



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

DEPARTMENT OF INTELLIGENT SYSTEMS

SIMULACE PROCESORU ARM PRO VÝUKU PROGRAMOVÁNÍ V ASEMBLERU

SIMULATION OF AN ARM PROCESSOR FOR THE EDUCATION OF PROGRAMMING
IN ASSEMBLER

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

ONDŘEJ ONDRYÁŠ

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. FILIP ORSÁG, Ph.D.

BRNO 2022

Zadání bakalářské práce



Student: **Ondryáš Ondřej**
Program: Informační technologie
Název: **Simulace procesoru ARM pro výuku programování v assembleru**
Simulation of an ARM Processor for the Education of Programming in Assembler
Kategorie: Vestavěné systémy
Zadání:

1. Prostudujte problematiku programování procesoru ARM v assembleru (zaměřte se na 32bitovou verzi ARMv8) a proveďte průzkum stávajících nástrojů simulace činnosti procesoru pro výukové účely. Seznamte se s možnostmi využití existujícího simulátoru instrukcí procesoru ARM pro výuku programování assembleru a jeho integraci do vývojového prostředí Visual Studio Code (VSC).
2. Navrhněte a implementujte architekturu software serveru pro vzdálené spouštění simulace instrukcí procesoru v programovacím jazyce C#.
3. Navrhněte a implementujte modul pro prostředí VSC, který bude umožňovat připojení k simulačnímu serveru a znázorní činnost instrukcí simulovaného procesoru. Zaměřte se na názornou prezentaci chování jednotlivých instrukcí.
4. Otestujte funkčnost klientské a serverové části a otestujte použitelnost řešení na vhodném vzorku uživatelů.
5. Získané výsledky testování shrňte, zhodnoťte klady a zápory navrženého řešení a navrhněte možnosti, jak systém dále zlepšit.

Literatura:

- Smith, Bruce. *Unified Assembly language ARM A32 assembly language: Guide to programming 32-bit ARM, thumb, floating point and Neon*. S.I.: BSB, 2017.
- Johnson, Bruce. *Visual studio code: End-to-end editing and debugging tools for web developers*. B.m.: John Wiley & Sons, Inc., 2019.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování práce viz <https://www.fit.vut.cz/study/theses/>

Vedoucí práce: **Orság Filip, Ing., Ph.D.**

Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.

Datum zadání: 1. listopadu 2021

Datum odevzdání: 29. července 2022

Datum schválení: 3. listopadu 2021

Abstrakt

Cílem této práce je vytvoření didaktického nástroje pro simulaci činnosti procesoru z rodiny Arm, který je integrován do editoru Visual Studio Code. Nástroj je určen pro výuku programování těchto procesorů na strojové úrovni. Implementuje službu umožňující překlad jazyka symbolických adres instrukční sady A32 a simulaci provádění jejích instrukcí. Využívá k tomu emulační jádro Unicorn a další nástroje s otevřeným zdrojovým kódem. Rozšíření pro editor poskytuje s pomocí služby podporu pro vývoj a ladění programů v tomto jazyce. Při tvorbě programu zobrazuje uživateli vysvětlivky pro použité instrukce a pomáhá s pochopením jejich funkcí. Při ladění umožňuje krokování a různé pohledy na vnitřní stav simulovaného procesoru, obsah registrů i paměti. Výsledkem práce je řešení, které je možné použít při výuce předmětu Pokročilé asemblery na FIT VUT. V budoucnu bude možné rozšířit jej o podporu jiných architektur a poskytnout tak nástroj pro usnadnění výuky i v dalších předmětech zaměřených na programování na strojové úrovni.

Abstract

This thesis aims to implement a didactic tool for simulation of an Arm-based processor integrated into the Visual Studio Code editor. The tool facilitates learning about the machine-level programming of these processors. It implements a service that provides an assembler and a simulator for the A32 instruction set. The service is built using the Unicorn emulation framework and other open-source tools. The editor extension uses the service to add support for the development and debugging of programs written in the assembly language. It shows descriptions of used instructions and helps the programmer understand their function. When debugging, it enables stepping through the code and provides various views of the state of the simulated processor, its registers and memory. The solution can be used in the Advanced Assembly Languages course at FIT BUT. It could be further improved in the future to support other architectures and provide an easy learning environment in other courses related to machine-level programming.

Klíčová slova

Arm, AArch32, A32, jazyk symbolických adres, assembler, instrukce, simulace, emulace, Unicorn, Visual Studio Code, rozšíření, jazykový server, LSP, ladicí adaptér, DAP, výuka, vzdálený překlad, ladění, krokování kódu

Keywords

Arm, AArch32, A32, assembly language, assembler, instruction, simulation, emulation, Unicorn, Visual Studio Code, extension, Language Server, LSP, Debug Adapter, DAP, education, remote assembling, debugging, stepping through code

Citace

ONDRYÁŠ, Ondřej. *Simulace procesoru ARM pro výuku programování v assembleru*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Filip Orság, Ph.D.

Simulace procesoru ARM pro výuku programování v assembleru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Filipa Orsága, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....

Ondřej Ondryáš
26. července 2022

Poděkování

Děkuji panu doktorovi Filipu Orságovi za vedení této práce a cenné rady. Děkuji všem svým nejbližším za psychickou podporu, kterou mi v posledních měsících poskytli. Speciální poděkování patří také Ing. Jakubovi Budiskému za věcné připomínky a Bc. Patriku Slámovi za jazykové korektury.

Obsah

1	Úvod	3
2	Procesory Arm a současné možnosti simulace jejich činnosti	4
2.1	Architektury a procesorová jádra Arm	4
2.2	Architektura Armv8	5
2.2.1	Režim AArch32, instrukční sady A32 a T32	6
2.2.2	Registry v režimu AArch32	7
2.2.3	Podpora SIMD a počítání s plovoucí řádovou čárkou	8
2.3	Simulace a emulace	10
2.4	Stávající nástroje pro simulaci činnosti procesorů Arm	10
2.4.1	Výukové simulátory	11
2.4.2	Komerční řešení	15
2.4.3	Emulace na systémové úrovni	16
2.5	Emulační jádro Unicorn	17
2.6	Jazyk symbolických adres a jeho překlad	17
2.6.1	Překlad jazyka symbolických adres	19
2.6.2	Překladače JSA procesorů Arm, the GNU Assembler	21
2.6.3	Formát ELF	21
2.6.4	Další možnosti překladačů	22
3	Editor Visual Studio Code a možnosti jeho rozšíření	23
3.1	Možnosti rozšíření editoru a jejich architektura	24
3.1.1	Architektura rozšíření	25
3.2	Rozšiřování jazykové podpory editoru	26
3.2.1	Deklarativní rozšíření	26
3.2.2	Programová rozšíření	27
3.2.3	Jazykové servery, Language Server Protocol	27
3.3	Architektura ladicích nástrojů editoru a její rozšiřování	29
3.3.1	Ladicí adaptéry	30
4	Návrh řešení	33
4.1	Požadované vlastnosti řešení	33
4.1.1	Integrace podpory jazyka do editoru	33
4.1.2	Spouštění a ladění	37
4.1.3	Reprezentace stavu procesoru a paměti v režimu ladění	40
4.2	Architektura navrhovaného řešení	43
4.2.1	Jazykový server	43
4.2.2	Překlad, spouštění a ladění jako síťová služba	43

4.2.3	Architektura běhové služby	44
4.2.4	Rozhraní mezi jednotlivými vrstvami řešení	44
5	Implementace	47
5.1	Překlad, spouštění a ladění na bázi emulačního jádra Unicorn	47
5.2	Rozšíření pro Visual Studio Code	62
5.3	Poskytování kontextových informací o kódu pomocí jazykového serveru . . .	65
6	Vyhodnocení řešení	72
6.1	Testování řešení	72
6.2	Možnosti vylepšení stávajícího řešení	75
6.3	Možnosti dalšího rozšíření	76
7	Závěr	79
	Literatura	81
	Přílohy	84
	Seznam příloh	85
A	Obsah paměťového média	86
B	Uživatelský manuál	87
C	Překlad a spuštění ze zdrojových kódů	98
D	Gramatika sledovatelných výrazů	101
E	Podpora požadavků protokolů DAP a LSP	102
F	Překládová služba (diagram tříd)	104
G	Emulační a ladicí služba (diagram tříd)	106
H	Rámce a proměnné (sekvenční diagram)	107
I	Emulační služba (stavový diagram)	108
J	Analýza operandů (pseudokód)	109
K	Srovnání možností ladění	111
L	Podpora instrukcí jazykovým serverem	114
M	Možnosti úpravy systému proměnných	116
N	Ukázky řešení	118
N.1	Zobrazení proměnných v režimu ladění	119

Kapitola 1

Úvod

Srdcem každého počítače je procesor, to je dnes již obecně známý fakt. Mnozí také alespoň tuší, že procesor je stěžejní součástí nejen počítačů, ale také telefonů a většiny spotřební elektroniky, se kterou každodenně přicházíme do styku. Právě v této oblasti v současné době dominují procesory založené na architekturách Arm. Tento status si vydobily mj. díky své skvělé energetické účinnosti a flexibilitě, která se projevuje v extrémně nízkých cenách čipů. Není tedy překvapivé, že se k těmto strukturám obrací vývojáři hardwaru i softwaru od domácích kutilů až k velkým firmám, které na ní staví úspěšná komerční zařízení.

Ruku v ruce s takto masovým rozšířením počítačové architektury přichází i potřeba vzdělávat programátory ve vývoji pro ni. Pro platformy na bázi Arm je vývojářům dostupné velké množství zdrojů, hardwarových i softwarových. Na procesorech Arm je možné provozovat známé operační systémy (Windows, Linux, macOS i další) a existují pro ně překladače či interprety většiny běžně používaných programovacích jazyků. Běžného vývojáře aplikací tak mnohdy nemusí vůbec zajímat, že jeho kód poběží na procesoru Arm. Je ale nutné si uvědomit, že kód napsaný v libovolném jazyce nakonec musí být převeden do formy instrukcí pro daný procesor. Aby celý ekosystém Arm mohl existovat, je zapotřebí odborníků, kteří zvládnou programovat tyto procesory na nejnižší možné úrovni.

Cílem této práce je vytvořit řešení usnadňující výuku programování procesorů Arm na strojové úrovni, v jazyce symbolických adres (*assembly language*, často slangově označovaném jako *assembler*). Záměrem je poskytnout studentům prostředí, ve kterém si mohou bez předchozí komplikované instalace a konfigurace rovnou vyzkoušet vývoj jednoduchých i komplexnějších programů v tomto jazyce. Řešení poskytuje jazykovou podporu v editoru, na kterou jsou studenti zvyklí z vyšších programovacích jazyků (zvýrazňování syntaxe, nášeptávač atp.). Významnou částí řešení je pak samotná simulace, během které je možné ladit kód a jednoduše zjišťovat aktuální stav simulovaného procesoru (obsah registrů, paměti) v didakticky přínosné reprezentaci.

Ve druhé kapitole je blíže představena architektura Arm a některé její vlastnosti a možnosti. Vysvětlena je související terminologie a také problematika různých profilů a instrukčních sad, které jsou součástí architektury Armv8. Dále jsou představeny současné možnosti její simulace, a to zejména v kontextu výuky. Nastíněny jsou zde také některé problémy, které je nutné řešit při překladu jazyka symbolických adres, a jejich řešení.

Ve třetí kapitole je představen editor Visual Studio Code a způsoby, jakými je možné rozšiřovat jeho funkce a podporu pro různé programovací jazyky a prostředí. Popsány jsou zde také protokoly Language Server Protocol a Debug Adapter Protocol, které editor pro doplňování jazykové podpory využívá.

Ve čtvrté kapitole jsou určeny funkce, které by měl nástroj pro výuku jazyka symbolických adres podporovat. Na základě uvedených požadavků je pak navržena architektura celého řešení. V páté kapitole je vysvětleno, jakým způsobem je navržený nástroj implementován. V šesté kapitole je popsán způsob, jakým byl nástroj testován, a jsou zde představeny některé možnosti jeho vylepšení a budoucího rozšíření.

Kapitola 2

Procesory Arm a současné možnosti simulace jejich činnosti

Slovo Arm vzniklo jako zkratka z *Acorn RISC Machine* a typicky se jím v současnosti označuje rodina procesorových architektur typu RISC, které navrhuje společnost Arm Ltd. Její předchůdce, Acorn Computers Ltd., byl založen v roce 1978 a zpočátku se věnoval výrobě tehdy populárních 16bitových počítačů. Jejich počítač BBC Micro, vydaný v roce 1981, byl ve Spojeném království mimořádně úspěšný a poskytl společnosti motivaci i finanční zájem pro začátek vývoje vlastního procesoru. Projekt procesoru využívajícího redukovanou instrukční sadu započal v roce 1983 a první čip ARM byl vyroben 26. dubna 1985. V roce 1990 se tým pracující na procesorech ARM přesunul pod novou společnost Advanced RISC Machines Ltd., dnes už jen Arm Ltd. V současné době využívají procesory Arm¹ prakticky všechny chytré telefony a postupně se zvedá jejich podíl i na trhu s osobními počítači a servery².

2.1 Architektury a procesorová jádra Arm

Architektury Arm jsou zástupcem počítačů s redukovanou instrukční sadou RISC (*Reduced Instruction Set Computer*). Tento typ architektury se vyznačuje omezením množství práce, kterou vykonává jedna instrukce ve smyslu počtu přístupů do paměti a počtu cyklů procesoru potřebných k jejímu provedení. Cílem omezení na jednodušší instrukce bylo mj. dosáhnout stavu, kdy bude jedna instrukce ideálně trvat pouze jeden takt procesoru, a zjednodušit tak využití řetězeného zpracování (*pipelining*), což je technika paralelismu na úrovni instrukcí zvyšující propustnost procesoru.

Těchto cílů dosahují RISC architektury různými způsoby. S pojmem RISC se do značné míry překrývá také pojem **load/store architektura**. V té může přístupy do paměti provádět pouze několik specializovaných instrukcí; aritmetické, logické a další datové operace mohou být prováděny pouze nad registry, kterých je typicky dostupné velké množství. To výrazně usnadňuje návrh procesoru a instrukční sady. Instrukce mívají pevnou délku (v případě Armv8 vždy 32 nebo 16 bitů, viz kap. 2.2). Paměť je typicky adresována jednoduššími adresními módy, kterých není mnoho.

Hlavní činností společnosti Arm Ltd. je návrh **architektur a procesorových jader**, která tyto architektury implementují. Arm samotný však nevyrábí fyzické čipy, jejich produktem jsou právě návrhy a popisy, tzv. *IP Cores* (*Intellectual Property*, duševní vlastnictví), které licencují třetím stranám. Architektury Arm jsou značně flexibilní ve smyslu podpory jednotlivých funkcí konkrétními implementacemi i ve smyslu chování procesoru, což umožňuje navrhovat různá jádra pro konkrétní třídy aplikací. Zjednodušení jader se pak

¹V minulosti byl využíván pravopis „ARM“, od srpna 2017 je ale oficiální podoba právě ta s velkým počátečním písmenem a ostatními malými, a to prakticky ve všech kontextech (v názvu společnosti, procesorů, instrukčních sad apod.) [8].

²[17] odhaduje, že v roce 2020 byly procesory Arm užity v téměř 100 % chytrých telefonů a 89 % tabletů. Podle [1] se v roce 2021 odhadoval 7% podíl na trhu PC a 4% podíl u serverů; oba zdroje se shodují na tom, že na poli PC se dá očekávat výrazný růst.

příznivě projevuje na ceně. V současné době nabízí Arm pod značkou *Cortex* 36 různých typů jader³. Jako přednosti svých architektur uvádí „bezpečnost, vysoký výkon a energetickou účinnost, rozsáhlý ekosystém a rozšířenost na trhu“ [7].

V této práci je pojem architektura obvykle užíván zaměnitelně s pojmem **instrukční sada** (angl. *instruction set architecture*). Ta definuje způsob, jakým je procesor řízen softwarem [5]. Poskytuje programátorovi prostředky, pomocí kterých interaguje s hardwarem. Jde o abstraktní model počítače, který určuje, jaké operace daná architektura umožňuje provádět, jak jsou jednotlivé instrukce kódovány, definuje použité datové typy a velikosti operandů, registry, způsob manipulace s pamětí, způsob ošetřování výjimek, možnosti skoků v programu a další nezbytné detaily, které přesně definují funkci procesoru z pohledu systému, programátora a programu. Někdy je však termínem architektura označena instrukční sada i způsoby její implementace (také označovány jako organizace nebo mikroarchitektura) [13, kap. 1.3].

Architektury Arm jsou označovány zkratkou *ArmvX*, kde *X* je verze. Aktuální verzí je Armv9, důležitými milníky v historii ale byly zejména Armv4, Armv6, Armv7 a Armv8. Verze Armv7 byla jako první definována ve třech verzích – **profilech**, které se zásadněji lišily v podpoře funkcí a systémových detailech (viz kap. 2.2). Až do verze Armv7 (včetně) byly architektury Arm výhradně 32bitové, pracovaly tedy s 32bitovými registry a adresním prostorem 2^{32} bajtů, tj. 4 GiB (s výjimkou rozšíření umožňujících různými technikami adresovat i větší paměťový prostor) [2, A3.1]. Verze Armv8 pak přinesla zásadní změnu: nový 64bitový režim s kompletně přepracovanou instrukční sadou a mnoha koncepčními rozdíly.

2.2 Architektura Armv8

Architekturu Armv8 poprvé představil Arm v roce 2011. První spotřebitelské produkty založené na této architektuře se dostaly na trh v roce 2013. Architektura nově definovala 64bitový režim, zároveň však zachovala kompatibilitu s předchozími 32bitovými verzemi. Z tohoto důvodu popisuje architektura dva výrazně odlišné **režimy provádění** (*execution states*): AArch32 a AArch64. Architektura určuje, jakým způsobem procesor přechází z jednoho režimu do druhého a v jakých případech tak může činit. Hlavním rozdílem je přirozeně bitová šířka sběrnic a registrů. Režim AArch64 však přinesl i významné změny v operačních režimech procesoru a způsobu nakládání s výjimkami, v organizaci registrů a paměti, v instrukčních sadách apod.

Paralelně ke dvěma režimům provádění existují od verze Armv7 také tři rozdílné varianty architektury, které jsou označeny jako **profily**. Ačkoliv jsou si profily koncepčně blízké, významně se liší v zaměření a mnoha detailech. Arm definuje následující profily [3, kap. A1.2]:

- profil **A** (*Application*): určen pro běžné aplikace a systémy, je charakteristický podporou systému virtuální paměti VMSA (*Virtual Memory System Architecture*) využívajícího jednotku správy paměti MMU (*Memory Management Unit*), až na výjimky všechny implementace podporují SIMD/FP instrukce [3, kap. A1.5] (viz dále);
- profil **R** (*Real-time*): určen pro systémy reálného času, podporuje jednodušší ochranu paměti PMSA (*Protected Memory System Architecture*) založenou na jednotce ochrany paměti MPU (*Memory Protection Unit*), ale neobsahuje MMU;

³Přehled nabízených jader: <https://www.arm.com/products/silicon-ip-cpu>, cit. 1. 6. 2022.

- profil **M** (*Microcontroller*): určen pro mikrokontroléry a vestavné systémy, volitelně může podporovat jednodušší variantu PMSA, podporuje pouze omezenou instrukční sadu T32, může obsahovat omezené SIMD nebo FP rozšíření [6, kap. A1.4].

V této práci se pod označením Armv8 vždy myslí profil **Armv8-A ve 32b režimu AArch32** (pokud není řečeno jinak). Toto zaměření jsem zvolil, protože je jeho znalost i v současné době v kontextu architektury Arm zásadní. V profilu Armv8-A je 64b režim nepovinný (neimplementuje jej např. jádro Cortex-A32), pro profil R byl doplněn relativně nedávno (první veřejná verze referenčního manuálu vyšla v září 2020 [4]) a podporuje jej pouze jádro Cortex-R82, v profilu M není vůbec podporován [6].

2.2.1 Režim AArch32, instrukční sady A32 a T32

Režim provádění AArch32 architektury Armv8 je nástupcem předchozích verzí architektury Arm. Je s nimi zpětně kompatibilní, umožňuje tedy spustit např. kód napsaný pro verzi Armv7 [3, kap. A1.1]. Jako povinné obsahuje různé funkce, které byly v předchozích verzích pouze rozšířením (např. podporu SIMD a FP operací), přičemž vylepšuje jejich možnosti. Některé koncepty naopak ruší, resp. jejich provádění pouze emuluje (např. podporu koprocessorů).

V režimu AArch32 je možné využít dvě instrukční sady: A32 a T32. Nejde o instrukční sady v typickém smyslu, jaký byl uveden výše. A32 (dříve *ARM*) a T32 (dříve *Thumb-2*) označují především způsob kódování instrukcí, přičemž ostatní aspekty architektury zůstávají téměř stejné. V sadě A32 jsou všechny instrukce dlouhé 4 B, zatímco sada T32 pro zvýšení hustoty kódu kombinuje 2B a 4B instrukce. Jeden program může obsahovat bloky využívající sadu A32 a jiné využívající sadu T32, procesor mezi těmito instrukčními sadami může přepínat, vždy však běží právě v jedné z nich [31, kap. 26].

Verze Armv8 definuje také UAL (*Unified Assembler Language*), což je upravená syntaxe jazyka symbolických adres, která umožňuje psát kód jednotným způsobem tak, aby jej bylo možné přeložit jak s využitím kódování A32, tak s využitím T32 bez nutnosti změn [31, kap. 26]. Zároveň je ale stále možné použít dřívější podoby jazyka, dříve napsaný kód tedy není nutné přepisovat.

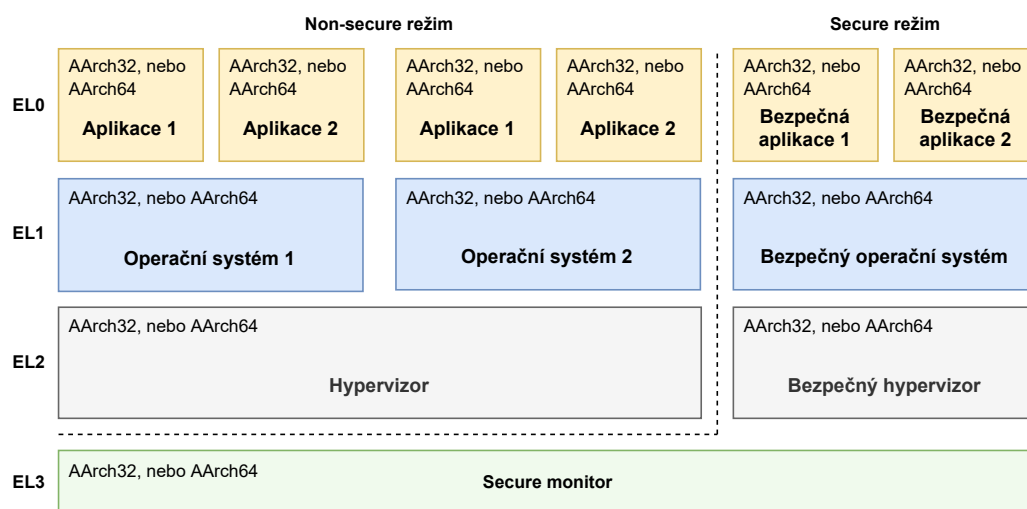
Operační módy a úrovně výjimek

Starší model výjimek architektury Armv7 je založen na operačních módech (*PE modes*). Ty slouží zejména pro řízení přístupu k systémovým zdrojům a pro poskytnutí běhového prostředí v různých situacích. Liší se především množinou instrukcí a registrů, které může kód využívat. Mezi jednotlivými módy procesor přechází typicky v reakci na výjimku nebo přerušení. Uživatelské aplikace běží v neprivilegovaném módu *User* s nejmenším oprávněním. Vyšší oprávnění mají módy *FIQ* a *IRQ* (pro obsluhu přerušení), *Abort*, *Undefined*, *Supervisor* a *System* [2, kap. B1.3].

V architektuře Armv8 je model výjimek a řízení přístupu ke zdrojům výrazně přepracován. Využity jsou tzv. úrovně výjimek (*Exception levels*), které jsou označeny jako EL0, EL1, EL2 a EL3. S rostoucím číslem roste úroveň oprávnění a dostupných systémových prostředků. Paralelně k nim jsou přítomny ještě dva bezpečnostní režimy (*Secure* a *Non-secure*), které poskytují ještě větší granularitu oprávnění. Jednotlivé úrovně výjimek mohou být implementovány v různých režimech (AArch32, nebo AArch64). Při přechodu z nižší do vyšší úrovně je možné přejít z AArch32 do AArch64 (např. 64b operační systém, který běží

v úrovni EL1, může spouštět 32b i 64b aplikace, které běží v úrovni EL0), ale ne naopak (32b OS nemůže spouštět 64b aplikace).

Procesory arch. Armv8 jsou vždy implementovány s využitím úrovně výjimek, ale pro kód, který na nich běží v režimu AArch32, je tato skutečnost transparentní. Architektura definuje komplexní pravidla pro mapování mezi úrovněmi výjimek a operačními módy. Tato práce se dále zabývá především simulací režimu AArch32 na aplikační úrovni, tedy v operačním módu *User*.



Obrázek 2.1: Úrovně výjimek architektury Armv8. Režim AArch64 může být na každé úrovni použit jen tehdy, když je vyšší úroveň také implementována v režimu AArch64. Zdroj: [3, obr. D1-1], upraveno.

2.2.2 Registry v režimu AArch32

Režim AArch32 v obou svých instrukčních sadách poskytuje 15 registrů o šířce 32 bitů pro obecné použití, označeny jsou R0 až R14. Registr R13 se také používá pro uložení ukazatele na zásobník a je možné k němu přistupovat pod jménem SP (*Stack Pointer*, ukazatel na zásobník). Registr R14 se používá pro uložení návratové adresy při skoku a je možné k němu přistupovat pod jménem LR (*Link Register*, registr pro návratové adresy). Speciální sémantika těchto registrů se projevuje pouze ve funkci některých instrukcí (např. instrukce POP pro čtení ze zásobníku implicitně využívá jako cílovou adresu hodnotu SP), v programu ale mohou být využity libovolným způsobem.

Dalším významným registrem je programový čítač PC (*Program Counter*). Ten nelze považovat za registr pro obecné použití, neboť zápisy do něj mění tok programu. V instrukční sadě A32 jej ovšem lze použít jako operand většiny instrukcí, které pracují s registry pro obecné použití, skok je tedy možné provést např. instrukcí pro sčítání: ADD PC, #4. Z tohoto důvodu je možné využívat pro něj i alternativní jméno R15. Arm ovšem doporučuje přistupovat k PC pouze prostřednictvím instrukcí, které jsou k tomu explicitně určeny [3, kap. E1.2.3, F5].

Některé registry mají více samostatných instancí pro jednotlivé operační módy (*register banking*, viz [3], kap. G1.9.2). Každý operační mód má vlastní dvojici registrů SP (R13) a LR (R14), mód FIQ má navíc ještě vlastní sadu registrů R8–R12.

Příznakové registry

Instrukční sada definuje také mnoho dalších registrů, které nesou informace o stavu procesoru a umožňují konfigurovat jeho chování. Většina z nich má příznakový charakter: nosičem informací jsou jednotlivé bity nebo menší skupiny bitů uvnitř jejich 32bitových hodnot. Většinu těchto registrů označuje referenční manuál jako systémové registry (*System registers*) [3, kap. G7.1].

Programátoři na aplikační úrovni se mohou často setkat se stavovými registry CPSR a APSR. Jsou v nich uloženy příznaky, které představují výsledek operace provedené instrukcí a které jsou využity např. při podmíněném větvení: *Negative* (nejvyšší bit roven 1, výsledek operace je v doplňkovém kódu negativní), *Zero* (výsledek operace je roven 0), *Carry* (bezznaménkové přetečení) a o *Verflow* (znaménkové přetečení).

Registr CPSR (*Current Program Status Register*) je možné číst a modifikovat pouze speciálními instrukcemi. V op. módu *User*, resp. na úrovni EL0, však program k tomuto registru přístup nemá – využívat může pouze jeho omezenou podmnožinu ve formě registru APSR (*Application Program Status Register*), jak naznačuje obrázek 2.2. K těmto registrům může program přistupovat pomocí instrukcí MSR a MRS, obvykle je ale využívá nepřímo pro podmíněné vykonávání jiných instrukcí (téměř všechny instrukce umožňují určení podmínkového kódu). Architektura definuje také registry SPSR (*Saved Program Status Register*), do kterých se při změně op. módu ukládá hodnota CPSR z módu *User*.

