VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

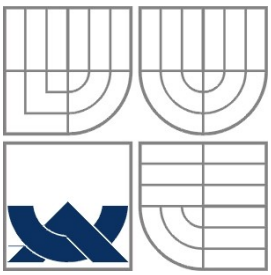
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

LADICÍ NÁSTROJ GENERICKÝCH SIMULÁTORŮ PROCESORŮ

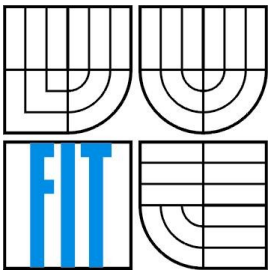
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE BC. MILAN WILCZÁK
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

LADICÍ NÁSTROJ GENERICKÝCH SIMULÁTORŮ PROCESORŮ

DEBUGGER FOR GENERIC MICROPROCESSOR SIMULATORS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. MILAN WILCZÁK

VEDOUCÍ PRÁCE
SUPERVISOR

ING. ZDENĚK PŘIKRYL

BRNO 2008

Abstrakt

Procesory s aplikačně specifickou instrukční sadou se stávají součástí každodenního života, přestože obvykle nejsou na první pohled vidět. Při jejich vývoji je potřeba nějak popsat jejich architekturu, instrukční sadu a chování. Aby jejich vývoj měl smysl, musí být možné pro tyto procesory vytvářet aplikace a při vytváření aplikací se dělají chyby. K jejich objevování slouží debuggery. Tato práce shrnuje některé základní informace pro vytváření debuggerů a popisuje implementaci debuggeru pro procesory vytvářené pomocí projektu Lissom.

Abstract

Application specific instruction set processors become part of every day life although it's not always visible at first sight. During their development it's needed to somehow describe their architecture, instruction set and behavior. To make their developement worth, it's necessary to be able to create applications for these processors and during application development errors are always made. Debuggers serve to discover and help fixing them. This paper summarises some basic information to debugger development and describes implementation for processors created using the Lissom project.

Klíčová slova

Lissom, debugger, DWARF, HW/SW co-design, procesory s aplikačně specifickou instrukční sadou.

Keywords

Lissom, debugger, DWARF, HW/SW co-design, application specific instruction set processors.

Citace

Milan Wilczák: Ladicí nástroj generických simulátorů procesorů, diplomová práce, Brno, FIT VUT v Brně, 2010

Ladicí nástroj generických simulátorů procesorů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Zdeňka Přikryla.
Další informace mi poskytl Karel Masařík.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Milan Wilczák
26. května 2010

Obsah

Obsah....1
1 Úvod....3
2 Principy debuggerů....4
2.1 Co je debugger....4
2.2 Architektura debuggerů....5
2.2.1 Typické GUI debuggeru....5
2.2.2 Jádro debuggeru....6
2.3 Hardwarová podpora ladění....6
2.4 Rozhraní operačního systému....7
2.5 Breakpointy....8
2.6 Zkoumání kontextu programu a dat....9
3 Ladicí formát DWARF....11
3.1 Obecné informace....11
3.2 Ladicí záznamy....11
3.3 Popis umístění....12
3.4 Záznamy o instrukcích, datech a typech....13
3.5 Informace o řádcích programu....13
3.6 Informace o rozložení rámce....14
3.7 Použití v projektu Lissom....14
4 Debugger GDB....15
4.1 Základní příkazy GDB....15
4.2 Podpora skriptování....15
4.3 Vestavěné simulátory procesorů....16
4.4 Použití v projektu Lissom....16
5 Architektura Lissom....17
5.1 Rozdělení nástrojů....17
5.2 Třívrstvá architektura....17
5.2.1 Prezentační vrstva....17
5.2.2 Middleware....18
5.2.3 Spouštění simulace....18
5.3 Komunikační protokol....19
5.4 Jazyk ISAC a assembler....19
5.4.1 Jazyky pro popis procesorů....19
5.4.2 Schopnosti jazyka ISAC....19
5.4.3 Možnosti assembleru....20
5.5 Distribuovaná simulace....20
6 Příkazová řádka....22
6.1 Používání příkazové řádky....22
6.1.1 Spuštění....22
6.1.2 Obecné zákonitosti....22
6.1.3 Základní režim....23
6.1.4 Ladicí režim....24

6.1.5 Dávkové zpracování....25
6.2 Návrh a implementace....26
6.2.1 Návrh tříd....26
6.2.2 Informace o příkazech....27
6.2.3 Postup zpracování příkazů....27
6.2.4 Automatické doplňování....28
7 Ladicí knihovna....29
7.1 Napojení simulátoru a ladicí knihovny....29
7.2 Architektura ladicí knihovny....29
7.3 Spustitelný soubor....30
7.3.1 Formát spustitelného souboru....30
7.3.2 Čísla řádků....30
7.3.3 Tabulka symbolů....31
7.4 Řízení vykonávání programu....31
7.5 Podpora breakpointů....32
7.5.1 Fyzické breakpointy....33
7.5.2 Logické breakpointy....33
7.5.3 Správce breakpointů....34
7.5.4 Protokol pro nastavování breakpointů....34
7.5.5 Zásah breakpointu....34
7.6 Zpracování výrazů....36
7.6.1 Syntax výrazu....36
7.6.2 Typový systém....37
7.6.3 Překladač výrazu....37
7.6.4 Podmínky v breakpointech....38
7.6.5 Uzly výrazů....38
8 Závěr....40
Literatura....41
Obsah přiloženého CD....42

1 Úvod

Při vývoji procesorů s aplikačně specifickou instrukční sadou (ASIP, application specific instruction set processors) je třeba, aby se pro něj daly vyvíjet aplikace. K tomu je potřeba překladač jazyka symbolických instrukcí nebo vyššího programovacího jazyka a linker. Protože téměř každá aplikace obsahuje chyby, je potřeba i nástroj pro jejich ladění, debugger. V projektu Lissom se tyto nástroje generují z popisu procesoru v jazyce ISAC (Instruction Set Architecture C), kde je popsán i příslušný jazyk symbolických instrukcí.

Projekt Lissom má za cíl vytvořit prostředí pro tvorbu procesorů ASIP za základě jazyka ISAC. Je možné ho používat pro architektury s pipeline, VLIW procesory (jejich význam roste) i složitější architektury. Prostředí je rozvrstvené, prezentační část je k dispozici i pro architekturu Eclipse. Podobný cíl mají i další projekty, například Shankya nebo LISAtek.

Hardware-software co-design je problematika souběžného návrhu technického a programového vybavení. Část aplikace běží na programovatelném procesoru, část v hardware nebo s hardwarovou podporou softwaru. Vývoj je kompromisem mezi rychlostí výpočtů a flexibilitou. ASIP se snaží maximalizovat výkon pro svou třídu aplikací při zachování flexibility, kterou mají procesory pro obecné použití. Často se používají jako součást jednočipových systémů.

Při ladění procesorů a nástrojů pro ně lze ladit obě součásti - procesor i program pro něj. Pomocí bezchybného programu lze kontrolovat, zda procesor zpracovává instrukce správně, naopak pro bezchybně napsaný procesor lze ladit aplikace. Tento debugger má, mimo jiné, prostřednictvím dávkového zpracování ve stylu debuggeru GDB umožňovat automatické ladění, resp. kontroly funkčnosti v obou směrech.

První kapitola se zaměřuje na obecná pravidla a schopnosti debuggerů. Cílem je vysvětlit, jak fungují běžné ladicí nástroje a co je potřeba pro jejich vývoj. Druhá kapitola popisuje formát DWARF, který slouží k uložení informací pro ladění. Třetí kapitola stručně popisuje debugger GDB, na němž lze vystavět grafický debugger i automatické testování pomocí dávkového zpracování. Čtvrtá kapitola představuje architekturu projektu Lissom, jeho součásti a použití. Pátá kapitola popisuje návrh a implementaci klienta pro příkazovou řádku. Šestá kapitola obsahuje popis ladicí knihovny a architektury simulační vrstvy.

2 Principy debuggerů

2.1 Co je debugger

Debuggerem označujeme nástroj, pomocí kterého lze v programech hledat chyby a poté je opravit. Umožňuje program zastavit na vývojářem definovaných místech, zobrazovat obsahy proměnných, někdy obsah i změnit. Debugger používá typicky původní vývojář, ale i tester, správce programu, v každém případě ale programátor.

Ladění probíhá opakovaným spouštěním programu (někdy po předchozích úpravách překladačem, aby výstup obsahoval informace pro debugger) ve spolupráci s debuggerem. Debugger řídí vykonávání aplikace pomocí speciálních prostředků nabízených operačním systémem. Vykonávání programu je typicky řízeno breakpointy a krokováním.

Breakpoint je speciální kód vložený do programu, který při vykonání vyvolá výjimku nebo jinak předá řízení debuggeru s tím, že se program zastaví. Debugger se o tom dozví a zkoumá, proč a kde k zastavení došlo - mohlo k tomu dojít kvůli breakpointu, výjimce (například dělení nulou, špatným přístupem do paměti, signálu, ...).

Debuggery lze dělit na samostatné a integrované nebo na příkazové a grafické. V současnosti se používají hlavně integrované debuggery v grafickém prostředí. Samostatný debugger je takový, že se spustí debugger, v něm se otevře program a debugger se stará o vypisování zdrojového kódu, proměnných, nastavování a zobrazování breakpointů a celé rozhraní. U integrovaného se o rozhraní stará IDE (Integrated Development Environment), které s debuggerem komunikuje. Výhodou je, že debugger pak může využívat pokročilých funkcí IDE a uživatel si nemusí zvykat na nové prostředí.

Příkazové debuggery přijímají vstupy z příkazové řádky a tedy lze přesměrovat vstup a výstup do souboru nebo roury, kterou může obsluhovat jiný program a tedy nad nimi lze vytvářet různé nadstavby a skripty. Grafické debuggery narozdíl od konzolových musí (resp. musely) řešit problém, kdy se program zastaví a kdy na tento program čekají události z grafického systému. Bez jejich zpracování nemůže pokračovat žádná aplikace v systému, tedy ani debugger. Hlavně ale musí zpracovávat zároveň události od GUI a zároveň od debugovacího API.

Důležitým principem debuggerů je tzv. Heisenbergův princip [1]. Ten říká, že činností debuggeru se nesmí změnit chování laděného programu. Tento princip je problém dodržet, zejména pokud chování programu závisí na čase nebo pokud kontroluje svůj kód, kdy by dělaly problémy breakpointy a musely by být implementovány hardwarem (například debugovací registry procesorů Intelu [4]).

Druhým principem je pravdivost debugování, tedy že to co debugger prezentuje jako chování a stav programu, musí být pravdivé. V opačném případě by byl debugger k ničemu, vývojář by pak hledal chybu někde jinde, než kde opravdu je. K tomuto problému může dojít hlavně u zobrazování obsahu proměnných kvůli ukládání registrů nebo použití registru pro dočasné uložení hodnoty proměnné.

Debuggery lze navíc ještě rozdělit na zdrojové a strojové podle úrovně, na jaké pracují. Například debugger BASICu může interpretovat přímo jednotlivé příkazy, to platí i pro různé

skriptovací jazyky. Na strojové úrovni pracují debuggery pro kompilované jazyky. Ty musí řádkům programu přiřadit skupiny instrukcí, které je reprezentují. K tomu používají ladicí informace generované překladačem.

2.2 Architektura debuggerů

Architektura debuggerů je rozdělena do několika vrstev. Na nejvyšší je uživatelské rozhraní. To představuje několik druhů oken rozdělených podle jejich funkce. Druhou vrstvou je jádro debuggeru, které zpracovává události z laděného programu a komunikuje s GUI. Na nižší úrovni je API operačního systému, které umožňuje práci s laděným programem.

2.2.1 Typické GUI debuggeru

Jedním z oken je zobrazení zdrojového kódu obvykle i s breakpointy. Smyslem je vytvořit iluzi, že se ladí program ve zdrojovém kódu a zakrýt realitu, že se pracuje se strojovým kódem procesoru. Toto okno je v integrovaných prostředích často editorem zdrojového kódu. Během provádění je označením řádku (šipkou nebo podbarvením) zobrazeno aktuální místo provádění.

Dalším z oken je zobrazení zásobníku. Zde jsou zobrazeny volání procedur a funkcí, nejlépe symbolicky jako jméno funkce a hodnoty parametrů, volitelně mohou být zobrazeny lokální proměnné. Některé procesory ukazatel zásobníku nemají, v takových případech se okno zásobníku nemá smysl.

Zobrazení breakpointů dává přehled o všech uživatelsky definovaných breakpointech. Breakpointy jsou kritickým nástrojem, protože umožňují řízeně zastavovat program. Breakpointy jsou obvykle definovány souborem a řádkem programu nebo názvem funkce. Ve výjimečných případech se použije přímo adresa. Breakpointy mohou být podmíněné, tj. při aktivaci se nejdříve ověří splnění podmínky a program se skutečně zastaví jen při jejím splnění. U breakpointů je možné i nastavit počet průchodů, než k zastavení dojde. Při zastavení programu se nalezne, který breakpoint se aktivoval, vyhledá se příslušný soubor a řádek programu a poté se v okně zdrojového kódu skočí na správný řádek.

Zobrazení CPU pracuje na nízké úrovni. Ukazuje program, jak ho vidí procesor a také stav procesoru a paměti. V okně jsou vidět přímo instrukce programu prokládané zdrojovým kódem a také jsou zobrazeny registry procesoru, obsah zásobníku (pokud je přítomen) a obsah paměti.

Okno proměnných zobrazuje proměnné, které jsou z aktuálního kontextu vidět. U strukturovaných typů je možné je rozbalit a zobrazit jejich složky.

Okno evaluátoru nebo inspektoru umožňuje zadat výraz, který se vykoná a zobrazí se jeho výsledek. Toto může způsobit vedlejší efekty, těm je obvykle třeba se vyvarovat. Pomocí tohoto okna lze vykonávat kód, který v programu ve skutečnosti není.

V příkazových debuggerech se nevyskytují okna, ale pomocí příkazů lze všechny tyto informace a funkce zobrazit a použít.

2.2.2 Jádro debuggeru

Tato vrstva obsluhuje všechny pohledy zmíněné výše. Laděná aplikace je z pohledu operačního systému proces. Debugger může nový proces spustit nebo se připojit k již existujícímu procesu, v obou případech se tento proces stane debugovaným. Na konci debugování jádro proces ukončí, nechá ho skončit normálně nebo se od něj odpojí.

Jádro je zodpovědné za správu tabulky symbolů, což je obvykle část spustitelného souboru, která obsahuje informace pro mapování symbolických jmen na adresy a datové typy, další části jsou určeny pro mapování čísel řádků v souborech na adresy a jiné ladící informace. Tyto ladící informace jsou nezbytné pro nastavování breakpointů, získávání obsahu proměnných a řízení vykonávání kódu.

Řízení vykonávání kódu zajišťuje iluzi vykonávání zdrojového kódu místo instrukcí, pokračování programu do příštího breakpointu, krokování po instrukcích nebo příkazech, zpracování výjimek v programu a vyhodnocování výrazů v kontextu debugovaného programu.

Vyhodnocování výrazů je proces, kdy se výraz spustí jakoby v rámci vykonávaného programu. Aby to bylo možné, musí debugger rozumět laděnému programu na úrovni překladače - musí znát pozice proměnných, datové typy a jak se mají volat funkce. Cílem je umožnit komplexní zjišťování stavu programu a jeho modifikaci. Ve spojení s breakpointy je možné chybu virtuálně napravit a po ověření bez potřeby rekompilace je možné opravu provést ve zdrojových souborech.

