VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

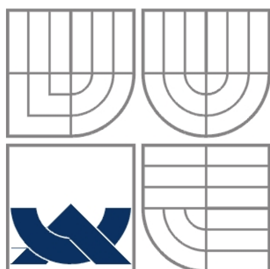
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VÝVOJOVÉ PROSTŘEDÍ PRO NANOPROCESORY

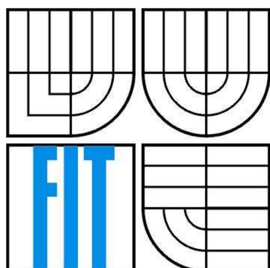
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Marek Plhák
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VÝVOJOVÉ PROSTŘEDÍ PRO NANOPROCESORY

DEVELOPMENT ENVIRONMENT FOR NANOPROCESORS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Marek Plhák
AUTHOR

VEDOUCÍ PRÁCE Ing. Jan Kořenek
SUPERVISOR

BRNO 2007

Abstrakt

Tato práce se zabývá návrhem a implementací knihovny, nad generickými vývojovými nástroji, zapouzdřující a také rozšiřující jejich funkčnost. Knihovna mimo svou funkci odrazit nově vznikající požadavky programátorů nanoprogramů je především vyvíjena s cílem zjednodušit implementaci vývojového prostředí. Navržená knihovna byla implementována a nyní slouží jako základ vývojového prostředí „geneditor“ vyvíjeného v rámci projektu Liberouter.

Klíčová slova

Generické vývojové nástroje, model nanoprocesoru, nanoprocesor, simulace, vývojové prostředí

Abstract

This thesis deals with analyse questions and implementation of library above the generic development tools. These tools extends and forms unified access structure for using applications. The library reflects not only requests coming as an orders from nanoprograms developers, but most important reason to create it is in making easy an implementation of now developing graphic development environment. This library was implemented, tested and set into graphics development environment named „geneditor“. All this was created within project of Liberouter developers group.

Keywords

Generic development tools, model of nanoprocesor, nanoprocesor, simulation, development environment

Citace

Marek Plhák: Vývojové prostředí pro nanoprocesory, bakalářská práce, Brno, FIT VUT v Brně, 2007

Vývojové prostředí pro nanoprocesory

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jana Kořenka.
Další informace jsem získal především díky členům vývojářského týmu Liberouteru.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Marek Plhák
8. května 2007

Poděkování

Chtěl bych touto formou poděkovat kolegům z projektu Liberouter, zejména Ivoši Hažmukovi, Petrovi Mikuškovi, Peterovi Stančekovi, Václavu Zachrovi za hodnotné informace a rady, bez kterých by tato práce nemohla být realizována.

© Marek Plhák, 2007.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů..

Obsah

Obsah 1
Úvod 2
1 Teoretický úvod 3
  1.1 Procesory 3
    1.1.1 Řídící jednotky CPU a MCU 3
    1.1.2 Typy procesorů RISC a CISC 3
    1.1.3 Nanoprocesory 4
  1.2 Vývojové nástroje pro nanoprocesory 5
  1.3 Vývojová prostředí pro nanoprocesory 6
  1.4 Analýza komponent a zdrojů 7
    1.4.1 Model nanoprocesoru 7
    1.4.2 Překlad programů 11
    1.4.3 Simulace běhu programů 12
2 Knihovna nad vývojovými nástroji 14
  2.1 Pozice knihovny ve vývojovém prostředí 14
  2.2 Návrh knihovny 15
    2.2.1 Konceptuální model 15
    2.2.2 Rozbor konceptu 16
    2.2.3 UML diagram tříd 19
  2.3 Implementační detaily 20
3 Závěr 21
Literatura 22
Seznam příloh 23
Přílohy 24

Úvod

Je tomu už více jak 40 let, kdy byl uveden na trh první procesor. Tehdy započal bouřlivý vývoj informačních technologií, které se i v dnešní době stále řadí mezi první nejvýznamnější technologické dovednosti vědy. V průběhu let kráčeli vývojáři po různých cestách, ne všechny ovšem vedly k úspěšnému cíli. Mnohé vývojové větve zanikly, jiné nedokázali splnit účel svého vzniku, naproti tomu další jsou zapsány v počítačové historii. Paradoxně v průběhu vývoje mikroprocesorů výrobci postupně opouštěli staré technologie a přecházeli k novým, aby se postupem času jiní výrobci vrátili k těmto opuštěným. Časem začaly informační technologie pronikat do každé věci, do každé myšlenky, do každého koutu světa, aby nám pomohly ušetřit čas a lidskou práci.

Podstatou vzniku nějakého procesoru musí být účel, kterému má sloužit. Časem začalo účelů takovou mírou nabývat, že tehdy doposavadní technologie nemohly zůstat beze změny. Očekával se rychlý vývoj technického vybavení, které si s sebou přímo z továrny neslo svůj účel, svůj program. Došlo tedy k oddělení hardwarového a softwarového vývoje do dvou technologicky nezávisle vznikajících, ale komunikačně propojených oborů. V poslední době se začaly opět tyto dříve oddělené obory spojovat, ale až v rámci finální fáze vývoje, takže oddělení je patrné po celou dobu vývoje produktu. Sloučení ve finální fázi je výhodné jak k dosažení vyšší rychlosti, díky použití účelných materiálů, tak s tovární výrobou související ekonomickou stránkou.

Tato práce se zabývá softwarovým vývojem v oblasti procesorů, řešících specifické problémy, zapadajících do kategorie RISC procesorů. S růstem výkonu a možností nabízených těmito procesory rostou i požadavky na programy. Jednak vznikají programovací jazyky, nebo se jazyky rozšiřují, v druhé řadě vznikají prostředí které poskytují programátorům vše potřebné pro vývoj programů. Tímto způsobem vznikly v projektu Liberouter generické vývojové nástroje, schopny rychle reflektovat nově vznikající schémata procesorů do modelů a dali tak vývojářům programů pohotový nástroj pro implementaci. Nástroje díky modelům mají nepostradatelný klad v možnosti nezávisle na dokončené, resp. funkční architektuře program realizovat.

Vývoj vzal opět rychlostí za své a rozsah aplikací a jejich ladění se stalo prioritou, která díky mnoha faktorům nemohla být splněna pouze generickými vývojovými nástroji. Další nevýhodou, vznikající průběžně od doby, kdy byly generické nástroje napsány se staly nástroje, nabalující se na generické vývojové nástroje během vývoje programu. Jedná se o nástroje softwarové realizace, testování s následným umístěním v hardwaru.

S možným řešením přichází realizace vývojového prostředí postaveném na vývojových nástrojích. Bohužel během implementace se začala první verze vývojového prostředí potýkat s mnoha nedostatky, které jsou způsobeny neuniverzálností řešení, vyplývající z univerzálnosti generických nástrojů. Projekt vývojového prostředí byl v tomto stádiu pozastaven, neboť robustnost řešení nabyla rozměrů, která nebyla v rámci čitelnosti a rozšiřitelnosti kódu akceptovatelná.

Novým požadavkem se tak stalo implementovat knihovnu, která by zapouzdřila funkce veškerých doposavaď používaných nástrojů při vývoji programů a ve finální fázi byla schopna vytvořit nejen zázemí pro vznikající univerzální vývojové prostředí, ale i jednoduché testovací aplikace. Toto se stalo tématem této bakalářské práce.

1 Teoretický úvod

Teoretický úvod hovoří o tématech, na jejichž obsah se mohu posléze v textu odkazovat. Prvním tématem se podíváme, co naše nanoprocesory vůbec jsou. Druhým, na vývojové nástroje, a nakonec, proč se vyvíjí naše řešení s ohledem na již hotové ve světě známé produkty.

1.1 Procesory

Projděme připomenutím a srovnáním typy procesorů, abychom si posléze přesně vymezili nám podstatné nanoprocesory.

1.1.1 Řídící jednotky CPU a MCU

Hlavním rozdílem mezi těmito dvěma typy, je oblast jejich použití. Na rozdíl od univerzálnosti CPU, MCU nabízí řešení specifické pro konkrétní systém. Pro lepší představu o vlastnostech uvedu jejich srovnání v tabulce níže.

Parametr	CPU	MCU
Periferní zařízení	Všechna jsou připojována externě	Malá možnost rozšíření, samy nabízejí mnoho periferií, převážně specifické pro konkrétní použití
Spotřeba	Vysoká	Oproti CPU velmi nízká.
Rozměry	Otevřenost architektury přináší větší nároky na rozměry pouzdra. Velký počet vývodů.	Rozměry jsou minimální v souvislosti s počtem vývodů, které jsou přímo závislé na konkrétním MCU pro dané řešení
Výkon	Vysoký.	Odpovídající požadavkům.
Použití	Obecné	Specifické

Tab. 1 – srovnání CPU a MCU

1.1.2 Typy procesorů RISC a CISC

Rozdělme si procesory podle toho, jaké služby musí být schopny splnit v konkrétním systému. Takto můžeme procesory kategorizovat na ty, s menší instrukční sadou, zřídka nabízejících více jak základní aritmetické instrukce, nazývané RISC a procesory s širokou paletu instrukcí, nalézajících převážně obecné uplatnění, nazývané CISC. Dalším podstatným rozdílem mezi těmito dvěma typy je rychlost provádění jednotlivých instrukcí. Zatímco u CISC není rychlost provedení instrukce pevně vymezena, u RISC je požadováno provedení jedné instrukce v jediném hodinovém taktu, tedy až na vyjímky, kdy instrukce přistupují k paměti.

Mezi další zajímavé rozdíly určitě patří prostor v čipu věnovaný řídícím obvodům pro zpracování instrukce, které u CISC tvoří okolo 60%, zatímco u RISC nesahají, díky své pevně vymezené délce a formátu přes 10%.