CPSR

31	30	29	28	27			23	22	21		19	16			9	8	7	6			3	0
N	Z	C	V	Q	000		SSBS	DIT	PAN	0	GE		000000		E	A	I	F	0	1		M

APSR

31	30	29	28	27						19	16											0
N	Z	C	V	Q	0000000					GE		000000000000								1		000

Obrázek 2.2: Registr CPSR a aplikační pohled do něj – registr APSR. Popisy dalších příznaků jsou uvedeny v [3], kap. G8.2.33.

Z dalších systémových příznakových registrů je v kontextu práce významný také reg. FPSCR (*Floating-Point Status and Control Register*), který obsahuje informace o jednotce pro počítání s plovoucí řádovou čárkou (včetně jejích příznaků N, Z, C, V) a její konfiguraci. Chování této jednotky ovlivňuje také reg. FPEXC (*Floating-Point Exception Control*).

K většině systémových registrů se přistupuje pomocí instrukcí MCR a MRC, které v předchozích verzích architektury sloužily jako jednotný způsob k manipulaci s registry koprocetorů. Systémové registry byly přístupné jako speciální logické koprocetory CP10–CP14. Princip koprocetorů byl v arch. Armv8 odstraněn, pro zachování zpětné kompatibility jsou však systémové registry stále zpřístupněny tímto způsobem [3, kap. G1.19.4].

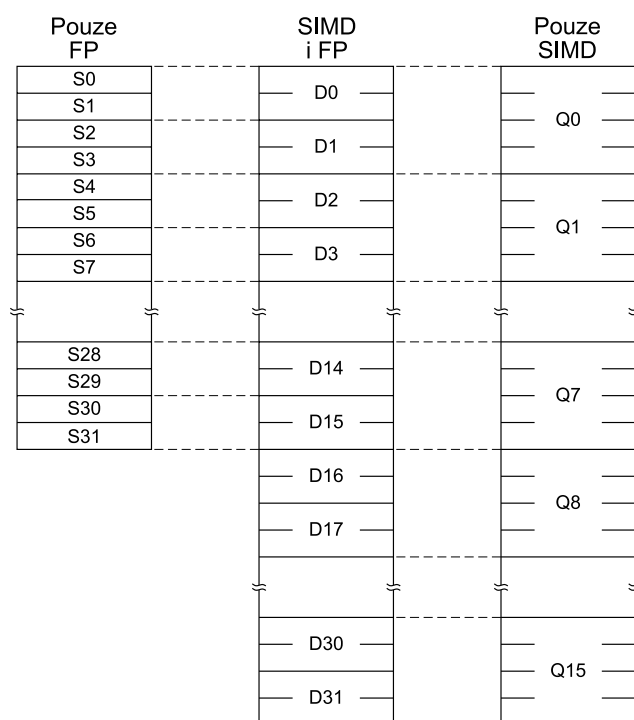
2.2.3 Podpora SIMD a počítání s plovoucí řádovou čárkou

Zkratka SIMD (*single instruction, multiple data* – jedna instrukce, více dat) označuje jednu z technik paralelního zpracování dat. SIMD instrukce umožňují najednou provádět operaci nad **vektorem** (uspořádanou n -ticí) dat. Tato technika se využívá při optimalizaci některých tříd algoritmů, časté využití nachází např. ve zpracovávání multimediálních dat.

Čísly s plovoucí řádovou čárkou FP (*Floating-Point*) rozumíme počítačovou reprezentaci reálných čísel. Jednu z jejích konkrétních podob definuje např. standard IEEE 754 [14], který implementuje i arch. Armv8.

SIMD a FP spolu nutně nemusí přímo souviset, ale různé implementace SIMD mohou podporovat počítání s FP čísly i s celými čísly. Arch. Armv7-A umožňovala implementovat zvlášť tzv. *Advanced SIMD* rozšíření a *VFP* rozšíření, a to v různých kombinacích možností a verzí, viz [2], kap. A2.6. Arch. Armv8-A sjednocuje podporu SIMD a FP a zavádí ji jako (téměř) povinnou pro všechny⁴ implementace pod označením *Advanced SIMD and floating-point support* (v práci je pro ni a pro související registry a instrukce využita souhrnná zkratka SIMD/FP). Obchodní označení pro tuto technologii je *Neon*.

SIMD/FP registry a datové typy



Obrázek 2.3: SIMD a FP registry. Registry třídy D jsou pohledem na poloviny registrů třídy Q; registry S jsou pohledem na poloviny registrů třídy D. Zdroj: [3, obr. E1-1].

SIMD a FP instrukce arch. Armv8 operují nad samostatnou sadou registrů. Fyzicky jde o šestnáct 128bitových registrů označených Q0–Q15. Každý z nich je logicky rozdělen na dvě 64b poloviny, ke kterým je možné přistupovat pod názvy D0–D31. Prvních 16 z nich je dále logicky rozděleno na dvě 32b poloviny, které jsou dostupné pod názvy S0–S31. Tyto tři skupiny jsou v práci označovány také jako **třídy** SIMD/FP registrů. Mapování jednotlivých tříd naznačuje obrázek 2.3.

Registry třídy S je možné použít pouze jako operandy FP instrukcí (ty pracují s jednou FP hodnotou, skalárem). Registry třídy D je možné využít jako operandy FP instrukcí, kde jejich hodnoty představují skalární 64b hodnotu, nebo jako operandy SIMD instrukcí, ve

⁴Referenční manuál připouští i implementace bez SIMD a FP podpory, ale ty jsou licencovány jen pro specializované aplikace a trhy [3, kap. A1.5].

kterých představují vektor několika hodnot menší šířky. Registry třídy Q je možné použít pouze v SIMD instrukcích, neboť Armv8 nepodporuje 128b FP čísla.

SIMD instrukce typicky umožňují pracovat s vektory 8b, 16b, 32b a 64b celých čísel (bezznaménkových i znaménkových) a s 16b (*half-precision*) a 32b (*single-precision*) čísla s plovoucí řádovou čárkou (v kódováních *binary16* a *binary32* podle standardu IEEE 754) [3, kap. A1.4.1]. FP instrukce, které jsou pouze skalární, umožňují pracovat kromě 16b a 32b čísel také s 64b čísla (*double-precision*) v kódování *binary64* [3, kap. A1.4.3/4/5]; dostupné mohou být také alternativní 16b formáty, které nejsou pro tuto práci důležité.

2.3 Simulace a emulace

Termíny simulace a emulace nemají pevnou definici a je možné nahlížet na ně různými způsoby. Emulaci popisuje [27] jako „*modelování operací jednoho počítače interpretačním programem na jiném počítači*“. [12, s. 179] definuje emulátor jako „*program běžící na jednom počítači, ale přesně zpracovávající instrukce pro jiný počítač*“. Simulaci pak [27] definuje jako „*napodobování děje, chování nebo nějaké soustavy modelem*“, obdobně ji popisuje také [28, s. 1259]. V obou případech tedy dochází k napodobování chování určitého systému, rozdílem je však především záměr použití.

Emulátor je v dnešním pojetí typicky software, jehož primárním cílem je napodobit jiné zařízení (obvykle nějaký počítač, resp. jeho instrukční sadu) natolik přesně, že je možné jej, úplně nebo v dostatečně velkém rozsahu, použít namísto napodobovaného zařízení. Přijímá stejná data, provádí stejný program a dochází ke stejným výsledkům jako počítač, který je jím napodobován [28, s. 535]. V ideálním případě uživatel „nepozná“, že je spuštěný program emulován, a podobně je i pro samotný program tato skutečnost transparentní. Emulátor musí dosáhnout stejného vnějšího chování, jaké vykazuje emulovaný systém (tedy replikovat jeho rozhraní), ale typicky přitom nemusí modelovat jeho vnitřní implementaci.

Cílem simulátoru je oproti tomu studium chování simulovaného systému, získávání nějakých nových poznatků [28, s. 1259]. Simulaci je možné popsat jako proces experimentování s modelem, při kterém se zkoumá nějaký konkrétní aspekt systému – nejen závislost výstupů na vstupech, ale také např. odezva, propustnost nebo celkový čas strávený v systému. To, jak simulační model odpovídá skutečnému zařízení, tedy závisí na jeho účelu: např. pokud by cílem hypotetického simulátoru arch. Armv8 mělo být pouze sledování změn obecných registrů v op. módu *User*, takový simulátor by nemusel obsahovat žádné informace o systémových registrech nebo SIMD/FP registrech.

Dá se tedy říct, že termín simulace je volnější než emulace a v jistém směru lze emulaci chápat jako konkrétní případ simulace. Tato práce popisuje implementaci simulátoru instrukční sady A32, jehož cílem je zkoumání chování jednotlivých instrukcí, co se týče jejich vlivu na stav procesoru. Využívá k tomu ale prostředky emulace: použitá knihovna se snaží přesně replikovat chování procesoru implementujícího architekturu Armv8. Hranice mezi těmito dvěma termíny však není v kontextu práce úplně ostrá.

2.4 Stávající nástroje pro simulaci činnosti procesorů Arm

Vzhledem k pozici, jakou procesory Arm na trhu zastávají, není překvapivé, že možnosti jejich emulace či simulace jsou rozsáhlé. Dostupná jsou jak komerční řešení přímo od společnosti Arm nebo některých výrobců čipů, tak různá otevřená, volně dostupná řešení. Nabízené simulátory a emulátory se přirozeně liší funkcemi a zaměřením, některé emulují

skutečné čipy, některé simulují pouze omezenou množinu instrukcí a podobně. Tato kapitola představuje několik řešení, která se zaměřují, podobně jako tato práce, na simulaci procesorů Arm pro výukové účely. V závěru naznačuje i další přístupy, které mají své využití spíše v praxi.

2.4.1 Výukové simulátory

Výukový simulátor nebo emulátor procesoru Arm by měl začínajícím programátorům maximálně napomoci při pochopení základních principů práce s jeho JSA. Poskytovat by měl přívětivý editor se zvýrazňováním a kontrolou syntaxe, vestavěnou dokumentací jednotlivých instrukcí a případně dalšími možnostmi, například vizualizací skoků. Při běhu programu pak musí umožnit minimálně krokování s nahlížením na stav registrů a paměti. Žádoucí je také podpora nějaké formy uživatelského vstupu a výstupu pro běžící program – např. simulací systémových volání `read` a `write` nebo simulací volání některých funkcí standardní knihovny jazyka C.

VisUAL

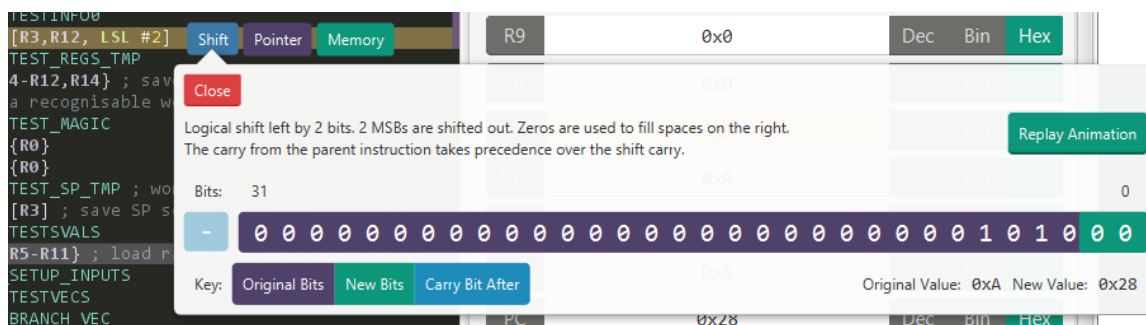
[VisUAL](https://salmanarif.bitbucket.io/visual/index.html)⁵ je výukový simulátor, který vznikl pro potřeby výuky na Imperial College London. Nabízí základní simulaci omezené podmnožiny instrukcí ze sady A32 a několik zajímavých didaktických funkcí:

- procházení historie hodnot registrů na jednotlivých řádcích kódu,
- vizualizaci ukazatelů (vysvětlivky pro pre-/post-indexové adresování),
- vizualizaci funkce jednotky pro posuvy a rotace (příklad je na obrázku 2.4),
- vizualizaci přístupů do paměti,
- vizualizaci zásobníku,
- vizualizaci skoků,
- kontrolu syntaxe,
- detekci nekonečných smyček,
- zobrazení seznamu symbolů,
- zobrazení jednoduchého popisu instrukcí.

Simulátor má jednoduché uživatelské rozhraní, které sestává z editoru kódu a z panelu s obsahem registrů. Zobrazit se dají také okna s obsahem simulované paměti a s přehledem symbolů. Editor není nijak rozšiřitelný, je pouze možné měnit barevné schéma a velikost písma. Okno editoru je interaktivní, pomocí podbarvování vizualizuje například skoky, breakpointy nebo aktuálně prováděný řádek. Není možné mít otevřených více souborů současně. Ukázka prostředí simulátoru je na obrázku 2.5.

Za hlavní nevýhodu simulátoru VisUAL považuji omezenost dostupných instrukcí. Simulátor podporuje většinu běžných instrukcí (aritmetické, logické atp.), z těch základních ale chybí například `MUL` pro násobení. Instrukce pro práci s čísly s pohyblivou řádovou

⁵Webová stránka projektu VisUAL: <https://salmanarif.bitbucket.io/visual/index.html>.



Obrázek 2.4: Simulátor VisUAL poskytuje vizualizaci funkce jednotky pro posuvy a rotace.

čárkou nebo SIMD instrukce nejsou podporovány vůbec. Simulátor také neumožňuje jakoukoliv interakci uživatele s programem pomocí vstupů a výstupů, podporována nejsou ani systémová volání nebo přerušování. Zobrazování dokumentace k instrukcím je značně jednoduché, neumožňuje náhled na kódování instrukce, neposkytuje podrobné informace o funkci instrukce a nebere v potaz použité podmínkové kódy.

Simulátor je implementován v jazyce Java. K dispozici jsou pouze binární balíčky pro Windows a macOS, nabízený je také instalační balíček pro Ubuntu. Zdrojový kód navíc není dostupný, spuštění simulátoru na různých počítačích je tedy problematické. Poslední vydání tohoto simulátoru je z prosince 2015, projekt není aktivně udržován. Při svém testování jsem přitom narazil na různé chyby vizuálního rázu (problikávající části okna, špatné pozicování vyskakovacích prvků), které znepříjemňovaly práci s nástrojem.

Existuje také nástupce simulátoru, [VisUAL 2](https://tomcl.github.io/visual2.github.io/)⁶. Jde o reimplementaci tohoto simulátoru s využitím jiných technologií (Electron, F#). K této verzi simulátoru jsou veřejně dostupné zdrojové kódy. Z didaktického hlediska bohužel tento projekt nepřináší oproti první verzi žádná zlepšení, instrukční sada je stále stejně omezená, naopak některé funkce chybí (dokumentace instrukcí, zvýrazňování skoků v kódu). Poslední verze byla publikována v prosinci 2019, v současnosti projekt nevykazuje žádnou aktivitu. Distribuované spustitelné soubory nejsou spolehlivé (na mém počítači s linuxovým systémem nebylo možné aplikaci spustit), překlad ze zdrojových kódů je netriviální.

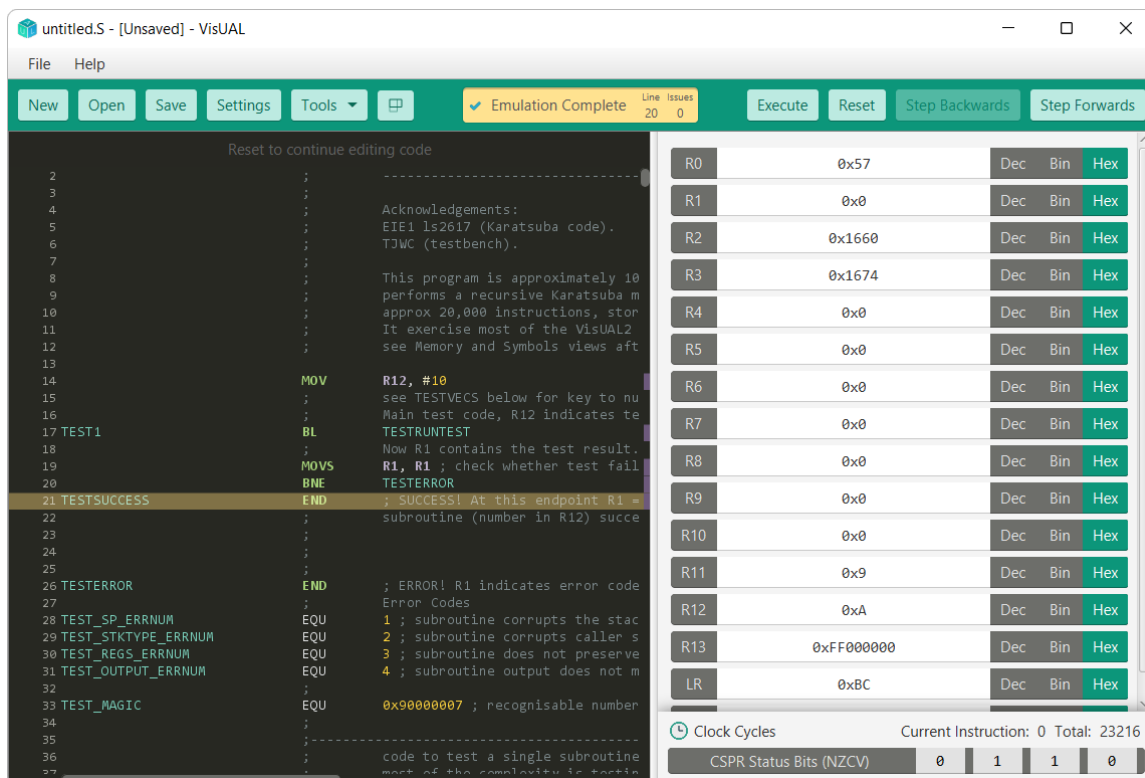
ARMStrong

Výukový simulátor [ARMStrong](https://github.com/linuxis9/ARMStrong)⁷ vznikl na univerzitě IUT de Valence. Podobně jako simulátor VisUAL byl také implementován v jazyce Java, v tomto případě však neobsahuje vlastní simulační kód, ale využívá jádro Unicorn (viz kap. 2.5).

Kromě zobrazování obsahu registrů a paměti nabízí tento simulátor také možnosti uživatelského vstupu a výstupu. Jednou z možností je využití simulovaných funkcí, které jsou zavolány při skoku na předem určenou adresu v paměti. Tímto způsobem je možné intergovat s uživatelem pomocí vestavěné konzole (výstup do konzole je vidět na obrázku 2.6) nebo ukládat a načítat obsah ze souborů. Podobně nabízí nástroj také funkce pro pozastavení programu a generování náhodných čísel. Druhou z možností je použití simulovaných spínačů, LED a osmisegmentových displejů, které jsou mapovány na adresy v paměti (také demonstrováno na obrázku 2.6).

⁶Webová stránka projektu VisUAL 2: <https://tomcl.github.io/visual2.github.io/>.

⁷Webová stránka (repozitář) projektu ARMStrong: <https://github.com/linuxis9/ARMStrong>.



Obrázek 2.5: Ukázka prostředí simulátoru VisUAL.

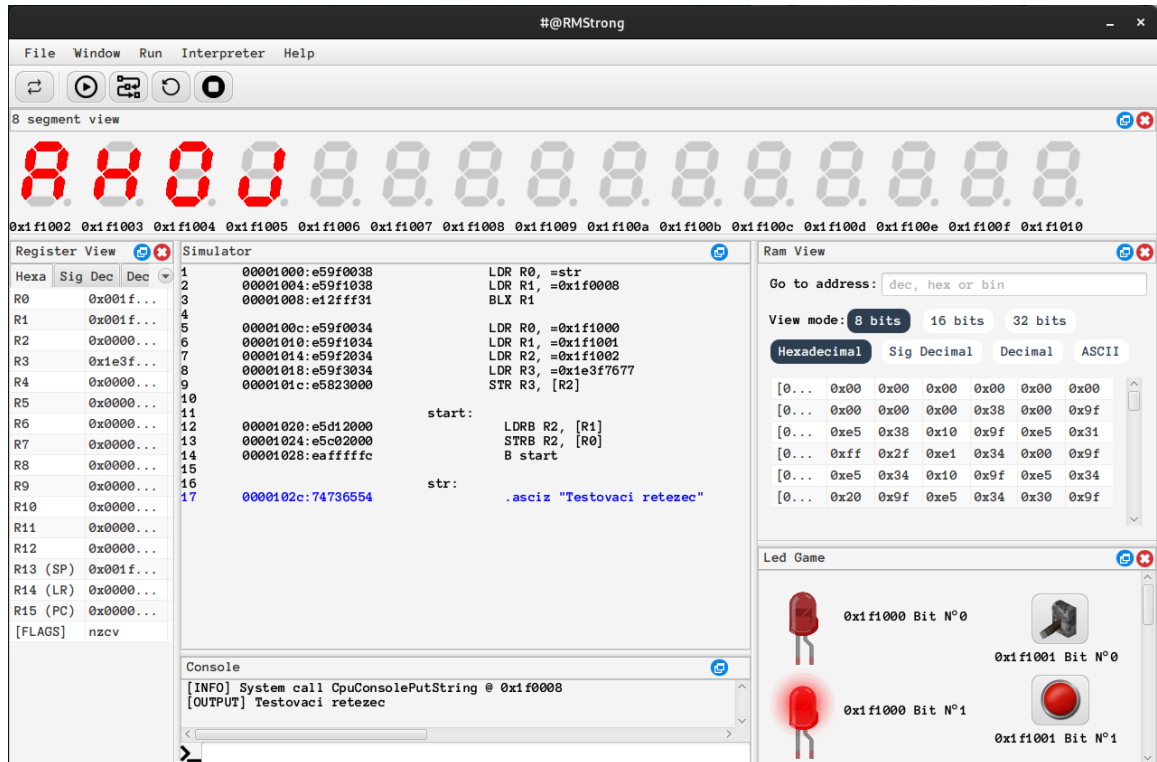
Protože simulátor využívá jádro Unicorn, nabízí teoreticky úplnou implementaci instrukční sady A32. Z hlediska podpory instrukcí je tedy výrazně vhodnějším nástrojem než VisUAL. Vestavěný překladač však není schopen některé instrukce zakódovat (např. SDIV⁸) a nástroj není ideální pro výuku použití SIMD a FP instrukcí, protože neumožňuje jednoduchý náhled na stav příslušných registrů (SIMD/FP jednotka navíc není při spuštění simulace povolena).

Nástroj nabízí také funkci interpretu, kdy je možné v reálném čase zadávat a spouštět instrukce. Tento režim je však od kódu v editoru zcela oddělený. Praktičtější by tato možnost byla, kdyby se daly takto dynamicky vkládat instrukce v režimu běhu programu, když je zrovna provádění pozastaveno.

Kromě uvedených funkcí simulátor ARMStrong nenabízí žádné další didaktické pomůcky při psaní a spouštění kódu. Vestavěný editor je pouze jednoduché textové pole bez dalších schopností (dokumentace k instrukcím, zvýrazňování syntaxe, chyb, skoků atp.), při spouštění je možné programem krokovat, není možné ale dopředu nastavit místa pro přerušení provádění (*breakpoints*). Uživatelské prostředí simulátoru je prosté a ne vždy úplně přehledné a odladěné (např. v panelu pro zobrazení paměti nejsou při jistém nastavení šířky vidět celé adresy jednotlivých buněk, což činí celý panel nepoužitelným; v okně s virtuálními přepínači se zobrazují špatné informace o pořadí bitů apod.).

⁸Instrukci SDIV jádro Unicorn použité v nástroji podporuje, což jsem ověřil manuálním zavedením zakódované instrukce do paměti a spuštěním simulace, která ji korektně provedla. Vestavěný překladač při pokusu o její použití ovšem vypíše chybu, prakticky tedy není při vývoji použitelná. Podobné chování nastává i s některými SIMD instrukcemi.

Použití simulátoru ARMStrong znesnadňuje také praktická neexistence dokumentace k jeho funkcím, které nejsou na první pohled zřejmé. Příkladem jsou zmíněné možnosti simulace systémových volání a vestavěných funkcí, které jsou zmíněny pouze jako součást příkladů v repozitáři projektu, konkrétní způsob jejich použití a fungování je však nutné vyčíst ze zdrojových kódů simulátoru.



Obrázek 2.6: Simulátor nabízí vizuální odezvu pomocí virtuálních diod LED, přepínačů a osmisegmentových displejů. Ve spodní části okna se nachází konzole, ve které je vidět řetězec vypsaný kódem programu.

CPUlator

Simulátor [CPUlator](https://cpulator.01xz.net/)⁹ je dalším nástrojem vhodným pro experimentování s programováním na strojové úrovni. Je implementován jako webová aplikace běžící v prohlížeči. Umožňuje simulovat podmnožinu instrukční sady Armv7 (simuluje jádro Cortex-A9) a také další instrukční sady (Nios II, MIPS32). Simulátor nepodporuje SIMD instrukce, podporovány však jsou FP instrukce.

CPUlator je zajímavý zejména svými rozsáhlými možnostmi simulace uživatelského vstupu a výstupu. V režimu simulace konkrétní vývojové desky (Altera DE1-SoC) poskytuje mnoho simulovaných periférií, mezi které patří například LED diody, přepínače, tlačítka, časovače, VGA obrazovka, audio (DAC převodník) nebo UART.

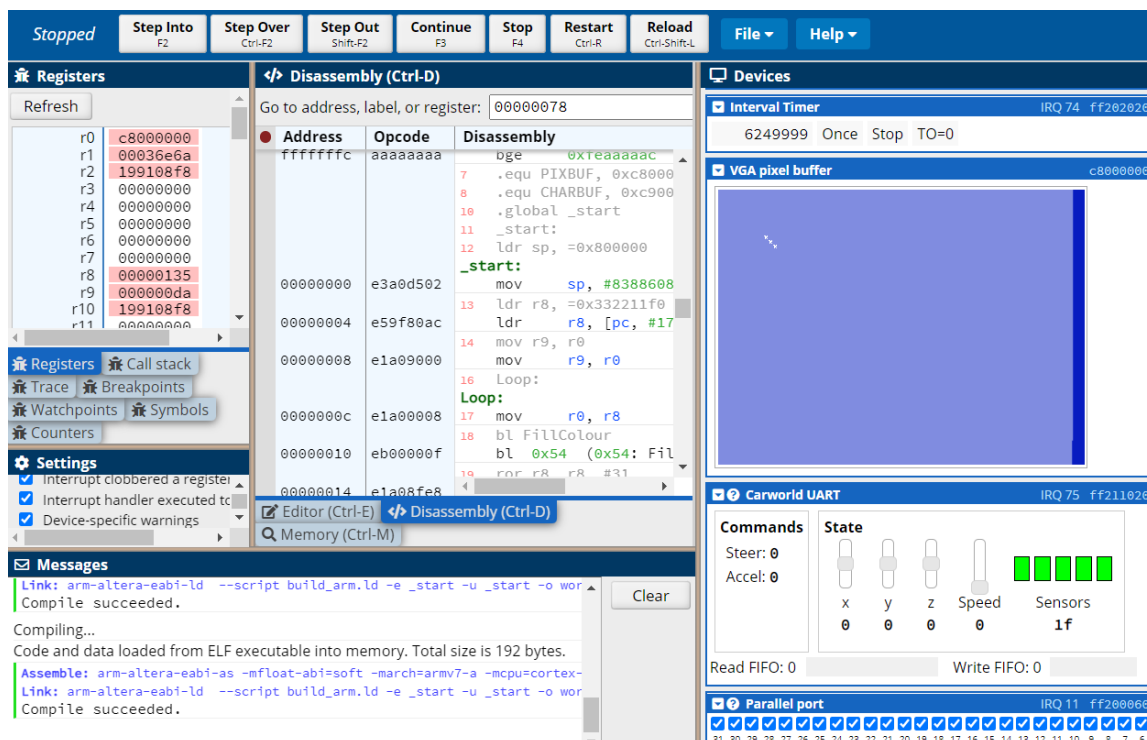
Na rozdíl od dříve zmíněných simulátorů využívá CPUlator pro překlad zdrojového kódu na serveru spouštěný překladač GNU Assembler bez jakýchkoliv modifikací. Vstupem simulačního jádra jsou zde běžné binární soubory ve formátu ELF. V simulátoru je tedy

⁹Webová aplikace CPUlator: <https://cpulator.01xz.net/>.

možné spouštět už zkompilevané binární soubory z jiných zdrojů. Při běhu je pak prováděný kód zobrazován v integrovaném disassembleru.

