečným stavem obvodu je tedy st3 a na výstupu je na portu i1 hodnota 1 a na portu i2 hodnota 0.

Posledním demonstrovačním příkladem je simulace obvodu popsaného PLA popisem. Jedná se o dekoder popsaný následující pravdivostní tabulkou:

```
0000 1111110
0001 0110000
0010 1101101
0011 1111001
0100 0110011
0101 1011011
0110 1011111
0111 1110000
1000 1111111
1001 1111011
1010 -------
1011 -------
1100 -------
1101 -------
1110 -------
1111 -------
```



obr. 16 – Časový průběh simulace PLA popisu

Obvod má čtyři vstupy I0 až I3 a sedm výstupů O0 až O6. Na výstupech jsou odpovídající hodnoty pravdivostní tabulky, např. při vstupním vektoru “0000“ je výstupní vektor “1111110“ nebo při vstupním vektoru “1111“ je výstupní vektor “XXXXXXX“.

6.2. Výkonnost simulátoru poruch

Pro měření výkonosti simulátoru byly použity testovací úlohy (benchmarky) z kolekce ISCAS (jsou obsaženy na CD). Pro každoutestovací úlohu jsem měřil čas simulace pro 100, 1000 a 10000 vstupních vektorů. Úlohy začínající na písmeno „c“ jsou kombinační obvody, úlohy začínající na písmeno „s“ jsou sekvenční obvody s full-scanem (jsou popsány kombinačně).

Měření bylo prováděno na notebooku ASUS A6JC, s procesorem Intel Core Duo 1.66GHz, 1GB RAM, OS WinXP Home, release kompilace simulátoru s optimalizací na maximální výkon.

úloha	počet vstupů	počet výstupů	počet hradel	počet poruch	čas simulace[s]		
					100 vektorů	1000 vektorů	10000 vektorů
c432	36	7	160	524	3,33	32,33	316,2
c499	41	32	202	758	3,45	36,13	324,66
c880	60	26	383	942	7,24	74,19	740,75
s27	7	4	11	32	0,02	0,13	1,23
s298	17	20	119	308	0,55	5,53	56,28
s344	24	26	169	342	0,98	9,53	95,52
s1494	14	25	647	1506	13	125,58	1270,44
s1423	91	79	657	1515	21,09	198	2021
s838	67	34	390	857	3,51	31,22	307
s9234	247	250	5597	1897	84,53	816,9	8068
s13207.1	700	790	7979	9815	924,7	8868	???
s15850.1	611	684	9775	8543	1505	14492	???

Tab. 3 – Měření výkonnosti simulátoru

U každé úlohy je zmíněn počet primárních vstupů a výstupů, počet logických hradel popisující funkčnost obvodu a také počet simulovaných poruch. Protože se simuluje třicetjedna poruch zároveň, je při každé simulaci na primární vstupy přiváděno (*počet poruch / 31 + 1) x počet generovaných vektorů* vstupních vektorů.
Z výsledku měření je vidět, že časy simulací rostou lineárně s počtem přiváděných vstupních vektorů. U náročných úloh (s13207.1 nebo s15850) je dokonce viditelné, že při větším počtu vstupních vektorů docházík superlineárnímu zrychlení. Při srovnání úloh s1494 a s1423 je také vidět, že simulační čas výrazně ovlivňuje také počet primárních vstupů a výstupů, neboť obě úlohy mají přibližně stejný počet hradel i poruchových míst, ale s1423 má více vstupů i výstupů a simulační čas je téměř dvakrát tak větší.

Výkon simulátoru pro simulaci poruch však není ideální. Úlohy s13207.1 a s15850.1 pro 10000 vektorů jsem už ani neměřil, neboť by je trvalo změřit téměř celý den. Při ladění a porovnávání výsledků jsem používal modifikovaný nástroj Atalanta-M (pocházející z VirginiaTech University), který navíc slouží jako generátor vstupních vektorů a faultlistů. Vyzvednu zde hlavně jeden simulační čas tohoto nástroje a to u úlohy s9234 pro 10000 vstupních vektorů. Tento fault-simulátor mi vrátil na testovaném stroji výsledky v čase **3,53s**. Je to hlavně z toho důvodu, že Atalanta je přímo navržená pro simulaci poruch, zatímco SystemC jsou obecné simulační knihovny založené na plánování procesů. Následující tabulka ukazuje simulační časy složitějších úloh měřené v Atalantě.

úloha		**čas simulace[s]**	
	100 vektorů	**1000 vektorů**	**10000 vektorů**
s13207.1	0,156	1,031	7
s15850.1	0,203	1,36	9,515

Tab. 4 – Výkonnost simulátoru Atalanta

I zde si lze všimnout superlineárního zrychlení při větším počtu vstupních vektorů, ale hned na první pohled je vidět, že Atalanta má toto zrychlení mnohem vyšší.

7. Závěr

Cílem diplomové práce bylo zaintegrovat knihovnu SystemC do EDA systému, naimplementovat rozhraní, které bude převádět struktury z tohoto systému do SystemC a provádět simulaci, a otestovat tento výsledný simulátor na zkušebních úlohách. K tomu bylo nutné vytvořit spustitelný program, který dostane na vstup zkušební úlohu a nějaké parametry simulace. Všechny tyto cíle se podařilo splnit.

Během práce však přicházely další nápady, které práci obohatily. Jedná se o simulaci poruch s využitím paralelizace a napsání další aplikace specializované především na simulaci poruch. U této aplikace pak přibylo ještě generování LFSR vektorů.

Simulační modul tedy umí odsimulovat obvody popsané pomocí symbolických funkcí, PLA a FSM popisem a strukturním popisem. Strukturní popis umí samozřejmě simulovat heterogenní struktury, tedy část obvodu popsaná pomocí PLA, část třeba symbolickými funkcemi, atd. Simulace poruch je také navržena pro heterogenní struktury, je tedy možné i v těchto strukturách simulovat poruchy.

Pro další vývoj na EDA systému by bylo vhodné, aby obsahoval nějaký generátor seznamu poruch, neboť simulační modul žádnou takovou funkci nemá a seznamy poruch se musí dodávat z cizích aplikací. Aby navíc nebyla potřeba používat externí prohlížeč časových průběhů, mohl by se do systému naimplementovat prohlížeč VCD souborů, jehož formát je poměrně jednoduchý.

8. Seznam literatury

[1] - Schmidt, J. Dokumentace systému EduArd (na přiloženém CD)
[2] - Yang S. Logic Synthesis and Optimization Benchmarks User Guide Version 3.0. MCNS, Research Triangle Park,NC 27709
[3] - SystemC User's Guide Version 2.0, , Open SystemC Initiative 2002, Synopsys Inc. (na přiloženém CD)
[4] - Functional Specification for SystemC 2.0, Open SystemC Initiative 2002, Synopsys Inc. (na přiloženém CD)
[5] - SystemC 2.0.1 Language Reference Manual Revision 1.0, Open SystemC Initiative 2003, Synopsys Inc. (na přiloženém CD)
[6] - Describing Synthesizable RTL in SystemC (Version 1.1, January 2002), Synopsys Inc. (na přiloženém CD)
[7] - Vierhaus S., Sieber H. T. Fault Injection Techniques and their Accelerated Simulation in SystemC, Brandenburg University of Technology Cottbus (na přiloženém CD)
[8] - Atalanta-M 1.1b Manual, Virginia Polytechnic & State University

A. Ovládání konzolových aplikací

1. Aplikace cirsim.exe

cirsim.exe [parametry]

Parametr –I *fn*: importuje do HUBu obvod popsaný v souboru *fn.*
Parametr –S *fn*: načtení konfiguračního souboru *fn.*
Parametr –O *fn*: nastaví název výstupního souboru na *fn.* Pokud není zadán, soubor se nazývá „simulation“.

Výstupní soubory mají název podle parametru zadaného za parametrem -O:
.vcd - časový průběh simulace
.ovc - sekvence výstupních vektorů (pouze pokud probíhá normální simulace)

Příkazy konfiguračního souboru:

Příkaz „`.library`“ - jméno knihovny, ze které chceme simulovat obvod

Příkaz „`.model`“ – jméno entity ze zadané knihovny, kterou chceme simulovat

Příkaz „`.step`“ – časový krok mezi dvěma vstupními vektory (v nanosekundách)

Příkaz „`.i`“ – počet vstupů simulované entity

Příkaz „`.clk`“ – definice globálního hodinového signálu. Parametrem je perioda v nanosekundách.

Příkaz „`.reset`“ – definice signálu globálního resetu. Nutným parametrem je počet změn (překlopení z 0 na 1 a naopak). Pokud je tímto parametrem 0, reset zůstává po celou dobu simulace v nedefinovaném stavu. Pokud je prvním číslem kladná hodnota, tak definuje počet překlopení globálního resetu. V tom případě následují časy v nanosekundách, v kterých se hodnota resetu překlápí. První čas je překlopení z nuly do jedničky.

Příkaz „`.probes`“ – seznam signálů, které se mají sledovat a zapisovat do výstupního časového průběhu. Jsou to názvy signálů oddělené mezerou.

Příkaz „`.end_probes`“ – konec seznamu sond.

Příkaz „`.faultlist`“ – parametrem je soubor se seznamem poruch ve formátu atalanta-M.

Příkaz „`.fault_result`“ – parametrem je název výstupního souboru, kam se zapíší detekováné poruchy.

Příkaz „`.names`“ – blok vstupních vektorů. Za .names následují jména vstupních portů. Těch je tolik, kolik jich definuje příkaz „.i“. Vstupní vektory jsou řetězce oddělené řádkovačem ve formátu popsaném na přechozí straně.

Příkaz „`.end`“ – konec bloku vstupních vektorů a zároveň celého konfiguračního souboru.

Příklad:

```
.library kiss_library
.model bbara
.step 50
.clk 50
.reset 0
.i 4
.probes
i1 i2 i3 i4
o1 o2
.end_probes
.names i1 i2 i3 i4
0111
0111
0111
0001
.end
```

2. Aplikace faultsim.exe

faultsim [možnosti] circuit_file

Parametr circuit_file: název souboru s popisem obvodu

Parametr -S: Není potřebný, je zde pouze pro kompatibilitu s atalantou.
Parametr -t *fn*: Testovací vektory jsou načteny ze souboru *fn*.Vyžadovaný parametr, pokud není použitý parametr -l.
Parametr -f *fn*: Seznam poruch je načten ze souboru *fn*. Vyžadovaný parametr."
Parametr -P *fn*: Informace o simulaci poruch jsou zapsány do souboru *fn*. Vyžadovaný parametr.
Parametr –l *poly seed num*: Generuje num vstupních LFSR vektorů. Parametry poly a seed jsou v osmičkové soustavě.