2.3 Hardwarová podpora ladění

Pro ladění musí hardware poskytovat několik nutných operací. Jedná se o podporu breakpointů, zachycení výjimek a možnost číst a zapisovat do hardwarových registrů, když nastane přerušení a to včetně programového čítače.

Prvním z prostředků poskytovaných hardwarem je možnost nastavit breakpoint, tedy nějakou instrukci, pomocí které je možné program zastavit a předat řízení (skrz operační systém) debuggeru. Toho lze dosáhnout specializovanou instrukcí, u procesorů Intel se jedná i *INT3*, u MIPS pomocí *BREAK*, u architektury Alpha *BPT*, jinde (například PowerPC) neplatnou instrukcí. Obvykle má tato instrukce stejnou délku jako nejkratší možná instrukce. Instrukce breakpointu vyvolá výjimku, kterou zachytí operační systém a ten předá informaci debuggeru. Některé architektury (například x86 [4]) podporují i hardwarové breakpointy, které se nastaví pomocí debugovacích registrů. Jejich výhodou je, že narozdíl od instrukcí pro breakpointy se nevkládají debuggerem do kódu programu a mohou být umístěny i na místa v ROM.

Breakpointy mohou být doplněny podporou pro krokování po instrukcích. Jedná se o to, že po návratu do laděného programu se provede pouze jedna instrukce a hned poté dojde k výjimce a řízení se přes operační systém dostane do debuggeru. Tato hardwarová podpora krokování usnadňuje, ale není nezbytně nutná. Stejného efektu lze dosáhnout nastavením breakpointu na další instrukci, to ovšem vyžaduje znalost umístění následující instrukce, tedy délky instrukce a u skoků adresu cíle skoku. Krokování po řádcích vždy vyžaduje použití breakpointů.

Další prostředek, který hardware musí podporovat, je zachycení hardwarových výjimek. Sem patří dělení nulou, chybný přístup do paměti, chybná instrukce a další. Tyto výjimky zachytí operační systém a předá je debuggeru.

Dalším podporou může být podpora watchpointů, tedy tzv. datových breakpointů. K nim dochází, když se proměnná (nebo výsledek výrazu) změní. Tato podpora může být buď přímo hardwarová, tedy dojde k výjimce při přístupu k plně definovanému rozsahu adres, nebo může být zajištěna pomocí stránkování. Watchpointy existují, aby bylo možné najít chybu způsobenou špatným zápisem do proměnné.

2.4 Rozhraní operačního systému

Debugger je z hlediska operačního systému obyčejná aplikace bez žádných speciálních práv. Aby bylo možné debugovat proces, musí k němu mít debugger přístup. Debugger nejdříve informuje operační systém, že chce vybraný proces ladit. Poté může pomocí speciálních volání číst a modifikovat paměť a registry procesu a obnovit vykonávání procesu. Když v procesu nastane výjimka nebo breakpoint, je proces zastaven a debugger informován.

Předávání řízení mezi debuggerem a aplikací je drahý proces z hlediska času, nemělo by k tomu tedy docházet zbytečně často. K přepnutí kontextu mezi debuggerem a debugovanou aplikací dochází nejen při předávání řízení, ale i při přístupu do paměti debugovaného procesu debuggerem, například při čtení obsahu zásobníku.

Pro práci debuggeru s aplikací se používá debugovací API operačního systému, různé operační systémy mají různá API. Například UNIXové operační systémy používají rozhraní ptrace. Mezi jeho operace patří (vynechávám předponu *PTRACE_*): *TRACEME* (zahájění ladění), *PEEKTEXT*, *PEEKDATA*, *POKETEXT*, *POKEDATA* (čtení a zápis do paměti programu), *CONT*, *KILL*, *SINGLESTEP* (řízení běhu programu), *ATTACH*, *DETACH* (připojení se k již běžícímu procesu), *GETREGS* a *SETREGS* (zjišťování a nastavování obsahu registrů). Debugger čeká pomocí volání wait, až se aplikace zastaví (jakoby efekt *SIGSTOP*). Novější UNIXové systémy používají rozhraní /proc, kde se používají standardní souborové operace a zejména funkce *ioctl*.

V případě debugovacího API Win32 se začíná debugovat buď vytvořením procesu pomocí *CreateProcess* se speciálním flagem nebo se použije volání *DebugActiveProcess* pro připojení existujícího procesu. Narozdíl od ptrace zde není možné se od procesu odpojit. Breakpointy a přístup do paměti procesu se zajišťuje pomocí volání *ReadProcessMemory* a *WriteProcessMemory*. Na události v aplikaci se čeká blokováním nebo dotazováním přes *WaitForDebugEvent*. Mezi debugovací události patří (vynechávám příponu *_DEBUG_*EVENT): *CREATE_PROCESS*, *CREATE_THREAD*, *EXCEPTION*, *EXIT_PROCESS*, *EXIT_THREAD*, *LOAD_DLL* a *UNLOAD_DLL*. Zásah do breakpointu se hlásí jako *EXCEPTION_DEBUG_EVENT*. Debugger může získat informace o kontextu debugované aplikace pomocí *GetThreadContext* a *SetThreadContext*, mezi tyto informace patří stav registrů včetně programového čítače a ukazatele zásobníku (ten pak lze zkoumat pomocí *ReadProcessMemory*). Řízení vykonávání je zajištěno operacemi *SuspendThread* a *ResumeThread*.

2.5 Breakpointy

Breakpointy jsou klíčem k řízení vykonávání programu. Téměř každý algoritmus řízení vykonávání zahrnuje v nějakém bodě breakpoint, z pohledu uživatele neviditelný, například *step into, step out* nebo *run to here.*

Uživatel může vkládat breakpointy na úrovni zdrojového kódu pomocí specifikace souboru a čísla řádku nebo na úrovni instrukcí zadáním přímo adresy nebo symbolu reprezentujícího první instrukci funkce. Některé vysokoúrovňové breakpointy mohou znamenat mnoho fyzických adres, to platí především pro C++ šablony nebo inline funkce. Na druhé straně může několik uživatelem definovaných breakpointů ukazovat na stejnou fyzickou adresu i stejný řádek zdrojového kódu. Některé breakpointy mohou být definovány jako dočasné, tzn. po prvním vykonání se zruší. U breakpointy lze definovat počet ignorování, tj. několik vykonání breakpointu nezpůsobí žádnou viditelnou akci, nebo podmínku, při jejímž splnění se program na breakpointu zastaví. V některých případech lze nastavovat breakpointy i na moduly (např. DLL knihovny), které ještě nebyly načteny, ke skutečné aktivaci těchto breakpointů dojde až při načtení těchto modulů. Interní breakpointy se musí chovat tak, aby o nich uživatel vůbec nevěděl.

Podle úrovně se rozdělují breakpointy na logické a fyzické. Logický breakpoint je informace, na jakém řádku a v jakém souboru se breakpoint nachází, případně symbolické označení funkce nebo přímo adresa instrukce. Logické breakpointy mohou obsahovat podmínky pro jejich aktivaci, informaci o počtu průchodů nebo zda je breakpoint aktivní. Fyzický breakpoint je umístěn v kódu laděného programu a informace o fyzickém breakpointu v debuggeru obsahuje jeho adresu, originální obsah paměti před nahrazením speciální instrukcí a počet logických breakpointů, které na něj ukazují. Logické breakpointy se mapují na fyzické ve vztahu N:M, tedy jeden logický lze mapovat na více fyzických a na jeden fyzický může ukazovat několik logických. Fyzický breakpoint může na jedno místo ukazovat pouze jeden. Kvůli urychlení mívá fyzický breakpoint ve svých informacích seznam všech svých logických breakpointů.

Vytvoření logického breakpointu je vcelku snadné, jedná se pouze o jednoduchou strukturu. Podle souboru a řádku, případně z tabulky symbolů se určí adresy fyzických breakpointů. Na tyto se bude odkazovat z logického breakpointu. Fyzické breakpointy se vytvoří, případně se jim inkrementuje čítač referencí. Pokud fyzický breakpoint neexistuje a je vytvářen, z dané fyzické adresy se načte aktuální obsah, uloží se do struktury a na adresu se uloží instrukce pro breakpoint.

Když se při běhu vyvolá výjimka, zkontroluje se programový čítač a debugger se pokusí najít příslušný fyzický breakpoint. Podle něj se najdou všechny logické breakpointy. U každého nalezeného breakpointu se ověří podmínky. První breakpoint se splněnými podmínkami se použije pro určení souboru a řádku pro okno zobrazení zdrojového programu.

Dočasné breakpointy jsou takové, že se po prvním zásahu smažou. Interní breakpointy nejsou viditelné uživateli, ale jsou důležité pro správnou funkci různých algoritmů debuggeru. Interní a dočasné breakpointy jsou určeny atributem ve struktuře logického breakpointu. Interní (a často zároveň dočasné) breakpointy se používají pro funkce jako *step into, step over, step out* nebo *run to here*.

Breakpointy lze využít tak, že při jejich aktivaci se provedou uživatelem definované akce a pak se běh programu obnoví. Těmito akcemi může být výpis proměnných nebo jiných dat. Dále je možné

breakpoint aktivovat až po jeho několikáté aktivaci. Pomocí kombinace podmíněných breakpointů a akcí je možné opravit program bez nutnosti rekompilace, změny se ovšem neuloží do spustitelného souboru.

Krokování je důležitou součástí debuggeru, umožňuje provádět kód řádek po řádku případně po instrukcích. Krokování lze rozdělit na dvě varianty podle toho, jak se chovají k volání funkcí a procedur: step into krokuje dovnitř funkce, step over se zastaví až po vykonání funkce.

Funkce *step into* na úrovni příkazů se realizuje pomocí simulování běhu po instrukcích a hledání řádku, který se vykoná jako další. Pokud virtuální ukazatel neukazuje na aktuální místo zastavení a zároveň představuje první instrukci nějakého řádku, algoritmus končí a vrátí tento řádek. Jinak se u neskokových instrukcí inkrementuje virtuální ukazatel programového čítače a pokračuje se další instrukcí. U skokových instrukcí se buď zjistí skutečný cíl skoku a pokračuje se instrukcí na cíli, nebo se skok provede hardwarovým krokováním. V obou případech se cyklus opakuje jako u další instrukce. Na řádek vrácený tímto algoritmem se vloží dočasný interní breakpoint a program pokračuje až do tohoto breakpointu.

Funkce *step over* funguje podobně, rozdíl je jen ve zpracování skokových instrukcí typu CALL. Tato instrukce se provede a hned za tuto instrukci nebo na návratovou adresu se vloží dočasný interní breakpoint, takže místo aby se krokovalo uvnitř volání, se volání vykoná celé a pokračuje se až za ním.

Funkce *step out* přečte ze zásobníku návratovou adresu a na ni umístí breakpoint. Funkce *run to here* jen umístí dočasný breakpoint na vybrané místo.

Krokování tímto simulováním se provádí kvůli tomu, že krokování je řádově pomalejší než běh plnou rychlostí do dalšího breakpointu.

Jako speciální druh breakpointu se dají považovat již zmíněné watchpointy. Ty umožňují sledovat změny dat v paměti. Místo vyvolání instrukcí breakpointu se vyvolají výjimkou přístupu do paměti, poté je nutné ověřit místo v paměti, zda se jedná o hlídané místo.

2.6 Zkoumání kontextu programu a dat

Nejdůležitější službou debuggeru při zastavení programu je odpověď na otázku "kde jsem?". Toho lze dosáhnout pomocí programového čítače, jeho převod na soubor a číslo řádku a zobrazení zdrojového kódu v okolí tohoto místa, nejlépe najetím na tento řádek v editoru IDE.

Druhou nejdůležitější je odpověď na otázku "jak jsem se sem dostal?". Tuto otázku dokáže zodpovědět zásobník, ale jen ve spolupráci s vhodnými debugovacími informacemi. Na zásobník se při volání podprogramu uloží návratová adresa, při návratu se tato adresa použije jako cíl skoku. Kromě návratové adresy se na zásobník ukládá předchozí ukazatel rámce a poté lokální proměnné a uložené registry. Před návratovou adresou jsou obvykle parametry funkce. Na základě zásobníku lze sestavit tzv. *stack trace*, tedy seznam aktivací funkcí a procedur. Pokud se používá ukazatel zásobníku a ukazatel rámce běžným způsobem, lze tento seznam vytvořit i bez debugovacích informací, ale budou chybět parametry funkcí. Rámcem se myslí data na zásobníku, kde je uložena návratová adresa a lokální proměnné. Běžný způsob znamená, že je na zásobníku uložena návratová adresa a hned potom předchozí ukazatel rámce. Aktuální ukazatel rámce ukazuje na předchozí

ukazatel rámce atd. Přitom je potřeba brát ohled na speciální případy, kdy se program zastaví v rámci prologu funkce, tzn. v části kódu, kde se rámec teprve utváří, nebo v epilogu, kde je to obráceně. Pokud by se na to nebral ohled, mohla by aktálně vyvolaná funkce v seznamu chybět.

Bez ohledu na vyspělost debuggeru je někdy nutné klesnout na úroveň instrukcí. Tehdy se zobrazuje disassemblovaný kód, tedy strojový kód převedený do člověku čitelné podoby. Vedle toho je vidět obsah registrů procesoru (obvykle hexadecimálně), lze je měnit. V okně bývá nezpracovaný obsah zásobníku, tedy výpis paměti kolem místa, kam ukazuje ukazatel zásobníku. U jiného výpisu paměti by měla jít nastavit zobrazená adresa.

Data globálních proměnných lze zobrazovat celkem snadno, adresy jsou uloženy v tabulce symbolů, v debugovacích informacích je o nich navíc uveden typ a v případě složitějších typů i popis těchto typů. Problém nastává u lokálních proměnných. Ty mohou být uloženy v registrech nebo na zásobníku, v některých případech v obojím, ale s aktuální hodnotou v registru (ze zásobníku se načte hodnota do registru pro dočasné rychlejší zpracování). Na zásobníku mohou být uloženy původní hodnoty registrů. Pro správnou interpretaci jsou potřeba debugovací informace o rozložení rámce. Po určení, kde je umístěna jaká proměnná, je možné získávat jejich hodnoty, případně je měnit.

Zajímavá situace u lokálních proměnných nastává při jejich použití pro watchpoint. Tehdy se fyzický watchpoint vytvoří při každém spuštění funkce a zruší při návratu.

Při vyhodnocování výrazů debuggerem se může stát, že bude potřeba vykonat nějaké funkce programu, ať už přímo voláním funkcí ve výrazu nebo prostřednictvím předefinovaných operátorů. V takovém případě se tyto funkce vykonávají v kontextu debugované aplikace. Může to fungovat tak, že se na zásobník uloží parametry, na instrukci pro návrat se vloží breakpoint a funkce se zavolá.

3 Ladicí formát DWARF

3.1 Obecné informace

Ladicí formát DWARF [3] byl vytvořen pro použití se spustitelným formátem ELF. Odtud také pochází jeho název, význam zkratky byl vytvořen až dodatečně: Debug With Arbitrary Record Format. Formát podporuje řadu jazyků, například C, C++, Fortran, Modula, Pascal, Ada a další. V této práci se zaměřím především na části použitelné pro assembler a jazyk C.

Ladicí informace jsou rozděleny do několika sekcí, názvy začínají .debug_. DWARF od verze 2 se zaměřuje na úsporu místa pro ladicí informace. Dosahuje se toho použitím programů v byte kódu pro generování tabulek informací, zkratkami a formátem LEB128 pro uložení čísel na minimálním počtu bitů.