Co se editoru kódu týče, CPULATOR nabízí některé z požadovaných vlastností, chybí ale například detailní popis instrukcí a jejich kódování nebo vizualizace skoků. Našeptávání je jen základní. Ladění je naopak propracované, umožňuje aktivaci různých kontrol, vkládání breakpointů, krokování, sledování volání funkcí (*call stack*) a sledování změn na konkrétních místech v paměti (ale ne v registrech).

K simulátoru je dostupná vcelku obsáhlá dokumentace¹⁰, která však popisuje spíše jeho architekturu a možnosti. Chybí v ní například konkrétní popis způsobu použití virtuálních periférií, který je nutné hledat v externích materiálech (to však není zcela překvapivé vzhledem k tomu, že simulátor napodobuje reálná zařízení).



Obrázek 2.7: Ukázka rozhraní simulátoru CPULATOR. Simulátor nabízí různé virtuální periferie, například VGA displej, časovače nebo senzory komunikující přes UART.

2.4.2 Komerční řešení

Pro ladění programů na reálných i simulovaných procesorech Arm je dostupná celá řada komerčních řešení, která jsou využívána v praxi. Typicky jde o balíky nástrojů, které nabízí výrobci čipů a systémů na bázi Arm nebo společnost Arm samotná. Jedním z nich je např. Keil MDK, což je řešení umožňující vývoj pro Cortex-M mikrokontroléry. Jeho součástí je vývojové prostředí, překladače, linker a další nástroje. Součástí vývojového prostředí je i simulátor instrukcí, případně je možné využít externí balíčky, tzv. *Arm Fixed Virtual Platforms*, které poskytují úplnou simulaci procesoru, paměti i periférií.

Tato řešení ovšem nejsou pro začátečnické experimenty a seznámení s architekturou příliš vhodná, a to zejména kvůli své komplexitě. Nastavení prostředí není triviální, neboť

¹⁰Dokumentace k simulátoru CPULATOR je dostupná na adrese <https://cpulator.01xz.net/doc/>.

už počítá s hlubokými znalostmi uživatele. Čistě didaktické funkce zde není možné očekávat. Konkrétně v případě Keil MDK je omezující také závislost na operačním systému Windows.

2.4.3 Emulace na systémové úrovni

S vývojem pro architektury Arm je přirozeně možné se seznámit i s pomocí nástrojů, které se vyžívají v praxi, a spouštěním programu i ladicích nástrojů přímo na systému, který je na arch. Arm založen. Vzhledem k jejich současnému zastoupení v oblasti osobních počítačů je však spíš nepravděpodobné, že by vývojář disponoval přímo stanicí s procesorem arch. Arm – v takovém případě ovšem může využít systémovou emulaci.

Systémové emulátory poskytují model celého počítače, tedy emulují procesor, paměť i periferie. To umožňuje emulovat celý operační systém, který se pak chová, jako by byl spuštěn na skutečném počítači. Vývojář tedy může v prostředí emulovaného systému provádět jak vývoj, tak překlad, spuštění a ladění programu. To by však bylo poněkud naivní, neboť emulovaný systém bude zákonitě pomalejší a vyvíjení přímo v něm by nebylo příliš komfortní. Praktičtější je tedy využití technik křížového překladu (generování strojového kódu jiného procesoru, než na kterém je spuštěn překladač) a vzdáleného ladění (schopnost ladicího nástroje poskytnout své rozhraní vzdálené stanici).

Jedním z dostupných nástrojů pro systémovou emulaci je QEMU. Jde o vyspělý, multiplatformní a široce konfigurovatelný emulátor (a virtualizační nástroj), který umožňuje vcelku efektivní emulaci celé řady architektur a na nich založených systémů. Nabízí emulaci procesoru, grafického výstupu, síťových rozhraní, USB sběrnic a dalších periférií. Pro emulaci jsou dostupné profily reflektující mnohá reálná zařízení (např. konkrétní vývojové desky).

Kromě systémové emulace umožňuje QEMU také *user-mode* emulaci, při které není emulován celý systém, ale pouze jeden uživatelský program, jehož systémová volání jsou překládána na volání hostitelského systému. Tento režim je však podporován, pouze pokud je hostitelským systémem GNU/Linux (případně BSD).

Použití QEMU pro základní studium nízkourovňového vývoje pro arch. Arm má různá úskalí. Úspěšně zprovoznit vzdálené ladění je možné několika způsoby, všechny ale vyžadují instalaci dalších nástrojů a vcelku složitou konfiguraci, v některých nemusí fungovat správně odchytávání výjimek atp. V editoru Visual Studio Code je možné připojovat se ke vzdálené instanci GDB, ale vyžaduje to nestandardní konfiguraci na straně laděného programu i editoru.

Z didaktického hlediska není tento přístup ideální, protože z principu žádné didaktické pomůcky neposkytuje. Student je tedy odkázán na programování v editoru bez jazykové podpory. Ladění s využitím GDB je např. v editoru Visual Studio Code podporováno vcelku dobře a srozumitelně, ale i přesto vyžaduje znalosti nástroje GDB, které nelze předpokládat. Výhodou tohoto přístupu ale může být velmi přesné napodobení chování skutečných zařízení a dostupnost prakticky všech možností instrukční sady Armv8.

2.5 Emulační jádro Unicorn

Unicorn je framework pro emulaci procesoru, který umožňuje emulaci samostatných instrukcí i celých programů, sledování jejich chodu a stavu emulovaného procesoru. Nabízí k tomu programátorovi rozsáhlé možnosti **instrumentace**¹¹: umožňuje volat uživatelem definované obslužné rutiny¹² po provedení instrukce, při přístupu do paměti nebo při zachycení výjimky či přerušení. Zároveň umožňuje v těchto rutinách během emulace dynamicky měnit obsah paměti nebo registrů [26]. Unicorn je otevřený software dostupný pod licencí GNU GPL verze 2.

Mezi klíčové vlastnosti frameworku Unicorn patří podpora emulace mnoha architektur (kromě Arm ve 32b i 64b režimu také např. x86 v 16b, 32b i 64b režimu, MIPS nebo RISC-V) i podpora mnoha hostitelských architektur a operačních systémů. Framework je implementován jako knihovna v jazyce C, která poskytuje jednoduché API. To je jednotné pro všechny emulované architektury. Existuje také celá řada knihoven v jiných jazycích (například Python, Rust, Java nebo C#), které zprostředkovávají funkce frameworku Unicorn¹³.

Základem frameworku Unicorn je emulátor QEMU, který je představen v kapitole 2.4.3. Zatímco QEMU poskytuje systémovou emulaci, Unicorn se zaměřuje čistě na emulaci procesoru. Unicorn je tedy v zásadě komponenta QEMU zajišťující emulaci procesoru, která je očištěná od vazeb na jiné části QEMU, upravená a doplněná o možnosti instrumentace, které QEMU postrádá, a jednoduché API pro její využití a manipulaci se stavem procesoru [25]. Unicorn zároveň těží ze skutečnosti, že je QEMU aktivně vyvíjený a široce používaný nástroj, a díky tomu nabízí široké množství cílových architektur s dobrým pokrytím instrukčních sad. Autoři frameworku poukazují také na jeho inherentní bezpečnost: ačkoliv se historicky v QEMU objevilo velké množství bezpečnostních chyb, prakticky žádné z nich se netýkaly samotné komponenty emulující procesor. Protože Unicorn používá pouze ji, má mnohem menší prostor k útoku (*attack surface*) [25]. Na druhou stranu, Unicorn je od QEMU ve svém kódu odchylen do takové míry, že není možné jej jednoduše udržovat synchronizovaný s kódem QEMU, opravy a nové funkce se do kódu frameworku tedy dostávají spíše sporadicky (aktuální verze vychází z QEMU verze 5.0.1, která vyšla v roce 2020).

Zdrojový kód 2.1 demonstruje použití API emulačního frameworku Unicorn pro zavedení a spuštění jednoduchého programu pro architekturu Arm. Na ukázce je vidět, že kompletní emulaci procesoru Arm je možné provést pomocí čtyř volání funkcí.

Nevýhodou frameworku Unicorn je nedostatek dokumentace. Webové stránky projektu obsahují pouze základní popis a v jeho repozitáři je dostupné stručné FAQ. Samotné API frameworku je přímo v hlavičkovém souboru zdokumentováno poměrně rozsáhle, chybí v něm ale upozornění na neočekávané chování v některých situacích. Chybí také popisy specifik jednotlivých architektur.

2.6 Jazyk symbolických adres a jeho překlad

Podstatou činnosti procesoru je, značně zjednodušeně, čtení binárních číslic z paměti a vykonávání operací, které různé posloupnosti těchto číslic reprezentují. Taková posloupnost je

¹¹Instrumentace zjednodušeně označuje vkládání speciálního kódu do laděného programu za účelem sledování (a případně ovlivnění) jeho běhu. Instrumentace může být prováděna v různých stádiích životního cyklu programu (při jeho tvorbě, při překladu, za běhu) [18].

¹²Tyto rutiny jsou v terminologii projektu Unicorn označovány termínem *hook* [26].

¹³Tyto knihovny jsou v terminologii projektu Unicorn označovány termínem *binding* [26].

```

1  uint32_t data[] = {
2      0xe3a00c11, // MOV R0, 0x1100
3      0xe3a0100c, // MOV R1, 0xc
4      0xe5801000  // STR R1, [R0]
5  };
6  uc_engine *uc;
7  uint32_t value;
8
9  uc_open      (UC_ARCH_ARM, UC_MODE_ARM, &uc);          // Vytvoření instance
10 //          (uc, adresa, velikost, oprávnění)
11 uc_mem_map   (uc, 0x1000, 4096, UC_PROT_ALL);          // Mapování emul. paměti
12 //          (uc, adresa, data, délka dat)
13 uc_mem_write(uc, 0x1000, data, 12);                    // Zápis do emul. paměti
14 //          (uc, adresa, max. adresa, timeout, počet instrukcí)
15 uc_emu_start(uc, 0x1000, 0, 0, 3); // Spuštění
16 //          (uc, adresa, cíl, délka dat)
17 uc_mem_read (uc, 0x1100, &value, 4);                    // Čtení z emul. paměti
18
19 assert(value == 0xc);

```

Zdrojový kód 2.1: Ukázka použití API frameworku Unicorn (s vynecháním kontroly chyb). Emulovaný program zapisuje do své paměti hodnotu 0xc, kterou pak může hostitelský program číst.

označena jako program ve **strojovém kódu**. Jedna konkrétní sekvence binárních číslic strojového kódu, kterou procesor vykoná přesně definovaným způsobem, se nazývá **instrukce**. Instrukce se typicky skládá z kódu instrukce (*opcode*) a z dalších číslic, **operandů**, které specifikují parametry dané operace [31, kap. 2].

Programování přímo ve strojovém kódu, tj. zapisováním binárních číslic, by pochopitelně bylo značně nepraktické. Pro vývoj na strojové úrovni se tak využívá **jazyk symbolických adres**¹⁴ (JSA). Jak název napovídá, tento jazyk zavádí symbolická – textová – označení zejména pro instrukce (mnemoniky), adresy jednotlivých částí kódu a dat (návěští) a další součásti procesoru, se kterými program pracuje (zejména registry) [31, kap. 2].

Mnemonika (*mnemonic*) je symbolické jméno označující jednu nebo více instrukcí. Například operace sčítání dvou registrů a sčítání registru s konstantou jsou popsány odlišným strojovým kódem, jde o dvě různé instrukce. Ty jsou ale v JSA zapisovány jednou mnemonikou (v tomto případě ADD), přičemž správné kódování (správnou instrukci) vybere překladač podle typů použitých operandů. [3, kap. C1.2.3] Termíny se ovšem v běžné řeči zaměňují a slovem instrukce se v kontextu JSA často myslí právě mnemoniky.

Symbol je v kontextu překladačů a objektových souborů jakýkoliv název, kterému programátor (resp. překladač) přiřadí nějakou hodnotou. Tou je obvykle adresa v paměti, překladač ale může umožnit ruční přiřazování libovolných hodnot. **Návěští** (*label*) je syntaktický prostředek, jehož sémantikou je deklarace symbolu označujícího adresu v paměti [10, kap. 5]. Pojmy návěští a symbol se někdy zaměňují.

¹⁴V češtině se používá také termín **jazyk symbolických instrukcí** (JSI). Anglickým ekvivalentem je *assembly language*. JSA se často, zejména v češtině, nesprávně označuje jako *assembler*, ačkoliv tento termín označuje především překladač JSA do strojového kódu.

2.6.1 Překlad jazyka symbolických adres

O JSA se typicky mluví jako o nejvíce nízkourovňových programovacích jazycích (většina příkazů v JSA typicky přímo odpovídá jedné konkrétní instrukci z cílové instrukční sady, resp. architektury). Překladač JSA¹⁵ i přesto zajišťuje minimálně:

- lexikální a syntaktickou analýzu,
- výběr instrukce, která odpovídá mnemonice a použitým operandům,
- zaznamenání definovaných návěstí,
- výpočet absolutních nebo relativních adres na místech, kde jsou použita návěstí,
- uložení instrukcí ve správném kódování.

Při překladu programu pro hypotetický jednoduchý počítač s instrukční sadou umožňující použití absolutních adres, který spouští pouze jeden program uložený v paměti na přesně daném a neměnném místě, budou výpočty adres odpovídajících symbolickým jménům v kódu triviální a výstupem překladače by mohl být přímo spustitelný kód. Počítače s operačními systémy však umožňují do paměti zavádět mnoho programů najednou a na různá místa, což proces komplikuje, protože v době překladu není známo, jaká konkrétní adresa v paměti bude symbolům odpovídat. Pro řešení těchto situací se využívají mimo jiné techniky relativního adresování a relokací kódu, které jsou zjednodušeně vysvětleny dále. Konkrétní detaily těchto technik a celého procesu linkování jsou závislé na cílové platformě a celá problematika je výrazně složitější a mimo rozsah této práce.

Relativní adresy

```
0x0    LDR R0, label
0x4    label: .word 512
```

Tento fragment kódu do registru R0 načítá hodnotu z místa v paměti, na které ukazuje symbol (návěští) `label`. Při překladu pro jednoduchý počítač z předchozího odstavce by mohl překladač do první instrukce doplnit jako operand přímo absolutní adresu `0x4`. Pokud by ale program byl zaveden do jiného místa v paměti, než na adresu `0x0`, takto uložená instrukce by nedávala smysl. V případě architektury Arm to ani není možné: instrukce kvůli omezením na délku neumožňuje zakódovat celou 32b paměťovou adresu.

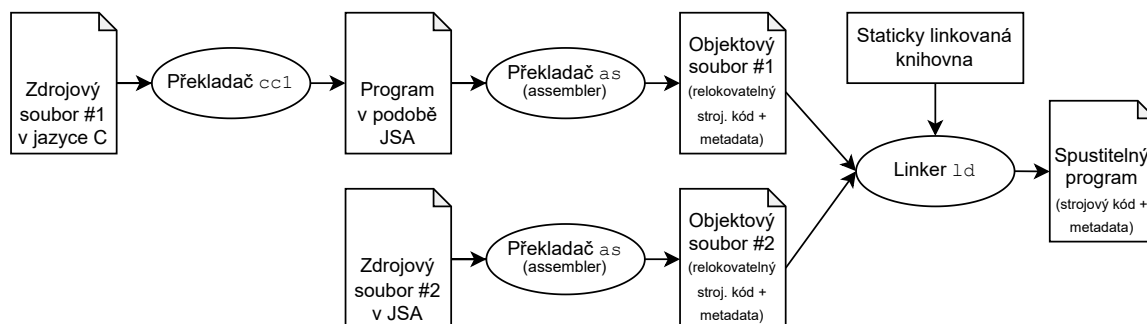
V tomto případě je možné využít **PC-relativní adresování**. Překladač v místě užití symbolu `label` spočítá rozdíl mezi adresou instrukce (resp. hodnotou programového čítače, se kterou bude procesor při vykonávání instrukce počítat a která vlivem řetězeného zpracování nebude nutně odpovídat adrese instrukce) a hodnotou tohoto symbolu (místem v paměti, na které ukazuje příslušné návěští). Tato instrukce je pak zakódována jako `LDR R0, [PC, #-4]`. Při vykonávání tedy bude čtení provedeno ze správného místa neohledně na to, na jaké adrese kód začíná.

¹⁵Pro překlad JSA se v angličtině užívá sloveso *to assemble* (doslova „sestavit, smontovat, shromáždit“). Z tohoto slovesa je odvozen také angl. termín pro překladač JSA: *assembler* („sestavovač“). Pro překlad vyšších jazyků se v angličtině používá sloveso *to compile*. V obou případech jde ale především o převádění jisté formy zápisu programu do jiné (v případě JSA do strojového kódu), proto se v této práci využívá pro oba procesy shodný český termín **překlad** (resp. **překladač**).

Relokace

Princip **relokací** se využívá (mimo jiné) kvůli možnostem využití externího kódu (knihoven) a segmentaci. Použití externího kódu v zásadě znamená, že je v kódu použita instrukce (např. `skok`), jejímž operandem je v tom kódu nedefinovaný symbol. Jeho hodnota (adresa cizího kódu) je doplněna až později, po přeložení. Překladač proto musí ke kódu samotnému doplnit metadata obsahující polohy takových instrukcí. Takový výstup překladače se nazývá **objektový soubor** [29].

Objektové soubory nelze přímo spustit, jsou určeny pro zpracování **linkerem**. To je program, který (mimo jiné) spojuje kód z jednotlivých objektových souborů a provádí relokace (doplňuje adresy). Teprve jeho výstupem je spustitelný soubor, což je v zásadě strojový kód opatřený metadaty (informacemi o symbolech, segmentech apod.) [29].



Obrázek 2.8: Princip překladač programu využívajícího více zdrojových souborů a knihovnu.

Spustitelné programy bývají z různých důvodů často rozděleny na více **segmentů**, typicky alespoň na dva: segment se spustitelným kódem a segment s nespustitelnými daty. Jeden objektový soubor může být rozdělen na **sekce** (*sections*). Linker přeskládává sekce jednotlivých vstupních souborů, spojuje jejich obsahy a výsledek uloží do vhodného segmentu. Ve výsledném spustitelném souboru jsou tedy části kódu z různých modulů uloženy v různém pořadí a na různých místech. To je další důvod, proč musí překladače vytvářet relokovatelný kód [10, kap. 4].

Následující fragment kódu povede na vygenerování objektového souboru s relokacemi:

```
.text
0x0    LDR R0, =label2
0x4    LDR R0, [R0]

.data
0x0    label1: .word 23
0x4    label2: .word 512
```

Jsou zde definovány dvě sekce (`.text` a `.data`), obsah každé z nich začíná na logické adrese `0x0`. Ve výsledném spustitelném souboru může být kód těchto sekcí umístěn na jakýchkoliv adresách. Přeložený kód první sekce má tvar:

adr.	obsah	ekvivalent v~kódu
0x0	E51F0000	LDR R0, [PC, #0]
0x4	E5900000	LDR R0, [R0]
0x8	00000004	

Pro analýzu tabulky relokací v objektovém souboru je možné použít nástroj `objdump`:

```
RELOCATION RECORDS FOR [.text]:
OFFSET      TYPE          VALUE
00000008    R_ARM_ABS32     .data
```

Na konci sekce, na adrese `0x8`, překladač vygeneroval 32bitovou konstantu s hodnotou `0x4`. V tabulce relokací je poté záznam, který linkeru oznamuje, že položka na adrese `0x8` v sekci `.text` má být relokována do sekce `.data`. Linker tedy tuto konstantu nahradí adresou, kterou přidělil počátku sekce `.data` a ke které přičte původní obsah položky, tj. `0x4`. Konstanta – nyní obsahující skutečnou adresu symbolu `label2` – je pak z první instrukce adresována PC-relativním adresováním.

2.6.2 Překladače JSA procesorů Arm, the GNU Assembler

Vzhledem k rozšířenosti a flexibilitě platformy Arm není překvapivé, že existuje celá řada řešení pro překlad jejího JSA. Překladače bývají typicky součástí větších balíčků nástrojů, které obvykle zahrnují také překladač jazyka C/C++, linker, nástroje pro ladění a další podpůrné programy. Tyto balíky často sestavují a distribuují samotní výrobci procesorů a řešení postavených na platformě Arm, některé z nich jsou komerční¹⁶, různá řešení nabízí i Arm samotný¹⁷.

V této práci je využit překladač GNU Assembler, označovaný také jako GAS nebo `as`. Ten je součástí balíku GNU Binutils, který obsahuje nástroje pro překlad, linkování a analýzu binárních souborů s programovými daty. Další nástroj z tohoto balíku, překladač GCC, je nejpoužívanější nástroj pro překlad programů v jazyce C¹⁸. Primárním cílem překladače GAS je pak právě poskytnout překlad do strojového kódu pro výstup překladače GCC [10, kap. 1.2]. Jde o svobodný software s otevřenými zdrojovými kódy. Nabízí širokou podporu pro různé cílové platformy: překladač GAS umožňuje generovat strojový kód pro 48 různých architektur [10, kap. 1].

2.6.3 Formát ELF

Executable and Linkable Format (ELF) je standardní formát pro uložení objektových a spustitelných souborů, který využívají např. linuxové systémy. Soubor v tomto formátu je výstupem překladače GAS i linkeru `ld`. Skládá se v zásadě z ELF hlavičky, tabulky programových hlaviček (*program headers*) a tabulky hlaviček sekcí (*section headers*). ELF hlavička obsahuje základní informace, např. o bitové šířce, endianitě, cílovém binárním rozhraní systému (ABI) a cílové architektuře, typu souboru nebo vstupní adrese programu. Programové hlavičky slouží operačnímu systému jako instrukce pro mapování programu, který je v souboru uložen, do paměti. Hlavičky sekcí rozdělují obsah souboru na části různého významu (např. programová data, tabulka symbolů, informace o relokacích, ladicí informace apod.). Za hlavičkami následují samotná binární data jednotlivých sekcí (resp. segmentů) [33].

¹⁶Komerční nástroje pro vývoj na platformě Arm nabízí např. společnost IAR: <https://www.iar.com/products/architectures/arm/iar-embedded-workbench-for-arm/>.

¹⁷Arm nabízí sadu nástrojů *Keil* pro vývoj vestavných zařízení založených na jádrech Cortex profilu M: <https://www.keil.com/>.

¹⁸Podle průzkumu společnosti JetBrains z roku 2021 používá pravidelně překladač GCC pro překlad jazyka C až 82 % vývojářů (druhým nejpoužívanějším překladačem je Clang, který používá pravidelně asi 34 % vývojářů) [15].

2.6.4 Další možnosti překladačů

Kromě instrukcí k sestavení a návěští se v kódu v JSA často objevují i další příkazy, které nejsou přímým protějškem nějaké instrukce, ale jistým způsobem ovlivňují chování překladače a jeho výstup. Terminologie se liší napříč překladači a platformami, ale zhruba je možné je rozdělit na dvě kategorie: pseudoinstrukce (také *pseudo-opcodes*) a direktivy překladače.

Na rozdíl od mnemonik a dalších symbolů, jejichž syntaxe, kódování a sémantika jsou přesně specifikovány architekturou, jsou direktivy a pseudoinstrukce specifické pro konkrétní překladač. Mnoho dostupných a používaných překladačů však vychází ze syntaxe překladače GAS.

Direktivy překladače a makra

Direktivy různým způsobem ovlivňují proces překladu a obsah výstupního objektového souboru. Umožňují například určení sekce, do které budou následující instrukce uloženy (`.text`, `.data`, `.bss`), označují návěští jako globální (`.global`)¹⁹ nebo umožňují do výstupu zapsat konstantní hodnoty různých bitových šířek – tímto způsobem je možné v paměti např. vyhradit prostor pro proměnné. V syntaxi GAS jsou direktivy uvozeny tečkou [10, kap. 7].

Část direktiv, které GAS nabízí, slouží pro tvorbu maker. Makra jsou sady pravidel, která umožňují parametricky generovat kód. Makro definuje vstupy a operace vedoucí na jejich transformaci na požadovaný výstup a umožňují tak programátorovi nahradit opakující se části kódu. Makra v GAS umožňují rekurzivní volání a podmíněné vyhodnocování, jejich vyjadřovací možnosti jsou tak vcelku silné.

Pseudoinstrukce

Pseudoinstrukce jsou specifickým druhem direktivy překladače. Na rozdíl od ostatních direktiv nejsou uvozeny tečkou a jejich syntaxe odpovídá běžným mnemonikám. Jejich chování může být definováno překladačem nebo instrukční sadou. Překladač GAS pro architekturu Armv8 v režimu AArch32 definuje pseudoinstrukce LDR (pro načítání libovolných konstant do registrů) a ADRL²⁰ [10, kap. 9.4.5].

Pseudoinstrukce definované instrukční sadou jsou typicky aliasy jiné instrukce s rozdílnou sémantikou, ale stejnou činností. Příkladem je ADD `<registr>`, PC, `<konstanta>`, což je alias pro ADR `<registr>`, `<návěští>` – sémantikou mnemoniky ADD je sčítání obecně, sémantikou ADR je PC-relativní adresování návěští, v obou případech je ale provedeno přičtení konstanty k registru PC. Definování takovýchto aliasů má význam hlavně pro zajištění přehlednosti kódu a uživatelsky přívětivějšího výstupu disassemblingu²¹.

¹⁹Návěští označené v objektovém souboru jako globální je „viditelné“ pro linker, a tedy využitelné z jiných modulů.

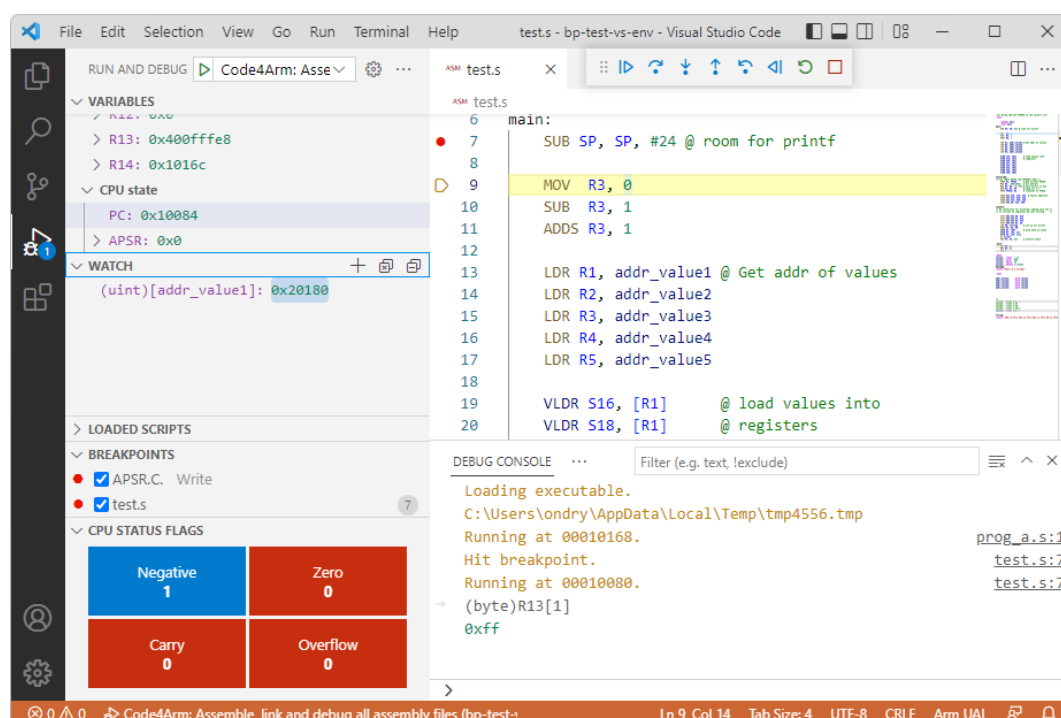
²⁰Dokumentace uvádí jako pseudoinstrukce také NOP a ADR [10, kap. 9.4.5], ty jsou ale v arch. Armv8 zahrnuty v in a ukční sadě [3, kap. F5.1.10, F5.1.126].

²¹Disassembling označuje zpětné generování JSA ze strojového kódu.

Kapitola 3

Editor Visual Studio Code a možnosti jeho rozšíření

Visual Studio Code (VSC) je rozšiřitelný editor zdrojového kódu. Jde o projekt s otevřeným zdrojovým kódem vyvíjený společností Microsoft. Jeho zdrojové kódy jsou dostupné pod licencí MIT¹. Je multiplatformní, dostupný je pro OS Windows, GNU/Linux a macOS. První stabilní verze vyšla v roce 2016 a od té doby se editor VSC stal jedním z nejpoužívanějších vývojových prostředí vůbec² [16, s. 1].



Obrázek 3.1: Prostředí editoru Visual Studio Code.