RISC	CISC
SPARC	System/360
ARC	VAX
ARM	PDP-11
AVR	Motorola 68xxx
MIPS	AMD
PIC	Intel x86
Power Architecture	

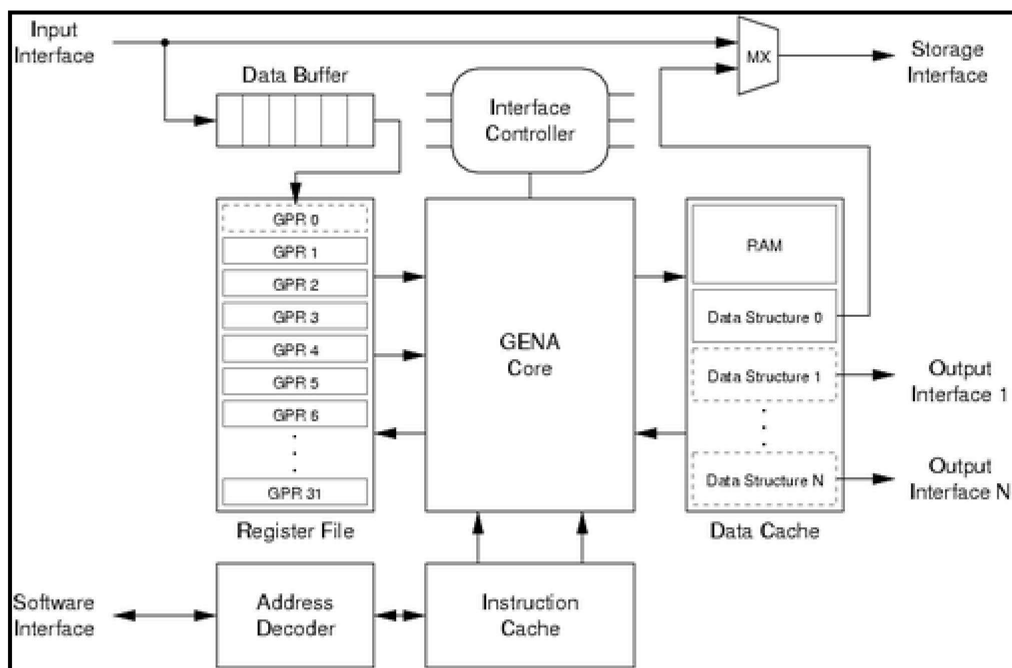
Tab. 2 – představitelé RISC a CISC

1.1.3 Nanoprocesory

Jedná se o mikroprogramové řadiče, které jsou navrženy vykonávat specifické funkce. Mají silný potenciál v rychlosti zpracování dat, čemuž vděčí jak svou technologickou úrovní, tak nízkou složitostí. V souhrnu zaručují programátorovi programu práci s pouze několika potřebnými instrukcemi. Program je nahráván přímo do paměti jádra, což vede k unifikovanosti v adresaci a přístupu k paměťovým slabikám. Podle všech těchto kritérií jsme jednoznačně schopni zařadit nanoprocesory do kategorie RISC procesorů. Co se technické implementace týče, jsou díky své malé velikosti často použity ve větším množství, jako paralelně pracující buňky konkrétního systému.

Příkladem takovéhoto nanoprocesoru je existující GENA jádro. Jádro obsahuje pouze instrukce pro základní aritmetické úkony, bitové operace a přesuny v asociované paměti.

Toto jádro se stalo základním kamenem procesorů analyzujících datový tok. Jako příklad takového nanoprocesoru zmíním HFE, z anglického „Header Field Extractor“ (viz. obrázek 1), pro zpracování hlaviček paketů, který má za cíl vytvořit unifikovanou zprávu o zpracovaném paketu.



Obr.1 – schéma nanoprocesoru HFE s jádrem GENA

1.2 Vývojové nástroje pro nanoprocesory

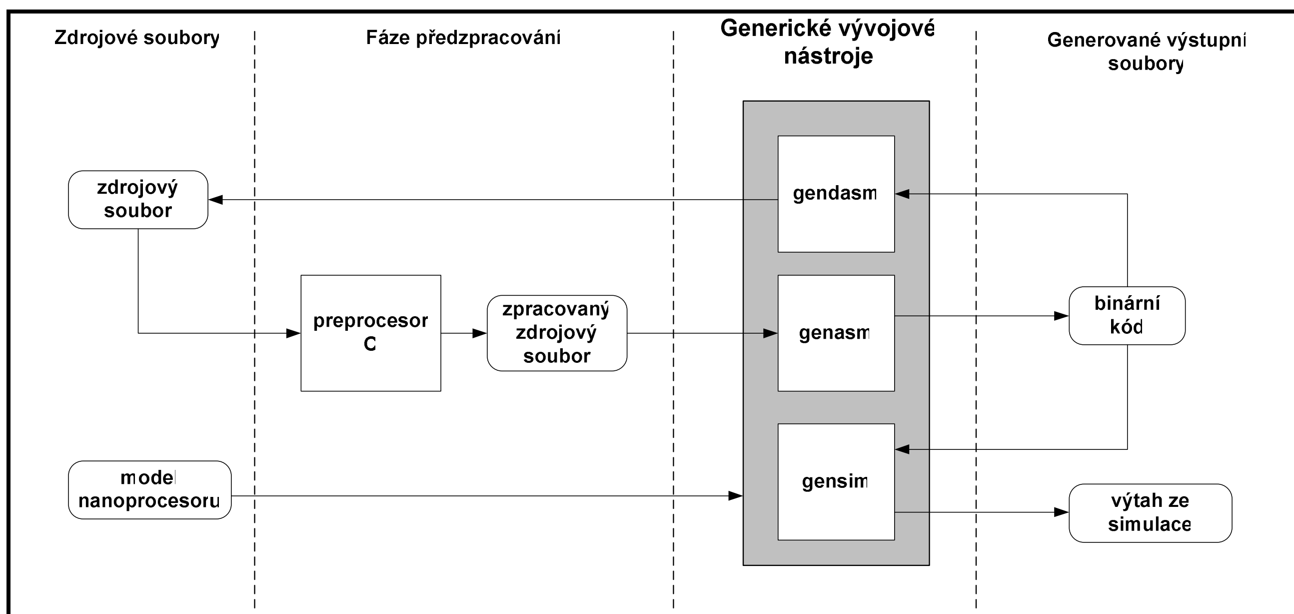
Jak již bylo v úvodu zmíněno, i přes nijak rozsáhlé programy určené pro tyto nanoprocesory, je jejich efektivnost a spolehlivost velmi zásadní. Vývoj programů sloužících pro dosažení takového stavu prošel ve vývojářském týmu Liberouter dlouhou cestu, o které bych se chtěl alespoň malou mírou zmínit.

Původním požadavkem na vývojové nástroje bylo reflektovat funkce a prostředky nabízené jádrem GENA. Na základě analýzy těchto požadavků byl implementován překladač a simulátor v jednom, běžící v textovém módu, obsluhovaný primitivními příkazy. Tento program je obecně označován jako „nsim".

Tímto, zdálo by se, že úloha vývoje nástrojů může skončit, neboť tento produkt nabízí vše potřebné a postačující pro vývoj v rámci GENA jader. Ovšem jak jsem se právě zmínil, je to právě ona specializovanost na GENA jádra, která zaručila konec v používání tohoto nástroje.

Nastala další etapa vývoje, ale tentokrát již v trochu jiném duchu. Bylo požadováno více dynamičnosti řešení. Vytvořit nástroj, který by byl schopen simulovat a vytvářet binární kód libovolného nanoprocesoru. V neposlední řadě se také hledí na možnost simultánního běhu několika nanoprocesorů a jejich interakce. Krom jazyku pro modelování nanoprocesorů, které jsou následně reprezentovány v podobě souborů, vznikly celkem tři nástroje. Souhrnně je nazýváme „generické vývojové nástroje".

Jedná se o překladač „genasm", zpětný překladač „gendasm" a simulátor „gensim". Nutno ještě dodat, že syntaxe kódu nanoprogramů umožňuje využít schopností a možností, které nabízí preprocesor jazyka C, čímž se rozšiřuje univerzálnost těchto nástrojů. Použití těchto nástrojů blíže znázorňuje obrázek níže.



Obr. 2 – Použití generických vývojových nástrojů

Bohužel, s růstem nutnosti vyvíjet rychle programy pro nanoprocesory, začal vzrůstat požadavek na vytvoření vývojového prostředí, obecného standardu z pohledu moderního vývoje aplikací. Faktorem, dále zdůrazňujícím tento požadavek, se stalo nedostatečné řešení simulátoru, jakožto ladícího nástroje, ve kterém se objevily oblasti, které, ne zcela simulátor pokrývá.

1.3 Vývojová prostředí pro nanoprocesory

Tak jako vznikající řešení z týmu Liberouteru, existují ve světě i jiná řešení, každé ovšem má svou dominantu v něčem jiném.

Co se týče známých produktů, jedná se převážně o aplikace, zaměřené na široké spektrum procesorů, nejen nanoprocesorů, čímž se chtějí stát všestrannými. V tomto shledávám jak výhodu, v široké použitelnosti, tak nevýhodu, která přináší mnoho nepotřebného.

Dalším problematickým bodem se jeví již hotové implementace programů pro nanoprocesory a určitě také struktura generovaného kódu, která je využívána jinými nástroji, pro zasazení do fyzické architektury. Aktuální kód je z nynějšího pohledu závislý na implementaci modelů a muselo by v takovémto případě dojít k mnoha transformacím a novým implementacím v jiných prostředích, které by kromě kvanta nadbytečných informací a nabízených možností zaručeně přibrzdily rychlý proud vývoje. V neposlední řadě se také jedná o finanční stránku věci, která by nejspíše předčila dosavadní.

Záměrem je tedy soustředit se pouze na nanoprocesory, na jejich modelování, simulaci a psaní programů pro ně. V návaznosti na generické vývojové nástroje se jedná o nástavbu, která poskytuje všechny služby těchto nástrojů a navíc je rozšiřuje o nové. Další nespornou výhodou je vlastní vývoj tohoto produktu, neboť se tak dá, v závislosti na přáních uživatelů, prostředí rychle rozšířit, resp. inovovat a přizpůsobit novým požadavkům.

1.4 Analýza komponent a zdrojů

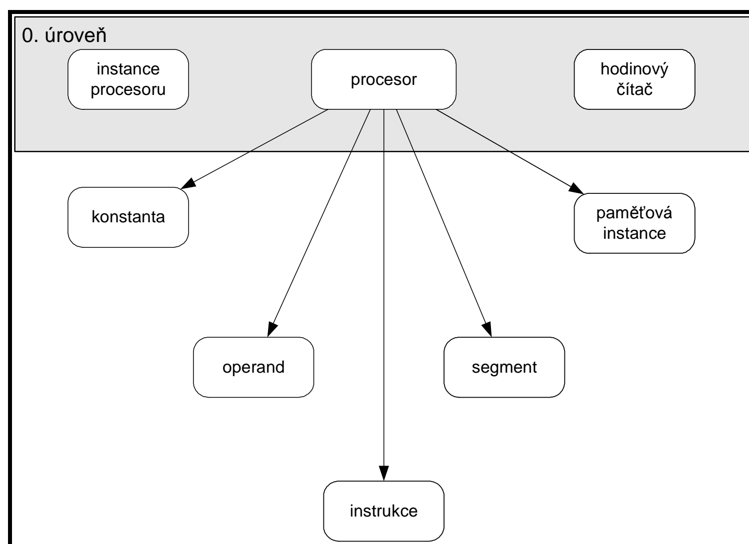
Před přistoupením k analýze se musíme zamyslet, nad cílem našeho snažení. Jednak musíme zjistit, co nám nástroje jsou schopny nabídnout. V druhé řadě, zjistit informace o modelu nanoprocesoru a vazbu těchto informací na zdrojové soubory, překlad, simulaci apod. . Co se týče této kapitoly, budeme k řešené problematice přistupovat jako doposavadní tvůrce programů pro nanoprocesory. Začneme s analýzou modelu, na které postavíme jazyk pro psaní programů uložených ve zdrojových souborech, které dále přeložíme, a v poslední fázi odsimulujeme.