LEB128 rozdělí číslo na bloky po sedmi bitech. Jednotlivé bloky se pak ukládají od LSB (Least Significant Byte) do MSB (Most Significant Byte), nejvyšší bit bloku určuje, zda bude následovat další blok. Znaménková čísla (SLEB128) se ukládají stejně jako bezznaménková (ULEB128), pouze se 7bitová část znaménkově rozšíří.

Tabulky s hodnotami použitých konstant najdete ve specifikaci formátu DWARF 2 v kapitole 7.

V této kapitole se pokusím popsat formát DWARF tak, aby bylo možné vytvořit ladicí informace pro program v jazyce C. Informace o jednotlivých symbolech jsou uloženy jako ladicí záznamy a uspořádány do stromu. Pomocí ladicích záznamů jsou uloženy informace o funkcích a jejich parametrech, o globálních a lokálních proměnných a také o datových typech včetně struktur a unií.Ladicí záznamy obsahují atributy, z nichž některé mají složitější sémantiku, například informace o umístění symbolu. Kromě ladicích záznamů jsou mezi ladicími informacemi uloženy informace o mapování řádků na adresy v strojovém kódu a o rozložení rámce pro případ, kdy je potřeba se podívat na funkci hlouběji na zásobníku.

3.2 Ladicí záznamy

Ladicí záznam se skládá z typu (prefix DW_TAG) a několika atributů (prefix DW_AT). U každého atributu je definována kategorie typu hodnoty.

Ladící záznamy jsou organizovány do stromu, kde každý uzel je ladicím záznamem. Každý záznam má definováno, zda má potomky. Pokud ano, další fyzicky uložený záznam je potomkem aktuálního záznamu, jinak je na stejné úrovni. Návrat na vyšší úroveň stromu je realizován nulovým záznamem (prakticky se jedná o jediný nulový byte).

Ladicí záznamy jsou uloženy ve dvou sekcích. Sekce .debug_abbrev obsahuje zkratky. Každá zkratka má svoje nenulové číslo (ULEB128), typ záznamu (1 byte), flag o potomcích a několik záznamů o atributech. Každý tento záznam obsahuje typ atributu (DW_AT) a typ hodnoty (DW_FORM). Speciální formou je DW_FORM_indirect, kdy přesná forma je určena jako první část hodnoty. Seznam atributů je ukončen nulou. Sekce .debug_info obsahuje ladicí záznamy. Záznam

obsahuje číslo zkratky a poté hodnoty atributů deklarovaných ve zkratce. Nulový záznam je reprezentován pomocí nulového čísla zkratky.

Konstanta je několik bytů bez určení významu. Konstanty mají 6 forem, které určují délku dat. Formy DW_FORM_data1 až DW_FORM_data8 slouží pro délky 1, 2, 4 a 8 bytů. Formy DW_FORM_sdata a DW_FORM_udata slouží k určení čísla ve formátu LEB128.

Adresa ukazuje do adresového prostoru popisovaného programu. Je reprezentována číslem o velikosti obvyklé pro danou architekturu.

Blok obsahuje několik bytů. Prvních několik bytů určuje zbývající délku bloku. Počet bytů pro počet je určen formami DW_FORM_blockX (X určuje délku: 1, 2 nebo 4 byty) a DW_FORM_block (velikost bloku určena pomocí ULEB128).

Flag indikuje přítomnost nebo nepřítomnost atributu. Je reprezentován jako jeden byte, nulová hodnota znamená nepřítomnost atributu.

Reference má dvě varianty. První ukazuje v rámci kompilační jednotky do vybrané sekce, druhá varianta ukazuje do jiné kompilační jednotky. První varianta je reprezentována formami DW_FORM_refX (X se nahradí 1, 2, 4 nebo 8 podle délky) a DW_FORM_ref_udata. Jedná se o offset do sekce vzhledem k začátku aktuální kompilační jednotky. Druhý typ reference používá formu DW_FORM_ref_addr a využívá relokaci jako to dělají symboly.

Řetězec je sekvence nenulových bytů ukončenou nulovým bytem (řetězec jako z jazyka C). Existují dvě formy. Forma DW_FORM_string ukládá data řetězce přímo jako hodnotu atributu. Forma DW_FORM_strp ukládá data řetězce v sekci .debug_str, hodnota v atributu je reprezentována jako čtyřbytový offset do této sekce.

3.3 Popis umístění

Popis umístění se dá považovat za složitější typ. Popisy umístění se používají pro určení místa, kde leží paměťová proměnná, ukazatel zásobníku nebo třeba návratová adresa z podprogramu. Umístění může být ve dvou formách, buď výraz nebo seznam. Výraz je uložen jako blok, seznam jako reference do sekce .debug_loc.

Výraz umístění je program v bytovém kódu. Instrukce pracují na principu zásobníku, výsledek programu, tedy umístění, je na konci uložen na vrcholu zásobníku. Parametry operací jsou součástí instrukce, ze zásobníku čtou informace jen některé kategorie instrukcí výrazu.

Instrukce pro registrové operace a literály vkládají na vrchol zásobníku registr nebo literál určený buď přímo instrukcí (X v názvu: DW_OP_regX nebo DW_OP_litX) nebo parametrem. Jednou z operací zařazených mezi literálové je určení adresy (DW_OP_addr) místo hodnoty.

Další určování adres je zařízeno prostřednictvím registrů (u DW_OP_addr to byl literál). Který registr se pro adresování použije je určeno parametrem instrukce (DW_OP_bregx) nebo přímo názvem instrukce (DW_OP_bregX). Místo registru lze použít i speciální umístění (DW_OP_fbreg) určené atributem DW_AT_frame_base aktuální funkce. K základu adresy se přičte offset zadaný dalším parametrem instrukce.

Instrukce pro zásobníkové operace umožňují kopírovat, přesunovat a mazat hodnoty na zásobníku. Zvláštní skupinou zásobníkových operací jsou dereferenční operace, které berou hodnotu na vrcholu zásobníku jako adresu a nahradí ji hodnotou na této adrese. Velikost hodnoty je určena buď parametrem nebo se použije velikost běžná pro danou architekturu.

Instrukce pro arithetické a logické operace použijí několik hodnot ze zásobníku, odstraní je a výsledek operace vloží na vrchol. Na vrcholu zásobníku je poslední operand operace.

Poslední skupina instrukcí slouží k řízení toku. Cíle podmíněných a nepodmíněných skoků jsou zadány relativně k následující operaci. Porovnávací instrukce odeberou z vrcholu zásobníku, první hodnotu na původním vrcholu zásobníku porovná s druhou a na vrchol zásobníku vloží jedničku nebo nulu podle výsledku porovnání.

Seznamy umístění jsou uloženy v sekci .debug_loc. Každý záznam je složen z počáteční a koncové adresy instrukcí a výrazu umístění, který platí v daném rozsahu adres. Adresy jsou uloženy ve formě DW_FORM_addr, výraz umístění jako blok DW_FORM_block2.

3.4 Záznamy o instrukcích, datech a typech

Před samotnými záznamy leží na začátku sekce .debug_info pro každou kompilační jednotku hlavička obsahující verzi DWARFu, offset do .debug_abbrev a velikost adresy pro architekturu.

Informace o kompilační jednotce obsahují cestu k zdrojovému souboru, jazyk, umístění kódu v paměti a odkazy na informace o řádcích, datových typech a makrech.

U vstupních bodech funkcí a procedur je určeno jméno, návratový typ a umístění kódu v paměti. Lexikální blok je v jazyce C ohraničen složenými závorkami a záznam umožňuje definovat proměnné v rámci bloku, sám o sobě obsahuje jen rozsah adres, kde je blok uložen. Pro návěští se ukládá jeho název a adresa.

Záznamy o datech slouží k uložení informací o proměnných, formálních parametrech funkcí a procedur a o pojmenovaných konstantách. V záznamu lze nalézt jméno, typ a umístění, případně i hodnotu.

Základní typy jazyka jsou určeny jménem, velikostí a způsobem zakódování (například float, celé číslo s nebo bez znaménka, adresa, apod). Ze základních typů se pak tvoří odvozené typy. Lze vytvářet pole (je třeba určit rozestup mezi prvky a volitelně i velikost celého pole), modifikované typy (ukazatele, const, volatile), aliasy typů (typedef) a struktury nebo unie. Informace o členech struktur nebo unií obsahují jméno, typ a umístění relativně k začátku struktury. Dále lze tvořit výčtové typy (obsahují jméno, velikost a seznam členů s jmény a hodnotami), ukazatele na funkce (formální parametry popsány stejně jako u normálních funkcí) a typy řetězců.

3.5 Informace o řádcích programu

Informace jsou uloženy v sekci .debug_line jako program v bytovém kódu. Cílem je ke každé adrese s instrukcí přiřadit řádek zdrojového kódu v souborech a také určit, zda na dané adrese příkaz začíná nebo procedura končí. Uložení těchto informací do matice by zabralo příliš mnoho místa, tato matice se virtuálně generuje právě programem pro konečný automat.

Automat má několik registrů, mezi ty důležité patří adresa, číslo souboru a řádek v souboru. Hlavička programu pro automat obsahuje seznam souborů a adresářů a konfiguraci instrukcí, která umožňuje používat speciální jednobytové instrukce pro automat.

Standardní operace mění obsahy registrů a mají i několikabytový zápis. Rozšířené operace jsou ještě delší, díky určení délky umožňují používat velké hodnoty. Speciální instrukce jsou jednobytové, mění v jednom kroku registry adresa a číslo řádku. Nemají konkrétní opcode, formát opcode je určen pomocí konfigurace v hlavičce.

3.6 Informace o rozložení rámce

Pomocí programu uloženého v sekci .debug_frame se podobně jako u informací o řádcích vytváří matice informací. V tomto případě se pro každou instrukci funkce určuje, kde jsou uloženy které registry a speciální hodnoty jako návratová adresa a ukazatel na začátek rámce. Tabulka informací tedy obsahuje sloupce umístění instrukce, ukazatele rámce a poté jednotlivé registry podle svých čísel. Jednotlivé hodnoty ve sloupcích jsou pravidla pro získání skutečné hodnoty.

Tato pravidla jsou:

- nedefinováno: pro registry, které nejsou uchovávány
- stejná hodnota: pro uchovávané registry, jejichž hodnota nebyla změněna
- offset: má jeden parametr offset, registr je uložen na adrese offsetu relativně k začátku rámce
- registr: jeden parametr určuje, ve kterém registru je hodnota uložena

Záznamy jsou rozděleny na dva druhy: společné informace a konkrétní rámec, přičemž ten odkazuje na záznam o společných informacích. Mezi ukládané informace o rámcích patří zarovnání kódu a dat, označení registru pro návratovou hodnotu a instrukce programu pro generování pravidel.

3.7 Použití v projektu Lissom

Tento formát pro uložení ladicích informací se nasazuje do překladačů jako implicitní místo staršího formátu STABS, který je pro člověka i s pomocí nástrojů mnohem hůře čitelný a tedy i hůře pochopitelný. Navíc DWARF má díky své struktuře lepší schopnosti.

V projektu Lissom má DWARF potenciál, ale jeho síla se projeví až ve spojení s překladačem jazyka C. Současně vyvíjená verze debuggeru ovšem bude podporovat zatím jen assembler, proto bude použit stávající formát, který pro účely ladění obsahuje jen symboly a jejich adresy, bez dalších ladicích informací. Myšlenky typového systému DWARFu jsou použity v části debuggeru, která se zabývá vyhodnocováním výrazů.

Formát spustitelného souboru bude popsán později v diplomové práci.

4 Debugger GDB

Debugger se ovládá zadáváním příkazů přes terminál, přičemž v interaktivním režimu (implicitní) vypisuje po každém příkazu podrobné informace o provedeném příkazu a aktuálním kontextu. Druhou možností je dávkový režim, kdy je množství vypisovaných informací omezeno.

4.1 Základní příkazy GDB

Debugger se běžně spouští s jedním parametrem – spustitelným souborem, který se bude ladit. Poté se nastaví breakpointy a program se spustí příkazem *start* nebo *run*, které přijímají parametry, které se mají předat spouštěnému programu. Parametry programu lze nastavit i parametrem debuggeru *--args*, poté se u příkazů *run* a *start* parametry nezapisují.

Breakpointy se nastavují příkazem *break* a jeho variantami [2]. Je možné vložit breakpoint na vybranou adresu, funkci, nebo soubor a řádek programu. Breakpointy mohou být podmíněné, kdy se aktivují (předají řízení uživatelskému rozhraní GDB) jen za určitých podmínek zadaných výrazem nebo počtem aktivací. Podmínky se k breakpointům nastavují dodatečně příkazem *condition*. K breakpointům lze dokonce nastavit příkazy (příkaz *commands*), které se provedou při její aktivaci.

Krokování zajišťují příkazy *step*, *next*, *stepi* a *nexti*, které krokují dovnitř nebo přes volání funkcí po řádcích nebo po instrukcích. Po každém zastavení se vypíše (v interaktivním režimu), kde se program zastavil. Pokračování v běhu do dalšího breakpointu se zajistí příkazem *continue*.

Pro zjišťování chyb se hodí příkaz *bt*, který vypíše přehled obsahu zásobníku, konkrétně jaké funkce byly zavolány a s jakými parametry. Výpis hodnot proměnných a paměti vůbec zajišťuje příkaz *print*, je u něj možné specifikovat formát výpisu, například hexadecimálně. Vypisované hodnoty se ukládají do historie hodnot, ke kterým lze pak ve výrazech přistupovat pomocí proměnných *$, $$* nebo *$$n*, kde *n* je pořadové číslo. Pro výpis paměti se používá podobný příkaz *x*, u kterého je obvykle potřeba specifikovat velikost elementu a jejich počet. Příkaz *set* funguje stejně jako *print*, ale nic nevypisuje a nemění historii hodnot.

Příkazy *print* a *set* pracují s výrazi, přičemž *print* poté vypisuje výsledek. Výraz může být složitý a dokonce může obsahovat i přiřazení a volání funkcí programu debuggerem. Přitom se dají používat proměnné programu (normální zápis jména) nebo proměnné jen pro účely debuggeru (začínají dolarem a poté následuje jejich jméno). Pro výrazy se přitom používá jazyk, ve kterém je program napsán, to ovšem platí jen pro podporované jazyky jako C, C++, Ada, Fortran a další.

4.2 Podpora skriptování

Příkazy lze načítat i ze souborů, při chybě pak skript okamžitě skončí. V interaktivním režimu to je možné pomocí příkazu *source*. Prováděné příkazy se nevypisují, ale jejich výsledky včetně interaktivních zpráv ano. Proto existuje parametr -batch, který omezí výpisy a zamezí interaktivním dotazům (předpokládá vždy souhlas) a navíc parametr -batch-silent, který nevypisuje nic kromě chybových hlášení.

Příkazy a soubory příkazů k vykonání lze předávat parametry -ex a -x. Cílem je, aby bylo možné debugger používat ve skriptech, které mohou provádět automatické testování nebo například automatický restart serverových aplikací s výpisem zásobníku a nastalých chyb.

Při načítání příkazů ze souboru se ignorují řádky začínající znakem # a prázdné řádky, které v interaktivním režimu způsobují opakování posledního příkazu. V souborech s příkazy lze používat příkazy if, else, while, end pro podmíněné příkazy a cykly. Dále se v nich obvykle používají příkazy *echo* pro jednoduchý výstup bez proměnných a příkaz *output*, který je variantou příkazu *print*, ale nemění historii hodnot a zobrazuje hodnotu výrazu bez dekorací.

4.3 Vestavěné simulátory procesorů

Debugger GDB umožňuje ladit programy určené pro jiné architektury než je hostitelská. Pomocí příkazu *target sim* lze určit simulátor procesoru, který se použije pro spuštění programu. Simulátory a simulovaná architektura jsou zakompilovány do debuggeru.