V základě jde o vcelku „hloupý“ textový editor, jehož síla spočívá v možnostech rozšíření: editor nabízí aplikační rozhraní, které umožňuje pro konkrétní programovací jazyk nebo platformu přidávat možnosti ladění a bohaté podpory při psaní kódu (např. zvýrazňování kódu, formátování, našeptávač a automatické doplňování jmen, refaktorování). Doplnky umožňují také upravovat vzhled editoru a přidávat do něj vlastní prvky uživatelského rozhraní.

¹Pod licencí MIT je nabízený volně dostupný zdrojový kód v repozitáři na adrese <https://github.com/microsoft/vscode>, který je označen názvem „Code – OSS“, přičemž názvem Visual Studio Code označuje Microsoft svou binární distribuci tohoto kódu, ve které jsou na rozdíl od OSS verze např. aktivované telemetrie a je dostupná pod licencí Microsoft Software: <https://code.visualstudio.com/License/>.

²O oblíbenosti editoru VSC svědčí také statistiky známého fóra Stack Overflow, kde v šetření z roku 2021 přes 70 % respondentů uvedlo, že tento editor v průběhu tohoto roku pravidelně používali [32].

Editor je postaven na platformě Electron, je tedy implementován s využitím webových technologií a jazyka TypeScript (jazyk navržený jako „typovaná nadmnožina“ JavaScriptu) [16, s. 12]. Tyto technologie jsou následně využity i při tvorbě rozšíření. S vhodnou kombinací rozšíření se tedy VSC může stát nástrojem s podobnými možnostmi, jaké nabízejí běžně využívaná integrovaná vývojová prostředí.

Editor VSC byl pro dosažení záměru této práce zvolen právě pro jeho rozšiřitelnost a širokou uživatelskou základnu. VSC pro přidávání jazykových možností využívá externí nástroje, se kterými komunikuje pomocí nezávislých rozhraní, tzv. *Language Server Protocol* a *Debug Adapter Protocol*, která jsou podrobněji popsána v kapitolách 3.2 a 3.3.

3.1 Možnosti rozšíření editoru a jejich architektura

Editor VSC je navržen způsobem, který umožňuje pomocí uživatelských rozšíření přidat nebo pozměnit většinu jeho chování. Rozšíření jsou programy v jazyce JavaScript, které jsou editorem obvykle³ spouštěny v běhovém prostředí Node.js a komunikují s editorem prostřednictvím rozsáhlého API. Dokumentace VSC [20] přibližně rozděluje možnosti rozšíření na následující kategorie:

- *Common Capabilities* (obecné možnosti): různé možnosti interakce uživatele a editoru samotného s rozšířením. Příkladem jsou např. registrace uživatelem spustitelných příkazů⁴, registrace klávesových zkratk, rozšiřování kontextových menu, zobrazování notifikací.
- *Declarative Language Features* (deklarativní rozšíření pro jazyky): umožňuje deklarativním způsobem (pomocí konfiguračních souborů) přidat základní podporu pro usnadnění editace kódu v nějakém jazyce. Tímto způsobem je možné pro jazyk přidat zvýrazňování (*syntax highlighting*), generování kódu podle šablony (*snippet completion*), automatické uzavírání párů závorek (*bracket autoclosing*, *autosurrounding*), členění kódu na bloky na základě párových značek (*folding*), vkládání komentářů (*comment toggling*) a odsazování. Tyto možnosti rozšíření jsou detailněji popsány v kapitole 3.2.
- *Programmatic Language Features* (programová rozšíření pro jazyky): umožňuje dynamické poskytování komplexních, kontextových informací o kódu. Mezi ty patří např. našeptávání a doplňování, navigace v kódu podle symbolů, zobrazování kontextových informací při označení symbolu (*hover information*), kontrola syntaktických a dalších chyb, formátování, sémantické členění na bloky. Tyto možnosti rozšíření jsou také detailněji popsány v kapitole 3.2.
- *Workbench Extensions* (rozšiřování UI): umožňuje přidávat prvky uživatelského rozhraní editoru (v dokumentaci označováno pojmem *workbench*). Přidávány mohou být

³VSC může být i s rozšířeními spuštěno také ve webovém prohlížeči, v takovém případě jsou rozšíření prováděna jeho běhovým prostředím *Web Worker API*. Specifika tohoto případu užití nejsou pro tuto práci důležitá. Další informace o principu spouštění rozšíření jsou dostupné v dokumentaci: <https://code.visualstudio.com/api/advanced-topics/extension-host>.

⁴Jako příkaz (*Command*) je v terminologii VSC označena pojmenovaná funkce poskytovaná editorem nebo rozšířením, která může být spuštěna ručně uživatelem, pomocí API editoru nebo pomocí rozšířením deklarovaného prvku rozhraní editoru (např. tlačítkem v nástrojové liště). Uživatel může všechny příkazy spouštět pomocí speciálního prvku rozhraní editoru, *Command Palette*.

jednoduché prvky (tlačítka, stavové indikátory v postranních lištách), stromové struktury (např. souborové stromy, stromy závislostí) nebo vlastní panely s libovolným obsahem definovaným pomocí HTML, CSS a JavaScriptu.

- *Debugging* (ladění): umožňuje tvořit rozšíření, která propojí editor s ladicími nástroji pro libovolný programovací jazyk. Rozhraní editoru VSC pro ladění a jeho rozšiřování je blíže popsáno v kapitole 3.3.
- *Theming*: umožňuje měnit vzhled editoru (barevná schémata UI a kódu).

Uvedené možnosti rozšíření jsou jen stručným přehledem, API editoru je skutečně komplexní a je podrobně popsáno v dokumentaci⁵.

3.1.1 Architektura rozšíření

Každé rozšíření editoru VSC se skládá minimálně ze dvou souborů: `package.json`, který definuje metadata rozšíření, a souboru se samotným kódem rozšíření v jazyce JavaScript, který exportuje alespoň funkci `activate`. S vývojem rozšíření souvisí kromě editorem poskytovaného API také koncepty *Activation Events* (aktivačních událostí) a *Contribution Points* (deklarací bodů rozšíření) [16, s. 147].

Aktivace rozšíření

Kód rozšíření není spuštěn hned při startu editoru, protože s větším množstvím rozšíření by to mohlo vést ke zbytečným prodlevám a nárokům na prostředky. Rozšíření jsou tak spuštěna – **aktivována** až v momentě, kdy uživatel provede nějakou akci, která je pro dané rozšíření relevantní. Rozšíření tedy ve svých metadatach musí definovat **aktivační události** [16, s. 148]. Aktivační událost je popsána jedním z předdefinovaných řetězců, který reprezentuje nějakou akci v editoru, např. spuštění uživatelského příkazu nebo přechod do režimu ladění. Jeho součástí může být parametr, který událost upřesňuje. Příklad sekce souboru `package.json` s definicemi aktivačních událostí je ukázán v kódu 3.1. Seznam všech možných aktivačních událostí je uveden v dokumentaci⁶.

```
"activationEvents": [  
  "onLanguage:asm-a32",  
  "onCommand:arm.showInstructionReference",  
  "onDebug"  
]
```

Zdrojový kód 3.1: Příklad definice aktivačních událostí pro rozšíření. Rozšíření bude aktivováno při otevření souboru, ve kterém bude detekován kód v jazyce `asm-a32`, při zavolání příkazu `arm.showInstructionReference` a při spuštění ladění.

Deklarace bodů rozšíření

Termínem *Contribution Points* (CP, body rozšíření) se označují deklarace v metadatach rozšíření, které poskytují editoru informaci o oblastech, jež jsou jím rozšiřovány [16, s. 155]. CP

⁵Dokumentace API editoru VSC: <https://code.visualstudio.com/api>.

⁶Dokumentace aktivačních událostí: <https://code.visualstudio.com/api/references/activation-events>.

jsou v souboru `package.json` definovány v sekci `contributes`. Jednotlivé CP nesou různé informace v závislosti na druhu rozšíření, který popisují. VSC poskytuje celou řadu CP, jejich úplný seznam je dostupný v [dokumentaci](#)⁷. V implementovaném řešení jsou využity například CP `commands` (pro deklaraci příkazů) nebo `languages` (pro deklaraci podpory nového programovacího jazyka).

3.2 Rozšiřování jazykové podpory editoru

VSC nabízí několik různých prostředků pro přidání podpory pro programovací jazyky. Tyto prostředky lze rozdělit do dvou kategorií: deklarativní (*declarative language features*) a programová rozšíření (*programmatic language features*).

3.2.1 Deklarativní rozšíření

Deklarativním způsobem lze přidat podporu základních funkcí pro usnadnění editace kódu a zvýšení jeho přehlednosti. Tento druh rozšiřování je omezen pouze na takové funkce editoru, které nevyžadují kontextovou analýzu kódu a vycházejí pouze ze syntaxe daného jazyka. Tímto způsobem je možné editoru poskytnout informace např. o znacích, které se typicky vyskytují v kódu v párech (závorky, uvozovky), nebo implementovat zvýrazňování klíčových slov. V kontextu JSA je možné využít deklarativní jazyková rozšíření např. pro zvýrazňování instrukcí, návěstí a dalších symbolů nebo pro doplňování hranatých závorek, které se využívají pro zápis pre-indexového adresování. Také je možné deklarativně definovat pravidla pro odsazování, podle kterých editor umožňuje automaticky formátovat kód, což se u JSA hodí opět při práci s návěstími. Pro deklarativní rozšiřování jazykové podpory je využit CP `languages` a doplňkový soubor s konfigurací jednotlivých funkcí. Příklad takového CP je uveden v kódu 3.2.

```
1 "contributes": {  
2   "languages": [  
3     {  
4       "id": "asm-a32",  
5       "extensions": [".s", ".asm"],  
6       "aliases": ["Arm Assembly", "asm"],  
7       "configuration": "./language-configuration.json",  
8       "filenames": ["asm-source"],  
9       "firstLine": "^\\\\.ASM-(A32|UAL)"  
10    }  
11  ]  
12 }
```

Zdrojový kód 3.2: Příklad deklarace *Contribution Point* pro jazyk. Na zvolený identifikátor `asm-a32` (řádek 4) mohou odkazovat aktivační události a jiné CP (viz např. kód 3.1). Editor může přiřazovat souborům jazyk podle přípony (řádek 5), celého jména souboru (řádek 8) nebo na základě prvního řádku odpovídajícího regulárnímu výrazu (řádek 9).

⁷Dokumentace bodů rozšíření: <https://code.visualstudio.com/api/references/contribution-points>.

3.2.2 Programová rozšíření

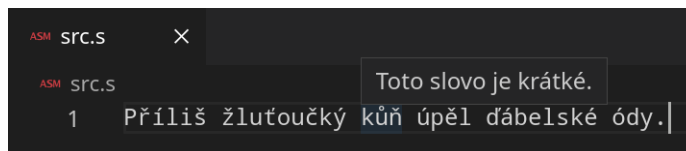
Programová rozšíření editoru umožňují poskytovat uživateli dynamickou zpětnou vazbu k aktuálnímu stavu editovaného kódu. Na rozdíl od deklarativních rozšíření, která pouze konfiguruji chování vestavěná v editoru, programová rozšíření vyžadují implementaci vlastní logiky. Ta v zásadě reaguje na určité události, které v editoru mohou nastat (např. změna obsahu, najetí myši nad symbol), provádí vlastní statickou analýzu aktuálního obsahu editovaného souboru a zasílá editoru vhodnou reakci (např. pokyn k tomu, že má být nějaký symbol v kódu označen jako chybný).

Pomocí programových rozšíření, v kombinaci s možností přidávat do UI editoru vlastní obsah, je možné implementovat skutečně komplexní podporu pro jazyk. VSC poskytuje 19 takto rozšiřitelných funkcionalit, jde o různé formy našeptávání, doplňování kódu, vyhledávání a přejmenovávání symbolů, navigace v kódu, formátování kódu. Programová rozšíření se mohou kombinovat s deklarativními a doplňovat je: například při zvýrazňování kódu mohou programová rozšíření o některých symbolech doplnit sémantické informace. Podrobný seznam možných programových rozšíření je dostupný v [dokumentaci](#)⁸.

Programová rozšíření je možné implementovat dvěma způsoby. Prvním z nich je využití API editoru přímo v kódu rozšíření, druhým je využití externí služby – jazykového serveru. Druhý způsob je popsán v kapitole [3.2.3](#).

Přímá implementace

Při své aktivaci může rozšíření zaregistrovat poskytovatele různých relevantních funkcionalit, které následně editor v odpovídajících situacích volá. Příklad takového rozšíření je uveden v kódu [3.3](#), ukázka jeho chování je pak na obrázku [3.2](#).



Obrázek 3.2: Informace zobrazená nad slovem po najetí kurzorem myši díky rozšíření implementovaném kódem [3.3](#).

3.2.3 Jazykové servery, Language Server Protocol

Alternativní možností, jak přidat do editoru dynamickou funkcionalitu pro editaci kódu, je využití tzv. **jazykového serveru** (*Language Server*). Jazykový server je oddělený program, který běží v samostatném procesu a přebírá zodpovědnost za analýzu kódu a vyhodnocování kontextových informací, které mají být uživateli v editoru zobrazeny. Editor, resp. rozšíření v něm, poté vystupuje jako klient, který se k serveru připojuje a komunikuje s ním pomocí protokolu *Language Server Protocol* (LSP). Typicky je proces jazykového serveru spuštěn rozšířením při jeho aktivaci, pro komunikaci mohou být využity různé prostředky meziprocesové komunikace [\[21\]](#).

Tento přístup má tři zásadní výhody. První je volnost při implementaci – jazykový server může být realizován v libovolném programovacím jazyce, stačí, aby umožňoval nějakou formu meziprocesové komunikace ve formátu daném protokolem LSP. Druhou výhodou je

⁸Dokumentace programových rozšíření je dostupná na adrese <https://code.visualstudio.com/api/language-extensions/programmable-language-features>.

```

1  class SimpleHoverProvider implements HoverProvider {
2      public provideHover(document: TextDocument, position: Position,
3                          token: CancellationToken): ProviderResult<Hover> {
4          const wordRange = document.getWordRangeAtPosition(position);
5          if (wordRange && document.getText(wordRange).length < 5) {
6              return new Hover("Toto slovo je krátké.");
7          }
8          return null;
9      }
10 }
11
12 // Registrace ve funkci activate(context: ExtensionContext)
13 context.subscriptions.push(vscode.languages.registerHoverProvider(
14     "asm-a32", new SimpleHoverProvider()));

```

Zdrojový kód 3.3: Příklad implementace jednoduchého programového rozšíření, které poskytuje informace po najetí kurzorem myši na slovo. Informace poskytuje třída `SimpleHoverProvider`, její instance je v editoru zaregistrována při aktivaci. Funkce `provideHover` je pak volána ve všech `asm-a32` souborech, když editor detekuje příslušnou akci. Neprázdná návratová hodnota je zobrazena v UI editoru, viz obr. 3.2.

oddělení logiky, která může být potenciálně značně náročná na výpočetní prostředky, z procesu editoru – neefektivní nebo příliš náročná implementace jazykové analýzy takto nemůže způsobovat problémy s výkonem editoru samotného. Třetí výhodou je znovupoužitelnost: oddělením logiky analýzy kódu od editoru a zavedením dobře popsané abstrakční vrstvy (kterou tvoří právě protokol LSP) je možné vcelku jednoduše použít jazykový server i pro jiné editory než VSC [21].

Language Server Protocol je protokol navržený společností Microsoft s cílem poskytnout jednotné rozhraní mezi editory⁹ a jazykovými servery, které je nezávislé na konkrétním nástroji nebo programovacím jazyce. Jeho myšlenkou je, že ačkoliv se vývojářské nástroje a editory liší, obecně poskytují podobné funkce – našeptávání, doplňování, vyhledávání definic symbolů a podobně – je tedy možné pro ně informace získávat jednotným způsobem. LSP standardizuje druhy objektů a zpráv, které si strany vyměňují pomocí jednoduchého protokolu pro vzdálené volání funkcí, JSON-RPC¹⁰ [19].

LSP vznikl jako podpůrný prostředek editoru VSC, proto přirozeně podporuje všechny funkce, které nabízí VSC ve svém API pro programová rozšíření. V současnosti jde však o samostatný projekt¹¹, který je nativně nebo pomocí rozšíření podporován celou řadou jiných editorů a IDE. Mezi ty patří např. Visual Studio, Eclipse IDE, Atom, Sublime Text, vim a neovim [23].

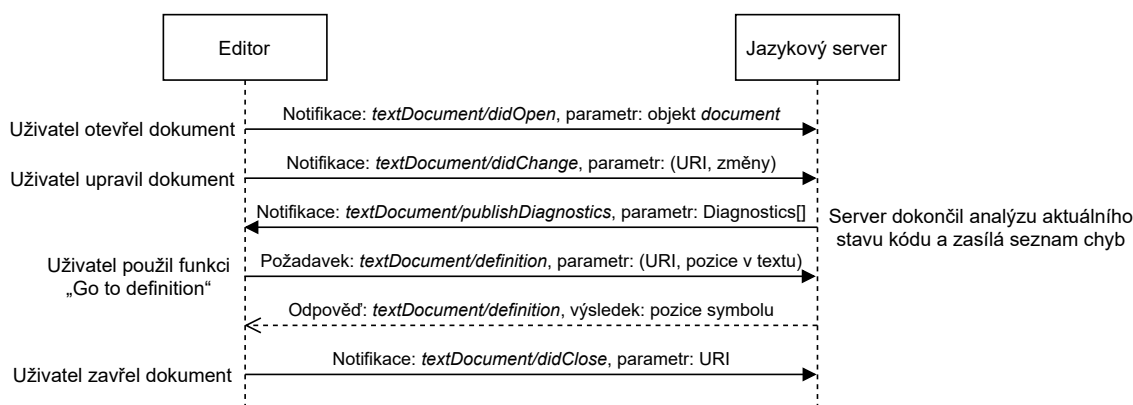
Na obrázku 3.3 je naznačen jednoduchý příklad komunikace mezi editorem a jazykovým serverem:

⁹Ve specifikaci LSP je použit obecnější termín *vývojové nástroje*.

¹⁰Informace o JSON-RPC jsou dostupné na adrese <https://www.jsonrpc.org/specification>. JSON-RPC je protokol určený k realizaci *vzdáleného volání procedur* (Remote Procedure Call, RPC), což jsou obecně techniky pro přenos řízení a dat do jiného programu, než je program volající [9].

¹¹Webová stránka projektu Language Server Protocol: <https://microsoft.github.io/language-server-protocol>. Specifikace protokolu je veřejně dostupná pod licencemi Creative Commons a MIT, do kterého je možné přispívat pomocí git repozitáře.

1. Uživatel otevře dokument daného formátu, editor se spojí se serverem a zašle zprávu (zavolá funkci) *textDocument/didOpen*, která nese URI otevřeného dokumentu, ID jazyka, označení verze dokumentu a jeho obsah.
2. Uživatel upraví otevřený dokument. Protože jsou tyto úpravy pravděpodobně provedeny pouze v kopii dokumentu v paměti editoru, je nutné synchronizovat je se serverem, aby mohl provádět analýzu nad aktuálním stavem dokumentu. Přenášet se může buď pouze informace o změně, nebo celý obsah dokumentu po změně.
3. Server po zaregistrování změny dokumentu provede jeho analýzu a najde v něm chybný obsah (např. neplatný symbol). Asynchronně zašle editoru zprávu *textDocument/publishDiagnostics*, ve které sděluje editoru, že je v daném souboru na daném místě chyba, která má být označena (např. podtržením).
4. Uživatel použije funkci „Go to definition“ pro přechod na definici označeného symbolu.
5. Server najde definici symbolu a vrátí ji v odpovědi na předchozí požadavek.
6. Uživatel zavře dokument, editor zašle zprávu *textDocument/didClose*, která značí, že je nyní platný obsah dokumentu opět uložen na disku.



Obrázek 3.3: Sekvenční diagram možné komunikace mezi editorem a jazykovým serverem [19].

3.3 Architektura ladicích nástrojů editoru a její rozšiřování

Jednou z klíčových vlastností editoru VSC, která jej dostává od pouhého editoru až k tomu, co přirozeně vnímáme jako vývojové prostředí, je bohatá podpora ladění. Integrace ladicích nástrojů přímo do editoru ulehčuje programátorovu práci (jejíž podstatnou částí je právě cyklus „úprava kódu – překlad – ladění“).

V režimu ladění poskytuje editor rozhraní pro mnoho typických činností prováděných při ladění: řízení chodu programu (krokování, pozastavování atp.), umístování různých druhů

*breakpointu*¹² (běžné, podmíněné, datové¹³, funkční¹⁴), nahlížení do vnitřního stavu programu a jeho modifikace (zobrazení zásobníku volání, stavu vláken, hodnot proměnných), zobrazování informací o výjimkách apod.

Překlad a ladění z pohledu uživatele

Překlad, spouštění a ladění jsou činnosti, které jsou pro každý jazyk odlišné, každá z nich je často prováděna odlišným způsobem, využívá různé nástroje. Architektura ladicích nástrojů ve VSC je proto značně flexibilní z pohledu uživatele i z pohledu rozšíření, které podporu pro jazyk poskytuje. Editor nabízí dvě cesty pro specifikaci kroků, které má pro spuštění překladu a ladění provést: *tasks* (úlohy), pojmenované činnosti jako spuštění externího procesu nebo příkazu rozšíření, a *launch configurations* (konfigurace spouštění).

Konfigurace spouštění určují parametry pro aktivaci konkrétního debuggeru¹⁵ pro daný projekt. Na rozdíl od úloh jsou určeny pouze pro spuštění a nastavení ladicího rozhraní editoru. Uživatel je specifikuje v souboru `launch.json`. Rozšíření VSC, které přidává možnosti ladění, určuje pomocí CP `debuggers`, jaké vlastnosti může uživatel v konfiguraci nastavit. Pokud editor detekuje v projektu soubor v jazyce, pro který nějaké rozšíření poskytuje možnosti ladění, nabídne uživateli vygenerování příslušné sekce v souboru `launch.json`.

3.3.1 Ladicí adaptéry

Termínem debugger je možné nazvat buď prostředí, ve kterém programátor provádí ladění, nebo množinu prostředků, které k ladění programátor používá, nebo program, který tyto prostředky poskytuje – spouští laděný program a umožňuje řídit a sledovat jeho chod. Tyto pohledy se často překrývají, například známý debugger `gdb` (The GNU Project Debugger) zajišťuje všechny tyto činnosti – spouští program a poskytuje prostředky k jeho ladění, které programátor používá prostřednictvím textového rozhraní.

Programovací jazyky a nástroje k jejich ladění se výrazně liší, zároveň však typicky nabízejí obdobné funkce (krokování, breakpointy a podobně). Aby nebylo nutné pro každý debugger implementovat v editoru speciální rozhraní, VSC implementuje generické, jazykově nezávislé rozhraní pro ladění (*generic debugger UI*). To s konkrétními debuggery komunikuje pomocí protokolu *Debug Adapter Protocol* (DAP), který abstrahuje specifika různých debuggerů a poskytuje jednotnou formu komunikace mezi editorem a debuggerem [24].

Protože existující debuggery tento protokol typicky nebudou implementovat přímo, součástí ladicí architektury je navíc tzv. **ladicí adaptér** (*Debug Adapter*), což je program, který komunikuje s editorem prostřednictvím DAP a překládá jeho požadavky do formy, kterou očekává existující debugger. Právě proto, že tento způsob použití je nejtypičtější, je protokol podle ladicích adaptérů pojmenován. To však neznamená, že nově vzniklé debuggery nemohou protokol implementovat přímo a zbavit se tak prostředníka.

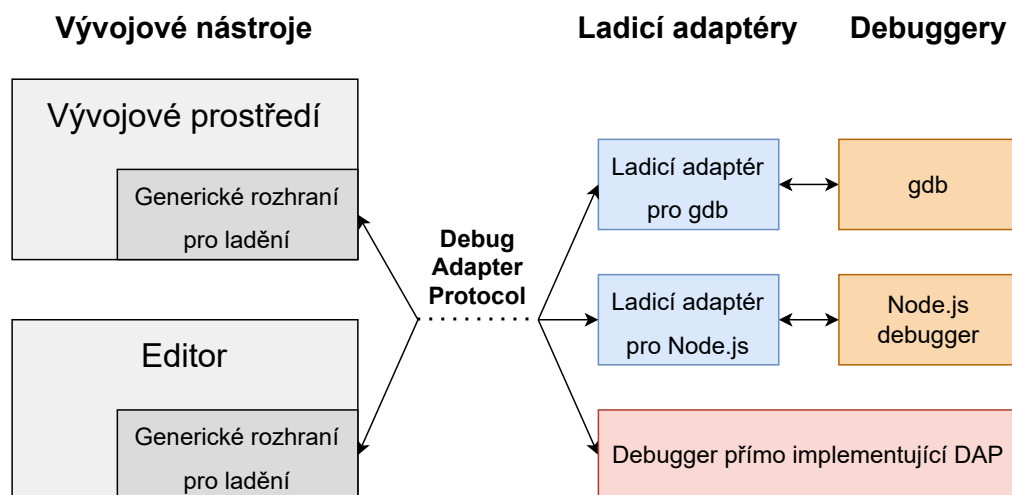
¹²Anglický termín *breakpoint* by se dal v češtině přeložit jako bod přerušení, jde o místo v kódu, v němž je chod programu při ladění pozastaven. Překládání tohoto termínu není typické a v této práci budu dále používat pouze anglické slovo.

¹³Datové breakpointy (*data breakpoints*) umožňují pozastavit program při změně jeho vnitřního stavu, typicky při změně nějaké proměnné.

¹⁴Funkční breakpointy (*function breakpoints*) umožňují pozastavit program při skoku do funkce s určeným názvem. Hodí se např. v případech, kdy je nutné zastavit provádění uvnitř kódu knihovny, jejíž zdrojový kód není dostupný [16, s. 96].

¹⁵Anglický termín *debugger* označuje program, který poskytuje prostředky pro ladění jiného programu. Do češtiny se typicky nepřekládá.

V principu je přístup k integraci ladicích nástrojů podobný tomu, který používá VSC pro poskytování jazykových možností (viz kap. 3.2.3). Podobně jako LSP, i DAP umožňuje nejen jednodušší implementaci editorů, ale i znovupoužitelnost ladicích nástrojů napříč různými editory, jak naznačuje obrázek 3.4. To zároveň nepřináší újmu na rozšiřitelnosti: v DAP je možné v případě potřeby použít vlastní nestandardní zprávy, jejichž obsluhu musí zajistit rozšíření v konkrétním editoru. Oddělení editoru a debuggeru s využitím řádně definovaného protokolu přináší také volnost při implementaci ladicího adaptéru nebo debuggeru, který může být implementován v libovolném vhodném jazyce, na rozdíl od běžných rozšíření VSC.



Obrázek 3.4: Diagram znázorňující použití protokolu DAP. Protokol umožňuje implementovat v editorech unifikované ladicí rozhraní nezávislé na jazyce a zároveň umožňuje využít tentýž debugger v mnoha různých editorech.

DAP také vznikl jako podpůrný prostředek editoru VSC, nyní jde ale o samostatný projekt¹⁶, který je nativně nebo pomocí rozšíření podporován dalšími editory (např. Visual Studio, Eclipse IDE, Emacs nebo vim [22]).

Ačkoliv je ladicí adaptér v principu zcela oddělená vrstva, není neobvyklé, že je implementován v jazyce TypeScript nad běhovým prostředím Node.js, stejně jako editor VSC (prostředky k tomu poskytuje např. knihovna `@vscode/debugadapter`). V takovém případě umožňuje editor provádět požadavky přímým voláním funkcí adaptéru, bez nutnosti komunikovat s ním prostřednictvím meziprocesové komunikace.

Specifikace protokolu dbá na zpětnou kompatibilitu. Všechny jeho implementace musí podporovat základní množinu funkcí, které byly definovány v jeho prvotních verzích. Podporu dalších, rozšiřujících funkcí z novějších verzí protokolu si klient a server oznamují pomocí tzv. **schopností** (*capabilities*). Při inicializačním požadavku oznámí klient své schopnosti, server v odpovědi zašle ty své a při komunikaci pak musí obě strany využívat jen ty funkce, které jim byly oznámeny druhou stranou.

DAP je textový protokol, ve kterém jsou jednotlivé zprávy a jejich data serializovány ve formátu JSON¹⁷. Tato serializovaná data jsou před odesláním opatřena hlavičkou obsahující

¹⁶Webová stránka projektu Debug Adapter Protocol: <https://microsoft.github.io/debug-adapter-protocol>. Specifikace protokolu je veřejně dostupná pod licencemi Creative Commons a MIT, do kterého je možné přispívat pomocí git repozitáře.