B. Faultlist formát

Příklad:

```
gate_A->gate_B /1
gate_A->gate_B /0
gate_A /1
gate_B /1
```

V tomto příkladě máme dvě hradla, gate_A a gate_B.
První řádka "gate_A->gate_B /1" popisuje s-a-1 poruchu na vstupním signálu hradla gate_B, které je připojeno na výstup hradla gate_A. Na druhém řádku je na stejném signálu popsána porucha s-a-0. Třetí řádek popisuje poruchu s-a-1 na výstupu hradla gate_A a na čtvrtém řádku je popsána porucha s-a-1 na výstupu hradla gate_B.

C. SystemC scheduler

The scheduler's task is to determine the order of execution of processes within the design based on the event sensitivity of the processes and the event notifications which occur.

The SystemC scheduler has support for both software and hardware-oriented modeling.

Similar to VHDL and Verilog, the SystemC scheduler supports delta cycles. A delta cycle is comprised of separate evaluate and update phases, and multiple delta cycles may occur at a particular simulated time. Delta cycles are useful for modeling fully-distributed, timesynchronized computation as found for example in RTL hardware. In SystemC, using notify() with a zero time argument causes the event to be notified in the evaluate phase of the next delta cycle, while a call to request_update() causes the update() method to be called in the update phase of the current delta cycle. Using these facilities, channels which model the behavior of hardware signals can be constructed.

SystemC also supports timed event notifications. Timed event notifications are specified using notify() with a time argument. A timed notification causes the specified event to be notified at a specified time in the future. Timed notifications are found in both VHDL and Verilog and are also useful in modeling software systems.

Lastly, SystemC supports immediate event notifications, which are specified using notify() with no arguments. An immediate event notification causes processes which are sensitive to the event to be made immediately ready to run (i.e. ready to run in the current evaluate phase.) Immediate event notifications are useful for modeling software systems and operating systems, which lack the concept of delta cycles.

The following steps outline the execution of the SystemC scheduler. More detailed pseudocode for the scheduler is included in Appendix A of this document.

1) Initialization Phase – Execute all processes (except SC_CTHREADs) in an unspecified order.

2) Evaluate Phase – Select a process that is ready to run and resume its execution. This may cause immediate event notifications to occur, which may result in additional processes being made ready to run in this same phase.

3) If there are still processes ready to run, go to step 2.

4) Update Phase – Execute any pending calls to update() resulting from request_update() calls made in step 2.

5) If there are pending delayed notifications, determine which processes are ready to run due to the delayed notifications and go to step 2.

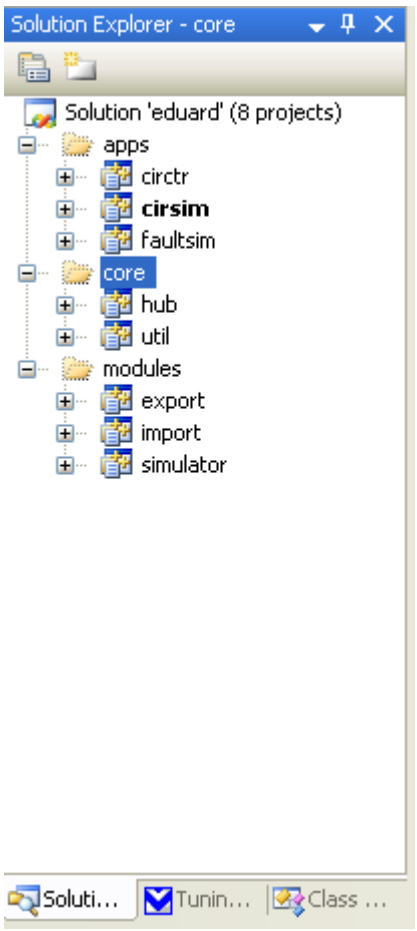
6) If there are no more timed notifications, simulation is finished.

7) Advance the current simulation time to the earliest pending timed notification.

8) Determine which processes are ready to run due to the events that have pending notifications at the current time. Go to step 2.

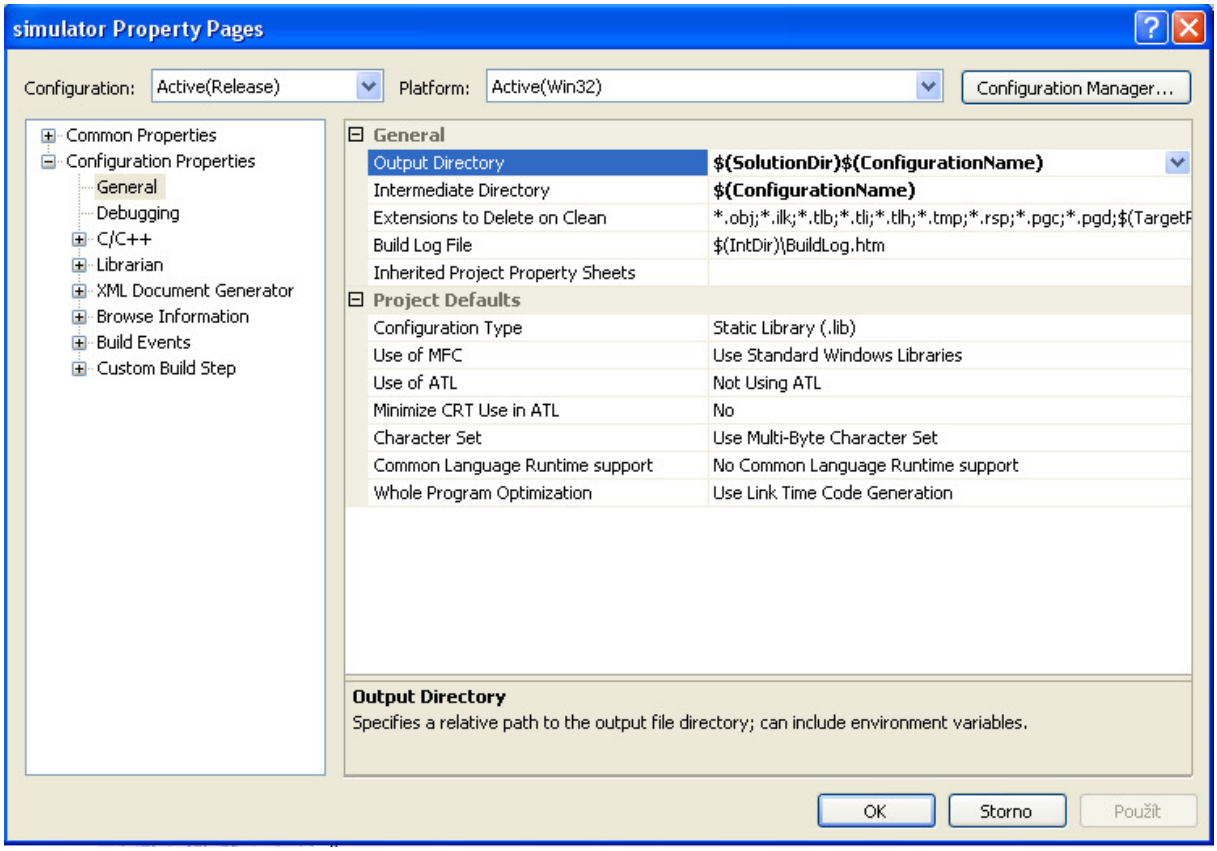
D. Nastavení projektu v MSVC 2005

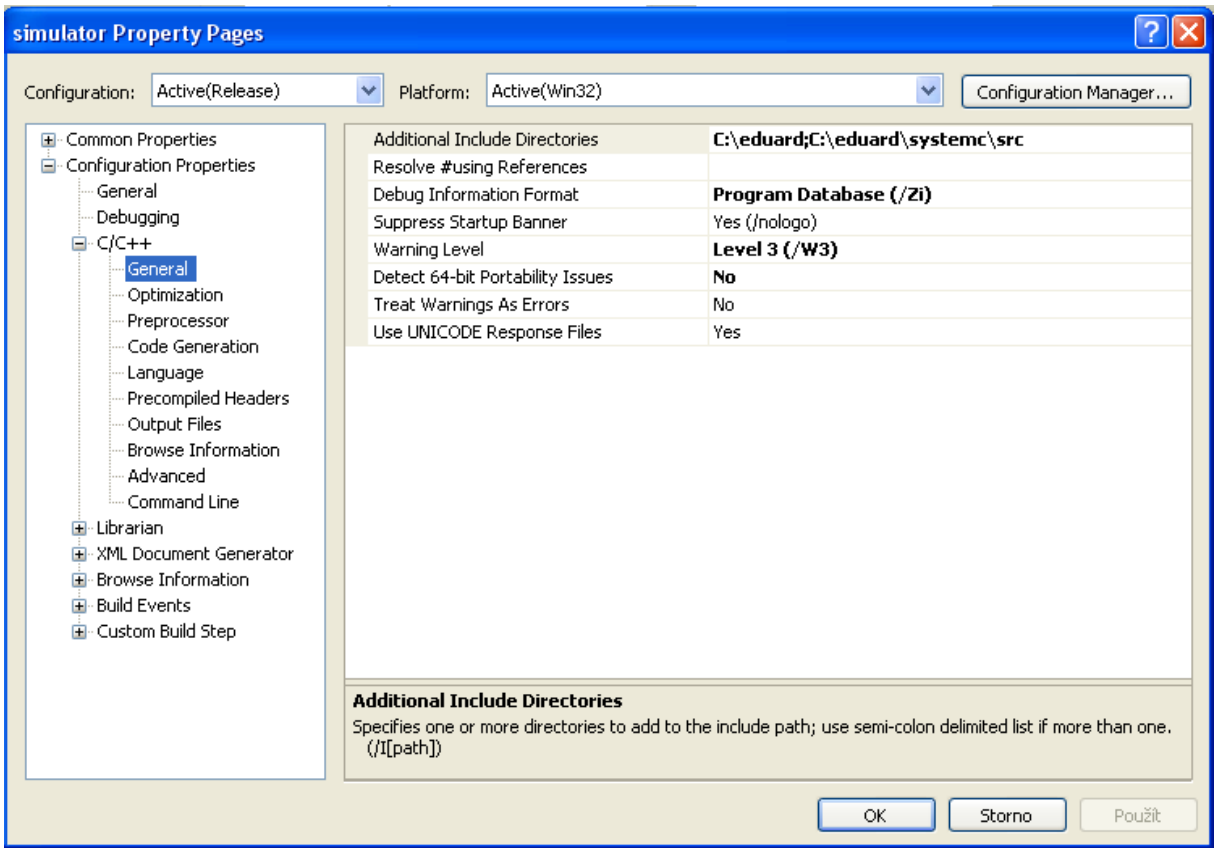
Celý projekt je rozdělen na tři části tak, jak to dokumentuje následující obrázek.