Samotný simulátor má rozhraní určené několika funkcemi začínajícími prefixem sim_. Ovšem ke spuštění programu musí být podporována architektura, která definuje registry, operace pro nastavování breakpointů a obnovování běhu programu, organizaci čísel v paměti (big a little endian) a další parametry. Kromě toho každý spustitelný formát má svou variantu pro každou architekturu. Architektury a spustitelné soubory jsou uloženy v knihovně BFD. Pro přidání nové architektury je třeba ji přidat do výčtového typu a přidat několik zdrojových souborů.

4.4 Použití v projektu Lissom

Programování simulátorů do debuggeru GDB by vyžadovalo kompilaci celého debuggeru, není možné zkompilovat jen malou část jako dynamicky linkovanou knihovnu. Vzhledem k častým rekompilacím by to bylo zbytečně časově náročné. Především ale by nebylo možné, nebo by bylo složité realizovat distribuovanou simulaci a rozdělení na třívrstvou architekturu, která se v projektu Lissom používá. Distrubuovanou simulací se myslí, že jednotlivé simulované procesory běží fyzicky na různých počítačích a sdílejí některé prostředky (jsou určeny v modelu).

Ovšem myšlenky GDB se využít dají. Stačí upravit stávající debugger v simulační vrstvě a vytvořit řádkového klienta, který bude podporovat dávkové zpracování. Střední vrstva nebude potřebovat téměř žádné změny, pouze se přidají příkazy, které se budou přesměrovávat na konkrétní simulátory již existujícími mechanismy.

5 Architektura Lissom

5.1 Rozdělení nástrojů

Projekt Lissom se snaží vytvořit prostředí pro vývoj procesorů a aplikací pro tyto procesory. Prostředí obsahuje knihovny, nástroje nezávislé na platformě a na nástroje generované.

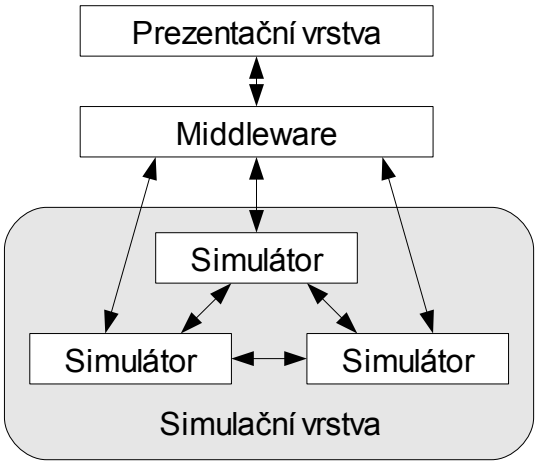
Knihovny jsou vždy nezávislé na konkrétním modelu. Knihovny třívrstvé architektury slouží ke zprostředkování komunikace mezi jednotlivými vrstvami. Druhou skupinu tvoří knihovny pro generování výstupů včetně nástrojů. Poslední skupina knihoven se používá při simulaci (sem patří i ladicí knihovna).

Nástroje nezávislé na platformě slouží jednak jako prostředí pro vývoj modelu a aplikací a jednak jako překladače a generátory modelově závislých nástrojů a jiných výstupů. Do první skupiny patří middleware a klienti pro příkazovou řádku a GUI. Druhá skupina zajišťuje překlad jazyka ISAC, linkování programů a generování modelově specifických nástrojů, případně jiných výstupů (například model ve VHDL, diagramy časování, apod.).

Generované nástroje jsou vytvářeny specificky pro konkrétní model, při změně v modelu je potřeba je přegenerovat. Jedná se o assembler (překladač jazyka symbolických instrukcí), disassembler, překladač jazyka C, simulátor procesoru, atd.

5.2 Třívrstvá architektura

Architektura projektu Lissom je rozdělena do tří vrstev - na prezentační, middleware a simulační vrstvu.



5.2.1 Prezentační vrstva

Prezentační vrstva se připojí k middleware, posílá mu příkazy a zobrazuje odpovědi. Tato vrstva zajišťuje vývojové prostředí. V projektu Lissom je reprezentována jednak nástrojem pro

příkazovou řádku a jednak modulem Eclissom pro grafické uživatelské rozhraní. Nástroj pro příkazovou řádku je jednodušší, střední vrstvě posílá příkazy a zdrojové soubory, od střední vrstvy pak zobrazuje výsledky a ukládá vytvořené soubory, například XML model nebo přeložený kód programu. Grafický klient má o něco bohatší možnosti, umožňuje navíc editaci modelu a programů a při ladění programů lze snadno zobrazovat i vykonávaný kód.

5.2.2 Middleware

Střední vrstva zajišťuje veškeré překlady, generování nástrojů, spouštění a komunikaci se simulátory. Při začátku sezení se ověří verze protokolu pro komunikaci s prezentační vrstvou a uživatel se přihlásí. Smyslem přihlášení je vytváření samostatných prostředí pro více uživatelů. Neprovádí se autentizace, ovšem jedno uživatelské jméno smí být v jednom okamžiku použito jen jednou. Každý přihlášený uživatel má svůj adresář, kde jsou uloženy jeho projekty a dočasné soubory.

Prostřednictvím střední vrstvy pak uživatel spouští překladač jazyka ISAC, nebo jiné nástroje (ať už globální nebo generované). Veškeré přenesené soubory jsou uloženy do adresáře uživatele a poté je spušten skript určený typem příkazu. Úkolem těchto skriptů je vytvořit prostředí pro samotný nástroj (překladač ISAC, generátor nástrojů, apod.) a předat mu ve správné podobě parametry. Výsledky nástrojů se pak pošlou zpět prezentační vrstvě. Toto je první z úkolů střední vrstvy.

Typický postup kompletního překladu modelu a aplikací začíná přeložením modelu v jazyce ISAC. Výsledkem je model přeložený do formy XML. Z ní se potom pomocí generátoru nástrojů vytvoří assembler, disassembler, simulátor a další generované nástroje. Ten vygeneruje zdrojové kódy modelově specifických částí nástrojů. Skript, který zavolal program pro generování zdrojových kódů nástrojů, je potom pomocí utility make přeloží. Následuje překlad simulované aplikace pomocí vygenerovaného assembleru a vytvoření spustitelného souboru pomocí linkeru.

5.2.3 Spouštění simulace

Druhým úkolem střední vrstvy je zprostředkovat simulaci procesorů. Prvním krokem je nainstalování simulátorů, ty tvoří simulační vrstvu. V konfiguračním souboru předaným prezentační vrstvou je seznam simulátorů k instalaci, u každého je uvedeno jeho jméno, soubor se simulovanou aplikací a umístění simulátoru v síti

Simulátor může běžet na stejném PC jako střední vrstva, v tom případě se simulátor jednoduše spustí, nebo na jiném PC, kdy se middleware pomocí SSH přenese program a simulátor na cílový počítač. Po spuštění simulátorů se střední vrstva na simulátory připojí a předá jim seznam ostatních simulátorů, které pak navážou spojení i mezi sebou.

Simulátory pak načtou svůj soubor se simulovanou aplikací, inicializují procesor a čekají na signál ke startu. Cílem tohoto čekání je umožnit nastavování breakpointů a jiných potřebných činností před zahájením samotné simulace.

Prostřednictvím middlewaru pak může prezentační vrstva jednotlivým simulátorům zasílat příkazy. Příslušný simulátor by v tu dobu měl být zastaven. Simulátor se zastaví při krokování nebo při dosažení breakpointu. Příkazy pro simulátory middleware pouze přeposílá, prvním parametrem

těchto příkazů je vždy jméno simulátoru, kterému je příkaz určen. Mezi tyto příkazy patří zejména krokování a práce s breakpointy. Výsledky příkazů pak střední vrstva obvykle beze změn přepošle prezentační vrstvě.

Po ukončení všech simulátorů lze získat jejich výsledky. Výsledky obsahují výstup ze simulátorů, dobu provádění, obsahy a počty použití jednotlivých registrů a pamětí.

5.3 Komunikační protokol

Komunikační protokol má formu zjednodušeného XML bez hlavičky a kořenového elementu. Používají se pouze značky bez atributů, není možné používat zkrácenou formu prázdné značky a všechny značky mají velká písmena. Důvodem těchto omezení je použití řetězcových operací pro parsování namísto knihovny pro zpracování plnohodnotného XML a to kvůli rychlosti zpracování.

Příkaz je tvořen jeho názvem a několika parametry. Pokud je některý z parametrů názvem vstupního souboru, je obsah tohto souboru zakódován a připojen na konec příkazu jako nový parametr.

Soubory jsou zakódovány tak, že jeho obsah se rozdělí na bloky pevné maximální velikosti (asi 100kB), bloky jsou pomocí knihovny ZLIB zkomprimovány metodou ZIP a zakódovány pomocí BASE64.

5.4 Jazyk ISAC a assembler

5.4.1 Jazyky pro popis procesorů

Jazyky pro popis procesorů se dají rozdělit na strukturální, behaviorální a smíšené. [6]

Strukturální jazyky se zaměřují na hardwarové součásti procesoru (registry, ALU, pipeline, řídící okruhy, ...) a jejich propojení. Mezi tyto jazyky patří MIMOLA, dají se sem zařadit i HDL jazyky jako Verilog nebo VHDL. Umožňují flexibilní a přesný popis mikroarchitektury, ale je problém extrahovat popis instrukční sady.

Naopak behaviorální jazyky jako ISDL nebo nML popisují instruční sadu a její chování. Instrukční sadu popisují často hierarchicky a pomocí atributovaných gramatik. Neumožňují modelovat žádné strukturní detaily, takže není možné z nich vytvořit cycle-accurate simulátor.

Kompromisem mezi těmito dvěma druhy jsou smíšené jazyky pro popis architektury. Nejsou tak dokonalé, co se týká detailů, ale je možné pomocí nich definovat jak strukturu, tak instrukční sadu. Navíc se používají i při verifikaci. Mezi tyto jazyky patří zejména LISA, ze které vychází i jazyk ISAC použitý v projektu Lissom.

5.4.2 Schopnosti jazyka ISAC

Model procesoru je popsán pomocí jazyka ISAC. Před vlastní kompilací je zdrojový kód modelu zpracován preprocesorem používaným pro jazyk C. Díky tomu je možné vkládat do modelu další soubory, definovat a používat makra, včetně těch s parametry. Jazyk ISAC prochází vývojem, během

tvorby této práce došlo k několika změnám, proto nebudu uvádět konkrétní syntax, jen přehled jeho schopností a základní strukturu.

První část modelu specifikuje zdroje procesoru. Mezi zdroje procesoru patří registry, paměti, cache, sběrnice, pipeline a mapa paměti. Registr může být označen jako čítač instrukcí nebo řídící registr. Místo prostého registru lze vytvořit i registrové pole. U pamětí, cache a sběrnic lze volitelně určit způsob sdílení zdroje s ostatními simulátory.

U všech zdrojů kromě mapy paměti lze definovat jejich parametry, například velikost, velikost bloku, latence nebo endian. Mapa paměti mapuje zdroje do jednoho velkého pole. Využívá ji především překladač jazyka symbolických instrukcí a loader programu do pamětí procesoru.

Druhá část popisuje operace v procesoru a jejich skupiny. Pomocí operací se tvoří veškeré chování procesoru a popis instrukcí. U instrukcí se vyskytuje jejich popis pro jazyk assembler, způsob zakódování a jejich chování. Chování se zapisuje v podmnožině ANSI C, je možné volat i funkce knihoven a přiložených zdrojových kódů v C. Při použití externích funkcí ovšem nebude možné je použít pro generování VHDL.

Ostatní operace určují jednotlivé akce procesoru, navzájem se mohou aktivovat (i se zpožděním, tím je implementována pipeline) a to i podmíněně. Některé operace jsou svým způsobem speciální. Jedná se jednak o operace se jmény *reset* a *main*, které slouží k inicializaci a provedení jednoho taktu v procesoru.

5.4.3 Možnosti assembleru

Assembler je navržen ve stylu GNU as.

Překladač assembleru se generuje z XML modelu procesoru. Každou instrukci v assembleru lze rozložit na strom operací jazyka ISAC. Zdrojový kód v assembleru je zpracováván po řádcích, na každém řádku je jedna instrukce. V případě procesorů VLIW (Very Long Instruction Word) může být instrukční slovo na několika řádcích, pokud direktiva *CODING* obsahuje znak nového řádku.

Kromě instrukcí mohou být ve zdrojovém kódu direktivy assembleru a symboly. Direktivy začínají tečkou. Mezi podporované direktivy patří *.section, .org, .line, .global* a *.equiv*. K definici dat slouží direktiva *.bit* následovaná počtem bitů a hodnotami k vložení.

Pro účely ladění je vhodné upozornit, že symboly, které nejsou direktivou .global označeny k exportu se neobjeví ve výsledném spustitelném souboru a tudíž je nebude možné v debuggeru použít.

5.5 Distribuovaná simulace

Cílem distribuované simulace je spolupráce několika najednou spuštěných simulátorů. O to, aby o sobě simulátory v jedné spuštěné simulaci věděly, se postará střední vrstva při jejich inicializaci.

Jeden ze způsobů spolupráce je sdílení zdrojů. V jazyku ISAC se toho docílí tak, že některé zdroje jsou označeny ke sdílení mezi simulátory. Simulátory k cizím zdrojům přistupují prostřednictvím funkcí ladicí knihovny, která také poskytuje přístup ostatním simulátorům k místním sdíleným zdrojům.

Druhý způsob spolupráce simulátorů je synchronní simulace. Aby byla možná, je třeba simulátor vygenerovat s touto podporou, tedy aby pomocí volání ladicí knihovny čekal na tik. Generátorem hodin je první simulátor uvedený v konfiguračním souboru simulace. Všechny simulátory s podporou synchronní simulace musí mít v konfiguraci nastavenou frekvenci. První (řídící) simulátor pak podle konfigurace rozesílá ostatním simulátorům signály pro provedení jednoho tiku. Pokud není synchronní simulace využita, jedná se o simulaci asynchronní. V takovém případě pracují simulátory relativně nezávisle a prezentační vrstva pouze čeká na ukončení všech simulátorů.

6 Příkazová řádka

6.1 Používání příkazové řádky

Příkazová řádka je jedním z klientů prezentační vrstvy. První verze příkazové řádky měla řadu nevhodných vlastností. Proto cílem refaktorizace příkazové řádky bylo přidat podporu pro dávkové zpracování a auto-complete, rozdělit ji na normální a ladicí režim a zpřehlednit její vnitřní strukturu.

6.1.1 Spuštění

Příkazová řádka se spouští pomocí cmdline2 a příkazy zpracovává ve stylu getopt (běžný způsob v prostředí Linux). Základní syntax pro spouštění vypadá následovně:

cmdline2 [-b] [-f příkazy] [-c konfigurace] [-i ipc] [-v] [-h]

Přepínač -b (nebo --batch) přepne program do režimu dávkového zpracování. V tomto režimu se vypisují pouze vyžádané informace a důležitá chybová hlášení. Dávkový režim je určen zejména ve spojení s načítáním příkazů ze souboru určeného přepínačem -f (nebo --file). Načítání ze souborů lze použít i bez přepínače -b.

Přepínače -h a -v vypisují nápovědu k syntaxi spuštění a verzi programu.