1.4.1 Model nanoprocesoru

Každý nanoprocesor, který má být odsimulován a má být pro něj proveden překlad programů, musí být zprvu namodelován. V této kapitole se zaměříme především na podstatné informace získatelné z modelu, které bude knihovna pro splnění svých závazků vůči vyšší vrstvě vyžadovat.

Modelový soubor má určitým způsobem definovanou strukturu. Jazyk pro popis modelu poskytuje široké spektrum možností a schopností pro modelování. Struktura může být rozšířena o nabízené prostředky C preprocesoru [ANSIC].

V podkapitolách uvádím všechny struktury, které jsou pro modelování nabízeny. Jistě nebude též od věci ukázat, v jakém vztahu jsou struktury modelovacího jazyka ke zdrojovému souboru . Hierarchický model struktury modelového souboru je zachycen v obrázku č. 6.



Obr. 3 – Závislosti v modelovém souboru

Ještě než vkročíme do podkapitol, chtěl bych vysvětlit, jak jsou koncipovány. Název kapitoly popisuje popisovanou strukturu v modelovacím jazyce. Následuje vysvětlení, obecný tvar, ukázka kódu a samozřejmě také, z jakého důvodu je konkrétní struktura v analýze modelového souboru potřebná. Při procházení ukázky kódu si všimněte tučnějšího písma, které vyjadřuje pouze ty pasáže, které jsou pro nás důležité. V uváděném obecném tvaru struktury se můžeme setkat se znaky z řad regulárních výrazů.

1.4.1.1 Procesor

Definováním struktury „procesor“ myslíme vstupní bod do modelu nanoprocesoru. Tato struktura (Obr. 6) je vstupním bodem pro definici součástí modelovaného systému.

Význam pro knihovnu

Každý procesor má svůj identifikační název, na základě kterého je možné vytvářet jeho instance, které musí být evidovány.

Obecný tvar

- processor <název> { .* }

```
processor GENA {
  // definice konstant, typů operandů, instrukcí, atd...
}
```

Příklad č.1 – Model nanoprocesoru, processor

1.4.1.2 Hodinový čítač

Hodinový čítač udává, kolik hodinových tiků nastalo od restartování procesoru. Pro více informací viz. [Wiwa].

Význam pro knihovnu

Pomocí hodinového čítače máme přehled o tom, kolik tiků již nastalo a můžeme tak např. implementovat hlídací časovací modul.

Obecný tvar

- instance <název>: clock

```
instance CLK: clock;
```

Příklad č.2 – Model nanoprocesoru, hodinový čítač

1.4.1.3 Instance procesoru

Pokud chceme použít nějakou ze struktur namodelovaného nanoprocesoru, musíme tak jako u tříd v programovacích jazycích vytvořit jejich instanci. Tato instance reprezentuje již konkrétní nanoprocesor.

Význam pro knihovnu

Jedná se o velmi důležitou definici, říkající, k jakému doslova „zařízení“ se bude, při zjišťování informací, v simulaci přistupovat.

Obecný tvar

- instance <název>: processor <název procesoru> clock <název hodinového čítače>

```
instance OUR_PROC: processor GENA clock CLK;
```

Příklad č.3 – Model nanoprocesoru, instance procesoru

1.4.1.4 Konstanta

Neboli definice názvu, kterému je přiřazena hodnota. Jakýkoliv další výskyt tohoto názvu je následně chápán jako hodnota přiřazená tomuto názvu.

Význam pro knihovnu

Při analýze modelového souboru potřebujeme znát překlad konstant na hodnotu.

Obecný tvar

- const <název> = <hodnota>

```
const R01 = 01;
```

Příklad č.4 – Model nanoprocesoru, konstanta

1.4.1.5 Segment

Blok, který fyzicky vyjadřuje jinou funkci vzhledem k ostatním. Příkladem může být kódový segment s posloupností instrukcí tvořících program a datový segment s libovolnými daty [Wiseg].

Význam pro knihovnu

Při překladu je nutné separovat segmenty ze zdrojových souborů. Je také nutné vědět, které segmenty jsou povoleny být použity ve zdrojových souborech. Následně při simulování je nástroji simulátoru (gensim) předán separovaný kód programu (separovaný ve smyslu rozdělený do segmentových souborů).

Obecný tvar

- segment <název=CODE> : <velikost>
  - implicitní název je CODE

```
segment      : 16; //CODE
segment DATA : 32;
```

Příklad č.5 – Model nanoprocesoru, segment

1.4.1.6 Paměťová instance

Definice pojmenovaného paměťového prostoru, se specifikovanou velikostí, případně i rozsahem.

Význam pro knihovnu

Díky znalosti paměťových instancí, jejich typu a rozsahu, je možné během simulace do nich importovat data, ale i z nich exportovat za účelem získání informací.

Obecný tvar

- vektorová instance: instance <název>: memory[<rozsah>][<velikost>]
- skalární instance: instance <název>: memory [<velikost>]

```
instance DATA: memory[8][32]; // pole o rozsahu 256 slabik, každá
                              // velikosti 32 bitů (4B)
instance Z: memory[1]; // 1 bitový příznak
```

Příklad č.6 – Model nanoprocesoru, paměťová instance

1.4.1.7 Řízení nanoprocesoru

Ze strany kódu běžícím na nanoprocesorech je nutné vědět, v jaké paměťové instanci je umístěn kód a neodmyslitelně také, jaká instance reprezentuje slabiku sloužící jako ukazatel na prováděnou instrukci (IP... „Instruction pointer").

Význam pro knihovnu

Naše knihovna také potřebuje mít přehled o tom, kde se nachází ukazatel na prováděnou instrukci, což je důležité nejen kvůli kontrolování bodů zastavení za běhu simulace, ale i k zjištění, jaká paměťová instance obsahuje kód.

Obecný tvar

- fetch
  - memory = <instance pro kód>
  - address = <instance pro ukazatel na prováděnou instrukci>

```
instance MAIN: memory[16][32]; // instance paměti pro kód
instance IP: memory[16]; // instance paměti sloužící jako ukazatel na
                         // instrukci
...
```

```
fetch {
  memory = MAIN; // paměťová instance, kde je kód
  address = IP; // paměťová instance ukazatele na instrukci (slabika)
}
...
```

Příklad č.7 – Model nanoprocesoru, řízení nanoprocesoru

1.4.1.8 Typ operandu

Definice typu operandu slouží dále, u definic instrukcí k určení, jakého datového typu je konkrétní parametr.

> Význam pro knihovnu
>
> Knihovna má možnost poskytnout programátorovi přehled, nejen nabízených instrukcí, ale také souvisejících datových typů pro parametry. Programátor vyšší vrstvy tak může například provést ověření, jestli je daný argument pro parametr instrukce správný, aniž by muselo dojít k vyvolání chyby při překladu zdrojového souboru.
>
> Obecný tvar
>
> - operand <název>(.*/<velikost>)

```
operand Imm16(E/16) immediate
{
  symbolic = E;
  encoding {
            if(value >> 16)
              throw 0;
            E = value[15..0];
  }
  semantics = E;
}
```

Příklad č.8 – Model nanoprocesoru, typ operandu

1.4.1.9 Instrukce

Každý procesor nabízí množinu operací, které jsou realizovány instrukcemi, které procesor je schopen interpretovat a operace provést. Instrukce jsou základními kameny modelu, který tímto získává svou funkčnost. Modeluje se jak syntaxe, tedy reprezentace v jazyku symbolických instrukcí, používaném ve zdrojových souborech, tak i sémantika, definující, co se a jak děje uvnitř nanoprocesoru. Tento vztah, sémantiky modelované instrukce a hardwarové realizace by měl být izomorfní.

> Význam pro knihovnu
>
> Kromě znalosti názvu instrukce, parametrů a jejich typů je důležité znát velikost instrukce v konkrétní paměťové instanci, kde bude použita. Díky znalosti velikost instrukce si můžeme dopočítat, jakému řádku s instrukcí ve zdrojovému souboru odpovídá jaký ukazatel na instrukci.
>
> Obecný tvar
>
> - operand <název>(.*/<velikost>(,.*/<velikost>)+)

```
instruction ADDI rD, rA, imm16
```

```
(E/INSTR_LEN = #110011_xxxxx_xxxxx_xxxxxxxxxxxxxxxx)
{
  rD: Reg(E[25..21]);
  rA: Reg(E[20..16]);
  imm16: Imm16(E[15..0]);
  semantics {
    REG[rD] = (REG[rA] + imm16)[REG_UNIT-1..0];
    Z = (REG[rD] == 0);

    if(REPI_M) {
      IP = (IP - instruction_length)[IP_SIZE-1..0];
      REPI_M = (REPI_M - 1)[MEM_UNIT-1..0];
    }
  }
}
```

Příklad č.9 – Model nanoprocesoru, typ operandu

1.4.2 Překlad programů

Pro překlad programů určených je vyvinut nástroj z kolekce generických vývojových nástrojů zvaný „genasm“.

Nástroji se předloží několik parametrů a on vytvoří binární kód. Prvním parametrem musí být název modelového souboru, následovaný zdrojovým souborem. Poté již stačí určit, jaké segmenty se mají exportovat do jakých souborů (oddělení je logické i z pohledu HW realizace, kdy segmenty mohou být úplně jiné paměti).

Seznam parametrů

Parametr	Význam
-s <soubor>	Předání zdrojového souboru
-o <segment=CODE> <soubor>	Exportuje segment daného názvu do souboru (implicitně je název segmentu nastaven na CODE)
-l <soubor>	Pořízení podrobného výpisu ze zdrojového souboru

Následující příklad ukazuje, jak spustit překladač nad modelovým souborem test1.ad, který definuje 2 segmenty (jeden datový a druhý kódový) a zdrojovým souborem test1.asm. Všimněme si také, že před samotným překladem musí dojít k předzpracování C preprocesorem.

```
cpp -P -traditional-cpp -E test1.asm -o test1.asmp
genasm test1.ad -s test1.asmp -o test1.code.hex -o DATA test1.data.hex
```

Příklad č.10 – Spuštění překladače

1.4.3 Simulace běhu programů

V návaznosti na „genasm“ překladač z předchozí kapitoly se budeme zabývat dalším nástrojem, kterým je míněn „gensim“, tedy simulátor.

Toto téma rozdělíme na dvě podčásti. Jedna se bude věnovat spouštění a parametrizaci simulátoru, druhá spuštění, řízení a získávání informací ve spuštěném simulátoru. Knihovna samotná musí umět jak automaticky sestavit parametry pro spuštění simulace, tak umět pomocí příkazů pro konzoli simulátoru, simulaci řídit, získávat a ukládat za běhu informace.

1.4.3.1 Komponenta simulátoru

Nástroji simulátoru musí být předáno, nad jakým nanoprocesorem se bude provádět simulace a jaký program na nanoprocesoru poběží. Simulátor se spouští nad 1..n modely, s čímž souvisí zajištění odpovídajících dat pro jednotlivé segmenty každého modelu.