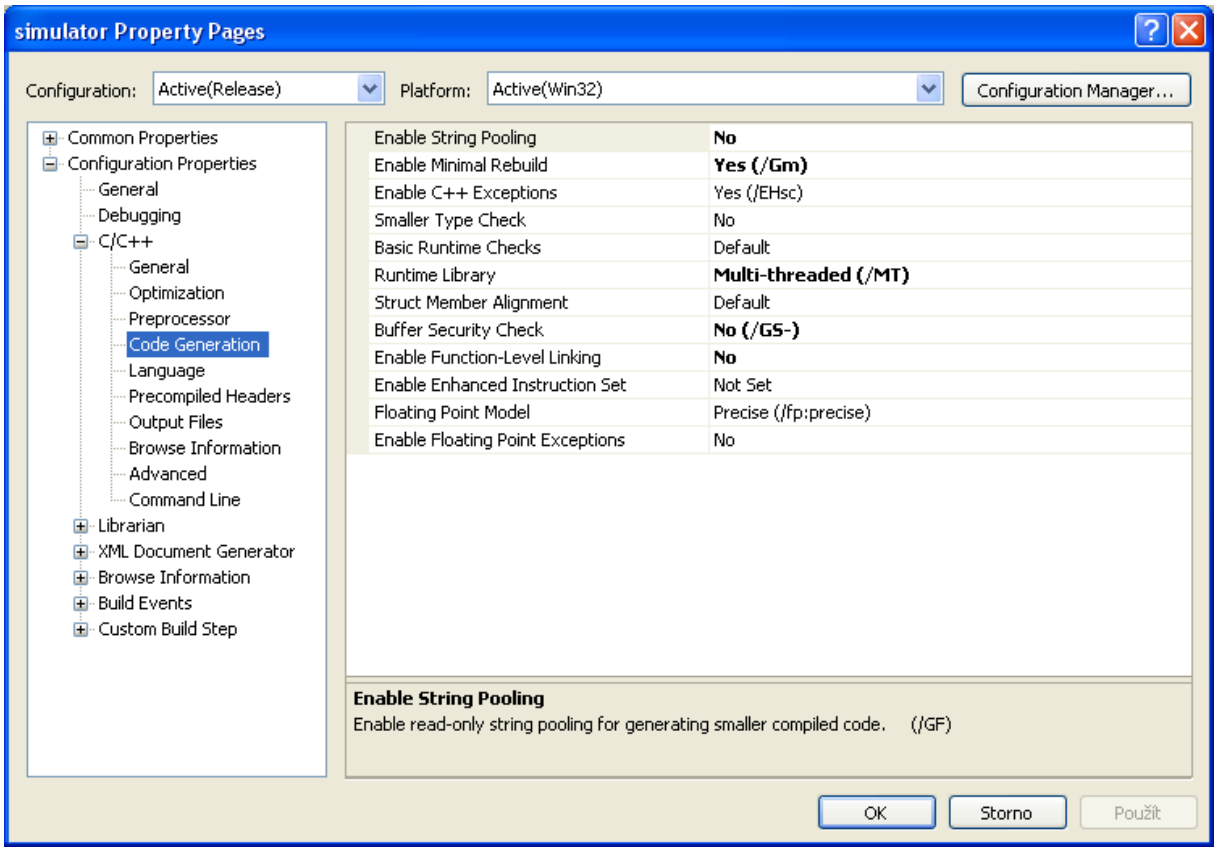


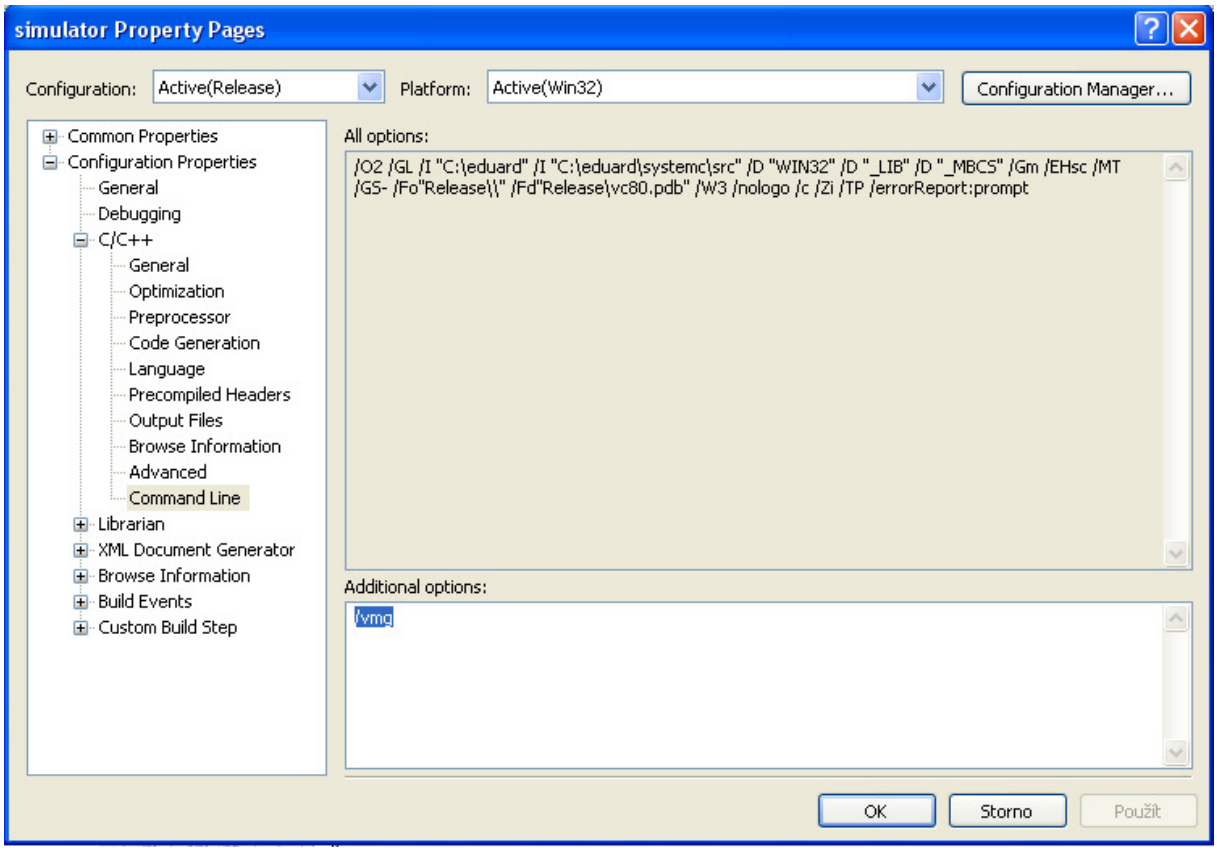
První složka „apps“ obsahuje projekty, které se mají kompilovat jako spustitelné .exe soubory. Další složky „core“ a „modules“ obsahují projekty, které se mají kompilovat jako .lib statické knihovny. Na následujících stránkách jsou nastavení projektu pro DEBUG i RELEASE kompilace zvlášť pro statické knihovny a zvlášť pro spustitelné aplikace.