Příkazová řádka se při startu připojuje ke střední vrstvě. Způsob připojení je určen pomocí konfigurace uložené v souboru config.xml nebo v souboru určeném pomocí přepínače -c (nebo --config). Na systému Linux je pro připojení k middleware možné použít IPC (meziprocesovou komunikace). V takovém případě je to uvedeno v konfiguračním souboru a pomocí přepínače -i (nebo --ipc-path) specifikuje cestu pro IPC (konkrétně funkci ftok). Běžnějším způsobem je ovšem použít protokol TCP/IP, kdy je v konfiguračním souboru uvedena IP adresa nebo doménové jméno serveru a port, na který se má klient připojit. Po připojení se musí uživatel přihlásit, jeho jméno je také uvedeno v konfiguraci.

6.1.2 Obecné zákonitosti

Nápovědu lze vyvolat příkazem *help*, program se ukončí pomocí *exit*. Některé příkazy mají parametry, které se mohou opakovat za sebou. Tyto parametry jsou v nápovědě zakončeny třemi tečkami. Ve všech případech se jedná o vstupní soubory a často je možné parametr úplně vynechat, protože většinou označují dodatečné hlavičkové a zdrojové soubory.

Pro načítání příkazů se používá knihovna *readline*, která v sobě zahrnuje podporu historie, řadu klávesových zkratek pro editaci a možnost používat automatické doplňování. V programu je implementováno doplňování názvů příkazů, vstupních a výstupních souborů včetně cest a některých dalších typů parametrů, například název projektu nebo název simulátoru.

Speciálním typem parametru je seznam voleb nebo operací. Ty se vyskytují u příkazů *cbt* (vytvoření nástrojů), *mksc* (vytvoření kompilovaného simulátoru) a *hdc* (kompilace hardware). Tento typ parametru obsahuje několik předdefinovaných možností, které lze za sebe řadit a oddělovat čárkou. Pořád se jedná o jediný parametr, takže nesmí obsahovat mezeru.

6.1.3 Základní režim

Základní režim příkazové řádky slouží pro překlad modelů a aplikací, instalaci simulátorů a jejich spouštění bez rozšířené podpory ladění. Příkazy v tomto režimu mají plnou podporu pro automatické doplňování příkazů a jejich parametrů.

Mnoho příkazů používá jako jeden ze svých parametrů název projektu, automaticky se doplňuje poslední použitý název. Názvy simulátorů se doplňují podle obsahu konfiguračního souboru zaslaného příkazem is (instalace simulátorů). Nezadané výstupní soubor se doplní tak, že se použije název prvního vstupního souboru včetně cesty a změní se mu přípona.

Příkaz se odešle a počká se na výsledek. Nakonec se zobrazí hlášení o úspěchu (a to i v dávkovém režimu) nebo chybová zpráva.

6.1.3.1 Příkazy pro kompilaci

Následující seznam ukazuje některé základní příkazy pro kompilaci a tvorbu nástrojů. Za pomlčkou je uveden název příkazu, ve vnořeném seznamu postupně parametry.

- Kompilace modelu - *isc*
  - model ISAC
  - hlavičkové soubory, které jsou v modelu ISAC vkládány direktivou #include
  - výstupní soubor XML, do kterého se uloží výsledek
- Vytvoření základních nástrojů - *cbt*
  - název vytvářeného projektu
  - parametry pro vytváření, typicky FASTSIM a další možnosti
  - model ve formě XML (výstup z příkazu *isc*)
  - hlavičkové a zdrojové soubory specifické pro daný model, které jsou potřeba ke kompilaci nástrojů
- Překlad aplikace v assembleru - *asm*
  - název projektu (je v něm vytvořen překladač assembleru)
  - vstupní soubor ASM
  - výstupní soubor OBJ se zkompilovaným kódem
- Linkování souborů do spustitelného souboru - *ld*
  - několik vstupních souborů OBJ
  - výsledný spustitelný soubor XEXE

6.1.3.2 Příkazy pro instalaci a spouštění simulátorů

Simulátory se nainstalují pomocí příkazu is. Prvním parametrem je soubor s konfigurací simulace. Další parametry tvoří spustitelné soubory, které se mají poslat střední vrstvě, aby je mohla předat simulátorům.

Simulátory se po nainstalování spustí příkazem sas. Pro ukončení všech simulátorů a získání jejich výsledků slouží příkaz eas, je vhodné ho použít až po dokončení všech simulací, jinak se simulátory násilně přeruší a nebude možné získat jejich výsledky.

6.1.4 Ladicí režim

Do ladicího režimu se ze základního režimu lze přepnout příkazem *gdb*, zpět do základního režimu se uživatel dostane příkazem *exit*.

V tomto režimu také funguje automatické doplňování, ale jen v omezené míře. Důvodem je, že příkazy obvykle mají natolik rozsáhlé schopnosti, že běžné rozdělení pomocí mezer na doplňování nestačí. Ladicí režim vychází z debuggeru GDB a snaží se používat jeho syntaxi příkazů.

6.1.4.1 Jednoduché příkazy

Protože v rámci simulace může být více než jeden simulátor a příkazy GDB předpokládají práci s jediným programem, pracuje se vždy s jedním vybraným simulátorem. Při vstupu do ladicího režimu se automaticky vybere simulátor, který je v konfiguračním souboru uveden jako první. Aktuální simulátor lze změnit pomocí příkazu *simulator*, jehož jediný parametr je název simulátoru, se kterým se poté bude pracovat.

Mezi jednodušší příkazy patří přehled informací - příkaz *info*. Jeho parametrem je druh informací, momentálně jsou implementovány pouze dva: *breakpoints* pro informace o breakpointech a *registers* pro výpis registrů a jejich hodnot.

Běh programu lze ovládat pomocí příkazů *run*, který spustí nebo pokračuje v simulaci, *continue* pro pokračování a *step* pro krokování po instrukcích. Narozdíl od GDB se při pokračování v simulaci nečeká na simulátory a je možné zadávat další příkaz. Toto chování ale pravděpodobně bude změněno na kompatibilní s GDB, protože lépe podporuje dávkové zpracování.

6.1.4.2 Nastavování breakpointů

Nový breakpoint se nastavuje příkazem *break*. Jeho umístění lze zadat několika možnými způsoby. První možností je uvést přímo adresu, na kterou se má breakpoint vložit, přičemž před adresu se vloží znak hvězdička, aby bylo možné tento způsob zadání rozeznat od ostatních. Samotné číslo se zadává jako v jazyce C, tedy implicitně dekadicky, uvozené nulou v osmičkové soustavě, nebo uvozené prefixem 0x v šestnáctkové soustavě. Novému breakpointu se přiřadí číslo, pod kterým se s ním bude dále pracovat. Příkaz *break* má za parametr pouze umístění, prozatím není možné v jednom příkazu kromě udání umístění určovat počet ignorování, podmínky ani další parametry breakpointu.

Breakpoint lze smazat příkazem *delete*, kdy se jako parametr použije číslo breakpointu. Druhou možností je příkaz *clear*, který má stejnou syntax jako příkaz *break* a pokusí se mezi breakpointy nalézt podle umístění ten správný.

Breakpointům je možné nastavovat počet průchodů, které se budou ignorovat. O tuto vlastnost se stará příkaz *ignore*, který přijímá celé číslo udávající nový počet ignorování. Ignorování průchodu má přednost před kontrolou podmínek.

Vybrané breakpointy lze deaktivovat bez jejich smazání. Příkazy z této skupiny přijímají jako (poslední) parametr číslo breakpointu. Do této skupiny patří příkazy *enable* pro povolení breakpointu,

disable pro zakázání (deaktivaci) a dočasné povolení pomocí variant příkazu *enable*: *enable once* a *enable delete*, které po zásahu breakpointu ho deaktivují nebo smažou.

6.1.4.3 Podpora výrazů

Aby byl debugger užitečný, musí kromě řízení vykonávání programu umožňovat zobrazovat data. V ladicím režimu příkazové řádky k tomu slouží příkaz *print*. Jeho parametrem je výraz ve stylu jazyka C, který se pošle aktuálnímu simulátoru k vyhodnocení a jeho výsledek se poté zobrazí. Způsob zobrazení je možné určit pomocí volitelného prvního parametru uvozeného lomítkem. Za lomítkem následuje písmeno určující formát (seznam ukazuje tabulka). Variantou příkazu print je příkaz *set*, který funguje stejně, ale výsledek nezobrazuje. Jeho název napovídá, že jeho smyslem je nastavování hodnot.

Formát	**Způsob zobrazení**
u	Dekadicky bez znaménka
d	Dekadicky se znaménkem
x	Hexadecimálně
o	Číslo v osmičkové soustavě
c	Znak a jeho číselná hodnota

Výraz (i v příkazu *print*) může obsahovat přiřazení, ale jen na nejvyšší úrovni, přiřazení nelze použít jako podvýraz jako je to možné v jazyce C. Jsou ovšem podporovány běžné operátory včetně indexování polí, ukazatelů a aritmetických operátorů. Vzhledem k absenci podpory jazyka C na úrovni vytváření aplikací pro simulátory nejsou podporovány například struktury.

Ve výrazech lze používat symboly (například proměnné, pole, apod.), implicitně mají typ int. Ze symbolu se použije adresa, která se poté použije jako index do mapy paměti procesoru. K registrům procesoru lze přistupovat prostřednictvím znaku dolar a následovaného názvem registru. Pro složitější specifikaci zdroje například přístup ke zdroji jiného simulátoru, indexovanému zdroji nebo dalším je potřeba použít variantu ${simulátor:zdroj[index]}.

6.1.5 Dávkové zpracování

Jak již bylo uvedeno, příkazy lze načítat ze souboru. V každém kroku se načte jeden řádek. Prvním akcí je přeskočení úvodních prázdných znaků, jako jsou mezery nebo tabulátory. Prázdné řádky a řádky s komentářem se ignorují. Komentáře začínají znakem #.

Pro účely dávkového zpracování existuje několik příkazů určených pro nastavení chování při chybách. Tyto příkazy a některé další vybrané se provádí vždy. Ostatní příkazy se provádí při chybách jen někdy, a to podle nastavení zpracování chyb.

Implicitním chováním je chyby ignorovat, tedy příkazy se budou vykonávat za všech okolností. Změnit chování při chybách je možné příkazem *error*, který funguje stejně v normálním i v ladicím režimu. Jeho parametr určuje konkrétní akci.

Základními režimy jsou *ignore* pro ignorování chyb a *exit* pro ukončení provádění při výskytu chyby. Druhou možností je přeskakovat příkazy pomocí režimu *skip*, kdy se příkazy budou přeskakovat až do výskytu příkazu *error catch* nebo *error delay*.

Rozdíl mezi nimi je v tom, že *error delay* si pamatuje, že nějaká chyba v minulosti nastala. Chyby se přitom kumulují. Tuto informaci pak využije příkaz *error activate*, který vyvolá chybu, pokud byla nějaká chyba zapamatována.

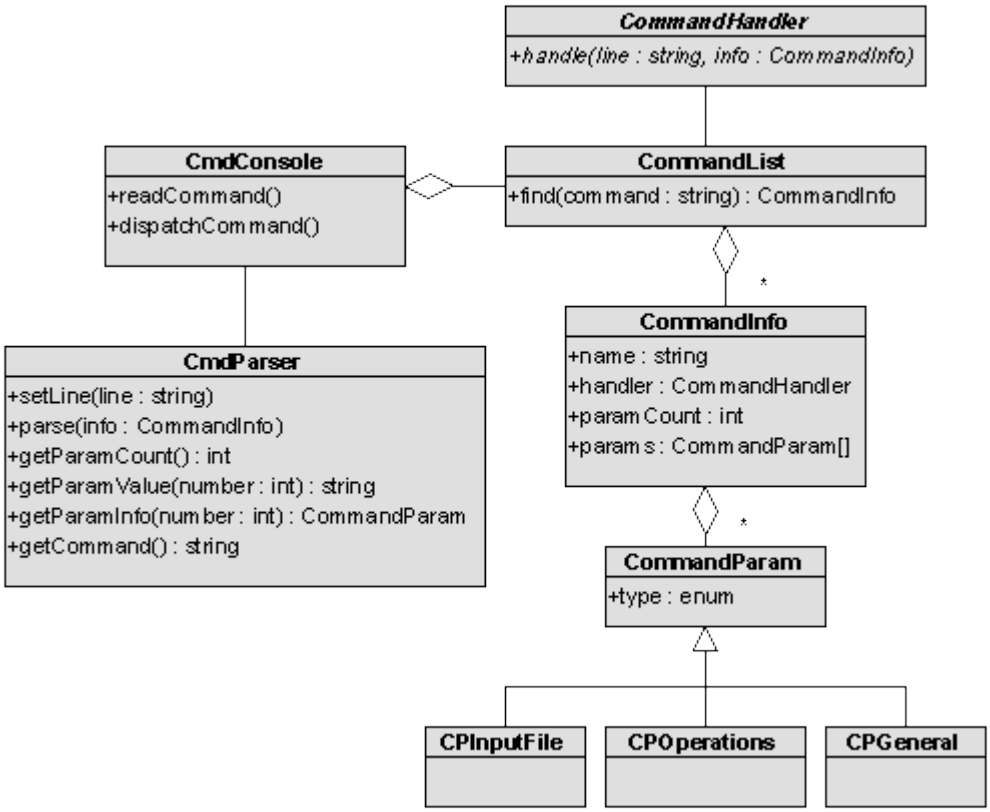
Smyslem *error delay* je možnost přeložit několik modelů a aplikací, kdy je možné, že nastane chyba a pomocí *error activate* zajistit, že se nespustí příkaz pro spuštění simulace.

Problémem chybového zpracování je prozatím v tom, že překladové příkazy vrací úspěch, pokud správně přijaly všechny parametry a nedají vědět o tom, že se překlad nezdařil kvůli chybám v souborech.

6.2 Návrh a implementace

6.2.1 Návrh tříd

Základní myšlenkou implementace příkazové řádky je intenzivní používání metainformací o příkazech a pro samotné vykonávání použití návrhového vzoru Příkaz. Základní rozvržení tříd ukazuje obrázek s diagramem tříd.



Obrázek 1: Diagram tříd příkazové řádky

Ovládacím centrem je třída *CmdConsole*. Kromě metod pro načtení a odeslání příkazu obsahuje různá nastavení, například dávkový režim, stav zpracování chyb a soubor, ze kterého se načítají příkazy. Obsahuje dva seznamy příkazů (*CommandList*), jeden pro základní režim, druhý pro ladicí.

6.2.2 Informace o příkazech

Seznam příkazů je používán především pro vyhledání informací o příkazu (*CommandInfo*) podle jména. Navíc může být potřeba vypisovat nápovědu, k tomu se opět použijí informace o příkazu (například popis funkce nebo ručně nastavená syntax) a jeho parametrech.

Každý příkaz má své jméno, seznam parametrů (*CommandParam*) a obslužný objekt vytvořený podle návrhového vzoru Příkaz (*CommandHandler*). Parametrů je několik druhů, liší se hlavně způsobem pro automatické doplňování. U parametru lze nastavit jeho jméno, které se zobrazí v automaticky generované nápovědě. Druhy parametrů ukazuje tabulka.