Kromě těchto základních možností parametrizace je možné také spustit simulaci v interaktivním módu, kdy běh je řízen konzolovými příkazy.

Seznam parametrů

Parametr	Význam
-i	Spuštění v interaktivním módu
-b <model>.<paměťová instance, nebo segment> <offset> <zdrojový soubor>	Načtení binárního kódu ze zdrojového souboru do paměťové instance konkrétního modelu. Zápis do paměti od adresy dané offsetem.
-o <model>.<paměťová instance, nebo segment> <offset> <velikost> <soubor>	Po provedení simulace bude vytvořen výstup do souboru, od daného offsetu a konkrétní velikosti v paměti

1.4.3.2 Komponenta simulátoru

Konzole simulátoru umožňuje v interaktivním módu řídit běh simulace. Postup řízení je analogický: Uživatel zadá příkaz, simulátor provede a vypíše informace jako reakci na konkrétní příkaz.

Přehled příkazů, které jsou z pohledu knihovny důležité je zachycen v tabulce níže.

Příkaz	Význam
dump <model>.<paměť> <offset> <soubor>	Uloží paměťovou instanci konkrétního modelu od daného offsetu do souboru
next <model>	Spustí následující instrukci konkrétního modelu. V případě běhu více modelů, pokud je název modelu vynechán vykoná se pouze jedna instrukce.
output [on|off]	Zapne/vypne hlášení, které je možné volat ze sémantické části definic instrukcí

pause [on|off] <model>	Zapne/vypne pozastavování ze sémantické části definic instrukcí. Určení konkrétního modelu je volitelné, implicitně platí pro všechny modely.
print <model>.<paměť> <offset1> <offset2>	Vypíše obsah paměti v hexa tvaru od offsetu1 po offset2, pokud offset2 není zadán, vypíše se pouze jedna slabika
status <instance modelu, hodin>	Zobrazí stav modelu (tj. pozastaven, běžící, atd.) v případě hodin zobrazí počet tiků od restartování.
run	Spustí simulátor bez možnosti pozastavení simulace. K pozastavení může dojít, pokud je nadefinováno instrukcí a to jen a pouze v případě, kdy nebyl použit příkaz pauze(viz. výše)
quit	Ukončí simulátor

2 Knihovna nad vývojovými nástroji

Ze specifikace navrženého vývojového prostředí je patrná jeho robustnost. Při požadavku na udržovatelnost a čitelnost kódu by bylo nevhodné skloubit všechny požadavky v jediné řešení. Tímto mám na mysli skutečnosti, využít již hotové generické vývojové nástroje, mít možnost rychlým způsobem implementovat nově vzniklé požadavky a vytvořit vhodný a univerzální základ pro aplikace zaměřené na kódování programů, modelování, ladění, ve shrnutí, aplikace tvořící vývojové prostředí.

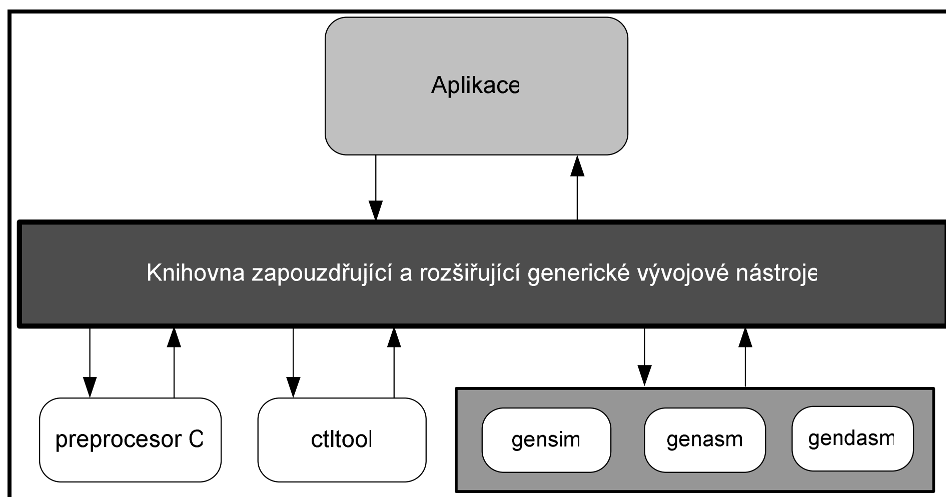
Z těchto požadavků vzniklo řešení, kterým se budeme zaobírat, jako hlavním tématem této práce. Jedná se o knihovnu nad generickými vývojovými nástroji.

2.1 Pozice knihovny ve vývojovém prostředí

Koncept vychází z úvahy nad schématem použití generických vývojových nástrojů. Víme, že budou existovat programy, které chtějí využívat funkcí těchto nástrojů. Víme také o robustnosti celého řešení a zároveň jsme si vědomi změn, které se mohou uskutečnit jak na straně nástrojů, tak ve vývojovém prostředí. Pro řešení není snad lepší varianty, než vytvoření knihovny, která by zapouzdřila funkce nástrojů a nabídla jednotné řešení programům.

Přesněji tedy tato knihovna musí obsáhnout funkce simulátoru, překladače, včetně neopomenutelné vazby na C preprocesor pro předzpracování souborů.

Tímto ale koncept nekončí. Dalším požadavkem je také existence vazby na hardware, která je v aplikaci, postavené na tomto základu velmi důležitá a neopomenutelná. Naštěstí v této oblasti komunikace s hardwarem již vývojáři udělali své a připravili celou sadu nástrojů, z nichž pro nás zajímavým a použitým bude nástroj „ctltool". Díky němu jsme schopni zjišťovat, ale naopak i nastavovat hodnoty v reálném technickém vybavení.



Obr.4 – koncept řešení

V nejvyšší úrovni vidíme aplikaci, která komunikuje s knihovnou, která v sobě zapouzdřuje spolupráci s vývojovými a komunikačními nástroji , které pro ni představují komponenty.

2.2 Návrh knihovny

Z předchozí kapitoly se nám otevřela brána k samotnému návrhu naší knihovny, neboť nyní přesně víme, co musí knihovna v nejmenší míře obsahovat. Díky již hotovým vývojovým nástrojům máme zjednodušenou práci o možnost je zařadit jako komponenty a využívat jimi nabízené služby.

Ve skutečnosti budeme odrážet uvedené kroky při tvorbě modelu nanoprocesoru, programů pro něj, překlad a simulaci, v naší knihovně.

Jednotlivé části jsou dále probrány v pododstavcích, jmenovitě:

1. Konceptuální model knihovny
2. Analýza modelového souboru
3. Analýza zdrojových souborů

2.2.1 Konceptuální model

Nejprve se postavíme před hotové věci, z kterých budeme muset sestavit určité vzájemně propojené celky.

Začněme modely. Model, víme z kapitoly o analýze zdrojů, definuje typy operandů pro parametry instrukcí, dále tedy navazující instrukční sadu, tedy instrukce, paměťové instance, segmenty a ještě také konstanty použité v modelu.

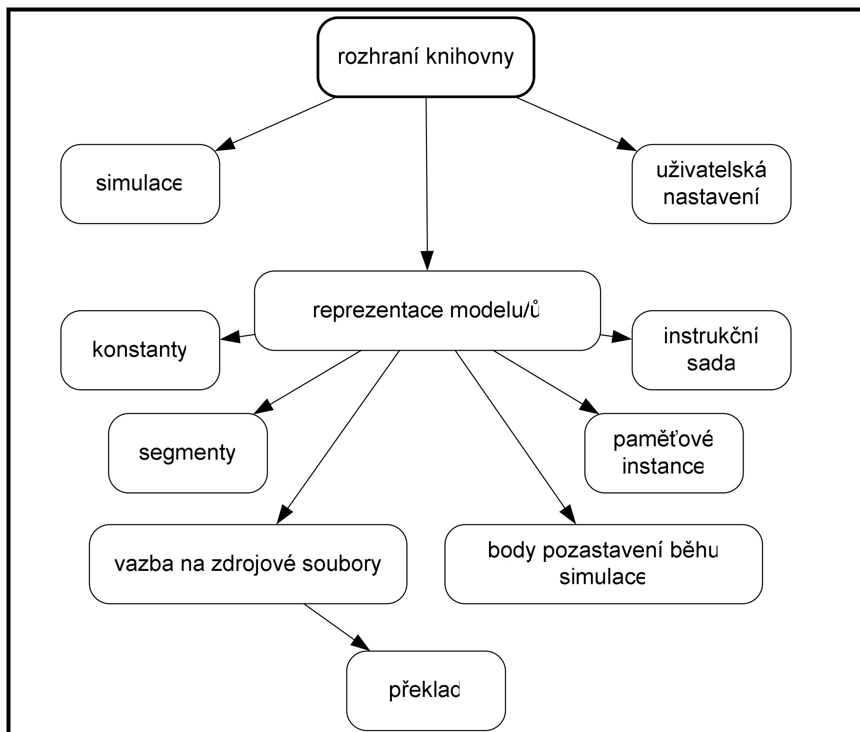
Přímo související s modely jsou dále zdrojové soubory. Jeden zdrojový program reprezentovaný jedním vstupním souborem se váže na jeden model.(v případě inkluze souborů uvnitř těla zdrojového souboru je možné uvažovat více zdrojových souborů tvořících jeden zdrojový program). Ze zdrojového souboru jsou pro nás podstatné instrukce z instrukční sady modelu, definované konstanty nahrazující identifikátory ve zdrojovém souboru hodnotou a nepochybně také rozčlenění kódu v segmenty. Shrnutím je tedy pro nás nutné evidovat ne jeden, ale více zdrojových souborů náležících jednomu modelu, které mohou být provázány. Prakticky může následně uživatel použít tento registr souborů i k jiným účelům než jen pro zdrojové soubory. Tím je myšlen prostor pro poznámkové soubory, apod. .

Dalšími body na které se musí knihovna soustředit je preprocesor jazyka C a komponenty generického vývojového prostředí, jmenovitě překladač a simulátor. Problém komunikace je naštěstí vyřešen díky možnosti použití souboru nástrojů(toolkit) Qt[QtDoc], nabízející mimo jiné i knihovnu pro spouštění aplikací, řízení vstupně výstupních proudů dat a obecně vůbec všechny potřebné prostředky pro řízení konzolových aplikací, jakými komponenty generických vývojových nástrojů nepochybně jsou. K parametrizaci komponent jen stručně říci, že se musíme ohlížet nejen na požadavky uživatele, tedy vlastní sestavení parametru, ale nabídnout i implicitní sestavení parametrů, což by mělo značnou mírou zjednodušit tvorbu programů pomocí této knihovny.