Nastavení RELEASE kompilace statických knihoven

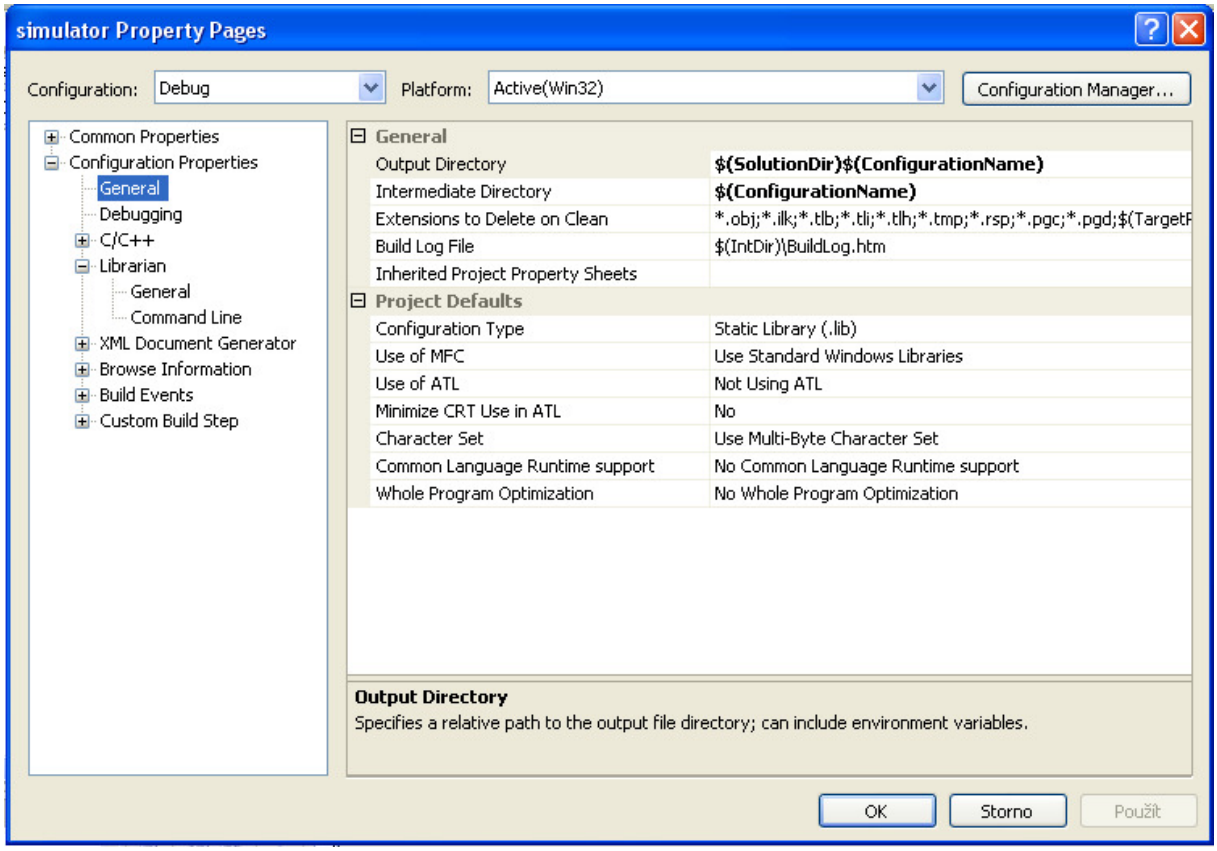


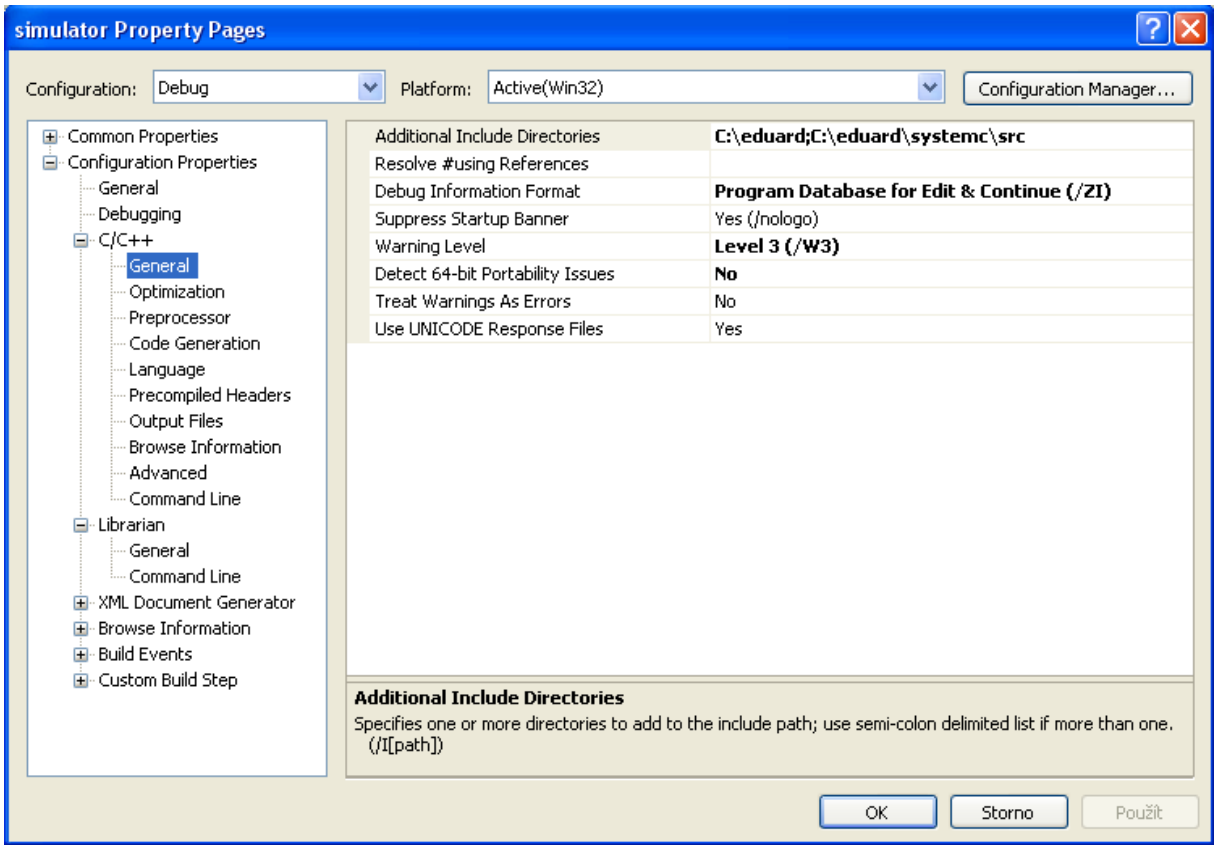


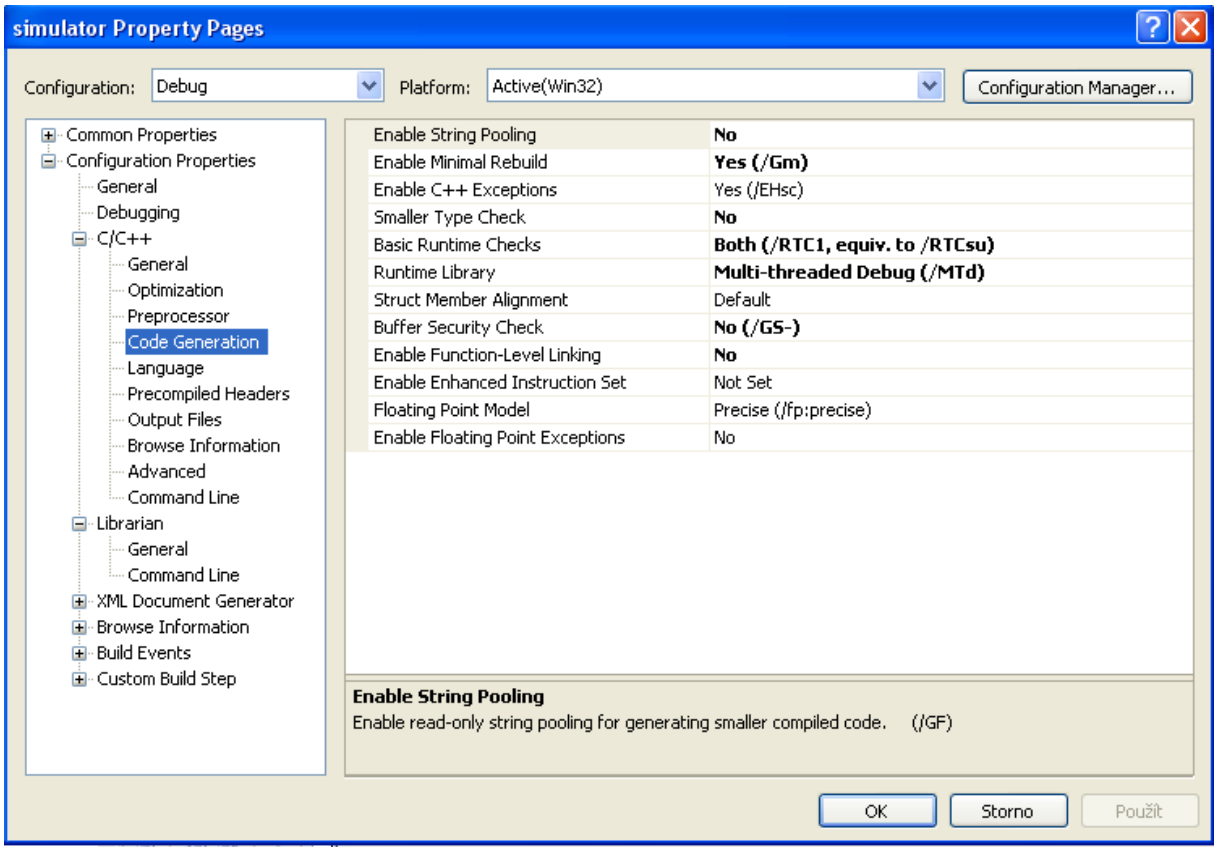

simulator Property Pages
Configuration: Active(Release)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Browse Information
Advanced
Command Line
Librarian
XML Document Generator
Browse Information
Build Events
Custom Build Step
Enable String Pooling No
Enable Minimal Rebuild Yes (/Gm)
Enable C++ Exceptions Yes (/EHsc)
Smaller Type Check No
Basic Runtime Checks Default
Runtime Library Multi-threaded (/MT)
Struct Member Alignment Default
Buffer Security Check No (/GS-)
Enable Function-Level Linking No
Enable Enhanced Instruction Set Not Set
Floating Point Model Precise (/fp:precise)
Enable Floating Point Exceptions No
Enable String Pooling
Enable read-only string pooling for generating smaller compiled code. (/GF)
OK
Storno
Použít

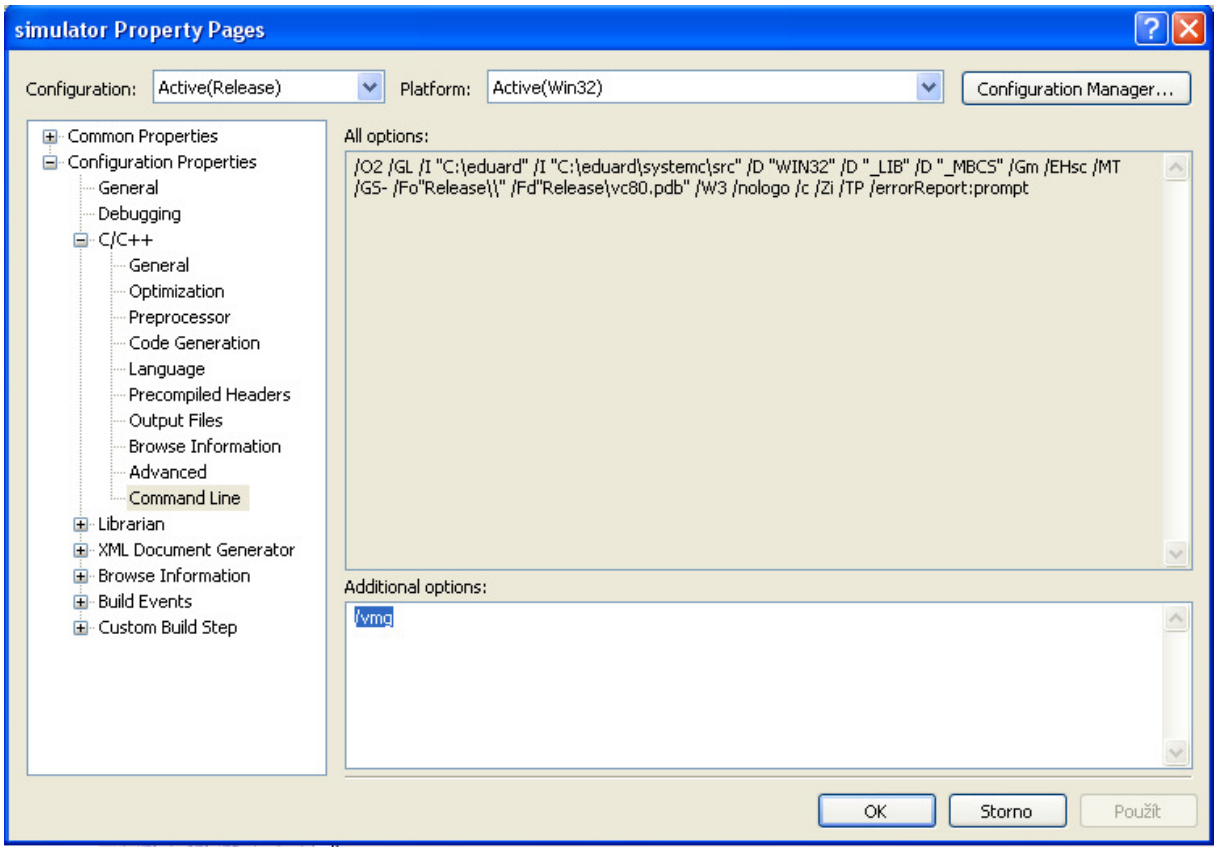

simulator Property Pages
Configuration: Active(Release)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Browse Information
Advanced
Command Line
Librarian
XML Document Generator
Browse Information
Build Events
Custom Build Step
All options:
/O2 /GL /I "C:\eduard" /I "C:\eduard\systemc\src" /D "WIN32" /D "_LIB" /D "_MBCS" /Gm /EHsc /MT /GS- /Fo"Release\\" /Fd"Release\vc80.pdb" /W3 /nologo /c /Zi /TP /errorReport:prompt
Additional options:
/vmg
OK
Storno
Použít