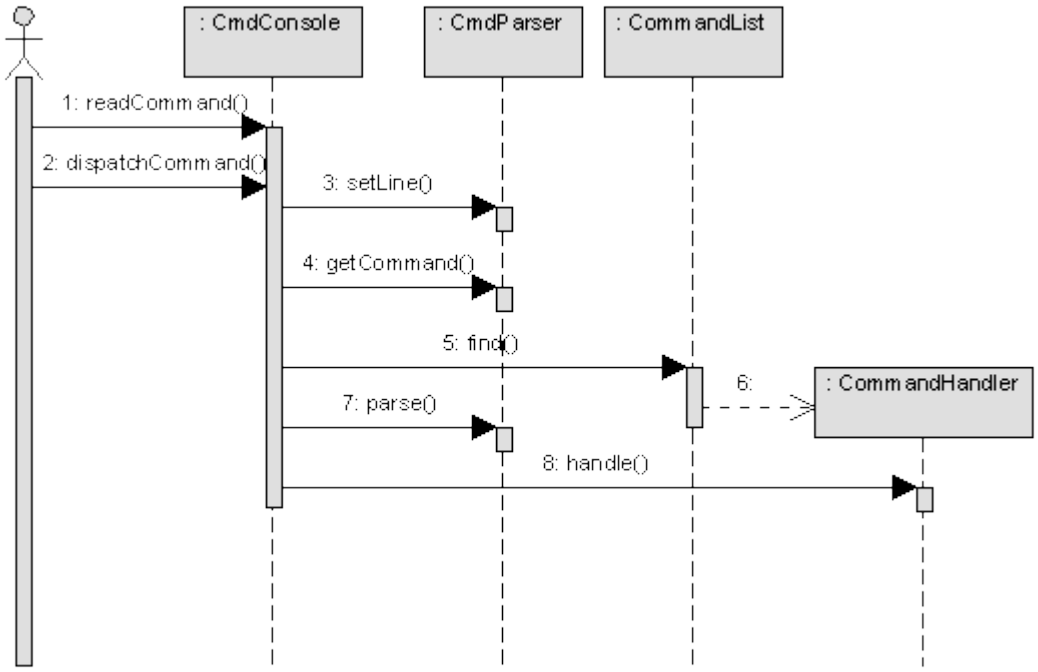
Druh parametru	**Význam a způsob doplňování**
Generic	Obecný parametr bez doplňování
InputFile	Vstupní soubor. Postupné doplňování cesty k existujícím souborům. Obsah souboru se odesílá jako další připojený parametr.
OutputFile	Výstupní soubor. Před odesláním příkazu se cesta zapamatuje a při přijetí výsledku se do tohoto souboru výsledek uloží. Doplňování funguje jako u InputFile, u prázdného parametru se doplní celá cesta vybraného vstupního souboru, kterému se změní přípona.
ProjectName	Název projektu. Doplňují se názvy použité dříve v tomto sezení.
SimulatorName	Název simulátoru. Seznam možností pro doplňování se vygeneruje při instalaci simulátorů.
Operations	Výběr z několika předdefinovaných variant, které je možné spojovat pomocí čárky, ale bez mezer.

6.2.3 Postup zpracování příkazů

Po inicializaci příkazové řádky podle parametrů předaných operačním systémem a podle konfiguračního souboru se spustí hlavní cyklus zpracování příkazů. Jeho iteraci ukazuje ve zjednodušené podobě sekvenční diagram.

Pomocí metody *readCommand* se příkaz načte. Přitom se odstraní bílé znaky na začátku a zkontroluje se, jestli řádek obsahuje příkaz, nebo jestli se jedná o komentář případně prázdný řádek. Řádek s příkazem se předá ke zpracování metodě *dispatchCommand*. Pomocí parseru příkazů *CmdParser* se zjistí jméno příkazu, které se využije k vyhledání informací *CommandInfo* pomocí metody *find* příslušného seznamu příkazů podle aktuálního režimu příkazové řádky. Nakonec se zavolá obsluha příkazu, která hned na svém začátku obvykle použije parser pro získání parametrů. Obsluhu příkazů zajišťuje abstraktní třída *CommandHandler* a její metoda *handle*, které se předá celý řádek s příkazem a informace o příkazu.

Třída CmdParser pro parsování příkazů pracuje ve dvou fázích - v první fázi vyvolané *setLine* se pouze oddělí název příkazu od zbytku parametrů, ve druhé fázi zpracované metodou *parse* se podle informací o příkazu parametry dekódují a určí se jejich typ (*CommandParam*).



Obrázek 2: Sekvenční diagram zpracování příkazů

6.2.4 Automatické doplňování

Pro načítání příkazů se používá knihovna *readline*. Tato knihovna umožňuje pokročilou editaci jednoho řádku a umí i automatické doplňování. Sama o sobě umí doplňovat pouze soubory, poskytuje ale rozhraní pro vytvoření vlastního systému doplňování. Základem je volání jedné funkce nastavené přes ukazatel, která vrací postupně jednotlivé možnosti, přičemž má jeden parametr - stav. Je definováno pouze to, že pro první možnost obsahuje nulu, pro ostatní jinou hodnotu. Konec seznamu řetězců pro doplnění se zajistí vrácením NULL. Doplňuje se vždy poslední slovo, jejich oddělovače jsou definovány ve speciální proměnné knihovny.

V příkazové řádce je využito stavu doplňování pro určení, zda se bude seznam doplňování nejdříve generovat případně vybírat, nebo zda již tento seznam je k dispozici a bude se pouze vracet aktuální možnost a posouvat ukazatel v seznamu. Pro toto oddělení je navržena třída *AutoComplete*. Jedná se o seznam řetězců rozšířený o atribut určující aktuální začátek slova a případně i prefix. Začátek slova se využívá k filtrování seznamu, aby se vracely pouze ty možnosti, které jím začínají. Při generování seznamu se pomocí parseru zjistí příkaz a parametr, na kterém je kurzor. Podle toho se vybere seznam možností, který se v případě potřeby naplní všemi možnostmi. Třída při žádosti o první prvek doplnění v metodě *resetCompletion* zkontroluje seznam a označí si prvky, které se budou vracet. Následná volání metody *nextCompletion* postupně vrací jednotlivé označené prvky.

7 Ladicí knihovna

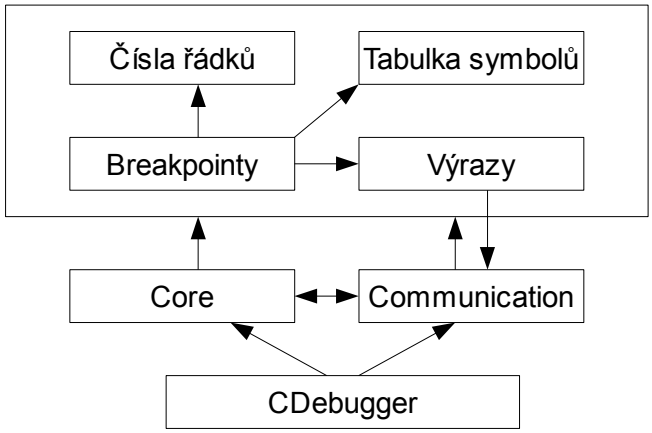
7.1 Napojení simulátoru a ladicí knihovny

Při startu simulátoru se po zpracování parametrů inicializuje procesor pomocí jeho operace *reset*. Poté se zavolá inicializace ladicí knihovny. Přitom se jí předávají v parametrech ukazatele na struktury s ukazateli na funkce pro přístup k mapě paměti, sdíleným i lokálním zdrojům, ukazatel na čítač instrukcí a jméno aplikace k odsimulování. Na konci inicializace se čeká na signál ke startu.

Po inicializaci knihovny a načtení programu se zahájí hlavní cyklus programu obsluhující operaci *main*. Před každým odesláním instrukce do dekoréru se zavolá ladicí knihovna, aby případně zastavila vykonávání při krokování nebo zásahu breakpointu. Během vykonávání může procesor potřebovat přistoupit ke zdroji jiného simulátoru. Toto se opět řeší pomocí ladicí knihovny. Po dokončení simulace se odešle úspěšnost a simulátor končí.

7.2 Architektura ladicí knihovny

Ladicí knihovnu lze rozdělit na několik základních součástí. Tyto součásti ukazuje obrázek, přičemž šipky ukazují, kdo koho především používá.



Obrázek 3: Součásti ladicí knihovny

Rozhraní tvoří třída *CDebugger* a funkce *dbgl_**, které kopírují její metody. Hlavním prvkem je jádro debuggeru, *CDebuggerCore*, které komunikuje se všemi ostatními součástmi. Veškerou komunikaci zajišťuje *CDebuggerCommunication*, přičemž pro distribuovanou simulaci si pomáhá zvláštní třídou. Při zpracovávání příkazů používá i ostatní části knihovny prostřednictvím veřejných ukazatelů ve třídě jádra debuggeru. Jádro debuggeru zajišťuje řízení vykonávání programu (spuštění simulace, krokování, pokračování v běhu) a pro breakpointy využívá třídu *CDbgBreakpoints* a její příbuzné. Ladicí informace (čísla řádků, tabulka symbolů) se získávají při načítání programu a jsou extrahované do vlastních tříd. Posledním velkým modulem je podpora zpracování výrazů. Používá se jednak v podmíněných breakpointech a jednak pro zpracování příkazů *print* a *set*.

Při inicializaci knihovny se běh rozdělí do dvou vláken. První (původní) vlákno zajišťuje běh simulace. Druhé (nové) vlákno má na starosti komunikaci, představuje ho metoda *RecvCommand*. V cyklu přijímá příkazy ze všech spojení a vykonává je. Pro zastavování simulace se používá mutex v jádře debuggeru, na kterém čeká vždy jen simulační vlákno. Komunikační vlákno tento mutex pouze uvolňuje.

7.3 Spustitelný soubor

7.3.1 Formát spustitelného souboru

Formát pro spustitelnou aplikaci je textový, orientovaný na řádky. Stejný formát se používá i pro relokatibilní objektové soubory. Číselné hodnoty jsou uloženy dekadicky bez znaménka. Uložené číselné informace (kód, data) jsou ve dvojkové soustavě, jedna hodnota na řádek. Textová informace je uložena na dvou řádcích. Na prvním řádku je délka řetězce, na druhém vlastní hodnota.

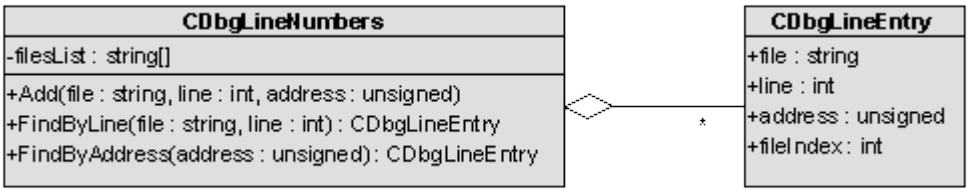
Hlavička souboru obsahuje informace jako počet sekcí, tabulek symbolů, endian a různé flagy. Každá sekce má své jméno, flagy a adresu v paměti. Data sekce, počet relokovaných údajů a počet údajů v tabulce řádků jsou uloženy za hlavičkou sekce, v hlavičce sekce jsou jejich délky a ukazatele na data.

Každý blok záznamů pro čísla řádků je uvozen počtem záznamů. Záznam obsahuje číslo řádku, název souboru a adresu relativní k začátku sekce. Tabulka symbolů je také uvozena počtem záznamů. Každý záznam o symbolu obsahuje jméno symbolu, viditelnost z hlediska linkeru, číslo sekce, relativní adresu a doplňující informace uvozené počtem řádků, na kterých jsou uloženy.

Kompletní popis formátu lze nalézt v interním dokumentu [7], zde bylo cílem ukázat jen principy.

7.3.2 Čísla řádků

Čísla řádků jsou uloženy pomocí třídy *CDbgLineNumbers*. Její strukturu ukazuje diagram. V zásadě se jedná seznam, ve kterém lze rychle vyhledávat podle adresy a podle čísla řádku v souboru. Nabízí se použít dvě hašovací tabulky, jednu pro každý směr.



Obrázek 4: Návrh tříd pro čísla řádků

Pro urychlení vyhledávání při použití jména souboru a zamezení zbytečného opakování jména souboru v záznamech se používá seznam všech použitých souborů, přičemž první z nich je brán jako základní. Toho se využije při nastavování breakpointů na čísla řádků bez uvedení souboru. V hašovací tabulce je součástí klíče index názvu souboru místo řetězce. V záznamu se název souboru vyskytuje, ale jedná se pouze o referenci, řetězec není záznamem vlastněn.

Možným vylepšením implementace je vyhledávání nejbližší adresy nebo řádku. Smysl by to mělo například při ladění v jazyce C, kdy se nastaví breakpoint na adresu, která nemá ekvivalent v informacích o řádcích, ale bylo by možné v editoru zobrazit pozici alepoň orientačně.

7.3.3 Tabulka symbolů

Tabulka symbolů funguje jako hašovací tabulka. Primární je hledat podle názvu symbolu, obrácené hledání může být také užitečné, zejména pokud je možné najít záznam i pro adresu, pro kterou sice záznam neexistuje, ale existuje záznam s nejbližší menší adresou. Toto existuje v GDB a používá se pro symbolické zobrazení adresy jako symbol a offset.

Tabulka symbolů je jednoúrovňová, to stačí pro použití globálních proměnných. Struktura třídy je ukázána na diagramu.

Obrázek 5: Návrh tříd pro tabulku symbolů

Pro podporu jazyka C bude potřeba jednak rozšířit záznam o typovou informaci a změnit implementaci z jednoúrovňové tabulky na hierarchickou (kvůli lokálním proměnným). K tomu by mělo stačit přidat výběr aktuálního kontextu (ve které funkci jsme, případně i v jakém bloku) a specifikace kontextu při přidávání symbolu. Hledání symbolů by používalo stejné rozhraní, hledalo by se přitom v aktuálním kontextu, jen by se použilo více úrovní tabulek.

7.4 Řízení vykonávání programu

Při inicializaci se zamkne simulační mutex a simulační vlákno čeká na jeho uvolnění. Komunikační vlákno mezitím zpracovává příkazy.

Příkaz *sas* v příkazové řádce způsobí, že střední vrstva odešle simulátoru příkaz *SML_DBL_START*, který má v parametrech informace o simulátorech uspořádané do čtveřic: jméno, IP adresa, port a frekvence. Tyto parametry se předají třídě pro distribuovanou simulaci, která zajistí navázání spojení. Poté se uvolní simulační mutex.

Příkaz *step* je možné použít, pokud simulátor čeká na obnovení běhu. Na začátku se zkontroluje, jestli je obnovení možné, poté se nastaví režim krokování a uvolní se simulační mutex. Při simulaci se přitom prostřednictvím funkce *dbg_is_step_mode* zavolá metoda jádra *is_step_mode*, která zkontroluje, zda je simulátor v režimu krokování nebo jestli je vyžádáno vynucené zastavení. Pokud ne, běh normálně pokračuje. Jinak se odešle asynchronní událost o zastavení na řádku nalezeném v modulu čísel řádků podle adresy určené programovým čítačem (pokud se jedná o krokování) a zamkne se simulační mutex.