¹⁷Samotný protokol je formálně specifikován prostřednictvím schématu ve formátu JSON: <https://microsoft.github.io/debug-adapter-protocol/debugAdapterProtocol.json>.

délku zprávy. Protokol definuje čtyři základní typy zpráv: požadavek (*Request*), odpověď (*Response*), chybovou odpověď (*ErrorResponse*) a událost (*Event*).

Požadavek vyjadřuje žádost o provedení nějaké akce, která je označena textovým identifikátorem. Tyto akce odpovídají úkonům, které uživatel při ladění provádí: příkladem požadavku je *Launch* (spustit), *SetBreakpoints* (nastavit breakpointy) nebo *Next* (další – proveď krok). Požadavky jsou označovány sekvenčními čísly, na každý požadavek musí server poslat odpověď.

Události jsou serverem asynchronně zasílané zprávy, které nesou informaci o změně stavu programu. Příkladem je *Output* (výstup programu nebo debuggeru) nebo *Exited* (program skončil).

Kapitola 4

Návrh řešení

Tato kapitola se zabývá funkcemi, které by měl didaktický nástroj pro výuku JSA procesorů Arm poskytovat, a vysvětluje, jakým způsobem budou k jejich dosažení využity možnosti představené v předchozích kapitolách. Řešení je rozděleno na několik vzájemně spolupracujících komponent: jazykový server, simulační službu pro překlad, běh a ladění a samotné rozšíření pro editor VSC (obsahující navíc ladicí adaptér). Kapitola 4.2 představuje, jak a proč spolu vzájemně komunikují.

4.1 Požadované vlastnosti řešení

Hlavní cílovou skupinou řešení jsou studenti FIT, kteří mají základní přehled o jazycích symbolických adres obecně (konkrétní zkušenosti mají s programováním v JSA architektury x86). Těm tedy řešení pomůže zejména s pochopením rozdílů mezi architekturami x86 a Arm. Řešení však bude připraveno i pro výuku programování na strojové úrovni bez předchozích znalostí programování v JSA.

Podstatnou vlastností řešení bude jednoduchost instalace a nastavení prostředí pro úspěšné spuštění a ladění programu. Jednou z předností editoru VSC je snadná instalace rozšíření, kterou je možné realizovat z vestavěného katalogu stiskem jednoho tlačítka. Celé řešení tedy bude možné nainstalovat tímto způsobem bez nutnosti manuální instalace či konfigurace dalších součástí. Důraz je kladen na podporu všech platforem, na kterých běží editor VSC (Windows, GNU/Linux i macOS na architekturách x86, amd64 i Arm).

Návrh řešení bude dbát na rozšiřitelnost a znovupoužitelnost jednotlivých komponent. V budoucnu tak bude možné jej například integrovat i do jiných editorů a potenciálně i do webových aplikací. Část zajišťující běh by mělo být možné, s relativně snadnými úpravami, použít dokonce i pro simulaci jiných architektur. O těchto možnostech pojednává kapitola 6.

4.1.1 Integrace podpory jazyka do editoru

Programátoři jsou v dnešní době zvyklí na vývojová prostředí s bohatou jazykovou podporou při editaci kódu. Pro účely výuky je tedy vhodné zajistit ji i pro JSA. Požadovanými vlastnostmi integrace jazyka do editoru jsou: zvýraznění syntaxe, našeptávání, dokumentace instrukcí, nápověda k signaturám, rozpoznání chyb, diagnostiky, práce se symboly a skládání bloků kódu.

Zvýrazňování syntaxe

Zvýrazňování syntaxe (*syntax highlighting*) zásadně usnadňuje čtení zdrojového kódu zejména barevným odlišováním jednotlivých částí textu podle jejich významu v daném jazyce. Studie [30] naznačuje, že zvýrazňování významně ulehčuje programátorovi orientaci v kódu a jeho pochopení.

<pre> 4 main: 5 ADDSCS R1, R0, R0, LSL R0 6 PUSH {LR} @ use pseudo directive 7 LDR R0, =string @ R0 points to string 8 BL 123 9 LDR R7, =value1 @ get address of param 10 LDR R8, [R7] @ load value1 into R8 11 PUSH {R8} // put on stack 12 VADD.F32 S0, S1, S2 13 14 /* 15 MOV R0, 0xff000000 16 ADD R0, 0xc 17 */ 18 19 ADD SP, SP, #12 @ balance stack 20 POP {PC} @ restore PC 21 22 __exit: 23 MOV PC, LR @ simple exit 24 25 abc: MOV R1, R0 26 test: MOVSCS R0, R0 27 28 .data 29 string: .asciz "Values are: %d, %d, %d and %d\n" </pre>	<pre> 4 main: 5 ADDSCS R1, R0, R0, LSL R0 6 PUSH {LR} @ use pseudo directive 7 LDR R0, =string @ R0 points to string 8 BL 123 9 LDR R7, =value1 @ get address of param 10 LDR R8, [R7] @ load value1 into R8 11 PUSH {R8} // put on stack 12 VADD.F32 S0, S1, S2 13 14 /* 15 MOV R0, 0xff000000 16 ADD R0, 0xc 17 */ 18 19 ADD SP, SP, #12 @ balance stack 20 POP {PC} @ restore PC 21 22 __exit: 23 MOV PC, LR @ simple exit 24 25 abc: MOV R1, R0 26 test: MOVSCS R0, R0 27 28 .data 29 string: .asciz "Values are: %d, %d, %d and %d\n" </pre>
---	---

Obrázek 4.1: Stejný kód bez zvýraznění syntaxe a s ním.

Pro dosažení zvýrazňování je nutné rozdělit text na segmenty zvláštního významu (*tokens*), na které jsou následně aplikována obarvovací pravidla. Rozdělování textu (*tokenizaci*), je možné provést deklarativně pomocí gramatik (viz část [Deklarativní zvýrazňování syntaxe](#) kap. 5.2) nebo programově algoritmem analyzujícím kód (*semantic highlighting*), což umožňuje přiřadit jednotlivým tokenům typy s využitím kontextových informací. Tyto přístupy lze kombinovat.

V JSA je možné rozlišit například tyto druhy tokenů:

- mnemoniky instrukcí,
- příznaky instrukcí:
 - příznak **S**, se kterým instrukce nastavuje příznaky ve stavovém registru,
 - kód podmínky pro vykonání instrukce,
 - specifikátor šířky kódování instrukce,
 - specifikátor datových typů SIMD a FP instrukcí,
 - specifikátor směru růstu adresy u instrukcí pro násobnou manipulaci s pamětí (LDM, STM);
- názvy registrů (obecných, SIMD, speciálních),
- číselné konstanty,
- druhy posunu/rotace,
- specifikátory zarovnání paměti (u SIMD instrukcí),
- návěští,

- direktivy překladače.

Požadavkem na řešení je tedy poskytnout zvýrazňování syntaxe jasně odlišující jednotlivé druhy tokenů, které navíc bude uživatelsky přizpůsobitelné.

Našeptávání

Našeptávání (*code completions*) je funkce, při které editor během psaní kódu uživateli zobrazuje návrhy slov, která by při aktuálním stavu kódu mohl na daném místě použít. Editor zobrazí našeptávací okénko buď na základě explicitní výzvy uživatele (typicky stiskem klávesové zkratky), nebo po zadání nějakého znaku, který odděluje symboly rozdílného druhu (v JSA to může být např. čárka, mezera nebo hranatá závorka).

V JSA se nabízí zejména našeptávání mnemonik a vhodných hodnot operandů. Našeptávání mnemonik navíc musí po detekci validní instrukce nabízet i varianty instrukce s pro ni platnými příznaky: např. po zadání aritmetické instrukce `ADD` našeptávač nabídne `ADDS` (varianta `S` pro nastavování příznaků ve stavovém registru) nebo `ADDEQ` (a další podmínkové kódy).

Řešení bude počítat také s případy, kdy je možné nabídnout jak jinou mnemoniku, tak i příznak už existující mnemoniky (např. po zadání `B`, což je platná mnemonika skokové instrukce (*Branch*), by mělo našeptávání nabídnout jak `BL`, což je úplně jiná mnemonika, tak i `BEQ`, což je původní mnemonika s příznakem – podmínkovým kódem).

Nabízeny budou pouze takové návrhy, které na daném místě dávají smysl. Důraz bude kladen také na co nejrozměšší řazení, aby návrhy, které programátor bude chtít využít pravděpodobněji, byly v seznamu nejvýše. Například vektorové instrukce často vyžadují příznak datového typu, který je od mnemoniky oddělen tečkou. U všech instrukcí ale může být použit také specifikátor šířky kódování `.W`, který je v A32 kódu validní, ale zbytečný. Našeptávač tedy po zadání tečky u vektorové instrukce nabídne nejprve specifikátory datových typů (navíc jen těch, které mohou být pro danou instrukci použity) a až pod nimi v seznamu bude specifikátor `.W`.

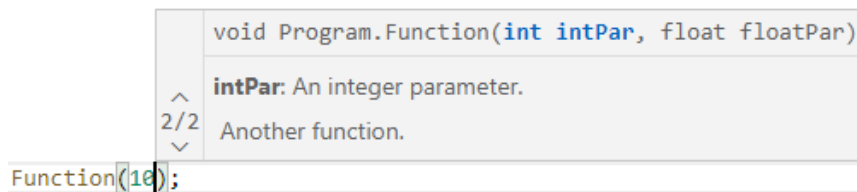
Součástí našeptávače bude také doplňkové okno s dokumentací k jednotlivým našeptávaným položkám. U mnemonik instrukcí využije řešení tuto možnost k zobrazení krátkého popisu instrukcí a zobrazení odkazu na úplnou dokumentaci.

Dokumentace

Řešení umožní snadné procházení seznamu všech instrukcí, vyhledávání v něm a zobrazení dokumentace jednotlivých instrukcí. Součástí balíčku s dokumentací instrukční sady jsou navíc popisy instrukcí v pseudokódu, které budou v rozšíření také dostupné.

Nápověda k signaturám

Nápověda k signaturám (*signature help*) je funkce editoru zobrazující kontextově závislé informace o použití nějaké konstrukce jazyka. Při vývoji v běžných imperativních programovacích jazycích, jakým je např. jazyk C#, se nápovědy k signaturám využívají zejména při volání funkcí. Okno zobrazené v editoru poblíž kurzoru obsahuje zejména informace o parametrech dané funkce (viz obr. 4.2). Důležitým aspektem je zvýraznění parametru, ve kterém je aktuálně umístěn textový kurzor, což usnadňuje programátorovi orientaci v zápisu volání funkce s větším množstvím parametrů. Může obsahovat také dokumentaci a v jazycích s přetěžováním typicky umožňuje zobrazovat informace o dalších variantách volání dané funkce.



Obrázek 4.2: Nápopěda k signaturám v kódu v jazyce C#. Zobrazené okno obsahuje dokumentaci funkce i konkrétního parametru. U přetížené funkce je možné listovat jednotlivými signaturami.

Při práci s JSA je možné na každou zapisovanou instrukci pohlížet jako na funkci, přičemž analogií k parametrům jsou operandy. Nápopědu k signaturám řešení tedy použije pro zobrazování informací o aktuálně zapisované instrukci a o jiných možnostech použití mnemoniky na daném řádku.

V rámci nápopědy poskytne řešení název a krátký popis instrukce, popis operandu, na kterém má uživatel zrovna kurzor, a odkaz na úplnou dokumentaci. Uživatel může listovat dalšími popisy instrukcí, které odpovídají dané mnemonice (např. většina aritmetických instrukcí má varianty pracující s registry nebo s konstantou).

Rozpoznání chyb a diagnostiky

Typickou schopností vývojových prostředí je detekce a zobrazení zejména syntaktických chyb v kódu. Editor VSC k tomu zavádí tzv. diagnostiky (*diagnostics*), což je možnost jazykového serveru zasílat editoru informace o segmentech kódu, které mají být jistým způsobem zvýrazněny a u kterých má být zobrazena nějaká zpráva. Nemusí jít nutně jen přímo o chyby, server takto může poskytnout uživateli libovolnou informaci vyplývající z analýzy kódu.

Řešení tímto způsobem především upozorní uživatele na chybně zapsané instrukce nebo jejich operandy a na další syntaktické chyby. Podobně upozorní také na neplatné použití příznaků, které jsou pro danou instrukci neplatné, na redefinice symbolů nebo na podezřele definované funkce (takové, které jsou definovány dřív, než je v předchozí funkci detekována instrukce nebo direktiva typicky ukončující funkci, např. `BX LR` nebo `.endfunc`).

Práce se symboly

Editor VSC poskytuje také několik funkcí pro práci se symboly v kódu. Typickými činnostmi, které mohou programátorovi usnadnit orientaci v kódu, je vyhledávání definice symbolu nebo naopak jeho použití napříč kódem. Užitečný může být také přehled všech symbolů definovaných v aktuálním souboru.

CodeLens

Editor umožňuje u libovolných míst v kódu zobrazit tzv. *CodeLens*, krátké doplňkové informace umístěné přímo v zobrazení zdrojového kódu. Řešení je využije pro grafické odlišení návěští, která programátor označí jako počátek funkce prostřednictvím direktivy `.type` nebo `.func`. Součástí zobrazeného textu bude také informace o počtu použití daného symbolu. Po kliknutí na zobrazený text *CodeLens* se otevře náhled na místa, na kterých je daný symbol použit (typicky tedy místa volání funkce).

Skládání bloků kódu

Další užitečnou funkcí editoru je možnost sbalovat a rozbalovat bloky kódu (*code folding*). Ve vyšších programovacích jazycích (např. C) blok kódu často tvoří dvojice složených závorek { }. V JSA se za blok kódu dá považovat např. fragment kódu reprezentující funkci (od návěští po nějakou instrukci, která typicky funkci ukončuje). Řešení umožní kromě skládání takových bloků také definici vlastních bloků prostřednictvím komentářů.

4.1.2 Spouštění a ladění

Základním požadavkem na řešení je, aby umožňovalo jednoduchý překlad kódu bez nutnosti konfigurace a poskytovalo co nejširší možnosti ladění. Při výuce základů JSA není žádoucí, aby studenti řešili problémy s překladem a linkováním. Samozřejmě je vhodné, aby řešení jistou míru kontroly nad tímto procesem poskytovalo, ale pro základní použití bude celý proces otázkou napsání zdrojového kódu a stisknutí tlačítka pro spuštění v editoru. V režimu ladění by pak řešení mělo obsáhnout maximum funkcí, které editor VSC nabízí a které zároveň dávají v kontextu JSA smysl. Některé z nich jsou shrnuty níže, kapitola 4.1.3 pak představuje způsoby, jakými řešení didakticky přínosně studentům prezentuje stav procesoru a paměti.

Jednotlivé funkce řešení budou v rozumné míře nastavitelné, aby pokročilejším uživatelům umožnily upravit zobrazování některých funkcí didaktického rázu, které pro ně mohou být už zbytečné, a nabídky větší možnosti experimentování.

Překlad a spuštění

Před spuštěním laděného programu obvykle uživatel vytváří konfiguraci spouštění (soubor `launch.json`). Rozšíření však poskytne předlohu této konfigurace, pokud jsou v otevřeném projektu rozpoznány soubory s JSA. Základním a jediným vyžadovaným konfiguračním parametrem budou zdrojové soubory. Řešení zajistí jejich přenos na server, překlad a linkování.

Pokročilejší uživatelé budou moci nastavit parametry příkazového řádku, které budou předány překladači a linkeru. V konfiguraci běhové služby však budou pro tyto uživatelské změny nastavitelná omezení, aby nemohl uživatel narušit chod celého řešení. Konfigurace spouštění umožní také pozměnit libovolné nastavení běhu a ladění oproti konfiguraci rozšíření.

Možnosti inicializace stavu programu

Řešení umožní konfiguraci chování běhového jádra při zavádění programu do paměti a jeho spouštění. Při zavádění programu je v emulované paměti alokován prostor pro zásobník. Uživatel může nastavit:

- Velikost zásobníku.
- Umístění zásobníku v adresním prostoru emulovaného programu: bude možné nastavit buď konkrétní fixní adresu, nebo její náhodné generování při každém či prvním spuštění programu.
- Způsob nastavení registru SP: Zásobník je možné plnit od nejnižší adresy k nejvyšší, nebo naopak (režim *ascending*, nebo *descending*). Hodnota SP může ukazovat na

první zaplněnou, nebo první nezaplněnou adresu (režim *full*, nebo *empty*). Použitý přístup ovlivňuje počáteční nastavení SP. Ačkoliv architektura Armv8 ukládá použití *full descending* zásobníku, uživatel může tímto nastavením experimentovat i s jinými režimy.

- Počáteční obsah zásobníku: zásobník může být před spuštěním vynulován, nebo může být celý paměťový prostor vyplněn náhodnými daty. To může studentům pomoci s pochopením chyb vzniklých přístupem do paměti, jejíž obsah předtím explicitně nenastaví.

Konfigurovatelné možnosti naplnění náhodnými daty poskytne řešení také pro registry R0–R12 a pro SIMD/FP registry D0–D31.

Program by měl vždy používat jen prostor, jehož využití ve svém binárním souboru definuje, resp. který mu operační systém na jeho žádost přidělí. Kvůli optimalizaci využití paměti je však často dostupná i jistá část adresního prostoru, o který program nepožádal, proto by k němu správně neměl přistupovat. Běhová služba nabídne nastavení striktního režimu, ve kterém je při přístupu do každého explicitně nealokovaného paměťového prostoru program ukončen s výjimkou. Studentovi to pomůže v hledání potenciálních chyb souvisejících s neoprávněnými přístupy do paměti.

Řízení chodu, krokování, breakpointy

Řešení umožní spuštěný program pozastavovat a poté jeho chod opět obnovit nebo v něm krokovat. Pozastavování je asynchronní: program je možné pozastavit kdykoliv, běhová služba pak správně vyhodnotí, na které instrukci k pozastavení došlo a zvýrazní ji.

Krokování je v kontextu JSA přímočaré: znamená provedení jedné instrukce a opětovné pozastavení chodu. Editor VSC nabízí poměrně běžnou trojici možností krokování: *step over*, *step into* a *step out*. Ty se běžně používají ke vstupování do funkcí a vystupování z nich. Vzhledem k tomu, že řešení nebude podporovat rozpoznávání funkcí a tato tlačítka není možné v editoru skrýt, budou všechna provádět stejnou činnost (krok na další instrukci).



Obrázek 4.3: Panel pro řízení chodu programu v režimu ladění. Zleva: *continue* (pokračovat v běhu), *step over*, *step into*, *step out*, *step back* (zpětný krok), *reverse continue* (pokračovat v běhu pozpátku), restart a ukončení programu.

Základní schopností ladicího nástroje je nastavování *breakpointů* – míst v programu, při jejichž dosažení bude provádění programu pozastaveno a uživateli budou nabídnuty možnosti analýzy stavu procesoru. Řešení tedy umožní nastavování breakpointů na jednotlivých instrukcích v kódu. Podporovat bude také *logpointy* – speciální druh breakpointů, které nezastavují provádění programu, ale zasílají uživatelem definovanou zprávu do konzole.

Zpětné krokování

Protokol DAP (a editor VSC) definuje také rozhraní pro akci zpětného kroku (*step back*) a běhu pozpátku, neurčuje přitom konkrétní sémantiku těchto akcí. Řešení je využije tak, že umožní vracet uživatelem provedené kroky: po zastavení chodu vyvolaném uživatelem

nebo dosažením breakpointu se v běhovém jádře bude při každém kroku ukládat stav emulovaného procesoru, a to až do chvíle, kdy uživatel obnoví chod programu funkcí *continue*. Tehdy se uložené stavy vymažou. Při použití funkce zpětného běhu (*reverse continue*) bude obnoven nejstarší uložený stav.

Disassembling

V některých situacích se může hodit také zpětný převod strojového kódu uloženého v paměti do JSA. Studentovi to může pomoci např. s pochopením faktu, že některé instrukce jsou pouze aliasy pro jiné (např. instrukce *PUSH*), nebo se způsobem, jakým pracuje pseudoinstrukce *LDR* (viz kap. 2.6.4). Editor VSC pro tyto účely nabízí okno *Disassembly*, do kterého je možné přejít z libovolného místa ve zdrojovém kódu. V tomto zobrazení se na každém řádku vypíše adresa v paměti, hexadecimální reprezentace strojového kódu, instrukce zpětně převedená do JSA a původní obsah odpovídajícího řádku zdrojového kódu. Je v něm možné umisťovat běžné breakpointy.

Vstup a výstup, simulované funkce

Úlohou počítačových systémů obecně je transformovat vstupy na výstupy, i výukový simulátor by tedy měl poskytnout možnosti interakce s prováděným programem. Při vývoji pro skutečné systémy využívají programy často textový vstup a výstup realizovaný např. s využitím funkcí standardní knihovny, která poskytuje rozhraní pro příslušná systémová volání.

Studenti, kteří řešení budou využívat, mají předchozí zkušenosti s voláním funkcí standardní knihovny jazyka C z prostředí JSA. Jedním z předpokládaných cílů studia programování v JSA arch. Arm je také pochopení její konvence volání¹. Jako ideální možnost realizace vstupu a výstupu se tedy nabízí poskytnout uživateli funkce této knihovny a simulovat chování, na které jsou uživatelé zvyklí z běžných programů s textovým rozhraním.

Pro využití skutečných implementací standardní knihovny jazyka C by řešení muselo buď zavést skutečné jádro operačního systému, nebo simulovat všechna jeho systémová volání. Takové přístupy jsou mimo rozsah této práce, o možnostech budoucího rozšíření pojednává kapitola 6.3.

Řešení ovšem nabídne alternativu v podobě tzv. **simulovaných funkcí**. Uživatel bude moci některé funkce ve svém kódu volat klasickým způsobem. Řešení při zjištění takového volání vykoná mimo emulaci odpovídající činnost, např. zašle do konzole v editoru výstup programu.

Pro textový vstup a výstup budou k dispozici následující funkce jazyka C: `getchar`, `putchar`, `ungetc`, `puts` a `printf`. Kromě nich nabídne také vlastní nestandardní funkce pro čtení čísel.

Obdobným způsobem poskytne řešení také možnosti dynamické alokace paměti prostřednictvím funkcí rodiny `malloc`. Kompletní přehled simulovaných funkcí a jejich chování je dostupný v kapitole **Simulované funkce** uživatelského manuálu v příloze B.

V budoucnu by bylo vhodné rozšířit řešení také o možnosti simulovaného vstupu a výstupu mapovaného do paměti MMIO (*Memory-Mapped Input/Output*), podobně jako to nabízejí některé stávající simulátory popsané v kapitole 2.4.

¹Konvenci volání procedur a funkcí stanovuje Arm jako součást *ABI* architektury Arm a je dostupná v dokumentu na adrese <https://github.com/ARM-software/abi-aa/blob/main/aapcs32/aapcs32.rst>.

4.1.3 Reprezentace stavu procesoru a paměti v režimu ladění

Reprezentace stavu procesoru a obsahu paměti je stěžejní funkcí ladicího nástroje určeného pro výuku. Studentům má řešení pomoci pochopit, že data v systému mohou být interpretována různými způsoby. Příkladem mohou být 64b SIMD/FP registry D0–D31: s jednou 64b hodnotou, která je v takovém registru uložena, se může pracovat jako s:

- jedním 64b celým číslem (v Armv8 označovaným jako *doubleword* – dvojslovo [3, kap. A1.4]),
- jedním 64b číslem s plovoucí řádovou čárkou ve formátu *binary64* (podle IEEE 754),
- dvěma 32b celými čísly (*word* – slovo),
- dvěma 32b čísly s plovoucí řádovou čárkou ve formátu *binary32* (podle IEEE 754),
- čtyřmi 16b celými čísly (*halfword* – poloviční slovo),
- osmi 8b celými čísly (*byte*).

Všechny celočíselné datové typy navíc mohou být interpretovány bezznaménkově nebo znaménkově (v doplňkovém kódu).

Řešení bude klást důraz na jasnou a dobře přístupnou prezentaci stavu procesoru. Využije k tomu podokno *Variables* (proměnné) v editoru VSC. To umožňuje debuggeru prezentovat libovolné dvojice název–hodnota ve stromové struktuře. To je možné využít právě pro zobrazování různých pohledů na hodnoty v registrech a paměti.

Řešení bude jako proměnné zobrazovat následující data:

- registry pro obecné využití R0–R14,
- SIMD/FP registry Q0–Q15, dělené na D0–D31 a na S0–S31,
- registry se stavem procesoru (programový čítač PC a příznakové registry APSR, CPSR, FPSCR a FPEXC),
- data na zásobníku,
- symboly označující data v datové sekci.

Zobrazení různých interpretací hodnot obecných a SIMD/FP registrů

Registry architektury Armv8 jsou 32bitové, což znamená, že kromě FP a celočíselných (znaménkových i bezznaménkových) interpretací celé hodnoty je možné na ně pohlížet také jako na nositele dvou 16b hodnot nebo čtyř 8b hodnot. Řešení umožní zobrazení některých těchto možných interpretací, uživatel navíc může v konfiguraci určit, které se mají zobrazovat. Hodnotu celého registru i všech zobrazených interpretací navíc uživatel bude moci nastavovat. Řešení bude podporovat vstup i zobrazení v dekadické, hexadecimální a binární soustavě.

Čísla s plovoucí řádovou čárkou v kódování *binary32* podle standardu IEEE 754 se skládají z tří částí: znaménkového bitu, 8b exponentu v kódu s posunutou nulou a 23b mantisy², která je celým číslem bez znaménka [14, kap. 3.4]. U takových čísel bude možné

²S mantisou se při výpočtech pracuje jako s 24b číslem, jehož nejvyšší bit je implicitně určen podle hodnoty exponentu, ukládá se tedy pouze spodních 23 bitů [14, kap. 3.4].

zobrazovat i nastavovat jako samostatné proměnné také tyto části, a to v binární a dekadické podobě.

Obdobné možnosti nabídne řešení i pro SIMD/FP registry. Stromová struktura proměnných v editoru bude respektovat způsob, jakým se tyto registry z jednotlivých tříd překrývají (viz kap. 2.2.3), např. registry třídy D v ní budou potomci registrů třídy Q. Zároveň stále umožní interpretovat jejich hodnoty v různých datových typech, podobně jako registry pro obecné použití. Řešení opět umožní podrobně nakonfigurovat, které SIMD/FP registry se mají zobrazovat, jakým způsobem se má zobrazovat jejich hodnota (jako celé číslo nebo jako desetinné číslo³) a které interpretace se pro jednotlivé třídy mají zobrazit.

Zobrazení příznakových registrů

Řešení umožní pomocí proměnných zobrazovat také příznakové registry, které reprezentují vnitřní stav procesoru (informace o výsledku některých provedených instrukcí, konfiguraci chování procesoru atp.). Tyto registry jsou, stejně jako obecné registry, 32bitové. Jejich hodnoty však představují bitová pole, v nichž jednotlivé bity nebo skupiny několika bitů nesou různé, nezávislé informace (příznaky). Dává tedy smysl i zde využít stromové struktury a kromě samotné hodnoty registru zobrazit jako jeho potomky také hodnoty jednotlivých polí. Ty budou pojmenovány svou zkratkou, jak ji uvádí referenční manuál arch. Armv8, po najetí kurzorem myši se zobrazí také jejich úplný název. Hodnoty, které mají speciální význam (jiný než pravda/nepravda), se zobrazí i s textovým popisem.

Ve výchozím stavu se budou zobrazovat registry APSCR a FPSCR (viz kap. 2.2.2), které nesou zejména pro programátora nejzásadnější informace o výsledcích provádění aritmetických a logických instrukcí. Uživatel si však bude moci povolit také rozšířené zobrazení, ve kterém se mu zobrazí celý registr CPSR, FPSCR a také konfigurační registr FPEXC.

Kromě zobrazení v podokně *Variables* přidá rozšíření i vlastní podokno, ve kterém bude vizuálně indikovat stav čtyř základních podmínkových příznaků N, Z, C, V.

Zobrazení stavu zásobníku

Řešení poskytne uživateli reprezentaci zásobníku a dat na něm. Opět k tomu využije strom proměnných, do kterého se budou dynamicky umisťovat hodnoty reprezentující datové položky na zásobníku. Jejich počet se určí podle rozdílu hodnoty SP a konce zásobníkového adresního prostoru (předpokládá se, že každá položka na zásobníku zabírá 4 B). K dispozici budou také interpretace těchto 32b hodnot ve smyslu různých datových typů podobně jako u obecných registrů.