Nastavení DEBUG kompilace statických knihoven

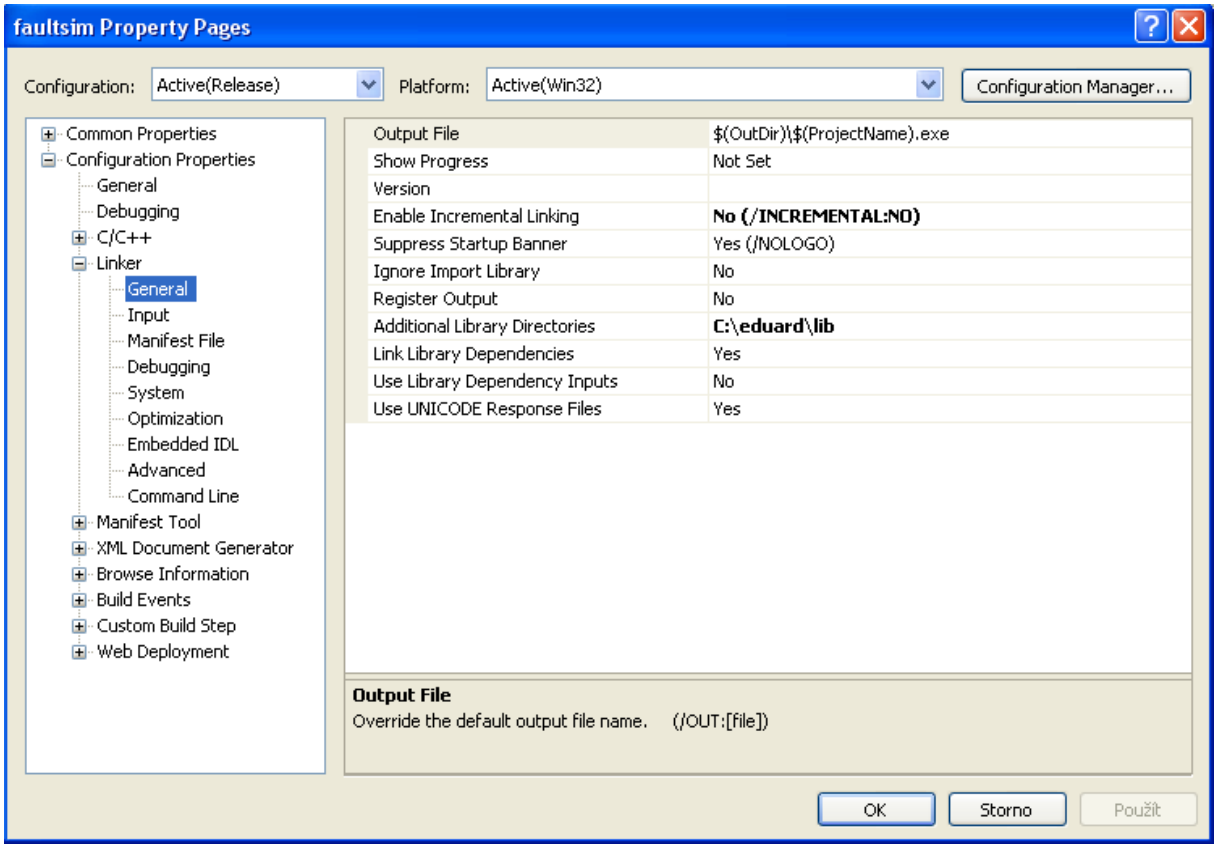


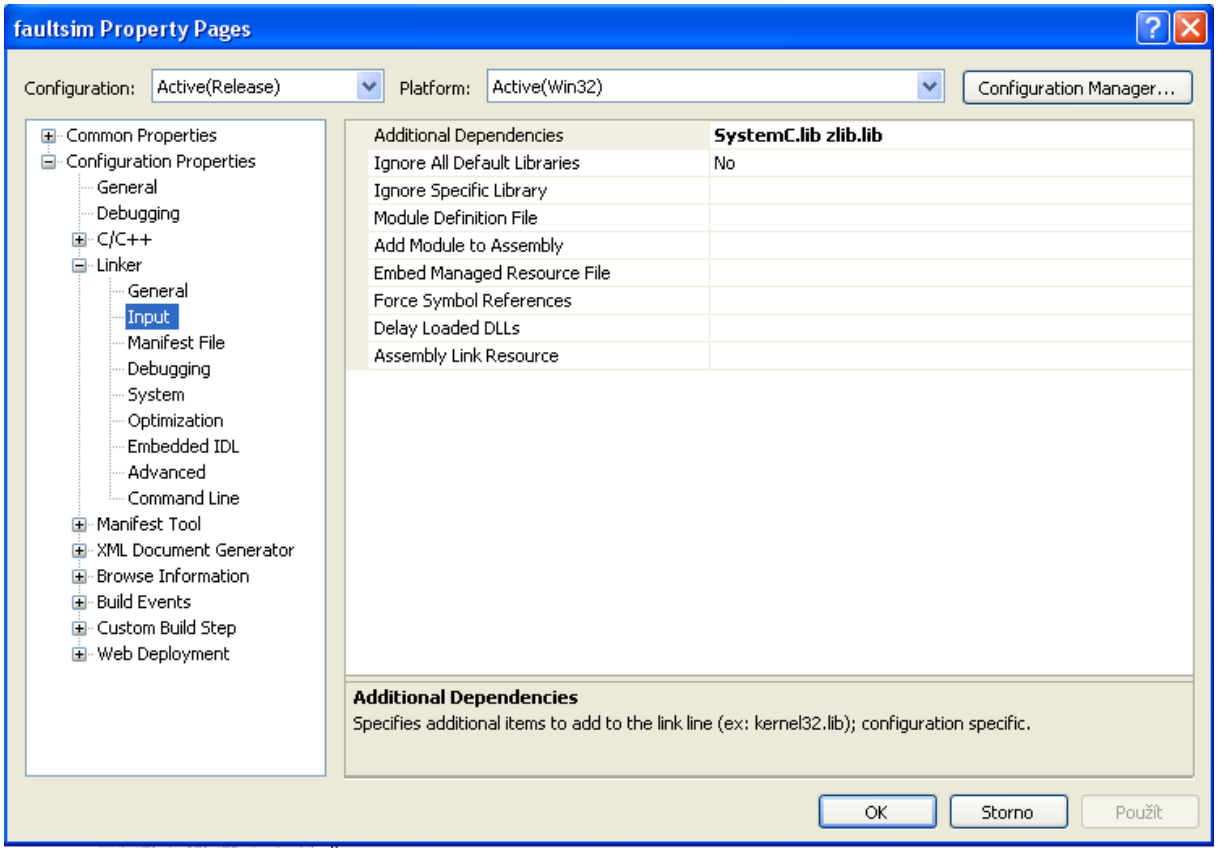


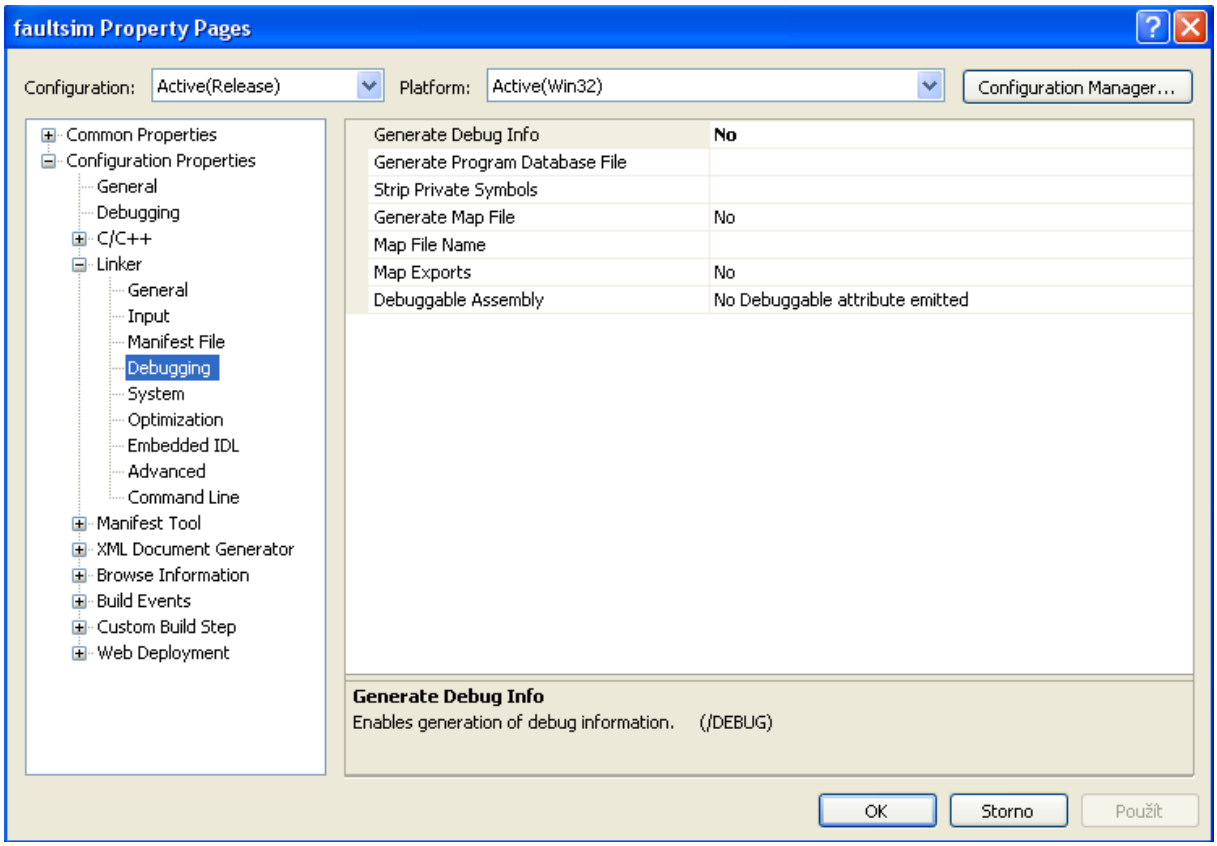

simulator Property Pages
Configuration: Debug
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Browse Information
Advanced
Command Line
Librarian
General
Command Line
XML Document Generator
Browse Information
Build Events
Custom Build Step
Enable String Pooling No
Enable Minimal Rebuild Yes (/Gm)
Enable C++ Exceptions Yes (/EHsc)
Smaller Type Check No
Basic Runtime Checks Both (/RTC1, equiv. to /RTCsu)
Runtime Library Multi-threaded Debug (/MTd)
Struct Member Alignment Default
Buffer Security Check No (/GS-)
Enable Function-Level Linking No
Enable Enhanced Instruction Set Not Set
Floating Point Model Precise (/fp:precise)
Enable Floating Point Exceptions No
Enable String Pooling
Enable read-only string pooling for generating smaller compiled code. (/GF)
OK
Storno
Použít