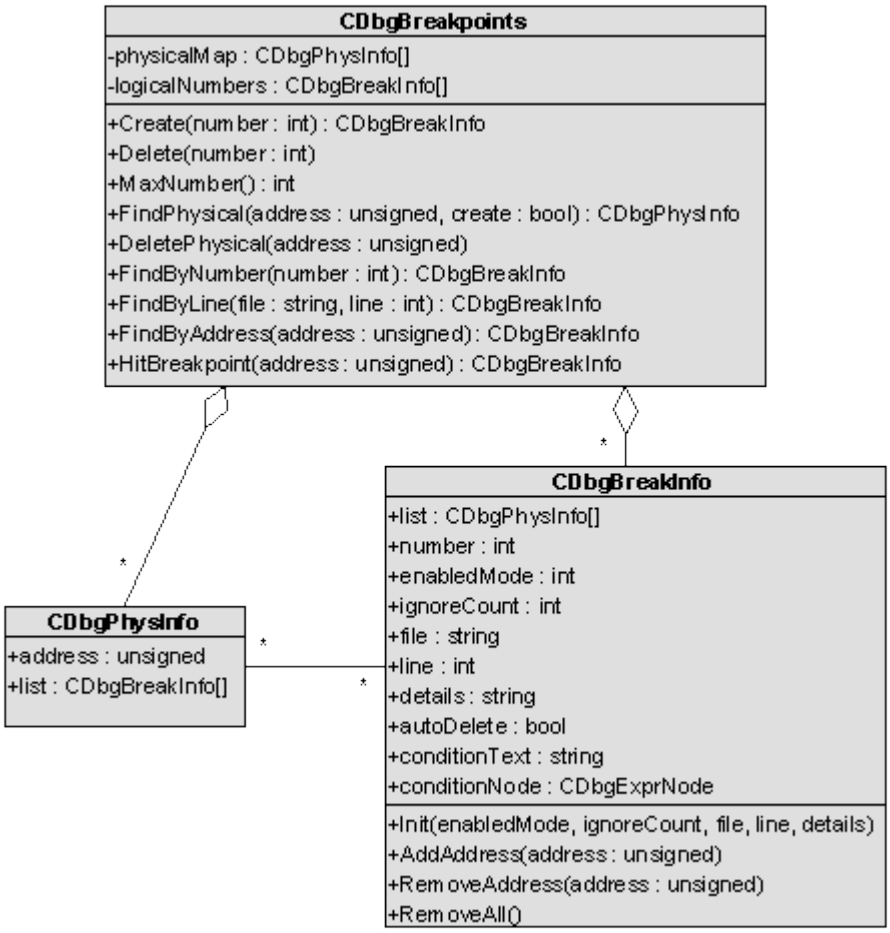
Druhým způsobem obnovení běhu je příkaz *continue*. Ten se zpracuje stejně jako *step*, rozdíl je v tom, že se vypne režim krokování místo zapnutí. Navíc se zruší žádost o vynucené zastavení.

Pokud *is_step_mode* vrátí false, tedy že se nemá krokovat, simulátor (podobně jako u krokování) ještě ověří pomocí metody *is_bp*, jestli je na aktuální pozici breakpoint. Pomocí metody třídy *CDbgBreakpoints HitBreakpoint* se pokusí nalézt informace o breakpointu na aktuální adrese. Pokud se žádné informace nevrátí, simulace pokračuje normálně. V opačném případě se odešle asynchronní zpráva se souborem a číslem řádku, na kterém k zastavení došlo. Nakonec se zamkne simulační mutex.

Při distribuované simulaci může být žádoucí, aby při zastavení jednoho procesoru na breakpointu se zastavily i ostatní. O tom, jestli k zastavení skutečně má dojít, rozhodne střední vrstva při přeposílání události o breakpointu. Všem simulátorům (včetně toho, na kterém došlo k zastavení) se odešle příkaz *SML_DBL_PAUSE*. Tento příkaz nastaví žádost o vynucené přerušení, která se využije při zavolání *is_step_mode* v simulačním vláknu. Vzhledem k tomu, že při jakémkoliv obnovení běhu (*step* nebo *continue*) dojde ke zrušení žádosti, nevadí provedení příkazu již zastaveným simulátorem.

7.5 Podpora breakpointů

Systém správy breakpointů se skládá ze správce breakpointů (*CDbgBreakpoints*), logických (*CDbgBreakInfo*) a fyzických breakpointů (*CDbgPhysInfo*). Diagram tříd ukazuje jejich rozhraní.

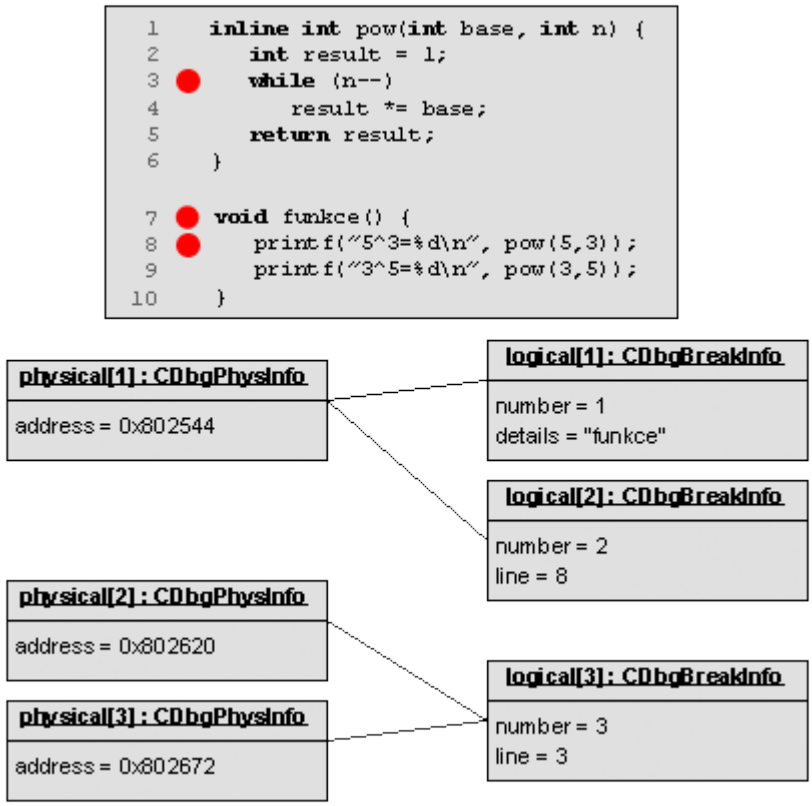


Obrázek 6: Diagram tříd breakpointů

7.5.1 Fyzické breakpointy

Breakpointy jsou rozděleny na fyzické a logické z toho důvodu, že breakpointy lze nastavovat na řádky i symboly, které nakonec mohou ukazovat na stejnou fyzickou adresu a naopak jedna inline funkce může být na několika různých adresách. Fyzický breakpoint je identifikován svou adresou a kvůli urychlení má v sobě zahrnut seznam logických breakpointů, ke kterým náleží.

Na obrázku je vidět, jak by mohl vypadat program a jeho breakpointy. Pro demonstraci je použit program v jazyce C, protože assembler nemá inline funkce.



Obrázek 7: Demonstrace fyzických a logických breakpointů

7.5.2 Logické breakpointy

Logický breakpoint (dále jen breakpoint) symbolizuje breakpoint nastavený příkazem *break* a jeho vlastnosti. Každý breakpoint má své číslo, to je automaticky generované při vytváření breakpointu, přičemž se vybírá nejnižší volné kladné číslo.

U každého breakpointu musí být známé jeho umístění. Breakpoint může být umístěn na číslo řádku, symbol nebo adresu. V prvním případě je umístění uloženo v atributech *file* a *line*. V ostatních případech je umístění uloženo v atributu *details* jako řetězec, který byl při vytváření breakpointu zadán. Ve všech případech se zjistí množina adres, na které je breakpoint potřeba vložit a tato se uloží do seznamu fyzických breakpointů.

K breakpointu lze přiřadit podmínku, která se vyhodnotí při každé kontrole breakpointu na zásah. Je uloženo jednak textové vyjádření podmínky a jednak rozparsovaná forma podmínky (strom výrazových uzlů).

Při kontrole breakpointu se také kontroluje (ještě před podmínkou), zda je povolen a jestli se nemá ignorovat. Pokud je breakpoint zakázán, kompletně se ignoruje. Breakpoint může být povolen trvale nebo dočasně. U dočasného povolení se po zásahu zakáže (*enable once*) nebo smaže (*enable delete*). Protože breakpoint je v době zásahu používán, musí se mazat opožděně. K tomu slouží atribut autoDelete, označuje breakpointy ke smazání. Druhou možností ignorování breakpointu je ignorování několika jeho zásahů. Počet těchto ignorování se nastavuje příkazem *ignore*.

O vytváření breakpointu se stará správce breakpointů, k inicializaci používá metodu *Init* a pro nastavení fyzického umístění *AddAddress*. Při změně umístění lze odstraňovat adresy pomocí *RemoveAddress* a *RemoveAll*. Ostatní vlastnosti breakpointu se upravují přímo nastavením jeho atributů.

7.5.3 Správce breakpointů

Správce breakpointů obsahuje seznam logických breakpointů a mapu fyzických breakpointů. Seznam je použit při vytváření a úpravách breakpointů, čísla breakpointů jsou indexy do tohoto seznamu. Logické breakpointy lze vyhledávat podle jejich čísel, adres a čísel řádků. Mapa fyzických breakpointů zajišťuje unikátnost fyzických breakpointů a hlavně se používá při testování zásahu breakpointu v metodě *HitBreakpoint*.

7.5.4 Protokol pro nastavování breakpointů

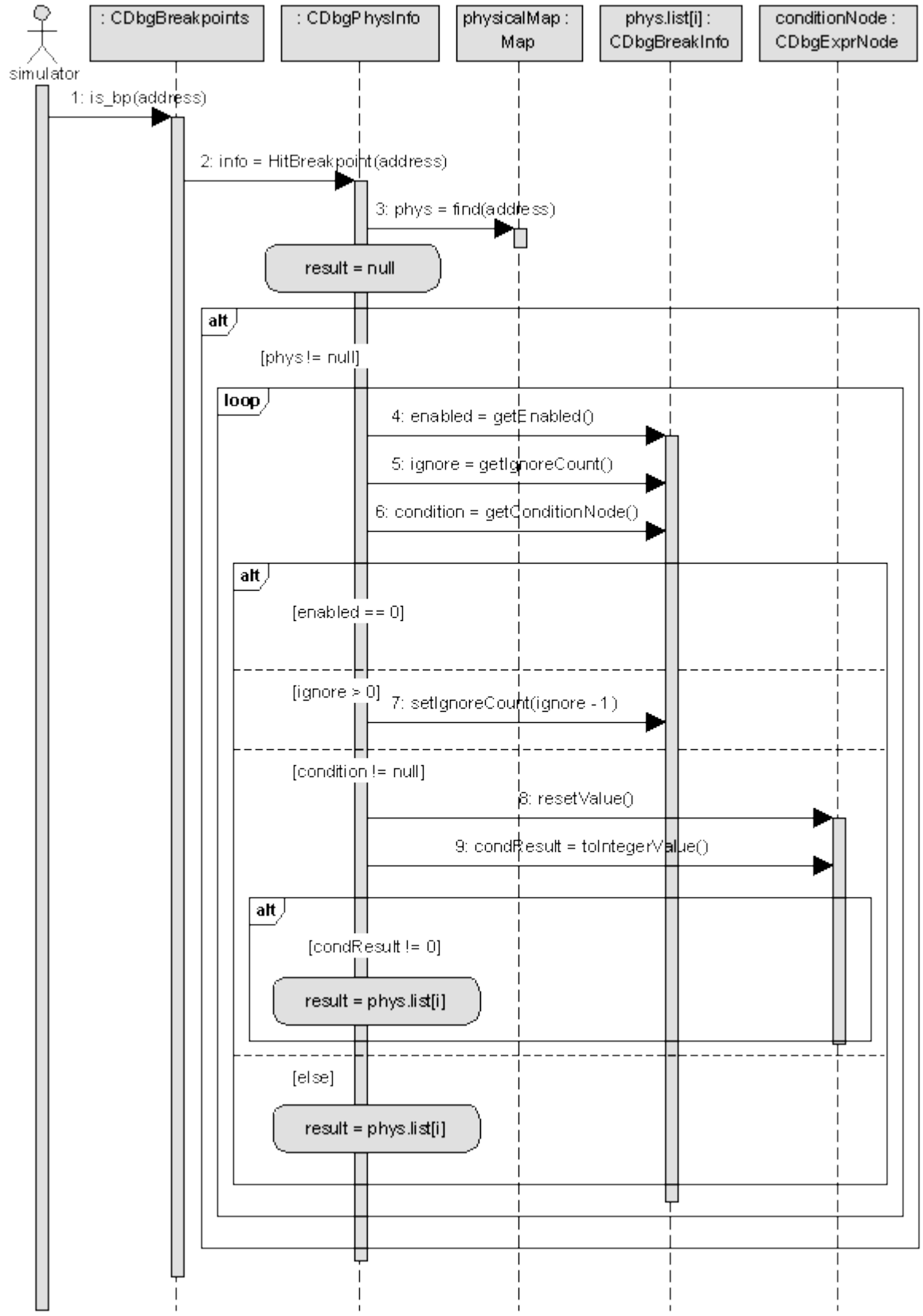
Vytváření a nastavování breakpointů je zajištěno příkazem *SML_DBL_BREAKPOINT*. Vždy obsahuje jméno simulátoru a číslo upravovaného breakpointu, případně nulu při vytváření pomocí *break* nebo rušení pomocí *clear*. Odpovídající příkaz GDB umožňuje v jednom příkazu breakpoint vytvořit, nastavit mu podmínku, případně i další vlastnosti. Aby toto bylo implementovatelné v příkazové řádce, jsou vlastnosti v parametrech postupně uloženy dvojice název vlastnosti a její nová hodnota v textové podobě tak, jak byla zadána v příkazové řádce.

Umístění breakpointu je vyjádřeno vlastností *location*. Možnosti zadání jsou fyzická adresa uvozená hvězdičkou a vyjádřená dekadicky nebo hexadecimálně s předponou 0x, název symbolu pro funkci nebo návěští a číslo řádku volitelně se specifikací souboru oddělené od sebe dvojtečkou.

Další vlastnosti jsou *enable* s hodnotami *disable*, *enable*, *once* a *delete*, vlastnost *ignore* pro ignorování určitého počtu zásahů a *condition* s výrazem pro vyhodnocení jako podmínka. Podmíněný breakpoint se aktivuje, pokud se výraz vyhodnotí jako nenulový. Poslední vlastností je *delete*, která ovšem nemá parametr. Slouží jako příkaz pro smazání breakpointu určeného pomocí čísla nebo umístění v *location*.

7.5.5 Zásah breakpointu

Testování zásahu breakpointu se zajistí voláním metody *HitBreakpoint*, která jako parametr přijímá hodnotu čítače instrukcí a vrací zasažený logický breakpoint nebo NULL. Sekvenční diagram ukazuje, co se děje při testu na zásah breakpointu.



Obrázek 8: Sekvenční diagram zásahu breakpointu

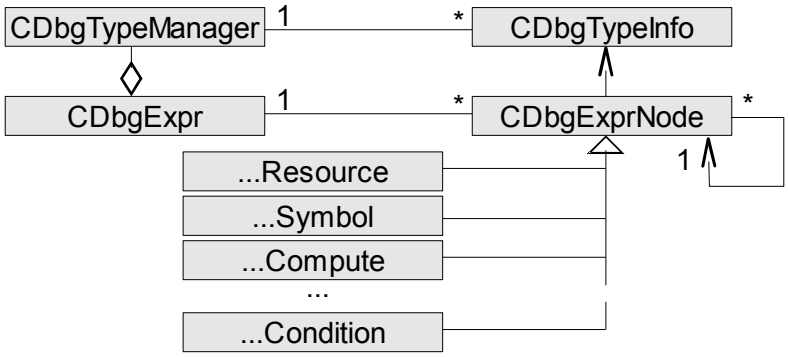
Prvním krokem je nalezení příslušného fyzického breakpointu pomocí mapy physicalMap. Poté se prochází jeho seznamem logických breakpointů. U každého se zkontroluje, jestli není zakázaný. U nenulového počtu ignorování se tento počet dekrementuje a logický breakpoint se přeskočí. Poté se vyhodnocuje již rozparsovaná podmínka.