Co se překladu samotného týče musíme se zamyslet, za jakým účelem bude uživatel knihovny k překladu přistupovat. V prvním případě kvůli překladu konkrétního zdrojového souboru, v druhém, překlad všech zdrojových souborů všech modelů. Tento přístup, pokud si povšimneme, je typický pro nejeden vývojový nástroj, kdy v rámci projektu můžeme kompilovat například moduly kódu separovaně ale i hromadně. Určitě bude tedy muset existovat návaznost kompilace na konkrétní zdrojový soubor, ale i na celý projekt!

Teď již zbývá pouze simulace. Simulátor nemá přímou návaznost na zdrojové soubory, ale zato na soubory s hexa kódem, který je určen pro datovou část paměťových instancí. Jsme si vědomi, že tyto soubory je možné získat jedině a pouze překladem zdrojových souborů s assemblerovským kódem a tedy musí existovat spojitost mezi překladem a simulací.

Popisu odpovídá konceptuální schéma knihovny níže.



Obr. 5 – konceptuální schéma knihovny

2.2.2 Rozbor konceptu

Nyní máme hotový koncept, ale to je jen začátek. Je nutné se ponořit hlouběji, s cílem získat více informací o konkrétních částech. Díky rozboru budeme schopni sestavit diagram tříd zachycující řešení knihovny ve strukturované a vzájemně provázané podobě.

Následujících několik podkapitol je zaměřeno na oblasti konceptu. Popisuje se, co a jak získat, pro dosažení určeného cíle. Kontrastnější pasáže označují nejpodstatnější informace.

2.2.2.1 Překlad

Pro **překlad** je potřeba mít **přístup k překladači**. Je také nutné určit, jaké **výstupní soubory** je požadováno očekávat, jako výsledek překladu, což odpovídá existenci několika segmentů ve zdrojovém souboru. Zároveň by měla být uskutečněna vazba na knihovnou reprezentovaný zdrojový soubor, jak říká i koncept, z něhož lze získat jak název zdrojového souboru, tak i seznam segmentů získaný během analýzy zdrojového souboru a mít možnost **sestavit parametry**.

Překladem je povolán nástroj „**genasm**“. **Parametr** je vhodné **sestavit automaticky**, ale také mít možnost zakomponovat volbu **uživatelské specifikace parametru**.

2.2.2.2 Knihovní zástupce zdrojového souboru

Pro reprezentaci zdrojového souboru v knihovně, je hned několik důvodů.

Jeden byl zde již jmenován, jakožto informace pro překladač o **segmentech**. V samotném segmentu si musíme umět zjistit jeho velikost a specielně u kódového segmentu, **jaký řádek s instrukcí ve zdrojovém souboru odpovídá jakému ukazateli na instrukci v modelu**.

Dalším důvodem je zajistit **stálost a platnost umístění** souboru v souborovém systému, což můžeme zajistit jeho otevřením, tedy zvýšením počtu otevřených relací pro soubor a zabránit tak

pohybům se zdroji během používání knihovny. Z obecných vlastností nás bude určitě zajímat **jedinečný název souboru**, který je vyjádřen **cestou k souboru**, která podmínku jedinečnosti plně pokrývá.

Může vzniknout otázka, jak rozpoznáme typ souboru abychom s ním mohli poprávu zacházet? Odpovědí je, že pouze **přípona asm** bude dále přebrána jako zdrojový soubor a pouze s tímto souborem je možné dále provádět překlad. Co se týče **volitelného nastavení**, nemělo by se zapomenout definovat soubor explicitně jako **přeložitelný** a **nepřeložitelný**. Toto nastavení by se mělo odrazit v hromadném překladu více zdrojových souborů, který by neměl chybět.

A co tak, možnost evidovat soubor, který neobsahuje zdrojový kód aplikace? Může to být soubor s daty pro instance modelu, informativní soubor s poznámkami, či texty obecně, a další, související s projektem, evidovaný jako zástupce v knihovně. Implicitně jsou tyto soubory podle podmínky výše označeny jako nepřeložitelné.

Celá struktura musí být podle všeho uvedeného **analyzována** před prováděním operací se zástupcem.

2.2.2.3 Množina zdrojových souborů pro konkrétní model

Na model se může vázat **několik zdrojových souborů**. Toto je podstatné evidovat. Především pohled ze strany modelu je v tomto důležitý. Ten se dívá na vstupní bod, na soubor, který obsahuje počátek kódového segmentu. Tento soubor musí být mimo jiné jako jediný takto označen, dále jako **vstupní bod programu**.

Nezbytným, pro pohodlnější překlad je **volání překladu** nad všemi registovanými zdrojovými soubory, kdy sám správce rozhodne, jaký soubor projde překladem, logicky pouze vstupní bod, který například inkluzí nabírá všechny ostatní soubory.

Implicitně se bude **první vložený** asm soubor označovat jako vstupní bod.

Protože se jedná o množinu je podímnkou, zastavit se u otázky, jakým způsobem evidovat položky. Pro malý počet bude plně postačovat **lineární seznam**. S tím souvisí i **identifikace**. Každému vloženému zástupci bude přidělen **identifikační klíč**, na základě kterého se bude vyhledávat se zaručenou jedinečností. Přístupové metody budou muset zajistit **přidání, odebrání, vyhledání**.

2.2.2.4 Konstanty modelu

Konstanty se získávají během analýzy modelového souboru. Podle analýzy modelového souboru je zcela patrné, že pro každou konstantu je důležitý veškerý její obsah tvořený **názvem** a **hodnotou**.

Protože se jedná o množinu, musí se vyřešit několik otázek. Za prvé, **registrace konstanty** při analýze modelu, za druhé přístup k registru, hlavně ze strany **mapování názvu na hodnotu**.

2.2.2.5 Segmenty modelu

Segmenty modelu jsou **bezhodnotové prvky**, jejichž jediný účel je čistě implementační. Pro nás podstatné je jejich úloha při překladu a načítání dat během simulace.

Tato množina vyžaduje pouze **registraci názvu segmentu** a **výčtový přístup**, čímž pokrývá veškeré požadavky na ni.

2.2.2.6 Instrukční sada modelu

Instrukční sada sestává z instrukcí, které jsou reprezentovány svým názvem, parametry a velikostí.

Přístup k jednotlivým takovým instrukcím chápaných jako struktury by měl být realizován jako **mapování názvu** (názvu instrukce) **na strukturu**. Přidávání instukce by mělo být z pohledu sekvenční analýzy struktury modelového souboru realizováno takto:

1. **registrace názvu instrukce**
2. **registrace parametrů pro tuto instrukci**
3. **nastavení velikosti instrukce**

2.2.2.7 Paměťové instance modelu

Opět se jedná o množinu struktur, definovaných svým názvem, dále v případě skaláru svou **velikostí slabiky**, u vektoru rozšířenou o **velikost celého pole**. Během analýzy modelu je jednoduché získat tyto informace v jednom kroku, takže fáze přidávání spočívá pouze v **uvedení názvu, velikosti slabiky a případné velikosti**, která **bude implicitně nastavena na jedna**, vyjadřující skalární veličinu.

Vyhledávání v této množině bude realizováno **mapováním názvu na strukturu**.

2.2.2.8 Body pozastavení běhu simulace

V úzké souvislosti se zdrojovými soubory, jejich částí označenou jako kódový segment a požadavku pozastavovat simulaci na jednotlivých řádcích je důležité toto reflektovat v rámci nějaké struktury. Musí zde být i odražena jednoduchost pro přístup, která je z aplikační vrstvy podmínkou. Tím je myšleno především přidávání bodu zastavení nikoliv pomocí ukazatele na instrukci, nýbrž pomocí odpovídajícího řádku v kódovém segmentu.

Přístup k prvkům tohoto seznamu bude zajištěn **pomocí mapování řádku na instrukční ukazatel**. Přidávání a odebírání bude realizováno **zadáním řádku v kódovém segmentu**. Překlad čísla řádku na ukazatel se provede automaticky na základě existující **překladové tabulky**, sestavené ze znalostí velikosti instrukcí kódového segmentu a jeho analýzy.

2.2.2.9 Reprezentace modelů

Model samotný obsahuje jednotlivé struktury, jmenovitě: **konstanty, instrukce, paměťové instance**. Je definován instanční reprezentací definice struktury modelu. Obsah této struktury je nutné sestavit podle **analýzy modelového souboru**.

Každý přidaný model musí mít definován svůj jedinečný identifikátor. Podmínkou je také registrace modelu, resp. jeho popisného souboru, tedy **přidání modelu obsaženém ve zdrojovém souboru**. Přístupu k němu **vyhledáním v mapování identifikátorem** a jeho **odstranění ze seznamu** rovněžtak.

Umožnit **překlad všech souvisejících zdrojových souborů** nad všemi modely je zajisté nezbytností, která musí být v implementaci nabídnuta.

2.2.2.10 Simulace

Dostali jsme se až k samotnému jádru celé knihovny, simulaci.
Knihovní reprezentace musí obsáhnout **automatické sestavení parametru pro simulátor** z řad generických vývojových nástrojů, tak zadání **uživatelského parametru**. Automaticky je parametr generován na základě registrovaných modelů a zdrojových souborů jim náležících.