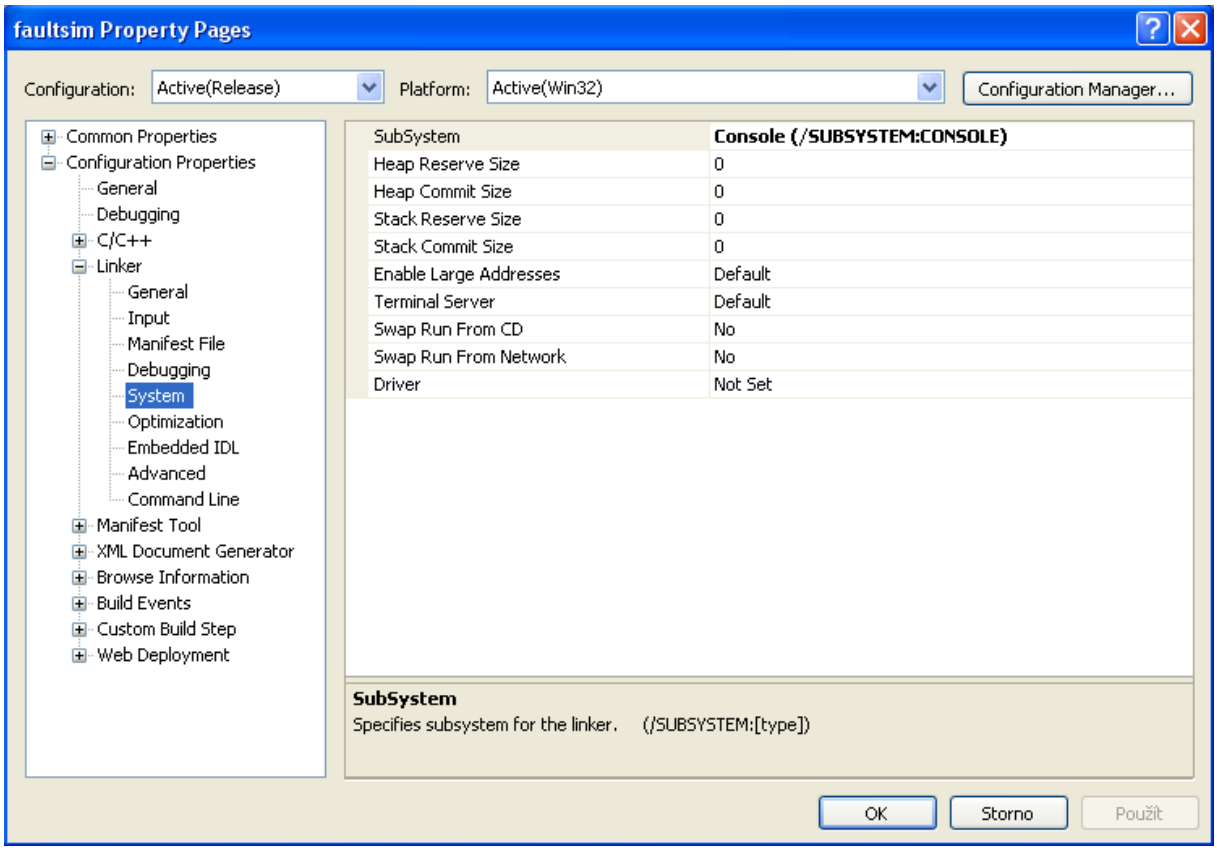

simulator Property Pages
Configuration: Active(Release)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Browse Information
Advanced
Command Line
Librarian
XML Document Generator
Browse Information
Build Events
Custom Build Step
All options:
/O2 /GL /I "C:\eduard" /I "C:\eduard\systemc\src" /D "WIN32" /D "_LIB" /D "_MBCS" /Gm /EHsc /MT /GS- /Fo"Release\\" /Fd"Release\vc80.pdb" /W3 /nologo /c /Zi /TP /errorReport:prompt
Additional options:
/vmg
OK
Storno
Použít

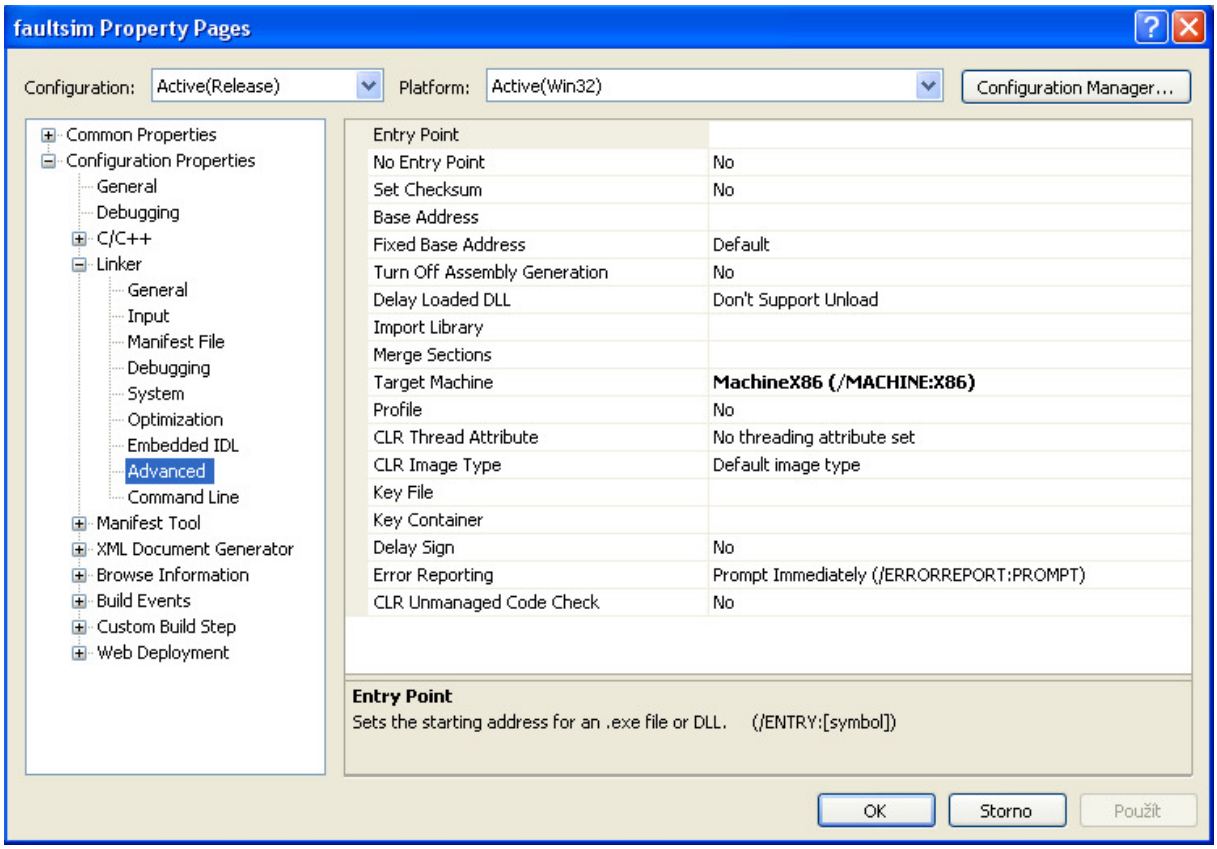
Nastavení RELEASE kompilace spustitelných aplikací (pouze linker)