Zobrazení symbolů označujících data v datové sekci

Řešení umožní jako proměnné zobrazovat také symboly, které jsou v kódu definovány v datové sekci, a hodnoty, které jsou v paměti uloženy na odpovídajících adresách. Zobrazí informaci o typu, kterým byly definovány, jejich adresu a hodnotu.

Symboly nenesou informaci o datovém typu, protože s tímto konceptem JSA nepracuje. Typ bude řešení odhadovat podle direktivy, která byla v kódu za definicí symbolu použita. Podporovanými direktivami budou: `float`, `single`, `double`, `word`, `long`, `int`, `short`, `hword`, `byte` a `asciz`. Čísla budou interpretována jako bezznaménková.

³Zobrazení ve formátu desetinného čísla nebude nabízeno pro registry třídy Q, protože arch. Armv8 nepodporuje 128b FP čísla [3, A1.4.1].

Uživatel může sledovat i symboly nepodporované tímto automatickým mechanismem (nebo interpretované jiným datovým typem) prostřednictvím vyhodnocování výrazů, viz dále.

Vyhodnocování a sledování výrazů

Další možností, kterou poskytuje editor VSC pro prezentaci dat laděného programu, je podokno *Watch*. To vypadá podobně jako podokno *Variables*, ale na rozdíl od něj si uživatel sám určuje, která data se zde mají zobrazovat. Činí tak zadáním textového **výrazu**, který editor zasílá debuggeru a jako odpověď dostává objekt podobný proměnné – výsledkem výrazu tedy může být i stromová struktura proměnných. Výraz je možné vyhodnotit také jednorázově, a to zadáním do podokna *Debug Console*. Syntaxi i sémantiku výrazů definuje použitý debugger. Pro editor je transparentní a možnosti využití výrazů jsou tedy široké.

Řešení bude podporovat několik typů výrazů:

- Registrové, které zpřístupní hodnotu celého registru nebo jeho části. Podporovány budou obecné i SIMD/FP registry. Výraz umožní určit, na položky jakého datového typu má být hodnota rozdělena, a případně kterou z nich zpřístupnit.
- Adresové, které zpřístupní hodnotu v paměti. Jako adresu bude možné použít hodnotu z registru, symbol nebo konstantu. Adresu bude možné posunout o hodnotu danou jiným registrem nebo konstantou a volitelným bitovým posunem či rotací, podobně jako to umožňuje instrukční sada A32. Bude možné určit datový typ položky, ke které je přistupováno, a tím i počet bajtů. Podporováno bude také čtení textových řetězců zakončených nulovým bajtem.
- Výrazy hodnoty symbolu, které zobrazí hodnotu (typicky adresu) libovolného symbolu.
- Výraz cesty k proměnné, který pouze zobrazí proměnnou zadaného jména, případně jejího potomka (např. zadáním *APSR.N* je zobrazena pouze hodnota příznaku *Negative* registru *APSR*).

Hodnoty registrových a adresových výrazů bude možné číst i zapisovat, u výrazů hodnoty symbolu by to nedávalo smysl.

Kompletní uživatelský popis možností výrazů je dostupný v manuálu v příloze **B**. Formální definice gramatiky výrazů je uvedena v příloze **D**.

Vyhodnocování výrazů ve zdrojovém kódu

V režimu ladění zobrazí řešení uživateli hodnotu registru nebo symbolu, když na něj najede kurzorem myši. Zobrazené okno bude obsahovat stejnou strukturu, jako odpovídající proměnná v podokně *Variables*.

Zastavování běhu při změně registrů a paměti

Užitečnou dovedností ladicích nástrojů je možnost automatického zastavování chodu programu při změně konkrétní proměnné (v případě JSA tedy registru nebo místa v paměti). V editoru VSC je tato funkce nazvána datový breakpoint (*data breakpoint*). Řešení nabídne sledování změn pro všechny typy proměnných a výrazů, u kterých to dává smysl.

4.2 Architektura navrhovaného řešení

Předchozí kapitoly naznačují, že se řešení bude skládat z několika spolupracujících větších celků. Z architektury rozšíření editoru VSC vyplývá, že tyto logické komponenty budou minimálně tři:

- jazykový server, který poskytne podporu při psaní kódu,
- komponenta poskytující překlad, běh programu (emulaci) a jeho ladění,
- samotné rozšíření pro VSC, které využívá předchozí dvě komponenty.

4.2.1 Jazykový server

Zavedení samostatného jazykového serveru je z hlediska celkového návrhu řešení přímočaré. Půjde o jeden samostatný program, který s editorem komunikuje přímo pomocí LSP, a to prostřednictvím přeměrovaného standardního vstupu. Rozšíření pro VSC bude nakonfigurováno tak, aby tuto komponentu spustilo při rozpoznání kódu v JSA (typicky tak činí podle přípony otevřeného souboru) a připojilo se k ní. Tento program tedy poběží přímo na počítači, na kterém je otevřený editor, a bude mít přímý přístup ke zdrojovým souborům, se kterými uživatel pracuje (může je číst ze souborového systému).

4.2.2 Překlad, spouštění a ladění jako síťová služba

Komplexnější částí řešení je samotný překlad, spouštění a ladění uživatelských programů. Záměrem je vytvořit řešení, které bude jednoduché k použití bez nutnosti jakékoliv konfigurace, a zároveň zajistit, že jej bude bez obtíží možné použít na všech platformách, na kterých je dostupný editor VSC. To není triviální úkol kvůli nástrojům, které do tohoto procesu vstupují (jmenovitě kvůli překladači, linkeru a emulačnímu frameworku Unicorn). Tyto součásti je nutné pro každou platformu přeložit a distribuovat zvlášť, při běhu pak mohou nastat problémy kompatibility s různými verzemi operačního systému.

Především z těchto důvodů jsem se rozhodl koncipovat řešení jako síťovou službu, která kompletně zajišťuje překlad, spouštění a ladění programů (dále bude označena jen jako **běhová služba**). Z pohledu uživatele i editoru půjde v podstatě o černou skříňku – vstupem služby jsou zdrojové kódy a uživatelská konfigurace, na základě kterých služba editoru přímo poskytuje ladicí rozhraní popsané protokolem DAP. Uživatel nemá přímou kontrolu nad procesem překladu a linkování, nedostane se k němu ani výsledný spustitelný soubor (ten služba přímo zavede do své emulační části).

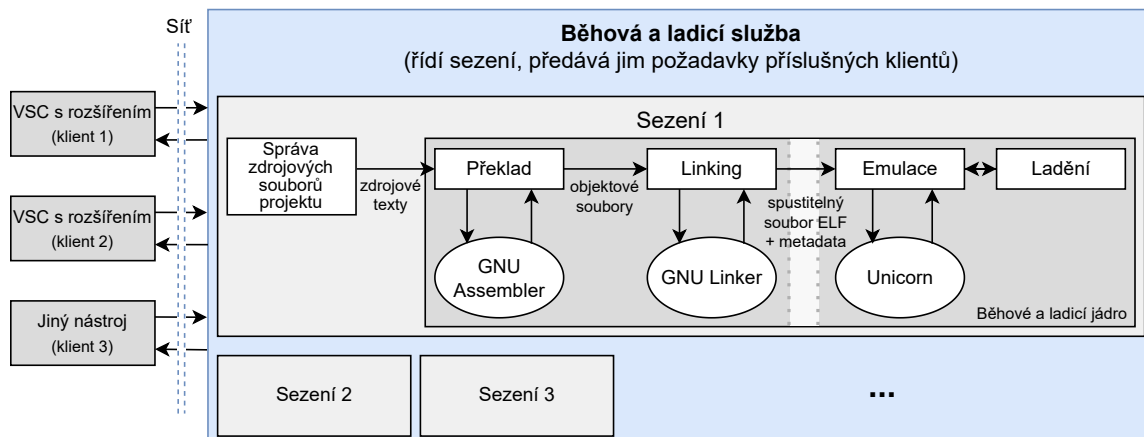
Výhodou tohoto přístupu je, že se pro uživatele z hlediska využívaných nástrojů „nemá co pokazit“. Pro každého uživatele bude chování stejné – nezávisle na systému, na kterém edituje kód. To je v kontextu výuky podstatná vlastnost, neboť to omezí čas strávený řešením individuálních problémů s překladovým a běhovým prostředím a umožní to tak soustředit se na skutečnou práci.

Koncepce síťové běhové služby otevírá cestu možnosti v budoucnu upravit řešení tak, aby mohlo být spuštěno přímo v prohlížeči, což by přineslo další usnadnění používání celého řešení ve výuce (viz také kap. 6.3).

Pokud by uživatel chtěl řešení využívat i bez připojování k vzdálenému serveru, není problém běhovou službu spustit přímo v jeho počítači a používat ji lokálně. Řešení bude podporovat oba tyto způsoby a přepínání mezi nimi.

4.2.3 Architektura běhové služby

Samotná běhová služba se skládá z několika dílčích částí popsanych výše. Protože je služba koncipována jako síťová, předpokládá se, že se k ní bude připojovat několik uživatelů najednou – musí tedy překlad a emulaci provádět ve více na sobě nezávislých instancích a zajistit předávání příchozích požadavků do správné z nich. Pokud služba neběží na stejném počítači jako editor, je také nutné zajistit předávání zdrojových souborů.



Obrázek 4.4: Koncept vzdálené překladové, běhové a ladicí služby. Služba (modrý obdélník) udržuje sezení (*sessions*; světle šedé obdélníky) – kontejnery pro projekt a překladové/běhové jádro (tmavě šedý obdélník).

S ohledem na znovupoužitelnost bude část překladu a spouštění implementována nezávisle na části udržující sezení. Vznikne tak **běhové jádro** – knihovna s popisem objektů, které bude vytvářet běhová služba pro každé sezení. Těmito objekty budou dva logické celky: překladová a emulační služba. Překladová služba spouští překladač a linker, jejichž výstupem je spustitelný soubor ve formátu ELF a další informace o překladu a linkování, které je možné využít pro ladění. Soubor ELF i doplňkové informace jsou načteny do paměti a následně předány emulační službě, která vytvoří instanci knihovny Unicorn, připraví prostředí pro emulaci a zpřístupní funkce ladění. Obrázek 4.4 demonstruje, jakým způsobem je běhová služba vystavěna.

4.2.4 Rozhraní mezi jednotlivými vrstvami řešení

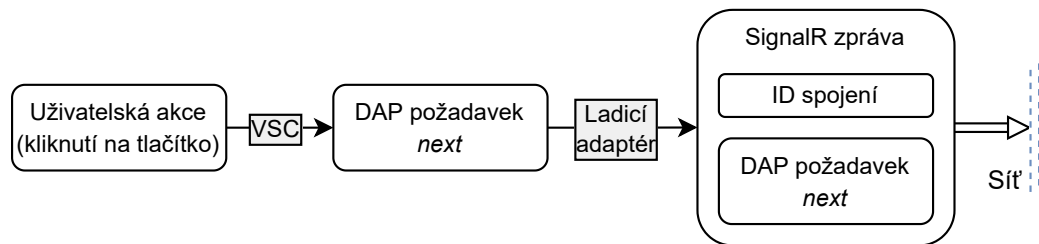
V takto rozvrstveném řešení je podstatné definovat způsob, jakým spolu jednotlivé vrstvy komunikují. Pro účely této kapitoly budu rozlišovat **rozhraní** ve smyslu sémantického prostředku (množiny typů zpráv, které si mezi sebou dvě vrstvy vyměňují) a **nosný prostředek** ve smyslu konkrétního způsobu, jakým jsou zprávy rozhraní zasílány (včetně případné serializace). V případě jazykového serveru se řešení drží specifikace LSP, která určuje jako nosný prostředek protokol JSON-RPC.

Běhová služba

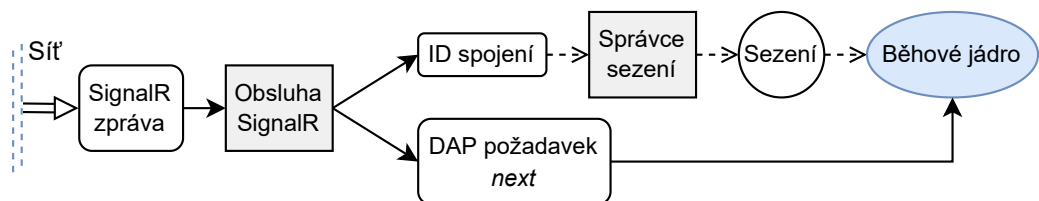
Jednou z vlastností protokolu DAP je, že neumožňuje rozlišovat různá sezení. Předpokladem totiž je, že editor při každém vstupu do režimu ladění spustí novou instanci ladicího adaptéru, která je poté ukončena spolu s režimem ladění (např. po ukončení laděného pro-

gramu). Řešení pracuje s konceptem jednoho serveru pro mnoho nezávislých klientů. Není tedy možné použít pro komunikaci mezi editory a běhovou službou protokol DAP přímo.

Rozhraní běhové služby bude odpovídat protokolu DAP, ale jako nosný prostředek pro jeho zprávy využije technologii *ASP.NET Core SignalR*. Jde o skupinu knihoven, které poskytují vzdálené volání procedur (RPC) mezi serverem a klienty. Je určena pro aplikace, které vyžadují efektivní komunikaci v reálném čase⁴. Protokol SignalR je dobře zdokumentovaný⁵ a pro několik jazyků jsou dostupné stávající klientské knihovny. Tato volba tedy nebrání znovupoužitelnosti řešení i v jiných nástrojích.



(a) Editor po kliknutí na tlačítko „další krok“ vystaví požadavek ve formě DAP zprávy. Předá jej jako argument obslužné funkci ladicího adaptéru. Tělo požadavku je serveru (běhové službě) zasláno jako zpráva protokolu SignalR.



(b) Server (běhová služba) po přijetí zprávy identifikuje klienta podle ID spojení.

Obrázek 4.5: Postup ladicího požadavku od jeho vzniku do zpracování běhovým jádrem. Obrázek a) znázorňuje požadavek na straně klienta (editoru), obrázek b) demonstruje jeho zpracování serverem (běhovou službou).

V rozšíření pro editor VSC bude zahrnut jednoduchý ladicí adaptér, který bude sloužit jako prostředník mezi editorem, který využívá DAP, a běhovou službou, která zprostředkovává rozhraní DAP pomocí SignalR. Tento adaptér bude implementován v jazyce TypeScript, editor jej tak bude spouštět uvnitř svého běhového prostředí a komunikovat s ním přímým voláním jeho funkcí (v zásadě jde hlavně o jednu funkci, `dispatchRequest`, jejímž parametrem je objekt reprezentující libovolný požadavek protokolu). Adaptér při zavedení naváže SignalR spojení se službou a příchozí zprávy z editoru bez pozměnění přepošle službě. Odpovědi na zprávy a asynchronní notifikace ze služby pak obdobně zašle zpět editoru. Princip tohoto zapouzdření je naznačen na obrázku 4.5.

V případě, kdy uživatel využívá vzdálenou běhovou službu, je žádoucí, aby si služba uchovávala sezení (informace o konfiguraci nástroje, zdrojových souborech, dříve přeložený program) po celou dobu, kdy je editor aktivní (je zbytečné přenášet při každém spuštění kód na server nebo provádět znovu překlad, pokud se kód nezměnil). Ladicí adaptér je

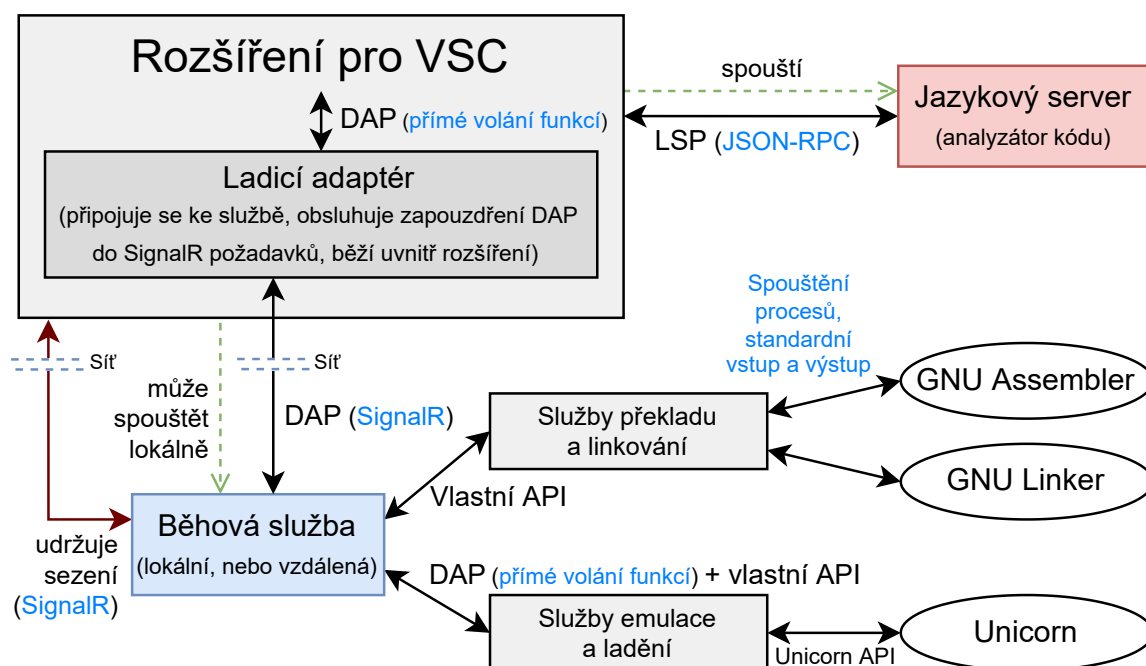
⁴Viz <https://docs.microsoft.com/en-gb/aspnet/core/signalr/introduction?view=aspnetcore-6.0>, cit. 6. 6. 2022.

⁵Dokumentace protokolu SignalR: <https://github.com/dotnet/aspnetcore/tree/main/src/SignalR/docs/specs>.

ovšem aktivní pouze po dobu jednoho spuštění a ladění programu, bude tedy nutné zavést ještě paralelní mechanismus pro udržování sezení po dobu života editoru a pro přiřazování vznikajících krátce žijících ladicích sezení k tomuto déle žijícímu sezení editoru.

Překladačová část běhového jádra žádné standardní rozhraní využívat nebude, zde půjde o vlastní jednoduché API, jehož vstupem budou zdrojové soubory a výstupem objekt reprezentující spustitelný soubor. Rozhraní běhové pak bude z velké části přirozeně opět odpovídat rozhraní protokolu DAP.

Celkovou architekturu řešení na nejvyšší úrovni znázorňuje obrázek 4.6. Jsou zde naznačeny tři hlavní části řešení (rozšíření, jazykový server a běhová služba), rozhraní mezi jednotlivými částmi i nosné prostředky, které jsou využity k přenosu zpráv.



Obrázek 4.6: Návrh celkové architektury řešení. Modré popisky naznačují, jaký nosný prostředek je použit pro daný typ komunikace.

Kapitola 5

Implementace

Tato kapitola blíže představuje technologie a postupy užité pro implementaci řešení navrženého v předchozí kapitole. Popisuje použití frameworku Unicorn pro implementaci běhové služby, poté vývoj samotného rozšíření pro editor VSC a v závěru implementaci jazykového serveru a techniky jazykové analýzy.

Pro implementaci běhové služby i jazykového serveru je využit jazyk C#. Ve zkratce jde o třídní, silně a převážně staticky typovaný, objektově orientovaný jazyk, který je založen na běhovém prostředí .NET. Nepřekládá se přímo do strojového kódu, ale do mezikódu (IL), který je do nativních instrukcí převáděn až za běhu. Běhové prostředí poskytuje mimo jiné správu paměti (*garbage collection*)¹. Zvolen byl zejména pro svou efektivitu z hlediska výkonu programů, rozsáhlé prostředky pro jejich vývoj a také na základě předchozích zkušeností autora. Použité prostředí .NET ve verzi 6 je multiplatformní a je dostupné mj. na operačních systémech Windows, macOS a GNU/Linux, a to na architekturách x64, x86 a Arm ve 32b i 64b verzi², což usnadňuje splnění jednoho ze základních cílů řešení, kterým je široká dostupnost na všech platformách podporovaných editorem VSC.

5.1 Překlad, spouštění a ladění na bázi emulačního jádra Unicorn

Struktura kódu odráží rozdělení běhové části na běhovou službu, která je implementována v projektu `Code4Arm.ExecutionService`, a na běhové jádro implementované v projektu `Code4Arm.ExecutionCore`, který má charakter knihovny. Toto rozdělení umožňuje vhodné oddělení zodpovědností a případné využití běhového jádra i v jiných projektech. S ohledem na to je v kódu zanesena také značná míra abstrakcí, která umožňuje konzumenta knihovny odstínit od implementačních detailů a usnadňuje případné budoucí změny v konkrétní implementaci.

Běhové jádro je uvnitř rozděleno na dva logické celky: překladovou službu ve jmenném prostoru `Assembling` a emulační (a ladicí) službu ve jmenném prostoru `Execution`. Tyto celky jsou na sobě ve své činnosti v podstatě nezávislé (vytváří a spouští se zvlášť), ale vstupem emulační služby jsou objekty `Executable` s daty a metadaty programu, které vytváří právě překladová služba. V přílohách **F** a **G** jsou uvedeny zjednodušené diagramy tříd překladové i emulační služby.

Překladová služba

Překladová služba řeší následující problémy: získání zdrojových souborů, řízené spuštění překladače GAS (které musí být provedeno s vhodnými parametry a kterému musí být zdrojové soubory předány), načtení výstupního objektového souboru i informací o průběhu překladu a ošetřování chybových stavů. Obdobně řídí také linkování pomocí `ld`, při kterém

¹Pro více informací o jazyce C# viz např. <https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>, cit. 7. 6. 2022.

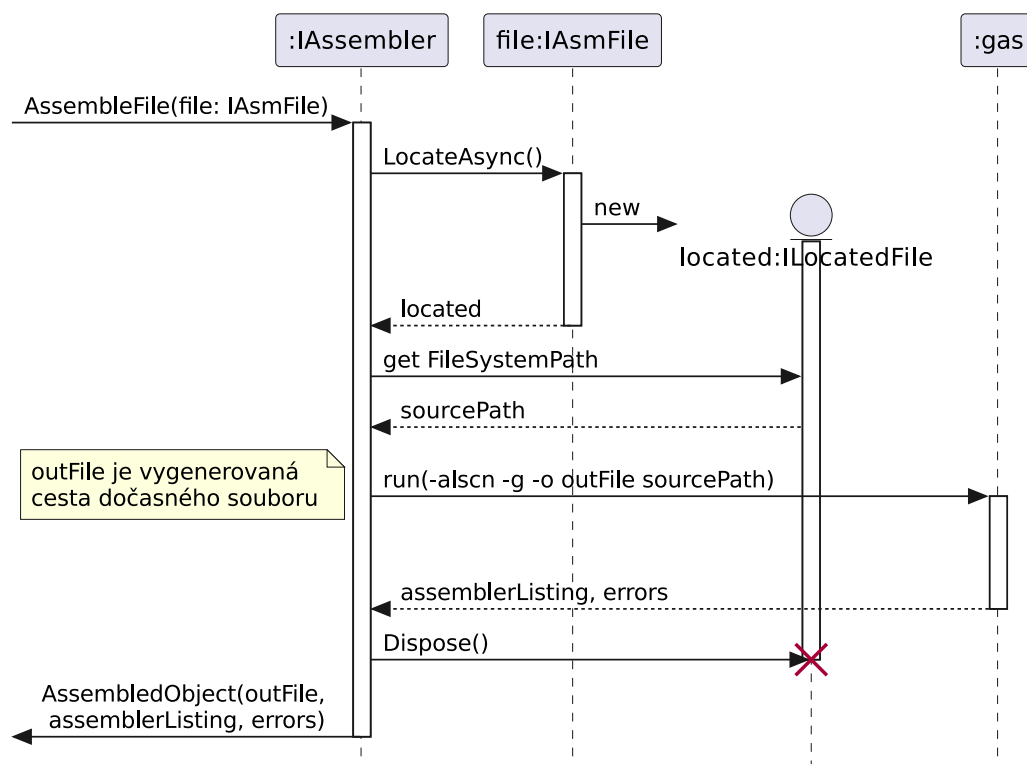
²Viz <https://docs.microsoft.com/en-us/dotnet/core/introduction>, cit. 7. 6. 2022.

navíc musí zajistit dostupnost simulovaných funkcí. V závěru načítá výsledný spustitelný soubor ELF a připravuje informace pro jeho použití v emulační službě.

Jedním z cílů návrhu bylo poskytnout obecné řešení, které nebude spoléhat na přítomnost aktuálních zdrojových souborů na disku. Běhové jádro proto zavádí abstrakci zdrojového souboru `IAsmFile`. Konzument služby poskytuje implementaci, která zajišťuje zpřístupnění souboru v souborovém systému³ a jeho případné zrušení. Implementace předpokládající běh jádra na stejném počítači, kde jsou uloženy zdrojové soubory, tedy nemusí dělat nic, naproti tomu jiná implementace může například načíst soubor z databáze a uložit jej do dočasného umístění.

Ladicí nástroje využívají pro získání některých informací o kódu výstup z překladače i z linkeru. API prostředí .NET umožňuje přesměrovat standardní výstup ze spouštěných procesů a číst jej. Při jeho nesprávném užití však může nečekaně docházet k uváznutí⁴.

Na obrázku 5.1 je uveden zjednodušený sekvenční diagram znázorňující komunikaci mezi objekty při překladu jednoho zdrojového souboru.



Obrázek 5.1: Sekvenční diagram překladu jednoho zdrojového souboru.

³Dalším možným přístupem by bylo požadovat dodání obsahu souboru a jeho předávání překladači pomocí přeměrovaného standardního vstupu, ale vzhledem k předpokladu, že bude často možné použít přímo existující soubor ze souborového systému, by tento přístup byl v mnoha případech zbytečně neefektivní.

⁴V řešení jsem pro zabránění uváznutí při spouštění procesů použil mnou upravenou verzi kódu, jehož původním autorem je M. R. Saeed a je dostupný na <https://stackoverflow.com/a/39872058> pod licencí CC BY-SA 4.0.

Role simulovaných funkcí při překladu

Uživateli budou k dispozici simulované funkce, čili pojmenované rutiny, které je možné z kódu programu volat standardním způsobem jako jiné funkce, ale jejich obsluha je provedena kódem simulátoru mimo emulaci. Z hlediska strojového kódu je funkce popsána adresou, na kterou je proveden skok. V JSA jsou tyto adresy reprezentovány symbolickými jmény (symboly). Pokud programátor použije symbol, který není v daném souboru nikde definován, předpokládá se, že adresu tohoto symbolu dodá jiný modul až při linkování, a do objektového souboru je na daném místě vygenerována pouze zástupná hodnota a informace v metadatech⁵.

Řešení realizuje simulované funkce tak, že v adresním prostoru emulace vytvoří blok, tzv. „trampolínu“, v němž jednotlivé adresy reprezentují dostupné funkce. Pro tento blok adres zaregistruje ve frameworku Unicorn obslužnou rutinu, která zachytí skok do něj a spustí příslušný kód simulátoru. Úlohou překladové služby tedy je poskytnout linkeru mapování symbolických jmen simulovaných funkcí na příslušné adresy z bloku „trampolíny“. Konzument překladové služby musí ještě před překladem dodat seznam funkčních simulátorů. Překladová služba jim přidělí adresy a vygeneruje tzv. *linker script* pro linker ld. Ten řídí proces linkování a umožňuje tak explicitně přiřazovat hodnoty symbolům. Vygenerovaný skript je jednoduchý, má např. následující podobu:

```
PROVIDE_HIDDEN(sprintf = 0xff000000);  
PROVIDE_HIDDEN(malloc = 0xff000004);  
PROVIDE_HIDDEN(puts = 0xff000008);
```

Počáteční a koncová adresa cílového bloku adres je součástí konfigurace překladové služby. Informace o přidělených adresách jsou pak uloženy ve výstupním objektu `Executable` překladové služby.

Využití formátu ELF

Výstupem procesu překladu a linkování je spustitelný soubor ve formátu ELF. Ten obsahuje jak binární data programu, tak informace nutné k jeho zavedení (především způsob rozložení programu v paměti). Překladová služba po dokončení linkování výsledný soubor ELF načítá do paměti. K tomu využívá knihovnu `ELFSharp`⁶, která je volně dostupná pod licencí MIT. Podle načtených hlaviček vzniklého ELF souboru vytváří objekt `Executable`. Ten zpracovává dostupné informace do podoby, kterou poté využívá emulační služba pro zavedení programu do paměti emulátoru. V rámci tohoto procesu především připravuje objekty `MemorySegment`, které určují způsob mapování paměti, a určuje počáteční a konečné adresy kódu.