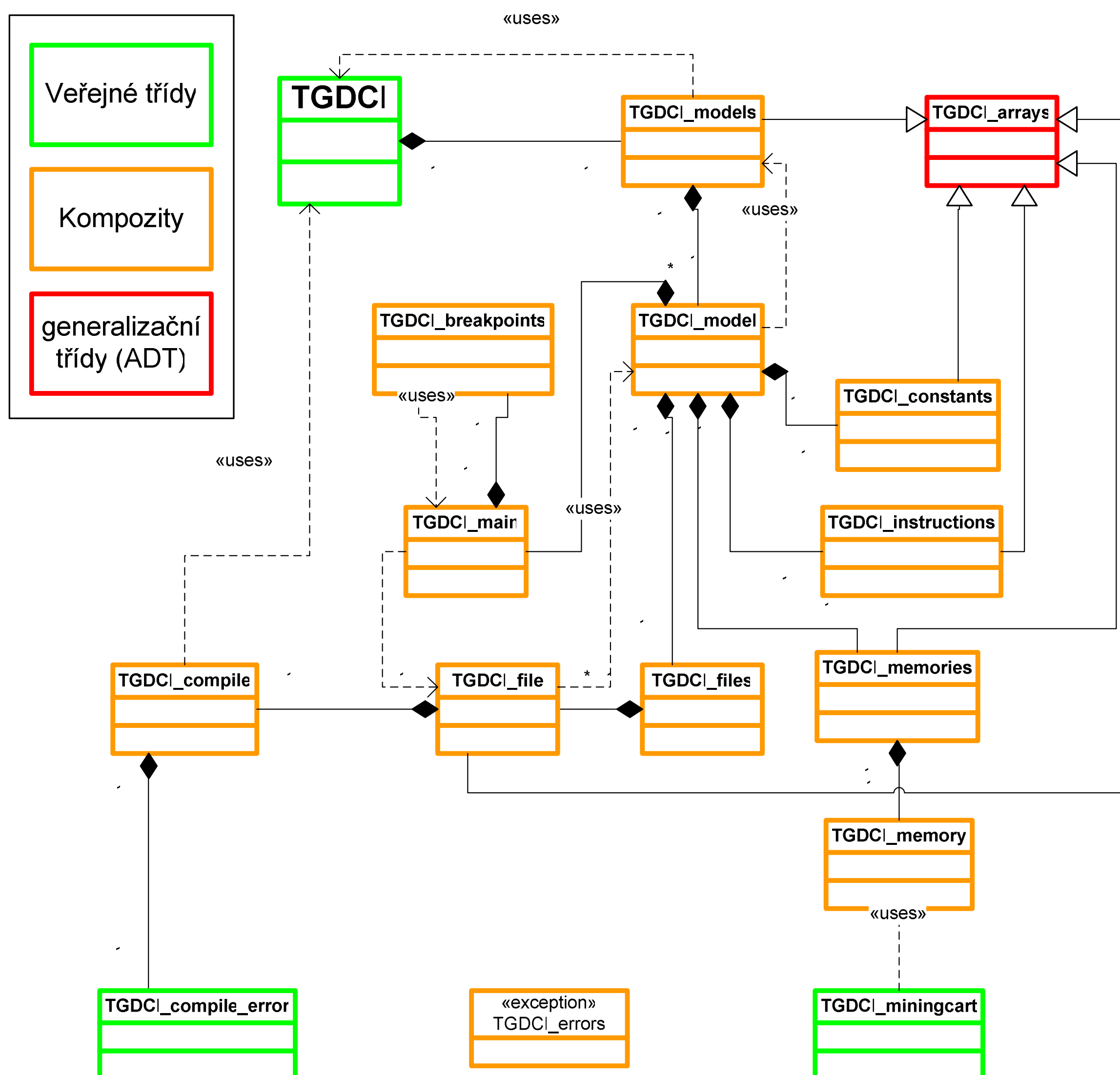
Tak jako je v nástroji „gensim“ zajištěno **spustit simulaci v některém z módů**, ať už **krokování,** či v **nepřetržitém běhu**, tak stejně nabízí i knihovna. Navíc podle specifikace knihovny

rozšiřuje o řízení **zastavení po inicializaci simulace**, to jest před provedením vůbec první instrukce, dále o řízení **zastavení před ukončením simulace** po provedení poslední instrukce, a v poslední řadě o to nejcennější, o řízení simulace pomocí **bodů zastavení**.

2.2.3 UML diagram tříd

Podle konceptu, s ním souvisejících struktur a jejich vlastností bylo navržena řešní, které by mělo být výchozím bodem pro implementaci knihovny. Struktura má hierarchický charakter a stejně tak i diagram této struktury, který ji vyjadřuje je takto sestaven.

Výše uvedený návrh byl během tvorby diagramu upraven a rozšířen do formy, která plně pokrývá problematiku, tudíž může obsahovat některé doposavaď nespecifikované skutečnosti.



Návrh zohledňuje i zapouzdření funkcí a vlastností knihovny a rozděluje tak navržené struktury do třech skupin. První, veřejná skupina tříd, poskytuje přístup k nabízeným funkcím a datům. Druhá skupina je kompoziční skupina, jejíž členové jsou kompozitem některé z tříd veřejných skupin. Jinak též, tyto třídy jsou přístupné pouze pomocí veřejných tříd. Třetí skupina je

skupina zajišťující implementační záležitosti, převážně abstraktních datových typů [Wiadt], sloužících jako generalizační třídy.

Název tříd začíná předponou gdci, zkratkou plného názvu „gen development environment communication interface“, tedy „komunikační rozhraní pro vývojové prostředí nad generickými vývojovými nástroji“.

2.3 Implementační detaily

Implementačním jazykem byl zvolen C++, s použitím souboru nástrojů Qt, který poskytuje kromě užitečných a implementačně čistých abstraktních datových typů, také třídy pro správu procesů, které jsou použity pro volání externích komponent.

Verze programu: 0.9b

Počet zdrojových souborů: 36, z toho 18 hlavičkových

Velikost zdrojových textů: 106.156 B

3 Závěr

Byla nastudována probematika programování programů pomocí generických vývojových nástrojů, zahrnující implementaci modelů nanoprocesorů, samotných nanoprogramů, jejich překlad a simulaci. Na základě získaných informací a analýzy dané problematiky bylo sestaveno řešení konstrukce knihovny, která byla posléze naimplementována.

Knihovna nabízí prostor pro testování, implementaci a ladění nanoprogramů s cílem přenechat na knihovně řešit algoritmizovatelné a automatizovatelné postupy při tvorbě.
Mimo oblast specielního účelu použití knihovny, v malých testovacích programech pro specifické situace, knihovna v plné míře doplňuje nedostatky původní implementace vývojového prostředí a dává tak prostor k nové implementaci založené na použití této knihovny jako jádra.

V současné době je knihovna využívána vznikajícím vývojovým prostředím geneditor, a díky pořád nově se objevujícím požadavkům z řad programátorů a uživatelů se postupně vyvíjí.

Rozšířením ladícího režimu pro konkrétní nanoprocesory a společně s úpravou modelu, který bude muset nést i hardwarově specifické informace se postupně realizuje komunikace mezi knihovnou a zařízením. Cílem tedy je, aby v celku vznikl produkt s funkčností rovnající se profesionálním vývojovým prostředím.

Literatura

[1] Tomáš Rybka, *Generické křížové nástroje pro procesory implementované*. Fakulta informatiky, Masarykova Univerzita, Brno, 2005.

[2] Petr Mikušek, *Návrh a implementace procesní jednotky pro analýzu vstupních paketů*. Fakulta informačních technologií, Vysoké učení technické v Brně, 2005

[ANSIC] *Programming Language - C, Rationale for American National Standard for Information Systems*, 1994
Dokument dostupný na URL http://www.lysator.liu.se/c/rat/title.html (květen 2007)

[Wireg] *Regulární výrazy, Wikipedie, otevřená encyklopedie*
Dokument dostupný na URL http://cs.wikipedia.org/wiki/Regul%C3%A1rn%C3%AD_v%C3%BDraz

[Wiadt] *Abstraktní datový typ, Wikipedie, otevřená encyklopedie*
Dokument dostupný na URL http://cs.wikipedia.org/wiki/Abstraktn%C3%AD_datov%C3%BD_typ

[Wiwa] *Watchdog, Wikipedie, otevřená encyklopedie*
Dokument dostupný na URL http://cs.wikipedia.org/wiki/Watchdog

[Wiseg] *Segmentace, Wikipedie, otevřená encyklopedie*
Dokument dostupný na URL http://cs.wikipedia.org/wiki/Segmentace

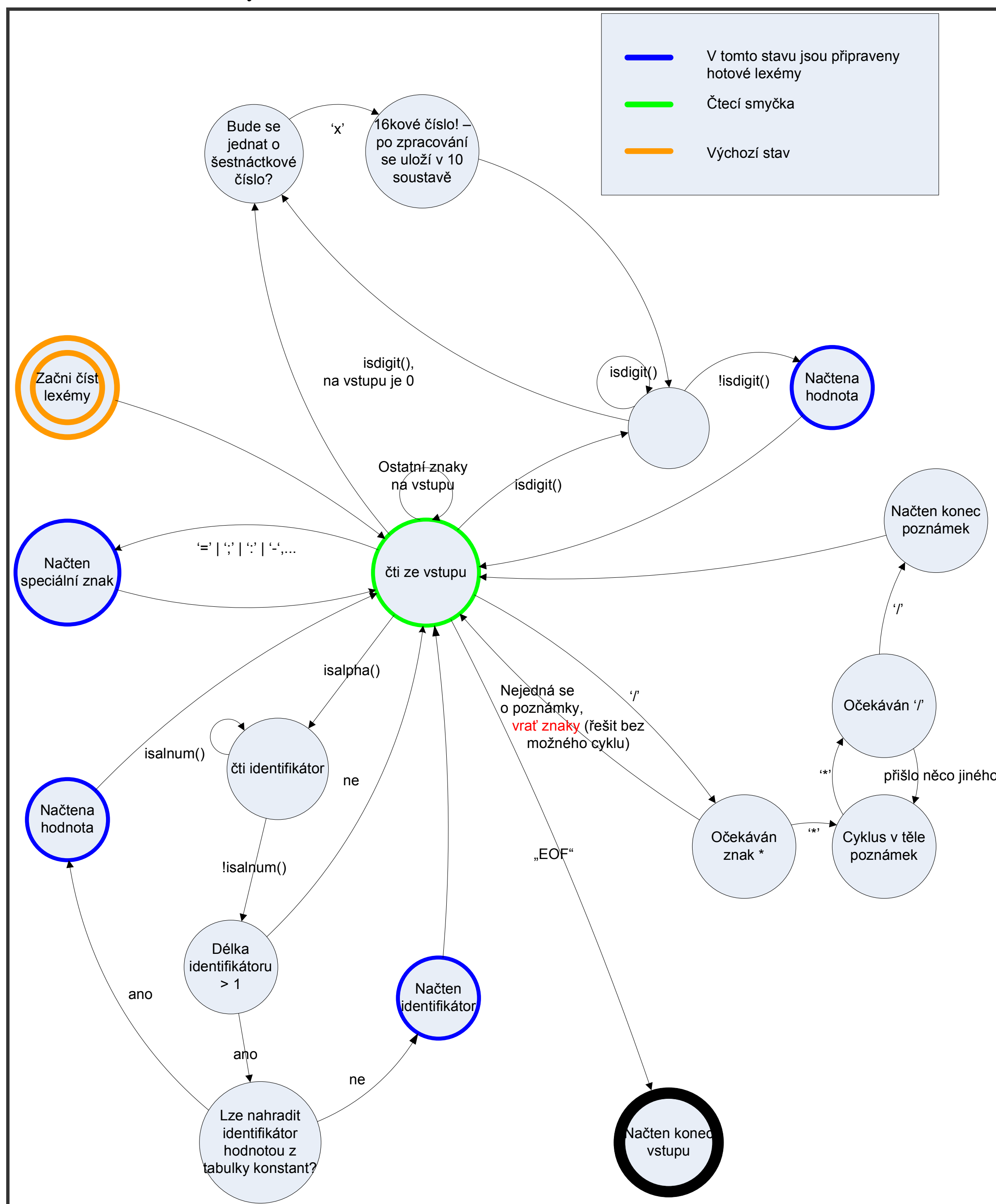
[QtDoc] *Dokumentace k toolkitu Qt v.4, Trolltech, Qt*
Dokument dostupný na URL http://doc.trolltech.com/4.0/index.html

Seznam příloh

Příloha 1. Lexikální analýza modelového souboru
Příloha 2. Syntaktická analýza modelového souboru
Příloha 3. Lexikální analýza zdrojového souboru
Příloha 4. Syntaktická analýza zdrojového souboru
Příloha 5. Zdrojový text modelového souboru
Příloha 6. Text zdrojového souboru v jazyku assembler
Příloha 7. Na CD: programová dokumentace, zdrojové texty a ukázkové příklady