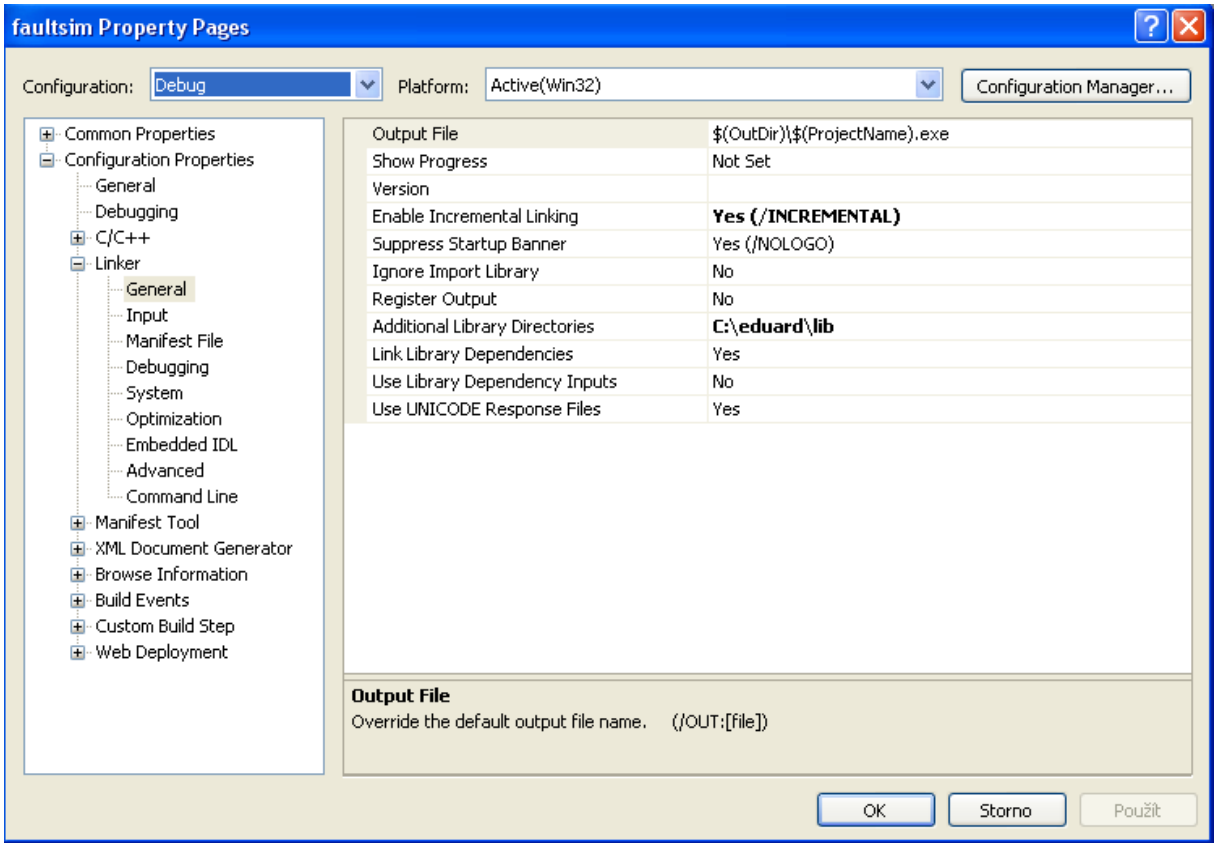


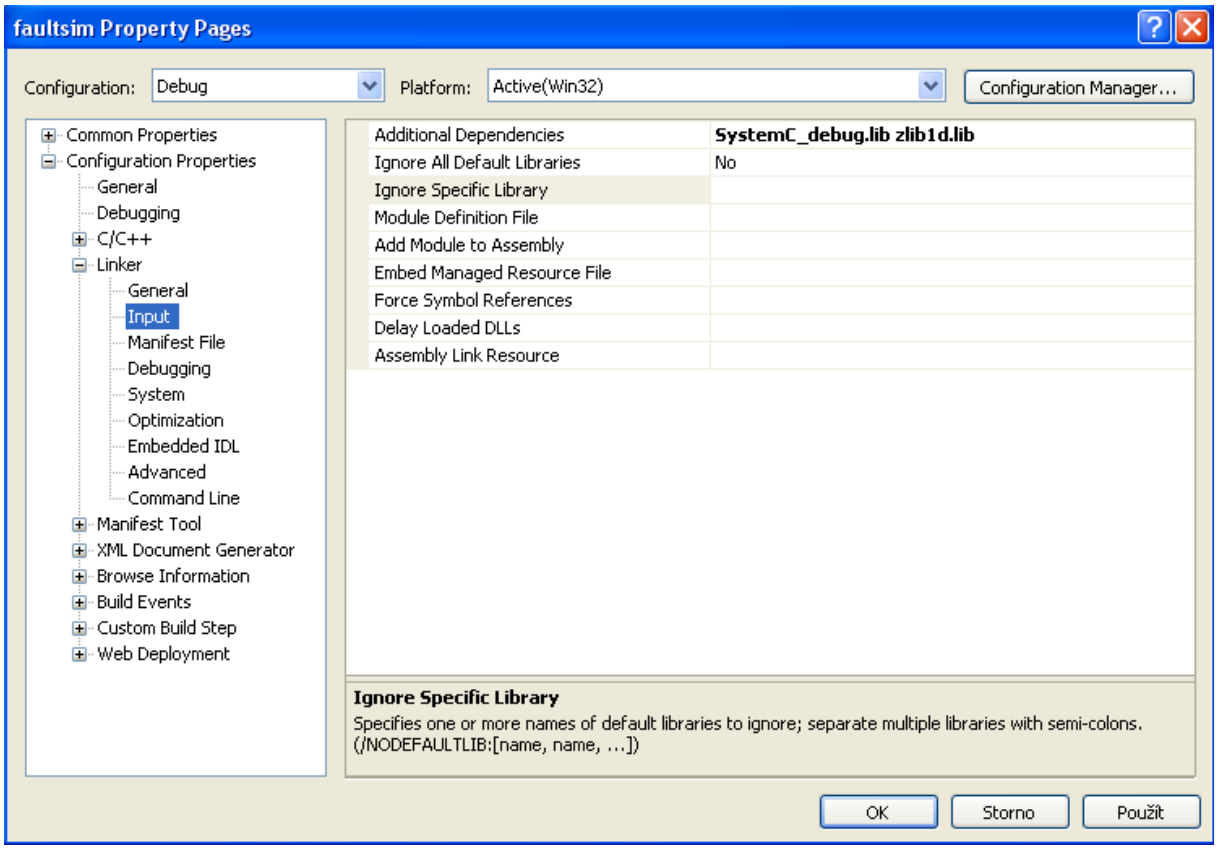

faultsim Property Pages
Configuration: Active(Release)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Advanced
Command Line
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Web Deployment
Generate Debug Info No
Generate Program Database File
Strip Private Symbols
Generate Map File No
Map File Name
Map Exports No
Debuggable Assembly No Debuggable attribute emitted
Generate Debug Info
Enables generation of debug information. (/DEBUG)
OK
Storno
Použít

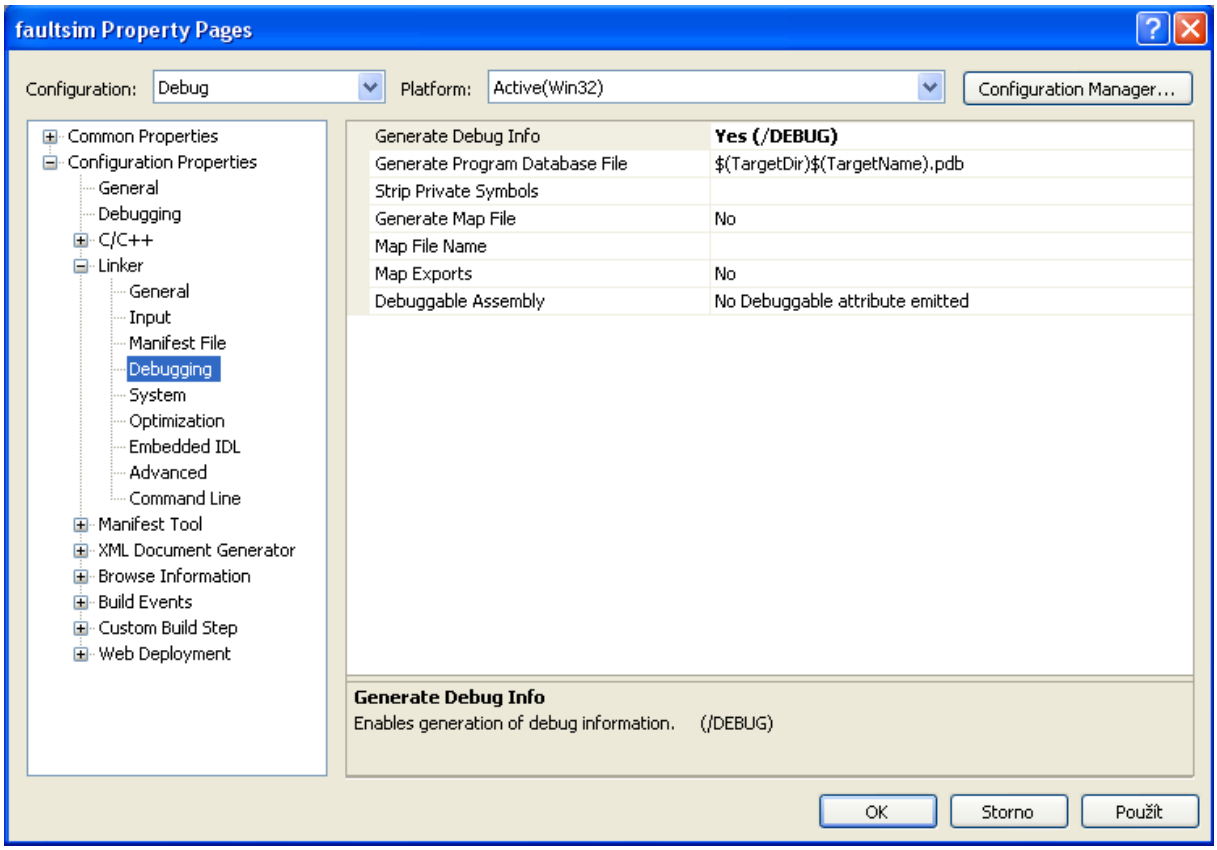

faultsim Property Pages
Configuration: Active(Release)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Advanced
Command Line
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Web Deployment
SubSystem Console (/SUBSYSTEM:CONSOLE)
Heap Reserve Size 0
Heap Commit Size 0
Stack Reserve Size 0
Stack Commit Size 0
Enable Large Addresses Default
Terminal Server Default
Swap Run From CD No
Swap Run From Network No
Driver Not Set
SubSystem
Specifies subsystem for the linker. (/SUBSYSTEM:[type])
OK
Storno
Použít