Framework Unicorn a jeho použití v jazyce C#

Emulační framework Unicorn je implementován v jazyce C. Poslední stabilní verze 2.0.0 vyšla v červenci 2022. V práci jsem využíval tuto verzi už v průběhu jejího vývoje, do nějž jsem také drobně přispěl (viz dále). Framework je v jiných programech použit jako knihovna.

⁵Některé překladače mohou vyžadovat explicitní označení takových symbolů jako „externích“, překladač GAS považuje automaticky za externí všechny nedefinované symboly.

⁶ELFSharp: <https://github.com/konrad-kruczynski/elfsharp>.

Překlad

Knihovnu je možné přeložit na systémech Windows, GNU/Linux i macOS, podporovány jsou překladače GCC a MSVC. Při překladu je možné nastavit, které cílové architektury budou v knihovně dostupné, čímž se výrazně sníží její velikost.

Na 64bitovém OS Windows byl zaznamenán problém při běhu služby, kdy volání funkcí knihovny z programu prostředí .NET v některých situacích způsobovalo pád celého programu. Příčina tohoto problému v době psaní práce není známá, otevřel jsem k němu diskuzi v repozitáři projektu Unicorn⁷. Řešením je použití specifické konfigurace překladače knihovny Unicorn, při které tento problém nenastává. Ta je však závislá na použití překladače MSVC, pro účely řešení tedy není v současné době možné překlady pro Windows provést křížově z jiných operačních systémů.

Překlad frameworku v současné době není možný pro Windows na platformě Arm. V prostředích systémů GNU/Linux a macOS nenastaly žádné problémy s překladem ani při běhu služby.

Problémy s frameworkem

Použití frameworku Unicorn znesnadňuje omezený rozsah dokumentace. Úplně chybí popisy specifik jednotlivých emulovaných architektur. Framework umožňuje např. nastavení typu procesoru, podle kterého jsou pak při emulaci dostupná některá rozšíření instrukční sady. Dostupné typy jsou označeny konstantami v hlavičkovém souboru, nikde však není zdokumentováno, jaké chování emulátoru nastavení konkrétního typu způsobí. Pro získání těchto informací je nutné analyzovat zdrojový kód frameworku. Příliš nápomocná není ani dokumentace QEMU, která ve verzi, na níž je Unicorn aktuálně postaven, tyto informace také neobsahovala. Vzhledem k flexibilní povaze architektury Arm, která umožňuje značné rozdíly mezi jednotlivými implementacemi, tak není snadné určit, jaké instrukce zvládne framework skutečně emulovat.

Neočekávané chování nastává při přístupu k některým registrům. API knihovny definuje číselné konstanty reprezentující jednotlivé registry⁸, které jsou poté využity k jejich čtení a zápisu. Nachází se v něm mj. konstanty pro registry S0–S31, D0–D31 i Q0–Q15. Při čtení registrů třídy S nebo Q však framework vracel vždy nulu, zápis také nebyl funkční. Ze zdrojového kódu frameworku jsem později zjistil, že operace s těmito registry nebyly vůbec implementovány, jejich použití tak bylo prázdnou operací, a to bez jakéhokoliv upozornění.

Doplnění podpory pro tyto registry bylo vcelku jednoduché, neboť byl způsob přístupu k těmto hodnotám popsán v komentářích v kódu QEMU. Upravený kód jsem poskytnul také pro zařazení do kódu frameworku v oficiálním repozitáři⁹.

Obdobný problém nastává i s dalšími zdánlivě dostupnými registry, např. s řídicími registry MVFR. Ty ovšem řešení nevyužívá, jejich podporu jsem tak do kódu frameworku prozatím nedoplňoval. Naopak API definuje některé názvy registrů, které se v architektuře ve skutečnosti nenacházejí.

⁷Viz <https://github.com/unicorn-engine/unicorn/issues/1631>.

⁸Hlavičkový soubor `arm.h` s konstantami architektury Arm: <https://github.com/unicorn-engine/unicorn/blob/dev/include/unicorn/arm.h>.

⁹Viz <https://github.com/unicorn-engine/unicorn/pull/1621>.

Použití v jazyce C#

Knihovna je implementována v jazyce C a využití jejího API je snadné, jak demonstruje příklad v kapitole 2.5. Volat funkce z nativních knihoven¹⁰ z kódu v jazyce C# (resp. obecně z programů využívajících běhové prostředí .NET) však není triviální. Programy prostředí .NET nemohou skákat přímo do jiného strojového kódu, neboť by tak (mimo jiné) byla narušena schopnost prostředí spravovat paměť. Interakce s ním umožňuje technika *P/Invoke*¹¹, což je sada API pro předávání dat a toku programu mezi řízeným a nativním kódem.

Typickým přístupem k využívání nativních knihoven z kódu prostředí .NET je tak implementace wrapperu – knihovny, která s využitím API *P/Invoke* a dalších technik jazyka C# pro nízkoúrovňovou práci s pamětí obalí volání nativního kódu, zajistí správu prostředků nativního kódu a poskytne API vhodné pro použití v řízeném kódu prostředí .NET. API wrapperu by mělo být idiomatické, tedy jeho použití by mělo být pro programátory zvyklé na jazyk C# přirozené, zároveň by však mělo ve vhodné míře zachovávat způsob použití původní knihovny.

Pro knihovnu Unicorn jistý wrapper existuje a je součástí jejího repozitáře. Nicméně po prvotním pokusu o jeho využití bylo odhaleno několik vážnějších problémů. Wrapper byl určen pro použití ve staré, nemultiplatformní verzi prostředí .NET. Nenabízí ani zdaleka plnou podporu API knihovny Unicorn a její doplnění by bylo obtížné. Řada jeho funkcí je implementována nekorektně nebo neefektivně (z hlediska operací s pamětí).

Z těchto důvodů jsem se rozhodl navrhnout a implementovat vlastní wrapper pro knihovnu Unicorn. Navržené řešení využívá pokročilých možností současné verze jazyka C# pro práci s pamětí a umožňuje efektivní použití frameworku Unicorn, při kterém nedochází ke zbytečným dynamickým alokacím a kopírování dat. Zpřístupňuje API frameworku téměř v celém jeho rozsahu, a to přívětivým způsobem, který je typově bezpečný a v jazyce C# idiomatický. Nabízí také rozšiřující funkce, např. pro podporu platform s odlišným pořadím ukládání bajtů do paměti. Součástí projektu je také experimentální statický analyzátor kódu, který kontroluje správné použití funkcí pro práci s registry.

Nový wrapper jsem už v době psaní práce samostatně uvolnil jako svobodný software pod názvem SharpCorn¹² a licencí LGPL verze 3. V blízké budoucnosti mám v plánu doplnit podporu ostatních cílových platform a případně jej poskytnout pro zařazení přímo do repozitáře projektu Unicorn.

Emulační služba

Základními úlohami, které emulační služba řeší, je příprava instance frameworku Unicorn, zavedení programu, spouštění emulace a obsluha jejího výsledku. Zavedení programu zahrnuje tvorbu emulované paměti, instrumentačních a ladicích prostředků (obslužných rutin pro různé události, adres ukončující emulaci atp.). Služba zajišťuje správné chování ve značně asynchronním kontextu použití – požadavky, které emulaci řídí, přicházejí nezávisle na aktuálním stavu emulace, a to potenciálně z různých vláken. Emulační služba je implementována ve třídě `ExecutionEngine`, jejíž kontrakt určuje rozhraní `IExecutionEngine`.

¹⁰V kontextu prostředí .NET se používají termíny *managed code* (řízený kód – jeho provádění řídí prostředí .NET) a jako protiklad *unmanaged code* (neřízený kód) nebo také *native code* (nativní kód – v kontrastu s univerzálním mezikódem, do kterého jsou .NET aplikace překládány).

¹¹Viz např. <https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke>, cit. 21. 6. 2022.

¹²Repozitář projektu SharpCorn: <https://github.com/ondryaso/sharpcorn>.

Příprava paměti

Příprava frameworku a emulované paměti je vcelku přímočará. Rozvržení paměti je určeno linkerem, uloženo v souboru ELF a připraveno pro službu ve vstupním objektu `Executable`. Ten obsahuje seznam segmentů k mapování, jejichž počáteční adresy i velikosti jsou zarovnány na 4KiB hranice, jak vyžaduje framework. Služba pak volá funkci `uc_mem_map`, která zajistí vytvoření přístupného bloku paměti v adresním prostoru emulovaného programu. Pro hostitele je tento proces transparentní (nemusí jít o přímé mapování souvislého bloku paměti hostitele do emulovaného adresního prostoru).

Při přípravě paměti je nutné vytvořit také několik bloků, které spustitelným souborem určeny nejsou: zásobník, prostor pro dynamickou alokaci paměti (*heap*) a „trampolínu“ pro simulované funkce, což je prostor naplněný instrukcemi návratu. Řešení umožňuje v konfiguraci nastavit chování při umísťování zásobníku, který může být umístěn náhodně nebo na konkrétní adrese. Náhodná adresa se volí tak, aby nedošlo ke kolizi s jiným segmentem paměti. Je možné také ovlivnit obsah zásobníku po spuštění či restartu programu. Zásobník může být naplněn náhodnými daty nebo vynulován, případně je možné zachovat data z předchozího spuštění. Všechny tyto činnosti zajišťuje metoda `MakeStackSegment`.

Příprava paměti je provedena ve dvou krocích. V prvním, po načtení programu, je paměť namapována. Ve druhém, před spuštěním, jsou do ní zkopírována data ze souboru ELF. Tím je zajištěno, že mapování paměti není zbytečně prováděno znovu, když je emulovaný program pouze resetován.

Příprava registrů

Po přípravě paměti jsou upraveny také hodnoty registrů. Uživatel může v konfiguraci určit, jestli budou jejich hodnoty vynulovány, nebo nastaveny náhodně. Důležité je nastavení správné adresy vrcholu zásobníku a také nastavení některých konfiguračních registrů emulovaného procesoru tak, aby byly povoleny SIMD/FP instrukce. Konkrétně je nutné nastavit příznaky v systémových registrech CPACR a FPEXC. Pak je procesor nastaven do operačního módu *User* zápisem do registru CPSR. Až poté je možné zapsat adresu zásobníku (ta je totiž typicky pro každý operační mód jiná).

Instrumentační prostředky

Při vytvoření instance frameworku Unicorn zavádí emulační služba obslužné rutiny pro události přerušení, neplatné instrukce a neplatné paměťové operace kdekoliv v celém adresním prostoru. Jde o jednoduché funkce, které příslušně nastaví v emulační službě příznak příčiny ukončení a následně ukončí emulační krok. Při aktivaci striktního režimu jsou přidány obdobné rutiny také pro událost jakéhokoliv přístupu do paměti v místech, do kterých by program přístup mít neměl.

Další významnou událostí, kterou je nutné instrumentovat, je skok do paměťového prostoru „trampolíny“ pro simulované funkce. Jde o část paměti, která je naplněná instrukcemi návratu (BX LR). Při jejím vytvoření je pro její adresní prostor zavedena také obslužná rutina (*code hook*), kterou framework Unicorn spouští před každým provedením instrukce. V této rutině je nalezen odkaz na metodu simulátoru, která je zaregistrována pro danou adresu instrukce, a tato metoda je spuštěna. Po jejím dokončení je řízení předáno zpět frameworku, který provede právě instrukci návratu.

Důsledkem interních mechanismů frameworku Unicorn je, že po ukončení emulace není zaručena správnost hodnoty programového čítače. To je ovšem pro účely ladění značně ne-

praktické, neboť jsou tak uživateli podávány nesprávné informace o tom, kde bylo spouštění zastaveno. Řešením je pro všechen spustitelný kód přidat instrumentaci (*code hook*), která je spuštěna po provedení každé instrukce kódu. Před spuštěním obslužné rutiny framework automaticky obnoví správnou hodnotu PC. Tatáž rutina je použita pro sledování změn registrů při použití datových breakpointů. Nevýhodou je, že takové řešení zpomaluje chod emulovaného kódu¹³. Při rozsahu programů, který je ve výuce očekávatelný, jsou však tyto dopady zanedbatelné.

Pomocí instrumentačních prostředků jsou ošetřeny také logpointy. Pro příslušné adresy je přidán *code hook*, který je spuštěn při vykonání dané instrukce. Jeho obsluha pouze zašle DAP požadavek *Output* s nastavenou zprávou (viz obr. 5.2b).

Spouštění emulace a obsluha výsledku

Samotná emulace programu má charakter cyklu, který je naznačen na obrázku 5.3. Proces začíná příchodem jednoho ze startovacích požadavků. Pro zajištění konzistence dat a chování v inherentně asynchronním prostředí (požadavky přicházejí nahodile a z potenciálně různých vláken) jsou obsluhy požadavků považovány za kritickou sekci, která je chráněna semaforem. Prvním krokem obsluhy je tedy kontrola aktuálního stavu (např. požadavek *Continue* může být proveden, jen když je emulace pozastavena) a obsazení semaforu. Po obsazení semaforu je provedena další kontrola stavu, který se mohl mezi obsazením a předchozí kontrolou změnit.

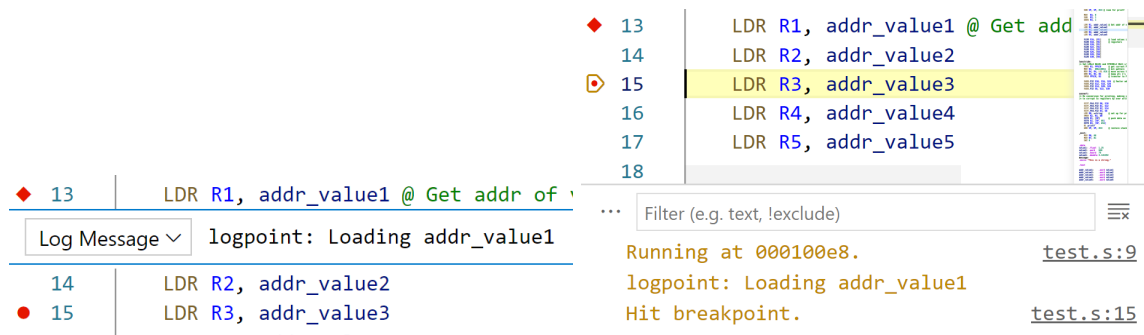
V případě požadavku *Launch* (při spuštění emulace) je program zaveden do paměti. U všech startovacích požadavků následuje nastavení ukončovacích podmínek. Framework Unicorn umožňuje ukončit emulaci, když programový čítač nabude jedné z určených hodnot (adres). Průběh emulace je takto přerušen na uživatelem nastavených breakpointech a také na adresách, které jsou v kódovém segmentu, ale neobsahují programová data (takové adresy jsou označeny speciálním symbolem `$d`, což provádí automaticky překladač). V rámci ošetření výsledku emulace se kontroluje, zda aktuální hodnota PC odpovídá nějakému z breakpointů. Pokud ano, zašle služba příslušnou událost DAP, což se projeví v UI editoru, viz obr. 5.2b.

Při pokračování v běhu po zastavení na breakpointu je nutné uvažovat, že je instrukce nastavena jako ukončovací a emulace se na ní tedy hned znovu zastaví. Řešením je dočasně tento breakpoint odebrat, provést emulaci pouze jedné instrukce, poté breakpoint znovu obnovit a až poté spustit další emulační krok.

Spuštěn je také časovač, který po nastavené době přeruší emulaci (aby špatně napsané programy, které uvážnou ve smyčce, zbytečně nezatěžovaly službu). Maximální dobu emulace je možné nastavit v konfiguraci spouštění. Tuto uživatelsky nastavitelnou hodnotu je možné omezit v konfiguraci běhové služby.

Po této přípravě, která je pro každý typ startovacího požadavku odlišná, následuje vždy volání metody `StartEmulation`, jejíž tělo je označeno jako emulační krok. Metoda volá API frameworku Unicorn pro spuštění emulace (`uc_emu_start`). Tato funkce blokuje provádění vláken až do ukončení emulace, proto je nutné celý emulační krok spustit v samostatném vlákne. Emulace je ukončena dosažením nějaké z ukončovacích adres, výjimkou v emulovaném kódu nebo zavoláním příslušné funkce API (`uc_emu_stop`), což je provedeno při přijetí DAP požadavku *Pause*.

¹³Viz <https://github.com/unicorn-engine/unicorn/blob/master/docs/FAQ.md>. Obsluha tohoto druhu instrumentace není triviální, v tomto případě mají navíc negativní výkonnostní dopady i časté přechody mezi nativním kódem frameworku a kódem prostředí .NET.



(a) Nastavení breakpointu a logpointu s vlastní zprávou. (b) Chování po spuštění. Při průchodu logpointem je v konzoli vypsána vlastní zpráva, program je zastaven až na breakpointu.

Obrázek 5.2: Breakpointy a logpointy.

Po ukončení emulace zjistí služba, jakému řádku ve zdrojovém kódu odpovídá poslední hodnota PC, aby mohla tuto informaci poskytnout klientovi. Součástí stavu služby je příznak příčiny ukončení, který je nastavován obslužnými rutinami nestandardních událostí (např. neplatného čtení z paměti) spolu s dodatečnými informacemi (např. neplatnou adresou). Podle hodnoty tohoto příznaku zasílá služba klientovi příslušné notifikace protokolu DAP. Nakonec je signalizován semafor, služba vystupuje z kritické sekce a čeká na další požadavek.

Emulační služba prochází během emulačního kroku různými stavy, které uchovává v příznaku **State**. Přechody mezi jednotlivými stavy v průběhu emulačního kroku znázorňuje stavový diagram v příloze I. V diagramu se nenachází počáteční stav **Unloaded**, ve kterém je služba před zavedením programu.

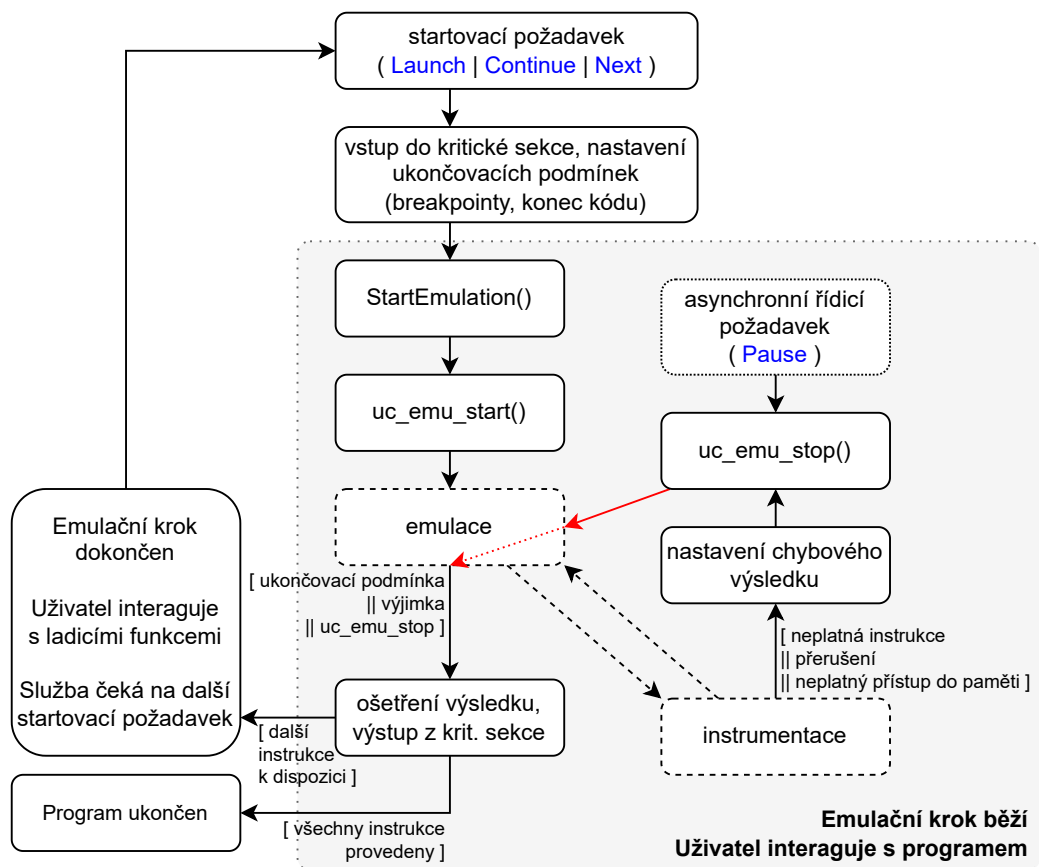
Asynchronní notifikace

Součástí protokolu DAP jsou, kromě požadavků a odpovědí na ně, také události (*Events*), které zasílá ladicí adaptér asynchronně klientovi. Protože emulační služba implementuje rozhraní DAP a je navržena tak, aby byla použitelná nezávisle na zbytku řešení, bylo nutné zvolit vhodný způsob, jakým události konzumentovi služby zasílat.

Využit byl behaviorální návrhový vzor **prostředník** (*Mediator*) [11, str. 273], a to v implementaci knihovnou MediatR¹⁴. Ten v zásadě funguje tak, že mezi komunikující objekty vkládá další objekt (prostředníka), který obstarává jejich komunikaci. V tomto případě emulační službě dodává odkaz na objekt prostředníka právě její konzument, a to v konstruktoru. Po vzniku události předá emulační služba prostředníkovi objekt popisující událost, tím pro ni její obsluha končí. Konzument si tak vhodným nastavením prostředníka může sám definovat, jak mají být události zpracovány (v případě řešení je konzumentem běhová služba, která události přeposílá klientům pomocí spojení SignalR).

Bylo možné použít i jiné návrhové vzory, např. vzor pozorovatel. Prostředník byl zvolen proto, že knihovna MediatR umožňuje výbornou integraci do systému vkládání závislostí, na kterém je postaven ASP.NET, nosná technologie běhové služby. Na stejném vzoru je navíc do značné míry postaven také jazykový server.

¹⁴MediatR: <https://github.com/jbogard/MediatR>.



Obrázek 5.3: Emulační cyklus. Modře jsou označeny požadavky DAP.

Ladicí služba

Většina metod protokolu DAP, které slouží k poskytování ladicích funkcí, je umístěna mimo samotnou emulační službu, a to ve třídě `DebugProvider` (ladicí služba). Její instanci vytváří emulační služba při svém vzniku a poskytuje ji jako svou vlastnost typu `IDebugProvider` (rozhraní udávající kontrakt ladicí služby). Implementuje rozsáhlý systém proměnných, výrazů a trasování (sledování změn registrů a paměti pro podporu datových breakpointů). Seznam rozšiřujících schopností (*capabilities*) protokolu DAP, které ladicí služba poskytuje, je uveden v příloze E.

Využití rozhraní protokolu DAP

Třídy, které reprezentují objekty definované protokolem DAP (tzv. modely), byly převzaty z projektu `OmniSharp C# Language Server Protocol`¹⁵, který je produktem sdružení .NET Foundation a je dostupný pod licencí MIT. Jsou v nich odstraněny nepotřebné části a jsou aktualizovány pro novější verzi protokolu. V rámci úprav byl využit také kód projektu `StringEnum`¹⁶, který je poskytován pod *public domain* licencí Unlicense. Převzatý kód je oddělen a označen v projektu `Code4Arm.ExecutionCore.Protocol`.

¹⁵Repozitář projektu: <https://github.com/OmniSharp/csharp-language-server-protocol>.

¹⁶Repozitář projektu: <https://github.com/gerardog/StringEnum>.

Využití systému proměnných protokolu DAP

V kapitole 4.1.3 byl představen systém proměnných, který v editoru VSC slouží k reprezentaci dat a stavu laděného programu. Tento koncept přirozeně využívá i protokol DAP. Proměnné jsou chápány jako stromová struktura, která je vázána pomocí číselných identifikátorů, tzv. **referencí proměnných** (*variables reference*, dále jen RP). Pro získání stromu klient použije požadavek *Scopes*, který vrátí seznam **rámců proměnných**. Rámce jsou v tomto případě např. registry, stav CPU nebo zásobník. Vlastností každého rámce je jeho RP. Pokud uživatel daný rámec v UI editoru rozbalí, vytvoří klient požadavek *Variables*, jehož parametrem je RP rozbaleného rámce. Server vrátí objekty *Variable*, které nesou zejména název a hodnotu proměnné (v obou případech jde o text, jejich obsah a význam je pro editor transparentní). Pokud má proměnná další potomky, součástí objektu *Variable* je také její RP. Tento proces je naznačen v diagramu v příloze H.

```

R3: 4294843346
├── signed bytes
├── unsigned bytes
│   ├── [0]: 210
│   ├── [1]: 27
│   ├── [2]: 254
│   └── [3]: 255
├── chars
├── signed shorts
│   ├── [0]: 7122
│   └── [1]: -2
├── unsigned shorts
│   ├── [0]: 7122
│   └── [1]: 65534
├── signed int: -123950
├── unsigned int: 4294843346
└── float: NaN
    ├── sign: 1 (-)
    ├── exponent: 11111111 => inf/NaN
    └── mantissa: 11111100001101111010010 => 8264658
    
```

(a) Kromě hodnoty celého registru jsou zobrazeny také znaménkové interpretace, rozdělení registru na části o menších datových šířkách a interpretace hodnoty v kódování *binary32*.

```

R3: 0xffffe1bd2
├── signed bytes
│   ├── [0]: -0x2e
│   ├── [1]: 0x1b
│   ├── [2]: -0x2
│   └── [3]: -0x1
├── unsigned bytes
│   ├── [0]: 0xd2
│   ├── [1]: 0x1b
│   ├── [2]: 0xfe
│   └── [3]: 0xff
├── signed int: -0x1e42e
└── unsigned int: 0xffffe1bd2
    
```

(b) Hexadecimální zobrazení hodnot.

```

R3: 111111111111111100001101111010010
├── signed bytes
│   ├── [0]: -00101110
│   ├── [1]: 00011011
│   ├── [2]: -00000010
│   └── [3]: -00000001
├── unsigned bytes
│   ├── [0]: 11010010
│   ├── [1]: 11011
│   ├── [2]: 11111110
│   └── [3]: 11111111
├── signed int: -000000000000000011110010000101110
└── unsigned int: 1111111111111100001101111010010
    
```

(c) Binární zobrazení hodnot.

Obrázek 5.4: Zobrazení hodnoty obecného registru a její různé interpretace jako *n*-tice menšího datového typu.

V systému se vyskytuje řada typů proměnných (např. registrová proměnná, interpretace 32b proměnné jako znaménkového čísla nebo exponent IEEE 754 kódování), jak naznačuje obrázek 5.4 a další obrázky v příloze N.1. V kódu je každý z nich reprezentován samostatnou třídou, která implementuje rozhraní *IVariable*. To předepisuje zejména název, kolekci potomků a metody *Evaluate*, *Get* a *Set*. Tyto třídy ve svém konstruktoru ihned vytvoří i své potomky, pokud nějaké mají mít. Tímto způsobem jsou od sebe chování jednotlivých typů proměnných jasně oddělena, objektový model zároveň poskytuje stromovou strukturu.

Jako RP jsou použita bitová pole, ve kterých jednotlivé skupiny bitů představují typ proměnné a různé hodnoty, které jednoznačně identifikují konkrétní proměnnou (např. ID registru nebo dvojice [ID registru; datový typ položek, na které je registr rozdělen]). Stromová struktura konkrétní proměnné se vytvoří jen jednou, typicky při prvním požadavku na daný rámec, načež zůstává uložena v paměti obvykle až do zrušení objektu ladicí služby. Každá proměnná s potomky je přístupná pod svým jedinečným RP. Ladicí služba si udržuje slovník, ve kterém je stromová struktura linearizována. Všechny objekty realizující proměnné jsou v něm tak vyhledatelné pomocí jejich RP.

Mnoho typů proměnných je inherentně pouze pohledem na jejich rodičovské proměnné, který bez rodiče nemůže existovat. Potomci tak mohou využívat naposled získanou a uloženou hodnotu rodiče. Například při vyhodnocení proměnné reprezentující registr je hodnota čtena z frameworku Unicorn pouze jednou, i když s ní pracuje velké množství potomků, což celé řešení zrychluje.

Vyhodnocování výrazů

Vyhodnocování uživatelem zadaných výrazů představené v kapitole 4.1.3 úzce souvisí s principem fungování proměnných. Výrazem je v kontextu DAP textový řetězec (jeho syntaxi i sémantiku určuje ladicí adaptér), který může být vyhodnocován v různých kontextech: mj. v podokně *Watch* (viz obr. 5.5), po zadání do ladicí konzole (*REPL – Read-Eval-Print Loop*, tj. cyklus vstup-vyhodnocení-výstup, viz obr. 5.5e) nebo jako výraz ve zdrojovém kódu, nad kterým uživatel podržel kurzor myši (*hover*, viz obr. 5.5c). Klient zasílá pro vyhodnocení požadavek *Evaluate*, přičemž formát odpovědi je podobný jako u proměnných, výsledky vyhodnocení jen nemají jméno. S proměnnými ovšem sdílí především princip získávání potomků, kdy výsledek vyhodnocení může obsahovat vlastní referenci proměnné (RP), pomocí které klient může získat potomky vyhodnoceného výrazu požadavkem *Variables*, jako by šlo o běžnou proměnnou.