Použitelný logický breakpoint se zapamatuje a pokračuje se dalším. To proto, aby nezáleželo na pořadí uložení logických breakpointů v seznamu a správně se upravily hodnoty ignorování. Po dokončení průchodu seznamem se vrátí zapamatovaný breakpoint nebo se vrátí NULL.

Vrácený breakpoint se pak využije v jádře debuggeru ke generování asynchronní zprávy o události zásahu breakpointu, která obsahuje jméno souboru a číslo řádku.

7.6 Zpracování výrazů

Modul zpracování výrazů se skládá z ovládacího centra, stromů uzlů, systému typů a z generovaného parseru. Diagram ukazuje vztahy mezi jednotlivými součástmi.



Obrázek 9: Diagram tříd zpracování výrazů

Ovládací centrum tvoří třída *CDbgExpr*. Zajišťuje volání parseru a poskytuje parseru a uzlům prostředky pro hlášení chyb a přístup ke zdrojům, symbolům a typovému systému pomocí statických metod. Ostatní součásti debuggeru používají nestatické metody. Metodou *compile* se výraz v textové podobě převede na strom uzlů výrazu a pomocí metody *evaluate* se pak tento strom vyhodnotí a výsledek výrazu se převede na řetězec podle zadaného formátu.

7.6.1 Syntax výrazu

Pro výrazy se používá podmnožina jazyka C, přičemž jsou podporovány běžné operátory včetně přetypování a podmínkového výrazu. Nejsou ovšem implementovány struktury a přiřazení je možné jen pomocí operátoru = na nejvyšší úrovni výrazu.

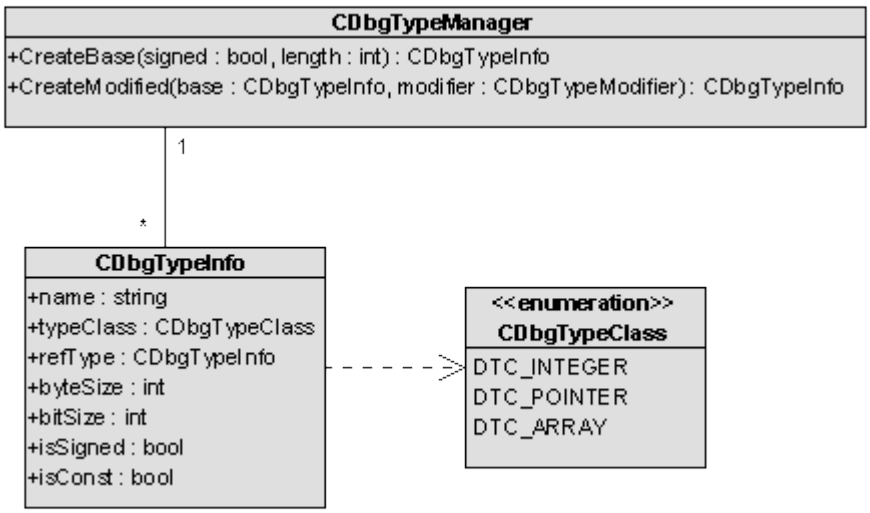
Ke zdrojům procesoru lze přistupovat pomocí znaku dolar následovaného jednoslovným identifikátem zdroje. K paměti lze přistupovat přes ukazatele a symboly. Přístup ke zdrojům, které mají složitější název nebo jsou například indexované, je potřeba použít složitější syntaxi. Tato syntaxe byla zvolena proto, aby gramatika pro parsování byla jednoznačná. Před dvojtečku lze napsat jméno simulátoru, součásti jména zdroje se oddělují tečkou, za jménem lze použít až dvě indexování

pomocí hranatých závorek, přičemž samotný index může být opět výraz. Syntax pro použití zdroje lze zapsat pomocí EBNF:

```
zdroj        = "$", jméno zdroje | ("{", rozšířeně, "}")
rozšířeně    = [simulátor], jméno zdroje, [index, [index]];
simulátor    = identifikátor ":";
jméno zdroje = { identifikátor, "." }, identifikátor;
index        = "[" výraz "]";
```

7.6.2 Typový systém

Typový systém tvoří graf. Základním kamenem jsou primitivní typy, to je například integer a jeho varianty (unsigned, short, long, ...). Z typů lze tvořit ukazatele na ně, konstantní varianty a pole. Zatím nepodporovanými způsoby definice typů jsou typedef a tvorba struktur, které budou obsahovat kromě svého jména seznam trojic offset, název a typ.

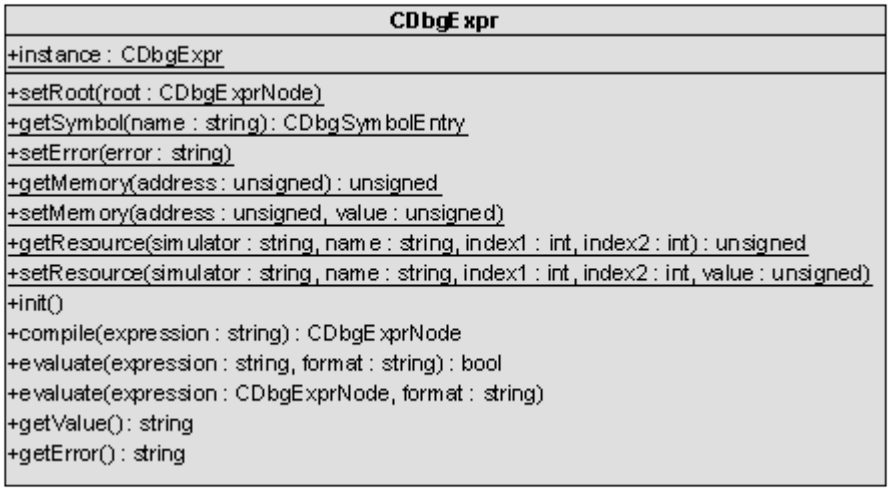


Obrázek 10: Třídy typového systému výrazů

Správcem typů je třída *CDbgTypeManager*, pomocí které se typy tvoří. Správce se sám postará o to, aby stejné typy byly existovaly pouze v jedné instanci. U každého typu lze zjistit jeho název, velikost, rodičovský typ (důležitý zejména pro ukazatele a pole) a krok pro inkrementaci (opět pro pole a ukazatele). U číselných typů se používá informace o přítomnosti znaménka a pořadí bytů v paměti (endian).

7.6.3 Překladač výrazu

Výraz se přeloží pomocí metody compile. Při zavolání se inicializuje lexikální analyzátor, aby zpracovával předaný řetězec, statický atribut *instance* se nastaví na *this* a spustí se parser, který tvoří uzly výrazu. Startovní symbol gramatiky pak předá celý strom třídě *CDbgExpr* pomocí statické metody *setRoot*. Všechny statické metody používají atribut instance k získání aktuálního objektu *CDbgExpr*, se kterým dále pracují.



Obrázek 11: Ovládací centrum výrazů

Lexikální analyzátor čte předaný výraz a parseru předává lexémy, případné hodnoty připojené k tokenům jsou uzly typů identifikátor, číslo nebo typ. Parser pak tvoří další uzly pomocí jejich konstuktorů. Strom uzlů vzniká tak, že některé parametry konstruktorů jsou uzly. Rodičovský uzel je zodpovědný za uvolnění všech svých podřízených uzlů. Právě v konstruktorech probíhá sémantická kontrola a kontrola typů.

Vyhodnocení výrazu se provede nad stromem uzlů metodou *evaluate*, která výsledek převede na řetězec podle zadaného formátu. Výsledný řetězec lze získat metodou *getValue*, případnou chybu lze zjistit pomocí *getError*. Druhou možností je zavolat metodu *getValue* kořenového uzlu, která způsobí rekurzivní vyhodnocení stromu výrazu. Výsledná hodnota je uložena přímo v kořenovém uzlu. Zatím jsou podporovány pouze primitivní typy a hodnota je tedy vždy uložena v atributu directValue, v budoucnu může být hodnota uložena i jinde nebo jiným způsobem. Pro získání celočíselné hodnoty slouží metoda *toIntegerValue*.

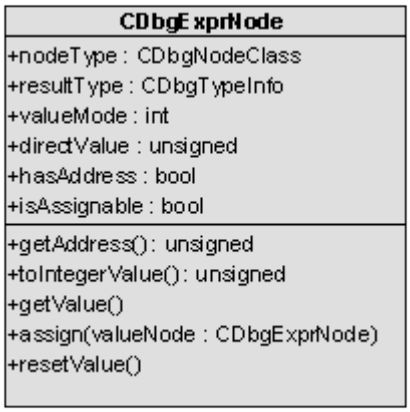
7.6.4 Podmínky v breakpointech

V breakpointech lze používat podmínky, jsou v nich uloženy ve formě stromu výrazu. Ten se vytvoří právě zavoláním compile a výsledkem je tento strom. Výhodou oddělení kompilace a vyhodnocení výrazu je rychlost díky absenci opakovaného parsování.

Protože v uzlech dochází ke kešování výsledků, je potřeba zavolat metodu kořene resetValue, která se rekurzivně zavolá i na podřízené uzly. Poté je možné zavolat metodu *getValue* (i prostřednictvím *toIntegerValue*) a získat aktuálně platný výsledek. Pokud je výsledek nenulový, breakpoint se provede.

7.6.5 Uzly výrazů

U všech uzlů se dá zjistit datový typ a jestli se jedná o L-hodnotu. Uzly jsou odvozeny od třídy *CDbgExprNode*. Samotné uzly určují chování při vyhodnocování (pomocí virtuálních metod) a obsahují odkazy na své parametry, ty jsou při parsování předány v konstruktoru.



Obrázek 12: Uzel výrazu

Informace o hodnotě jsou uloženy v atributech *valueMode* a *directValue*. První z nich určuje, jak je hodnota uložena, jestli vůbec, druhá obsahuje samotnou hodnotu, pokud se ovšem jedná o typ uložitelný do jednoho čísla o délce asi 64 bitů. Hodnota se při vyhodnocování kešuje, takže při opakované žádosti o hodnotu se použije již uložená. Smazat hodnotu z cache lze pomocí metody *resetValue*.

Vyhodnocování probíhá rekurzivním voláním metod *getValue* nebo *getAddress* (pro L-hodnoty s adresou). Metoda *assign* slouží k uložení do L-hodnot (i bez adres, například zdroje procesoru). Samotná třída *CDbgExprNode* má implementovány všechny metody, takže stačí implementovat jen nutné metody a zbytek nechat na rodiči. Tato funguje i metoda *toIntegerValue*, která vyhodnotí výraz pomocí *getValue* a vrátí jeho číselnou hodnotu.

8 Závěr

Seznámil jsem se s principy tvorby debuggerů, formátem pro ladicí informace DWARF a debuggerem GDB. Přestože je formát DWARF určen pro vyšší programovací jazyky, využil jsem jeho myšlenky ohledně typového systému. Debugger GDB posloužil jako vzor pro vytvoření ladicího rozhraní pro příkazovou řádku.

Vytvořený debugger slouží k ladění aplikací napsaných v jazyce symbolických instrukcí. Je ale navržený tak, aby ho bylo možné snadno rozšířit na ladění programů v jazyce C. Pro podporu jazyka C bude potřeba rozšířit typový systém o strukturované typy (struktury a unie) a aliasy typů (typedef), které se pak využijí v modulu zpracování výrazů. Problémy mohou nastat kvůli optimalizacím, kdy příkazy a instrukce nejsou ve stejném pořadí, tehdy může být poněkud matoucí práce s breakpointy a krokováním.

Nově implementovaná příkazová řádka podporuje dávkové zpracování. Kromě načítání příkazů ze souborů a možnosti omezení výpisů je možné omezeným způsobem ošetřovat chyby. Pro důkladné ošetření chyb je potřeba správně informovat chybový systém o problémech, to je zatím problém zejména u příkazů provádějících překlad. Dávkové zpracování by mohlo být v budoucnu rozšířeno o pokročilejší možnosti skriptování, jako jsou podmínky a cykly. Ladicí část příkazové řádky nečeká na zastavení běhu simulátorů, to může být pro dávkové zpracování problém. Zde bude třeba odchytávat asynchronní události a zařídit čekání na ně. Poté bude možné ladit program pomocí skriptů.

Ladicí knihovnu jsem rozšířil o nové možnosti a upravil stávající systém pro breakpointy a ladicí informace. U breakpointů lze nastavovat různé parametry, a to včetně podmínek. Breakpointy jsou navrženy tak, aby bylo možné zpracovávat i inline funkce. V původní knihovně byla z ladicích informací pouze podpora pro čísla řádků, tu jsem upravil a přidal tabulku symbolů. Tabulka symbolů je zatím jednoduchá, pro podporu jazyka C ji bude potřeba upravit na víceúrovňovou kvůli lokálním proměnným.

Jednou z významných součástí ladicí knihovny je podpora zpracování výrazů, které umožňují číst a zapisovat do zdrojů procesorů, umí pracovat se symboly, ukazateli a je možné snadno doplnit podporu pro struktury. Tato součást má kromě příkazů print a set význam hlavně pro podmíněné breakpointy. Prostor pro vylepšení v této oblasti je také pro lepší implementaci obsluhy příkazu info registers, která má vypsat seznam zdrojů a jejich hodnoty.

Všechny části byly testovány na modelech procesorů VEX, MIPS a ARM5. Tyto modely jsou přímo určeny k testování, všechny mají svou variantu aplikace pro výpočet CRC. Z hlediska užitečnosti pro testování debuggeru obsahují podmínky a cykly, tedy lze testovat různé omezení pro breakpointy. Často používají registrová pole, což je vhodné pro testování podpory výrazů (zejména určení zdroje procesoru).

Literatura

[1] ROSENBERG, Jonathan B. *How Debuggers Work*. New York, NY, USA : John Wiley & Sons, Inc., 1996. 256 s. ISBN 0-471-14966-7.

[2] Free Software Foundation. *Debugging with GDB : The GNU Source Level Debugger* [online]. Ninth Edition. 2010 [cit. 2010-05-10]. Dostupné z WWW: <http://sourceware.org/gdb/current/onlinedocs/gdb/>.

[3] *DWARF Debugging Information Format* [online]. [s.l.] : UNIX International, 1993 [cit. 2010-05-10]. Dostupné z WWW: <http://www.dwarfstd.org/doc/dwarf-2.0.0.pdf>.

[4] X86 debug register In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 11 June 2006, 14 March 2010 [cit. 2010-05-10]. Dostupné z WWW: <http://en.wikipedia.org/wiki/X86_debug_register>.

[5] Přikryl, Zdeněk; Hruška, Tomáš; Masařík, Karel: Distributed Simulation and Profiling of Multiprocessor Systems on a Chip. *WSEAS TRANSACTIONS on CIRCUITS and SYSTEMS*. August 2008, Issue 8, Volume 7, s. 788-799. Dostupný také z WWW: <http://www.wseas.us/e-library/transactions/circuits/2008/27-677.pdf>. ISSN 1109-2734.

[6] Mishra, Prabhat; Dutt, Nikil D. *Functional Verification of Programable Embedded Architectures*. Springer Science+Business Media, Inc., 2005. 180 s. Dostupné z WWW: <http://books.google.com/books?id=jGhZOBJvm3AC&printsec=frontcover>. ISBN 0-387-26143-5.

[7] KOLÁŘ, Dušan. *Návrh výstupního formátu pro assembler a linker* [online]. [s.l.] : [s.n.], 5.2. 2004, 7.1. 2008 [cit. 2010-05-19]. Dostupné z CVS: <cvs://lissom.aps-brno.cz/cvs/lissom/doc/tool/vystupni format-0.00.3.doc>. Interní materiál.

Obsah přiloženého CD

Příloha 1. Zdrojové texty