Přílohy

Příloha 1. Lexikální analýza modelového souboru

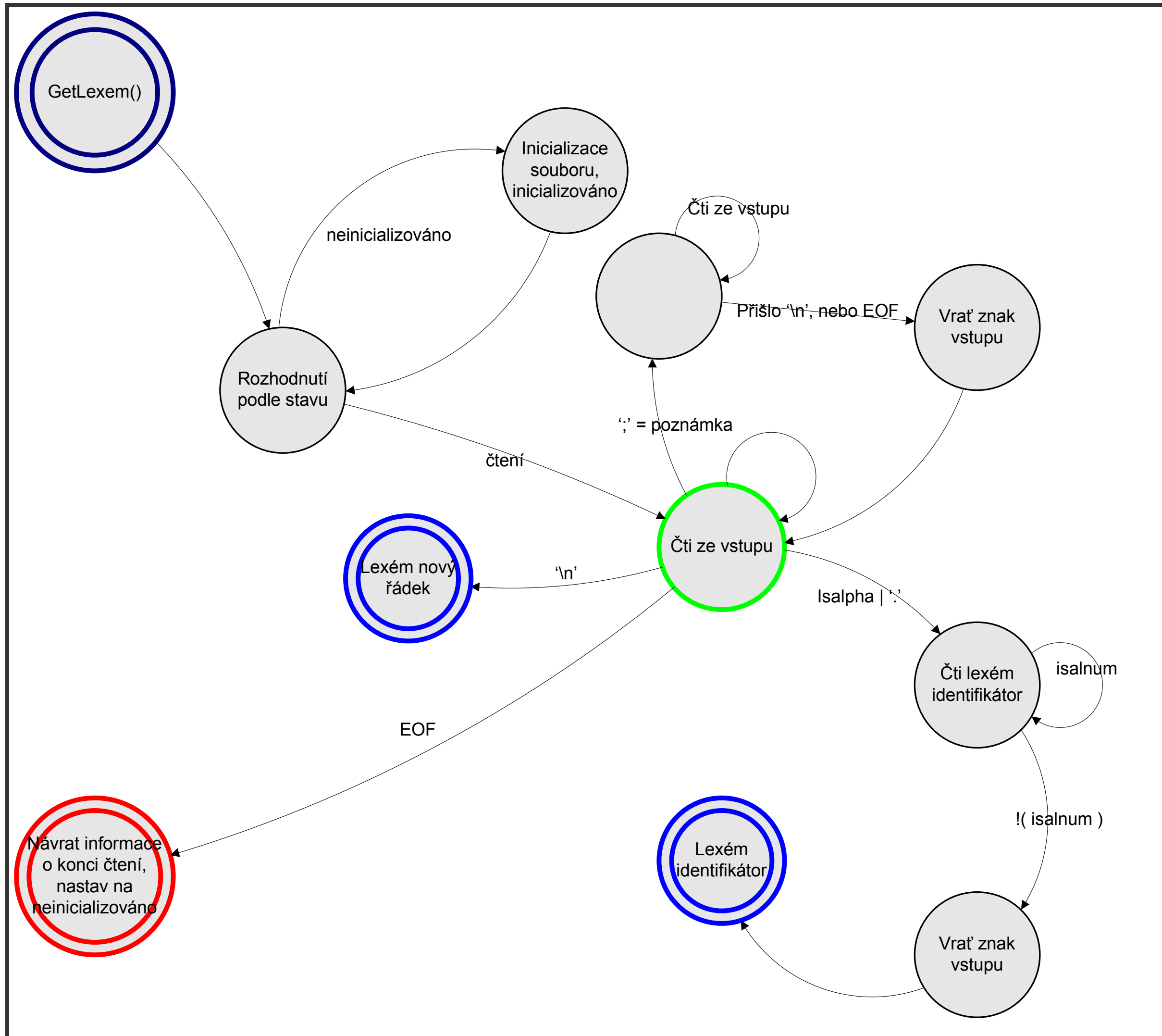


Příloha 2. Syntaktická analýza modelového souboru

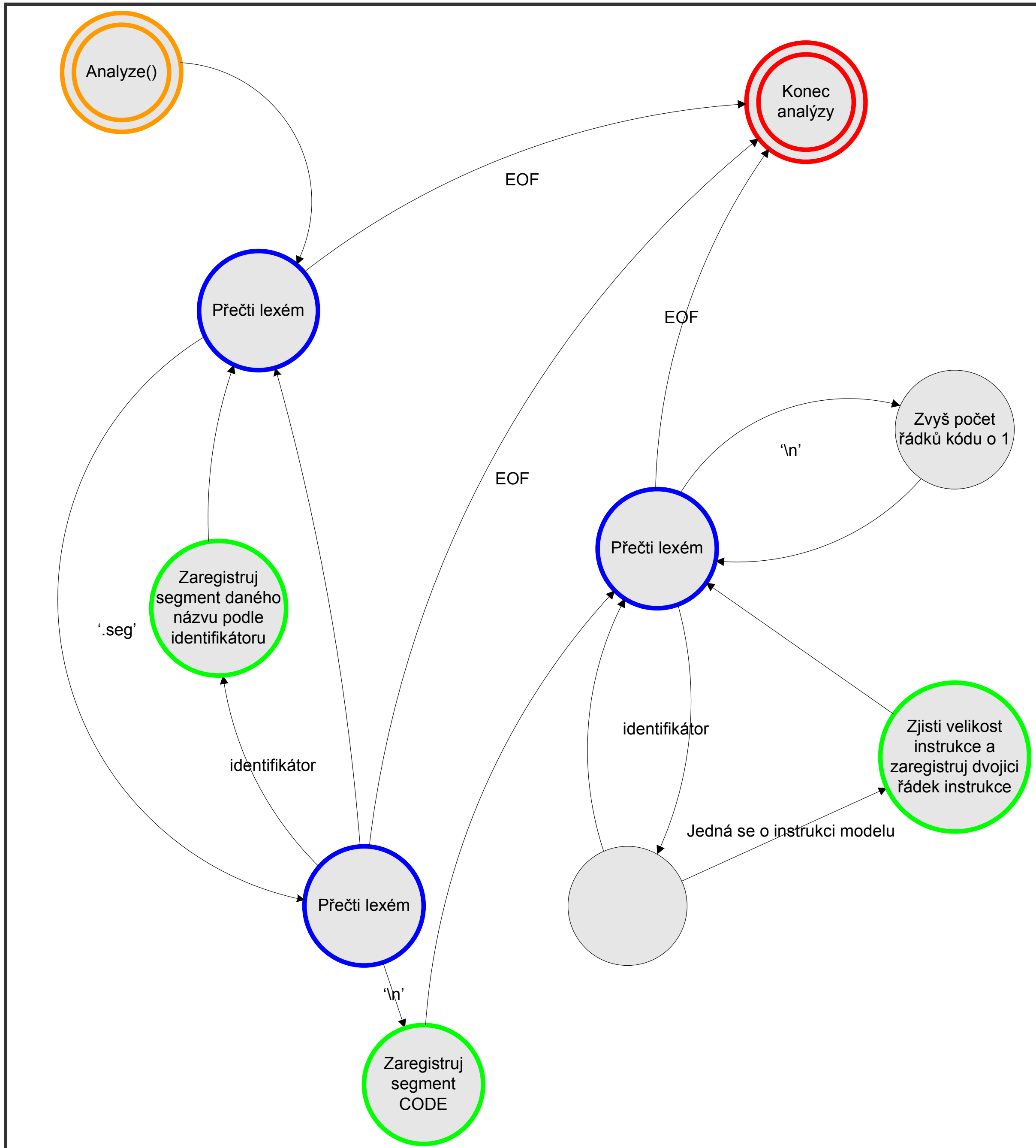
Vyhodnocovací cyklus
Nastavení atributů objektu
Očekávací stav – pokud není splněna podmínka, skáče se do stavu označeného modře...

Spuštění analýzy
Očekáván lexém „processor“
Jiné lexémy
přišel
Očekává se název
Ostatní Lexémy
Hledání konce bloku – i zanořených
Očekávána hodnota
hodnota
„=“
Očekáváno „=“
Ulož konstantu
název
Očekáván název
„const“
‘{‘
}, všechny zanořené uzavřeny
Konec analýzy
„EOF“
„segment“
Očekáváno číslo
„:“
číslo
Zaznamenej informaci o segmentu
Očekávána dvojtečka
Očekáván název nebo dvojtečka
Čekání na známé lexémy
Neznámé lexémy
Jedná se o funkci
‘{‘
Očekáván parametr (ins02)
Očekáván první parametr
‘,’
název
Očekáván oddělovač parametrů
Není další parametr
Čekej na levou kulatou závorku (ins04)
Libovolný lexém
„instruction“
Očekáván název
‘=‘
Nepotřebné Lexémy
Očekávej konec, nebo další
Očekávej název
‘,’
Čekej na lexém ‘/’
Ostatní lexémy
(
‘/’
Jestliže se jedná o název funkce, dosaď parametry funkce
Očekávej číslo
číslo
‘(‘
‘)’
Zvětši velikost instrukce
Zaznamenej instrukci
Nastav vlastnost struktury main
Neznámý lexém
Čekání na známý lexém, resp. pravá složená závorka „}“
„}“
„fetch“
„{“
Očekávána levá složená závorka „{“
ano
Je název instancí paměti? (memory instance)
jiné
„address“ | „memory“
Očekáván znak „=“
jiné
„=“
Očekáván název
„instance“
Zaznamenej instanci clock
„clock“
Ulož informaci o instanci procesoru
ano
Jedná se opravdu o název instance procesoru a hodin?
Očekáván známý lexém
název
Očekáván název
„processor“
„memory“
Očekávej název
Očekávána hranatá levá závorka „[“
„[“
Očekávána hodnota
„]“
hodnota
Očekávána pravá závorka „]“
přišlo cokoliv jiného
Počet dvojic hr. závorek v intervalu 1..2
Očekávej lexém clock
„clock“
Očekávej název
Ulož instanci memory
ok