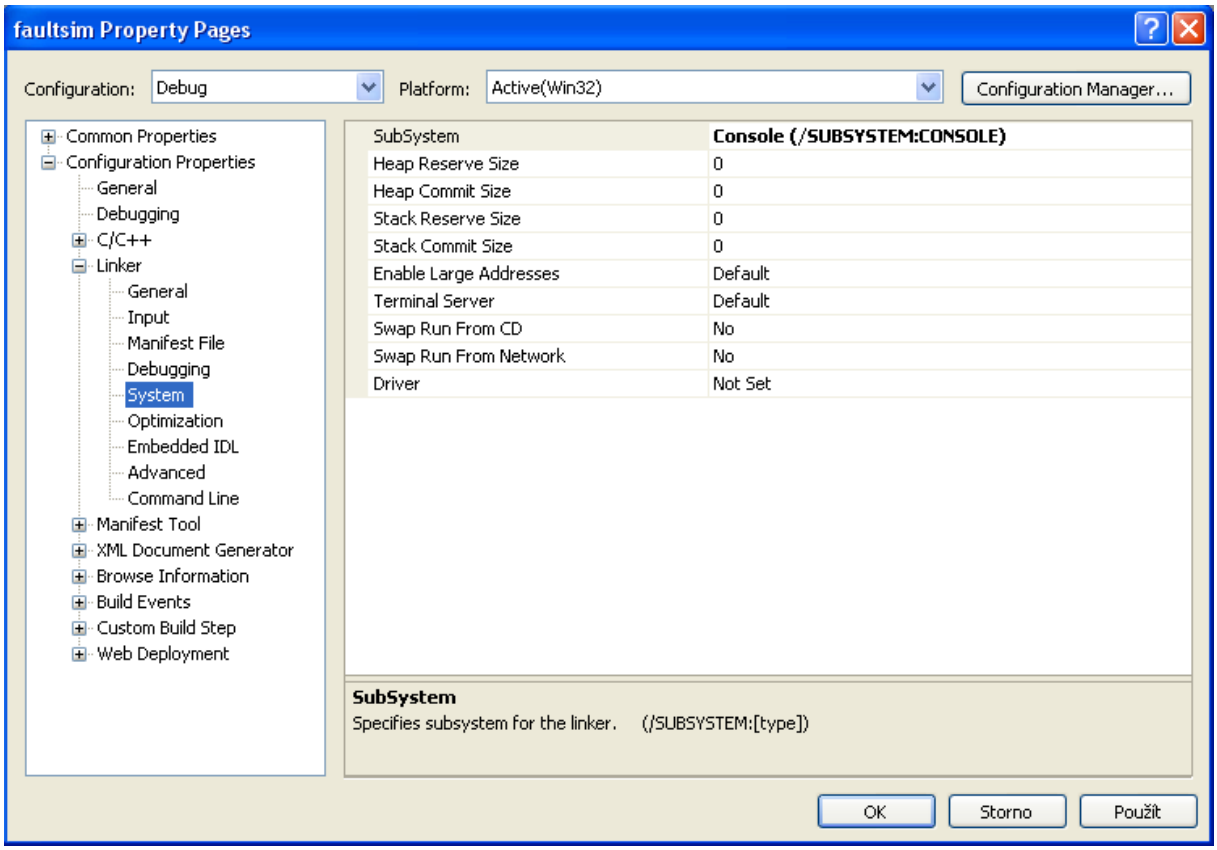


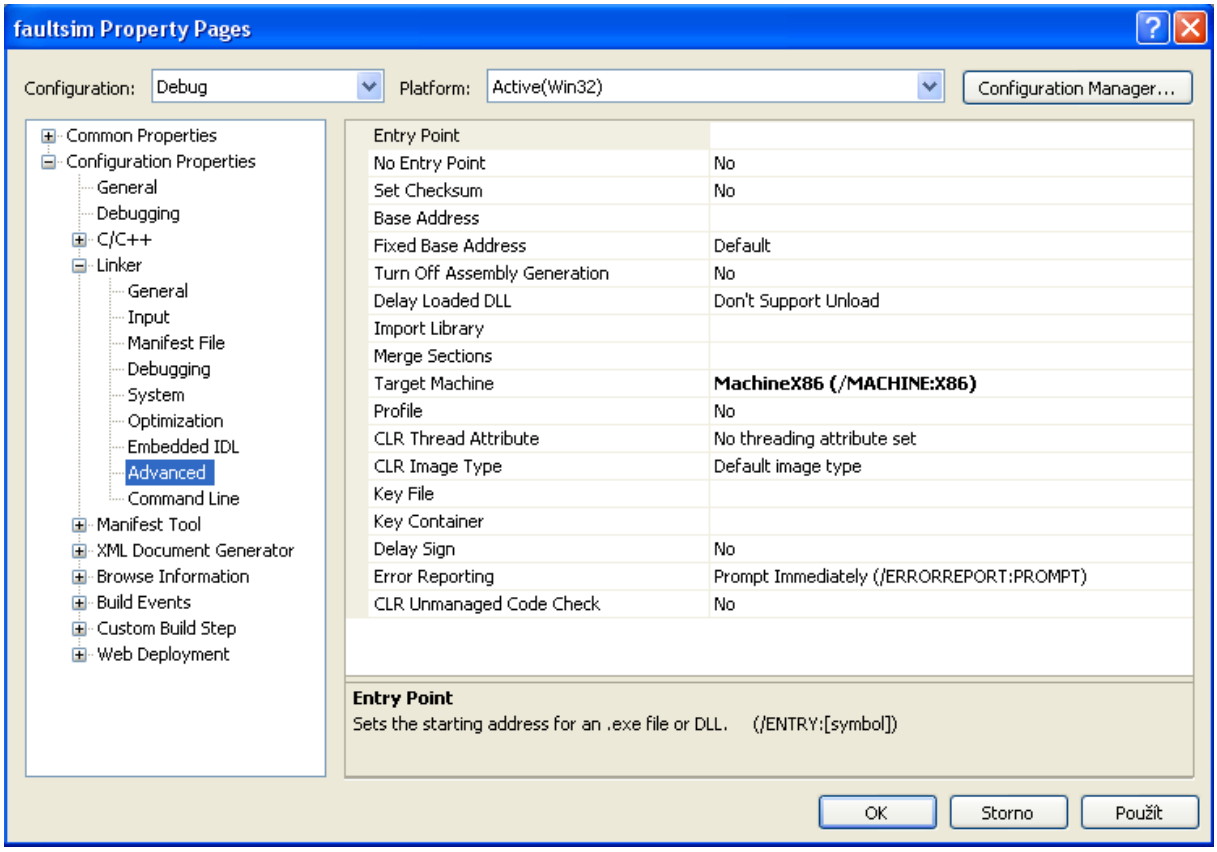
Nastavení DEBUG kompilace spustitelných aplikací (pouze linker)




faultsim Property Pages
Configuration: Debug
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Advanced
Command Line
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Web Deployment
Additional Dependencies
SystemC_debug.lib zlib1d.lib
Ignore All Default Libraries
No
Ignore Specific Library
Module Definition File
Add Module to Assembly
Embed Managed Resource File
Force Symbol References
Delay Loaded DLLs
Assembly Link Resource
Ignore Specific Library
Specifies one or more names of default libraries to ignore; separate multiple libraries with semi-colons.
(/NODEFAULTLIB:[name, name, ...])
OK
Storno
Použít


faultsim Property Pages
Configuration: Debug
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Advanced
Command Line
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Web Deployment
Generate Debug Info
Yes (/DEBUG)
Generate Program Database File
$(TargetDir)$(TargetName).pdb
Strip Private Symbols
Generate Map File
No
Map File Name
Map Exports
No
Debuggable Assembly
No Debuggable attribute emitted
Generate Debug Info
Enables generation of debug information. (/DEBUG)
OK
Storno
Použít


faultsim Property Pages
Configuration: Debug
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Advanced
Command Line
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Web Deployment
SubSystem | Console (/SUBSYSTEM:CONSOLE)
Heap Reserve Size | 0
Heap Commit Size | 0
Stack Reserve Size | 0
Stack Commit Size | 0
Enable Large Addresses | Default
Terminal Server | Default
Swap Run From CD | No
Swap Run From Network | No
Driver | Not Set
SubSystem
Specifies subsystem for the linker. (/SUBSYSTEM:[type])
OK
Storno
Použít


faultsim Property Pages
Configuration: Debug
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Advanced
Command Line
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Web Deployment
Entry Point
No Entry Point | No
Set Checksum | No
Base Address
Fixed Base Address | Default
Turn Off Assembly Generation | No
Delay Loaded DLL | Don't Support Unload
Import Library
Merge Sections
Target Machine | MachineX86 (/MACHINE:X86)
Profile | No
CLR Thread Attribute | No threading attribute set
CLR Image Type | Default image type
Key File
Key Container
Delay Sign | No
Error Reporting | Prompt Immediately (/ERRORREPORT:PROMPT)
CLR Unmanaged Code Check | No
Entry Point
Sets the starting address for an .exe file or DLL. (/ENTRY:[symbol])
OK
Storno
Použít

E. Zdrojové kódy 4bitové sčítačky (VHDL)

Soubor halfadder.vhdl:

```
library IEEE;
use ieee.Std_Logic_1164.all;
use ieee.numeric_std.all;

entity halfadder is
  port (
     A: in std_logic;
     B: in std_logic;
     S: out std_logic;
     Carry: out std_logic
    );
end halfadder;

ARCHITECTURE halfadder_arch OF halfadder IS

BEGIN
      S <= A xor B;
      Carry <= A and B;

END halfadder_arch;
```

Soubor adder.vhdl:

```
library IEEE;
use ieee.Std_Logic_1164.all;
use ieee.numeric_std.all;

entity adder is
  port (
     A: in std_logic;
     B: in std_logic;
     C: in std_logic;
     SUM: out std_logic;
     Carry: out std_logic
    );
end adder;

ARCHITECTURE adder_arch OF adder IS

signal c1, c2, s: std_logic;

component halfadder
  port (
     A: in std_logic;
     B: in std_logic;
     S: out std_logic;
     Carry: out std_logic
    );
end component;

BEGIN
      H1_map: halfadder
            port map(A => A,
                     B => B,
                     Carry => c1,
                     S => s);
```

```
      H2_map: halfadder
            port map(A => C,
                        B => s,
                        Carry => c2,
                        S => SUM);

      Carry <= c1 or c2;

END adder_arch;
```

Soubor 4bitadder.vhdl:

```
library IEEE;
use ieee.Std_Logic_1164.all;
use ieee.numeric_std.all;

entity FourBitAdder is
  port (
     IN_1: in std_logic_vector(3 downto 0);
     IN_2: in std_logic_vector(3 downto 0);
     SUM: out std_logic_vector(3 downto 0);
     Carry: out std_logic
    );
end FourBitAdder;

ARCHITECTURE FourBitAdder_arch OF FourBitAdder IS

signal c: std_logic_vector(2 downto 0);

component halfadder
  port (
     A: in std_logic;
     B: in std_logic;
     S: out std_logic;
     Carry: out std_logic
    );
end component;

component adder
  port (
     A: in std_logic;
     B: in std_logic;
     C: in std_logic;
     SUM: out std_logic;
     Carry: out std_logic
    );
end component;

BEGIN
      H_map: halfadder
            port map(A => IN_1(0),
                        B => IN_2(0),
                        Carry => c(0),
                        S => SUM(0));
      A1_map: adder
            port map(A => IN_1(1),
                        B => IN_2(1),
                        C => c(0),
                        Carry => c(1),
                        SUM => SUM(1));
      A2_map: adder
```

```
            port map(A => IN_1(2),
                        B => IN_2(2),
                        C => c(1),
                        Carry => c(2),
                        SUM => SUM(2));
      A3_map: adder
            port map(A => IN_1(3),
                        B => IN_2(3),
                        C => c(2),
                        Carry => Carry,
                        SUM => SUM(3));

END FourBitAdder_arch;
```

F. Příklad konečného stavového automatu (bbara.kiss2)

```
.i 4
.o 2
.p 60
.s 10
--01 st0 st0 00
--10 st0 st0 00
--00 st0 st0 00
0011 st0 st0 00
-111 st0 st1 00
1011 st0 st4 00
--01 st1 st1 00
--10 st1 st1 00
--00 st1 st1 00
0011 st1 st0 00
-111 st1 st2 00
1011 st1 st4 00
--01 st2 st2 00
--10 st2 st2 00
--00 st2 st2 00
0011 st2 st1 00
-111 st2 st3 00
1011 st2 st4 00
--01 st3 st3 10
--10 st3 st3 10
--00 st3 st3 10
0011 st3 st7 00
-111 st3 st3 10
1011 st3 st4 00
--01 st4 st4 00
--10 st4 st4 00
--00 st4 st4 00
0011 st4 st0 00
-111 st4 st1 00
1011 st4 st5 00
--01 st5 st5 00
--10 st5 st5 00
--00 st5 st5 00
0011 st5 st4 00
-111 st5 st1 00
1011 st5 st6 00
--01 st6 st6 01
--10 st6 st6 01
--00 st6 st6 01
0011 st6 st7 00
-111 st6 st1 00
1011 st6 st6 01
--01 st7 st7 00
--10 st7 st7 00
--00 st7 st7 00
0011 st7 st8 00
-111 st7 st1 00
1011 st7 st4 00
--01 st8 st8 00
--10 st8 st8 00
--00 st8 st8 00
0011 st8 st9 00
-111 st8 st1 00
```

```
1011 st8 st4 00
--01 st9 st9 00
--10 st9 st9 00
--00 st9 st9 00
0011 st9 st0 00
-111 st9 st1 00
1011 st9 st4 00
```

G. Obsah přiloženého CD

Umístění adresáře/souboru	**Obsah**
/atalanta	generátor vstupních vektorů a simulátor poruch
/benchs	testovací úlohy
/demos	demostrační ukázky implementovaných aplikací
/doc	použitá literatura
/gtkwave	freewarový prohlížeč VCD souborů
/lib/systemc-2.2.0.tgz	knihovna SystemC včetně zdrojových kódů
/src	zdrojové kódy s projektem v MSVC 2005
/dp.doc	tato diplomová práce
/dp.pdf	tato diplomová práce