Příloha 3. Lexikální analýza zdrojového souboru



Příloha 4. Syntaktická analýza zdrojového souboru



Příloha 5. Zdrojový text modelového souboru

```
/*
 * Demonstration processor.
 * Author(s): Tomas Rybka
 * This processor has been designed to demonstrate most important features
 * of gentools programs.
 *
 * The processor utilizes an accumulator architecture, almost all operations
 * are performed on a ACC register.
 *
 * There is a demonstration assembly program provided: test1.asm.
 * It implements the quick-sort algorithm. It is written for easy
 * presentation of assembler, disassembler and simulator as well.
 *
 */

processor TEST
{

        /*
         * Physical memory layout:
         *    0x000..0x0FF: program & data
         *    0x100..0x1FF: stack from top
         */

        const MEM_UNIT = 8;
        const PHYS_ADDR = 9;
        const IP_SIZE = 8;

        segment : MEM_UNIT;
        segment DATA : MEM_UNIT;

        instance MAIN: memory[PHYS_ADDR][MEM_UNIT];
        instance DATA: memory[PHYS_ADDR][MEM_UNIT];
        instance ACC: memory[MEM_UNIT];
        instance IND: memory[MEM_UNIT];
        instance SP: memory[PHYS_ADDR];
        instance IP: memory[IP_SIZE];
        instance ZF: memory[1];
        instance OF: memory[1];

        fetch {
                memory = MAIN;
                address = IP;
        }

        event initialization {
                IP = 0;
                SP = (1 << PHYS_ADDR) - 1;
        }

        event fetched {
                if ((IP + instruction_length) >> 8)
                        terminate;
                IP = (IP + instruction_length)[7..0];
                clock 1;
        }

        /*
         * Operand types
         */

        operand Reg(E/2) symbolic
```

```
{
        ACC:    #01;
        SP:     #10;
        IND:    #11;
}

operand DataAddr(E/MEM_UNIT) address DATA
{       symbolic = E;
        encoding {
                if (value >> MEM_UNIT)
                        throw 0;
                E = value[MEM_UNIT-1..0];
        }
        semantics = E;
}

operand CodeAddr(E/MEM_UNIT) jump address
{       symbolic = E;
        encoding {
                if (value >> MEM_UNIT)
                        throw 0;
                E = value[MEM_UNIT-1..0];
        }
        semantics = E;
}

operand Imm(E/MEM_UNIT) immediate
{       symbolic = E;
        encoding {
                if (value >> MEM_UNIT)
                        throw 0;
                E = value[MEM_UNIT-1..0];
        }
        semantics = E;
}

operand Mem(Cls/2, E/MEM_UNIT = #00000000) struct [base]
{
        base: Reg(Cls);
        semantics {
                if (base == 0b10)
                        return SP;
                if (base == 0b11)
                        return IND;
                return ACC;
        }
}

operand Mem(Cls/2, E/MEM_UNIT) struct [base,ofs]
{
        base: Reg(Cls);
        ofs: Imm(E);
        semantics {
                if (base == 0b10)
                        return SP + ofs;
                if (base == 0b11)
                        return IND + ofs;
                return ACC + ofs;
        }
}

instruction    NOP
        ( E/MEM_UNIT = #10000000 )
{
```

```
}

instruction    LDA    src
        ( E/MEM_UNIT = #01000001,
          EADR/MEM_UNIT )
{
        src: DataAddr(EADR);
        semantics {
                ACC = src[MEM_UNIT-1..0];
        }
}

instruction    LDI    src
        ( E/MEM_UNIT = #11000001,
          EADR/MEM_UNIT )
{
        src: DataAddr(EADR);
        semantics {
                IND = src[MEM_UNIT-1..0];
        }
}

instruction    CMP    src
        ( E/MEM_UNIT = #00100001,
          EADR/MEM_UNIT )
{
        src: DataAddr(EADR);
        semantics {
                ZF = (ACC == src);
                OF = (ACC < src);
        }
}

instruction    LDA    mem
        ( E/MEM_UNIT = #010001xx,
          EADR/MEM_UNIT )
{
        mem: Mem(E[1..0], EADR);
        semantics {
                ACC = DATA[mem];
        }
}

instruction    LDI    mem
        ( E/MEM_UNIT = #110001xx,
          EADR/MEM_UNIT )
{
        mem: Mem(E[1..0], EADR);
        semantics {
                IND = DATA[mem];
        }
}

instruction    CMP    mem
        ( E/MEM_UNIT = #001001xx,
          EADR/MEM_UNIT )
{
        mem: Mem(E[1..0], EADR);
        semantics {
                op: uint[MEM_UNIT] = DATA[mem];
                ZF = (ACC == op);
                OF = (ACC < op);
        }
}
```

```
instruction    STA    mem
        ( E/MEM_UNIT = #011001xx,
          EADR/MEM_UNIT )
{
        mem: Mem(E[1..0], EADR);
        semantics {
                DATA[mem] = ACC;
        }
}

instruction    SWAP   mem
        ( E/MEM_UNIT = #011010xx,
          EADR/MEM_UNIT )
{
        mem: Mem(E[1..0], EADR);
        semantics {
                pause string("Swapping: ", ACC, " <-> ", DATA[mem], "\n");
                tmp: uint[MEM_UNIT] = ACC;
                ACC = DATA[mem];
                DATA[mem] = tmp;
        }
}

instruction    PUSH
        ( E/MEM_UNIT = #00000100 )
{
        semantics {
                if (SP <= 0x100) {
                        print("TEST: Stack overflow\n");
                        terminate;
                }
                DATA[SP = (SP - 1)[PHYS_ADDR-1..0]] = ACC;
        }
}

instruction    POP
        ( E/MEM_UNIT = #00000101 )
{
        semantics {
                if (SP >= 0x1FF) {
                        print("TEST: Stack underflow\n");
                        terminate;
                }
                ACC = DATA[SP];
                SP = (SP + 1)[PHYS_ADDR-1..0];
        }
}

instruction    INC
        ( E/MEM_UNIT = #00000110 )
{
        semantics {
                OF = (ACC == (1 << MEM_UNIT)-1);
                ACC = (ACC + 1)[MEM_UNIT-1..0];
                ZF = (ACC == 0);
        }
}

instruction    DEC
        ( E/MEM_UNIT = #00000111 )
{
        semantics {
                OF = (ACC == 0);
```

```
                ACC = (ACC - 1)[MEM_UNIT-1..0];
                ZF = (ACC == 0);
        }
}

jump instruction      _BRANCH{cond:uint[3],d:int}   addr
        ( E/MEM_UNIT = (mask[MEM_UNIT])(0x10 + (cond << 5))[7..0],
          ETGT/MEM_UNIT )
{
        addr: CodeAddr(ETGT);
        semantics {
                if (cond == 0b001) {
                        if (!ZF) return;
                } else if (cond == 0b000) {
                        if (ZF) return;
                } else if (cond == 0b100) {
                        if (OF || ZF) return;
                } else if (cond == 0b110) {
                        if (!OF || ZF) return;
                } else if (cond == 0b101) {
                        if (OF && !ZF) return;
                } else if (cond == 0b111) {
                        if (!OF && !ZF) return;
                } else {
                        print("TEST: Internal compare error\n");
                        terminate;
                }
                IP = addr[IP_SIZE-1..0];
        }
}

instruction    JE     = _BRANCH{0b001,0};
instruction    JNE    = _BRANCH{0b000,0};
instruction    JA     = _BRANCH{0b100,0};
instruction    JB     = _BRANCH{0b110,0};
instruction    JAE    = _BRANCH{0b101,0};
instruction    JBE    = _BRANCH{0b111,0};

jump instruction      JMP    addr
        ( E/MEM_UNIT = #00001000,
          ETGT/MEM_UNIT )
{
        addr: CodeAddr(ETGT);
        semantics {
                IP = addr[IP_SIZE-1..0];
        }
}

instruction    CALL   addr
        ( E/MEM_UNIT = #00001100,
          ETGT/MEM_UNIT )
{
        addr: CodeAddr(ETGT);
        semantics {
                if (SP <= 0x100) {
                        print("TEST: Stack overflow\n");
                        terminate;
                }
                DATA[SP = (SP - 1)[PHYS_ADDR-1..0]] = IP;
                IP = addr[IP_SIZE-1..0];
        }
}

jump instruction      RET
```

```
		( E/MEM_UNIT = #00001101 )
	{
		semantics {
			if (SP >= 0x1FF) {
				print("TEST: Stack underflow\n");
				terminate;
			}
			IP = DATA[SP];
			SP = (SP + 1)[PHYS_ADDR-1..0];
		}
	}

	jump instruction	HALT
		( /MEM_UNIT = #00000000 )
	{
		semantics {
			terminate;
		}
	}


}

instance CLK: clock;
instance PROC: processor TEST clock CLK;
```

Příloha 6. Text zdrojového souboru v jazyku assembler

```
; Author(s): Tomas Rybka
; A quick-sort implementation, in the TEST processor language (test1.ad).
;


	.seg

	LDA	end
	DEC
	PUSH
	LDA	start
	PUSH
	CALL	qsort
	HALT

	.db	0	; force an error

;
;	[SP+1] = start
;	[SP+2] = end
;
qsort:
	; initialize local variables
	LDA	[%SP,1]	; read starting address
	CMP	[%SP,2]
	JBE	int_ok	; nonempty interval, continue
	RET
int_ok:	PUSH		; make a variable
	LDA	[%ACC]	; load a pivot
	PUSH		; make a variable with a pivot value
	LDA	[%SP,4]	; read end address
	PUSH		; make a variable

	; from now:
	; SP+0 = right
```

```
        ; SP+1 = pivot
        ; SP+2 = left
        ; SP+4 = start
        ; SP+5 = end

        ; find a new leftmost greater one
div:
        LDA     [%SP,2]; load a left index
ld:     LDA     [%ACC] ; load a value
        CMP     [%SP,1]; compare with a pivot
        JAE     ld_ex  ; less then pivot? continue
        LDA     [%SP,2]; load the left index
        INC            ; increment it
        STA     [%SP,2]; store the new value
        JMP     ld     ; check another one
ld_ex:

        ; find a new rightmost less one

        LDA     [%SP]  ; load a right index
rd:     LDA     [%ACC] ; load a value
        CMP     [%SP,1]; compare with a pivot
        JBE     rd_ex  ; greater than pivot? continue
        LDA     [%SP]  ; load the right index
        DEC            ; decrement
        STA     [%SP]  ; store the new value
        JMP     rd     ; check another one
rd_ex:

        ; validate a change

        LDA     [%SP,2]; load the left index
        CMP     [%SP]  ; compare with the right index
        JA      sort   ; left > right? exit
        LDA     [%ACC] ; load a left value
        LDI     [%SP]  ; load a right index into aux register
        SWAP    [%IND] ; swap with value at 'right'
        LDI     [%SP,2]; load a left index
        STA     [%IND] ; store a new left value
        LDA     [%SP,2]; load a left index
        INC            ; increment
        STA     [%SP,2]; store the new left index
        LDA     [%SP]  ; load a right index
        DEC            ; decrement
        STA     [%SP]  ; store a new right index
        JB      snd
        CMP     [%SP,2]; compare with the left index
        JAE     div    ; right >= left? continue
sort:
        LDA     [%SP]
        CMP     [%SP,4]
        JBE     snd
        PUSH
        LDA     [%SP,5]
        PUSH
        CALL    qsort
        POP
        POP
snd:
        LDA     [%SP,5]
        CMP     [%SP,2]
        JBE     exit
        PUSH
        LDA     [%SP,3]
```

```
        PUSH
        CALL    qsort
        POP
        POP
exit:   POP
        POP
        POP
        RET

;
; Data for the program
;

        .seg    DATA
start:
        .db     14
        .db     1
        .db     122
        .db     47
        .db     40
        .db     254
        .db     9
end:
```