Je zřejmé, že spolu systém výrazů a proměnných musí spolupracovat. Výrazy, které jsou podporovány, obecně vedou buď na existující proměnnou (všechny registrové výrazy), nebo na novou instanci nějakého typu proměnné (např. adresa symbolu nebo paměťový výraz). Díky tomu lze třídy reprezentující jednotlivé typy proměnných použít i pro výrazy, nutně bylo pouze implementovat algoritmus pro vytvoření příslušného objektu proměnné pro daný textový vstup.

Metody pro vyhodnocování výrazů jsou součástí třídy *DebugProvider*, ale pro usnadnění orientace v kódu jsou odděleny v samostatném souboru *ExpressionEvaluator.cs*. Syntaxi podporovaných výrazů popisuje gramatika v příloze D. Tato gramatika je regulární a jazyk, který generuje, je tedy možné rozpoznat regulárním výrazem, čehož řešení využívá.

Problematickým se zde ukázal návrh systému proměnných, ve kterém objekt reprezentující proměnnou při svém vzniku sám vytváří také všechny své potomky. V porovnání s proměnnými poskytují výrazy dynamičtější pohled na data. Použitím typu a specifika-toru zobrazení je možné různými způsoby ovlivnit potomky výrazu. Dosažení tohoto chování ovšem není při aktuální implementaci proměnných možné, protože kvůli způsobu přidělování RP může existovat pouze jeden objekt reprezentující požadovanou proměnnou a její podstrom. Tento problém částečně řeší např. zavedení speciálního typu proměnné, který obaluje jiný objekt proměnné a umožňuje dynamicky měnit své potomky. Toto řešení je ale omezené pouze na jednu úroveň stromu.

```

▼ WATCH
  R0: 0x123456ff
  ▼ (short)R0: 0x123456ff
    ▼ signed shorts
      [0]: 0x56ff
      [1]: 0x1234
    (byte)R0[2]: 0x34
    (byte)R0[2]:b: 110100
    (byte)R0[2]:d: 52
  ▼ (float)R0:ieee: 5.6905216E-28
  ▼ float: 5.6905216E-28
    sign: 0 (+)
    exponent: -91 (00100100)
    mantissa: 3430143 (01101000101011011111111)

```

(a) Výrazy s obecnými registry.

```

▼ WATCH
  D0: 0x3ff4000000000000
  S1: 0x3ff40000
  ▼ (uint)D0: 0x3ff4000000000000
  ▼ unsigned ints
    [0]: 0x0
    [1]: 0x3ff40000
  ▼ (float)D0: 1.25
  ▼ floats
    [0]: 0
    [1]: 1.90625
  ▼ (double)D0:ieee: 1.25
    sign: 0 (+)
    exponent: 0 (01111111111)
    mantissa: 1125899906842624 (010000...

```

(b) Výrazy se SIMD/FP registry.

```

0x20178
> signed bytes
> unsigned bytes
signed int: 0x20178
unsigned int: 0x20178
Hold Alt key to switch to editor language hover
LDR R1, addr_value1 @ Get addr of values

```

(c) Při najetí kurzorem myši nad řetězec „R1“ v kódu je zobrazena odpovídající proměnná.

```

▼ WATCH
  [R0]: 0x53
  (int)[R0]: 0x20303153
  (string)[R0]: S10 is %f
  (byte)[R0, 1]: 0x31
  (byte)[R0, 1, LSL 1]: 0x30
  (byte)[R0, R1, ROR 1]:b: 1111100
  (float)[value1]: 1.25
  (double)[value1, +10]: 3.141592
  (float)[value4, -10]: 1.25
  (double)[value4]: 3.141592
  &value1: 0x20178
  &value4: 0x20182

```

(d) Paměťové výrazy.

PROBLEMS	OUTPUT	DEBUG CONSOLE
→ R0	0x123456ff	
→ (byte) R0[2] =	0x0	
→ R0	0x120056ff	
→ APSR.N	0	
→ CPSR.M	User	
→ (short) R0[2]	Invalid subtype index.	

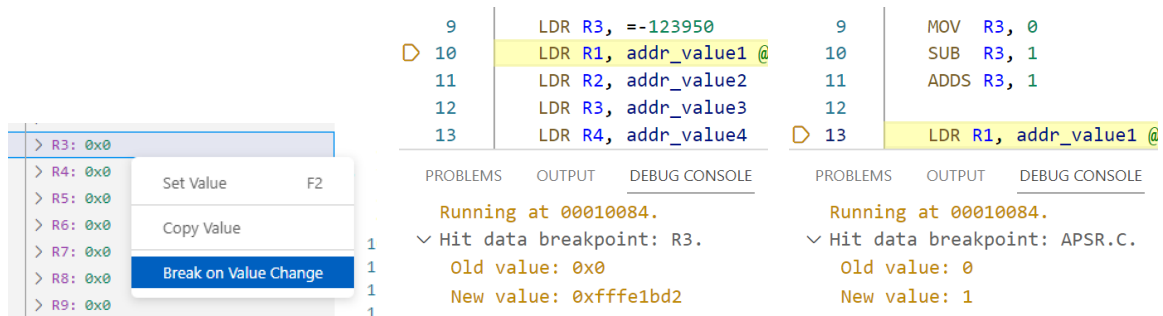
(e) Vyhodnocování v konzoli. Řádky označené šipkou jsou uživatelským vstupem. Hodnoty výrazů lze také nastavovat.

Obrázek 5.5: Vyhodnocování výrazů pomocí okna *Watch*, v kódu (*hover*) a v ladicí konzoli (REPL).

Sledování změn registrů a paměti (datové breakpointy)

Ladicí služba umožňuje sledovat změny hodnoty jakékoliv proměnné nebo výrazu a pozastavit při nich provádění programu. V kódu je pro takto sledovatelné položky využít termín *traceable* (sledovatelný, trasovatelný). Z hlediska implementace dává smysl použít pro trasování návrhový vzor **pozorovatel** (*Observer*) [11, str. 293]. Trasovatelné objekty (proměnné) jsou zde producenty zpráv o změně hodnoty, pozorovatelem je pak především ladicí služba samotná a v některých případech také potomci těchto proměnných.

Trasovatelné proměnné jsou v zásadě dvojího druhu: paměťové a registrové. Paměťové proměnné je možné trasovat pomocí instrumentačních prostředků frameworku Unicorn tak, že je na příslušnou adresu v paměti přidána obslužná rutina pro zápis, jejíž aktivaci zajistí framework. V tomto případě je pouze nutné v rámci ošetření výsledku emulačního kroku posunout PC o jednu instrukci dál, jinak by další spuštění emulace mohlo být zastaveno na stejné instrukci zápisu do paměti. Pro sledování změn stavu registrů však Unicorn žádné prostředky nenabízí. Využita je zde skutečnost, že je při běhu instrumentováno provádění každé instrukce. Ladicí služba během něj tedy prochází zaregistrované datové breakpointy tohoto typu a na každém volá funkci `TraceStep`, která načte aktuální hodnotu registru, porovná ji s předchozí zaznamenanou hodnotou a případnou změnu ohlásí pozorovatelům.



(a) Nastavení datového breakpointu.

(b) Při změně hodnoty je program pozastaven.

(c) Sledovat je možné také příznakové registry.

Obrázek 5.6: Nastavování a funkce datových breakpointů. Po provedení instrukce, která mění sledovaný registr, je program zastaven a informace o předchozí i nové hodnotě jsou vypsané do konzole.

Disassembling pomocí knihovny Capstone

Pro podporu zpětného převodu strojového kódu do JSA je využita knihovna Capstone¹⁷, která je dalším projektem autora frameworku Unicorn. Je implementována v jazyce C s dostupnými knihovnami pro jiné jazyky, které zprostředkovávají její funkce prostřednictvím tzv. *bindings*. Knihovna je volně dostupná pod licencí BSD. Pro volání z jazyka C# je použita knihovna (wrapper) Capstone.NET¹⁸, jejímž autorem je Ahmed Garhy a je volně dostupná pod licencí MIT.

¹⁷Knihovna Capstone: <https://github.com/capstone-engine/capstone>.

¹⁸Knihovna Capstone.NET: <https://github.com/9ee1/Capstone.NET>.

44:	VCVT.F64.F32 D1, S12			37	VADD.F32 S8, S22, S30
0x100e4	EE B7 1A C6	vcbt.f64.f32 d1, s12		38	
45:	VCVT.F64.F32 D2, S14			39	convert:
0x100e8	EE B7 2A C7	vcbt.f64.f32 d2, s14		40	/* Do conversion for printing,
46:	VCVT.F64.F32 D3, S8			41	/* to corrupt Sx registers by c
0x100ec	EE B7 3A C4	vcbt.f64.f32 d3, s8		42	
47:	LDR R0, =string @ set up for printf			43	VCVT.F64.F32 D0, S10
0x100f0	E5 9F 00 68	ldr r0, [pc, #0x68]		44	VCVT.F64.F32 D1, S12
48:	VMOV R2, R3, D0			45	VCVT.F64.F32 D2, S14
0x100f4	EC 53 2B 10	vmov r2, r3, d0		46	VCVT.F64.F32 D3, S8
49:	VSTR D1, [SP] @ push data on stack			47	LDR R0, =string @ set u
0x100f8	ED 8D 1B 00	vstr d1, [sp]		48	VMOV R2, R3, D0
50:	VSTR D2, [SP, #8]			49	VSTR D1, [SP] @ push
0x100fc	ED 8D 2B 02	vstr d2, [sp, #8]		50	VSTR D2, [SP, #8]
51:	VSTR D3, [SP, #16]			51	VSTR D3, [SP, #16]
0x10100	ED 8D 3B 04	vstr d3, [sp, #0x10]		52	BL printf
52:	BL printf			53	ADD SP, SP, #24 @ restc
0x10104	EB 00 00 19	bl #0x10170		54	
53:	ADD SP, SP, #24 @ restore stack			55	_exit:
0x10108	E2 8D D0 18	add sp, sp, #0x18		56	MOV R0, #0
56:	MOV R0, #0			57	MOV R7, #1
0x1010c	E3 A0 00 00	mov r0, #0		58	SWI 0
57:	MOV R7, #1			59	
0x10110	E3 A0 70 01	mov r7, #1		60	.data
58:	SWI 0			61	value1: .float 1.25
0x10114	EF 00 00 00	svc #0		62	value2: .word 666

Obrázek 5.7: Okno *Disassembly* se strojovým kódem zpětně převezeným do JSA.

Mapování adres na řádky kódu pomocí ladicích dat DWARF

Důležitou schopností ladicího nástroje je obousměrný převod mezi adresou instrukce v paměti a polohou této instrukce ve zdrojovém souboru. Mapování polohy v souboru na adresu je využito při nastavování breakpointů, v opačném směru pak např. při krokování nebo pozastavení programu. Pro jeho dosažení využívá řešení ladicí informace ve formátu DWARF.

DWARF označuje formát souboru, který nese informace sloužící k podpoře ladění na úrovni zdrojového kódu. V zásadě jde o stromovou strukturu, do které překladač ukládá různé informace o typech, datech a funkcích, které se ve zdrojovém souboru vyskytují, a kromě nich právě také informace o vztahu adres a zdrojových souborů. V kontextu JSA je DWARF užitečný právě pro získávání těchto informací, které v tomto formátu generuje překladač GAS a ukládá je do výstupního objektového souboru ELF (pokud je použit přepínač `-g`).

Ke čtení informací o řádcích kódu z dat ve formátu DWARF je využita malá část zdrojového kódu projektu SharpDebug¹⁹, jehož autorem je Vuk Jovanovic. Kód je dostupný pod licencí MIT v repozitáři projektu. Původní kód projektu SharpDebug byl modifikován a zredukován o části, které nejsou pro práci relevantní. Naopak bylo přidáno vlastní rozhraní pro jeho využití. Převzatý kód se nachází v odděleném projektu `Code4Arm.ExecutionCore.Dwarf`.

Odhadování datového typu symbolů

Jako proměnné jsou zobrazeny také symboly z datové sekce. To však není úplně přímočaré, neboť symbol je pouhým zástupným jménem pro nějakou hodnotu (typicky adresu), nenese tedy žádnou informaci o datovém typu. Při ladění ve vyšších programovacích jazycích se pro určení typů proměnných, které často odpovídají symbolům, používají dodatečné ladicí informace (např. ve formátu DWARF). Ty ovšem u kódu napsaného v JSA nejsou dostupné, neboť JSA z principu své nízkoúrovňovosti s pojmy datový typ nebo proměnná v podstatě

¹⁹Repozitář projektu SharpDebug: <https://github.com/southpolenator/SharpDebug>.

nepracuje. Uživatel by tedy musel ladicí informace v příslušném formátu sám vytvořit. Řešení tedy datový typ symbolů odhaduje podle direktivy, která následovala za definicí symbolu návěštím (např. pro kód `value: .word 16` je odhadnutým typem 32b celé číslo).

Informace o direktivách emitujících data jsou získány z výstupu překladače (*listing*). Pro určení skutečné adresy symbolu je však nutné počítat i s jeho přemístěním při linkování. Tyto informace jsou získány z výstupu linkeru. Překladačová služba oba výstupy zaznamenává a požadované informace z nich extrahuje při přípravě objektu `Executable`.

Běhová služba

Běhová služba spravuje spojení s klienty a běhová jádra, která pro klienty vytváří. Zajišťuje obousměrné předávání požadavků DAP. Má dva režimy fungování: lokální a vzdálený. Zásadní rozdíl mezi nimi je v době života a způsobu správy souborů. V obou případech komunikuje běhová služba s rozšířením výhradně pomocí technologie SignalR.

V lokálním režimu spouští rozšíření přímo na počítači uživatele instanci služby, a to těsně před začátkem ladění. Po dokončení programu je proces služby ukončen. Nedochází zde tedy k žádnému udržování sezení²⁰. Lokálně běžící instance služby přistupuje ke zdrojovým souborům přímo v souborovém systému, není nutné provádět žádnou synchronizaci mezi editorem a službou.

Ve vzdáleném režimu je služba spuštěna dlouhodobě typicky mimo systém uživatele. Editor před prvním spuštěním ladění žádá o vytvoření sezení, v rámci kterého služba udržuje soubory projektu a spojení s editorem až do ukončení editoru. Před každým spuštěním ladění je provedena synchronizace souborů. Editor oznámí seznam souborů z konfigurace spouštění a jejich verze, služba seznam porovná s aktuálním stavem sezení a vyžádá si soubory, jejichž verze se změnila. Pokud se žádné zdrojové soubory nezměnily, využije předchozí přeložený program, což celé řešení zrychluje. Po spuštění ladění zajistí ladicí adaptér jeho připojení do existujícího sezení. Aktuální implementace zdrojové soubory ukládá do dočasného umístění v souborovém systému, které je po ukončení sezení odstraněno.

Konfigurace služby a běhových jader

Běhová jádra poskytují celou řadu konfiguračních možností pro všechny své části (překlad, linkování, spouštění a ladění), viz příloha B. Pro určení konfigurace jednoho konkrétního jádra je využit jednoduchý systém **vrstev**. Výchozí hodnoty konfiguračních klíčů jsou přečteny z konfiguračního souboru běhové služby `appsettings.json`. Uživatel pak může tyto hodnoty měnit jak v rámci konfigurace editoru VSC, tak i v konkrétní konfiguraci spouštění. Konfiguraci editoru zasílá rozšíření službě jednou pro každé sezení, případně znovu po její změně. Konfigurace spouštění se posílá při každém spuštění ladění. Před spuštěním jsou pak na výchozí hodnoty postupně jednotlivé vrstvy aplikovány. Některá nastavení ladicích nástrojů je možné měnit i při běhu programu, čehož je dosaženo tak, že rozšíření registruje v editoru pozorovatele události změny konfigurace a v reakci na tuto událost zasílá službě požadavek na změnu.

²⁰Uvnitř služby se stále s objektem představujícím sezení pracuje, jeho životnost je ale omezena na dobu trvání spojení s ladicím adaptérem.

Ošetřování výjimek

Běhové jádro oznamuje některé chybové stavy prostřednictvím mechanismu výjimek jazyka C#. Chybovými stavy se zde rozumí reakce na neplatné uživatelské vstupy (např. chybný výraz k vyhodnocení). Výjimky vzniklé při běhu v emulovaném programu jsou ošetřovány v rámci emulačního kroku a oznamovány službou pomocí příslušných prostředků protokolu DAP.

Běhová služba všechny výjimky zachytává jednotným způsobem prostřednictvím **filtru** požadavků SignalR. Je to prostředník vložený mezi příjem SignalR zprávy a její zpracování, který obaluje volání obslužné metody požadavku a pozměňuje odpověď. V tomto případě je volání zabaleno do běžného **try-catch** bloku, v případě zachycení výjimky je vytvořena chybová zpráva protokolu DAP, která obsahuje bližší informace podle typu výjimky.

5.2 Rozšíření pro Visual Studio Code

Rozšíření pro editor VSC zajišťuje především spouštění jazykového serveru a komunikaci s běhovou službou, případně její spouštění v lokálním režimu. Součástí rozšíření je také jednoduchý ladicí adaptér, který předává požadavky protokolu DAP mezi editorem a službou. Rozšíření zajišťuje zasílání konfigurace běhové službě. Dále jsou přímo v rozšíření implementovány některé funkce pro podporu při vývoji: automatické vytváření konfigurací ladění, načítání a zobrazování dokumentace instrukcí a zobrazování podmínkových příznaků NZCV při ladění. Rozšíření deklarativním způsobem definuje konfigurační klíče všech částí řešení, což umožňuje jednoduchou konfiguraci pomocí GUI editoru. Obsahuje deklarativní popis syntaxe jazyka, díky kterému je základní zvýrazňování syntaxe dostupné ještě před spuštěním jazykového serveru nebo při jeho nedostupnosti.

Spouštění serverů

Protože jsou jazykový server i lokální běhová služba distribuovány spolu s rozšířením jako binární soubor pro běhové prostředí .NET, musí rozšíření zajistit, aby bylo v počítači toto prostředí nainstalováno. Původně k tomu mělo být využito rozšíření *The .NET Install Tool for Extension Authors*²¹, které zajišťuje přítomnost požadované verze a umožňuje zjistit cestu k jeho spustitelnému souboru. Toto rozšíření však neumožňuje stáhnout prostředí s podporou pro ASP.NET a SignalR. Proto byla část kódu rozšíření, která zajišťuje samotný proces stahování a konfigurace prostředí, drobně modifikována a integrována do kódu řešení. Převzatý kód je volně dostupný pod licencí MIT a je umístěn v adresáři `vscode-extension/src/vscode-dotnet-runtime-library`.

O spouštění jazykového serveru, komunikaci s ním a jeho ukončování se automaticky stará knihovna `vscode-languageclient`. Rozšíření jí pouze předává cestu ke spustitelnému souboru a správné parametry příkazové řádky. Spouštění lokální instance běhové služby rozšíření zajišťuje vlastním kódem pomocí API běhového prostředí Node.js. Před každým spuštěním najde pomocí knihovny `portfinder` volný port, na kterém bude služba naslouchat a ke kterému se ladicí adaptér v rozšíření připojí.

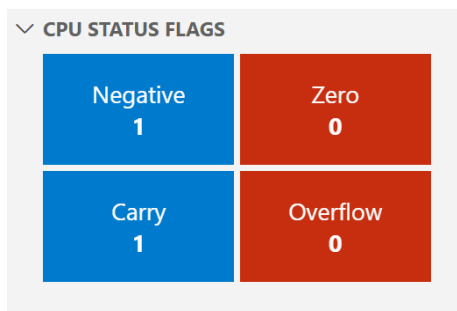
²¹Repozitář rozšíření The .NET Install Tool for Extension Authors: <https://github.com/dotnet/vscode-dotnet-runtime>.

Podokno s podmínkovými příznaky

API editoru VSC umožňuje přidávat podokna, ve kterých je zobrazen prakticky libovolný HTML obsah (*webviews*). Při jejich implementaci je však nutné počítat s jistými specifiky jejich chování, které se týkají použitých externích prostředků (např. stylů), skriptů a životního cyklu obsahu. Jakékoliv změny uvnitř obsahu okna jsou při jeho skrytí zahazeny, a okno tedy musí pomocí skriptu uchovávat svůj stav. Skripty v nich nemohou využívat API editoru a se svým mateřským rozšířením komunikují výhradně pomocí zasílání zpráv serializovatelných do formátu JSON. Je nutné dbát na vhodnou integraci vlastních podoken do uživatelského rozhraní editoru ve smyslu stylování a barev.

API editoru neumožňuje rozšířením při ladění přímo číst aktuální hodnoty proměnných, umožňuje však přidat třídu sledující komunikaci mezi editorem a ladicím adaptérem (*debug adapter tracker*). Pro získávání potřebných dat tedy sleduje rozšíření vyměňované požadavky DAP. Při zachycení odpovědi na požadavek *Variables* ověří, zda je v jejím těle uvedena proměnná *APSR*, a pokud ano, zašle zprávu s hodnotou proměnné podoknu, jehož obslužný skript změní odpovídající HTML obsah.

Pokud by uživateli nevyhovovaly použité barvy, může je změnit pomocí nastavení barevného schématu v editoru. Rozšíření deklaruje v CP `colors` vlastní klíče barev a jejich výchozí hodnoty, v CSS stylu podokna k nim přistupuje pomocí editorem automaticky vytvořených CSS proměnných.



CPU STATUS FLAGS	
Negative 1	Zero 0
Carry 1	Overflow 0

Obrázek 5.8: Podokno s podmínkovými příznaky.

Dokumentace instrukcí

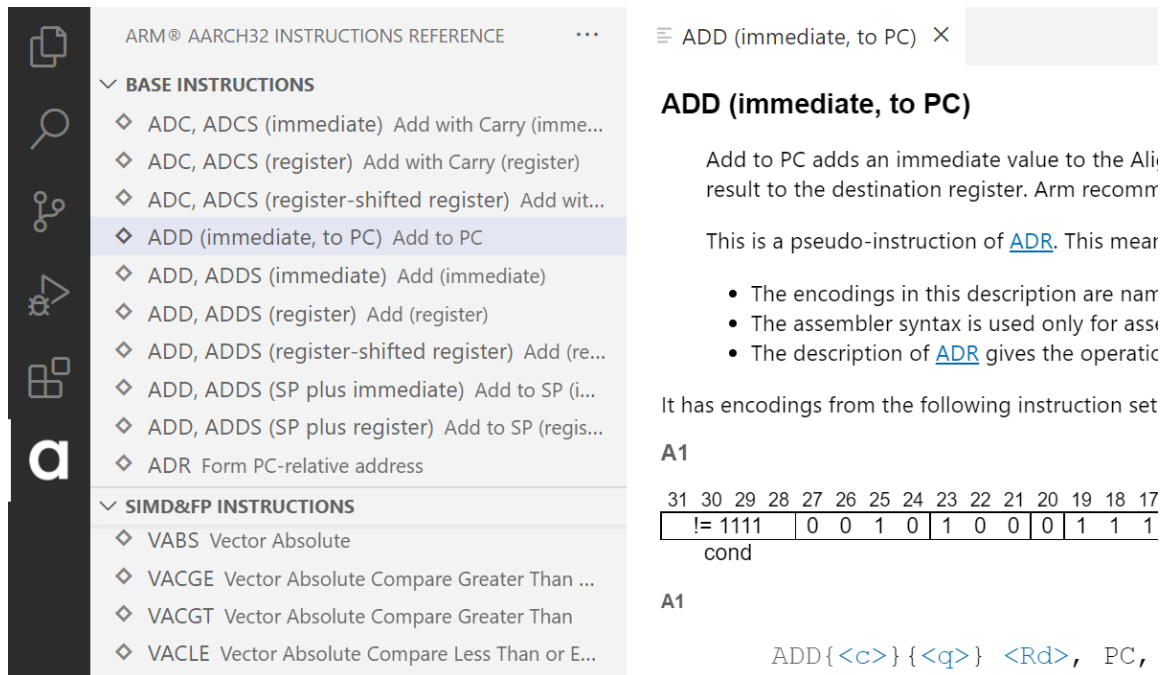
Úplný popis zápisu, kódování a sémantiky instrukcí je dostupný především v referenčním manuálu architektury Armv8. Kromě toho jej ale společnost Arm poskytuje také ve strojově čitelných formátech XML a HTML. Instrukce jsou zde popsány textově, jejich sémantika je formalizována pseudokódem ve speciálním jazyce²². Popis v tomto kódu je natolik přesný, že je podle něj možné např. implementovat funkční simulátor architektury nebo formálně verifikovat její vlastnosti. Součástí je také znázornění kódování instrukcí. Balíčky s tímto popisem je možné stáhnout na [webu Arm](https://developer.arm.com/downloads/-/exploration-tools)²³.

Rozšíření využívá právě XHTML verze popisu instrukční sady, ve které každé instrukci odpovídá jeden soubor. Nabízí se tedy tento HTML soubor zobrazit pomocí vlastního podokna (*webview*). Rozšíření musí zajistit načtení seznamu instrukcí, správu instancí ote-

²²Další informace o formátu a využití tohoto popisu viz např. <https://alastairreid.github.io/ARM-v8a-xml-release/>.

²³<https://developer.arm.com/downloads/-/exploration-tools>

vřených podoken a navigaci mezi nimi, protože ve *webview* nelze provádět běžnou navigaci pomocí odkazů.



Obrázek 5.9: Vlastní panel se seznamem instrukcí a otevřený panel s dokumentací.

Rozšíření načítá seznam všech instrukcí z rejstříku, který je součástí balíku, a to pomocí jednoduchého regulárního výrazu. Pro zobrazení konkrétní instrukce pak načítá konkrétní soubor s dokumentací a před přiřazením do těla podokna jej modifikuje tak, že vymaže zbytečné záhlaví a zápatí a vloží do něj vlastní styl a pomocný skript. Styl slouží pro přizpůsobení barev pro prostředí editoru VSC. Skript zajišťuje navigaci v dokumentaci – odkazům na jiné stránky přiřazuje obsluhu události `onclick`, ve které zasílá zprávu rozšíření. To zajistí otevření požadované stránky v novém panelu. Podobu seznamu instrukce i otevřeného panelu s dokumentací ukazuje obrázek 5.9.

Soubory dokumentace nelze distribuovat spolu s rozšířením, protože jsou předmětem autorského práva společnosti Arm, která by je musela pro toto použití licencovat. Rozšíření tedy při aktivaci explicitně vyzve uživatele ke stažení souborů, čímž na něj přenesse odpovědnost za tento úkon. Samotné stažení a rozbalení balíku provede rozšíření automaticky po potvrzení uživatelem.

Deklarativní zvýrazňování syntaxe

Preferovanou technikou editoru VSC pro zvýrazňování syntaxe je použití tzv. **TextMate gramatik**²⁴. Jsou to v zásadě kolekce regulárních výrazů, které mohou být rekurzivně zanořené (jejich vyjadřovací síla je tak vyšší než u běžných regulárních výrazů). Ty jsou postupně aplikovány na řádky zdrojového textu. Každý z nich přiřazuje odpovídajícím segmentům v kódu (tokenům) nějaký klíč (*scope*). Klíčem jsou pak v barevném schématu

²⁴Viz https://macromates.com/manual/en/language_grammars a <https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide>.

editoru přiděleny barvy. Tento proces běží neustále v rámci editoru, a to vždy při změně souboru, proto je jeho reakční doba velmi nízká.

Pomocí komplexnější jazykové analýzy prováděné v jazykovém serveru je možné dodávat dodatečné informace o sémantických tokenech, kterým jsou přiřazovány obdobné klíče (*scopes*). Jeden úsek textu tak může mít přiřazeno více klíčů a pro obarvení se použije ten nejspecifičtější. Doba odezvy jazykového serveru je však obecně vyšší, proto je vhodné co největší část syntaxe postihnout právě deklarativním způsobem v gramatice.

Použitá TextMate gramatika pro JSA instrukční sady A32 se nachází v souboru rozšíření `syntaxes/arm-ual.tmLanguage.json`. Je odvozena z gramatiky, jejímž autorem je Dan Underwood²⁵ a která je volně dostupná pod licencí MIT. V gramatice však byly provedeny rozsáhlejší změny, zejména byla vygenerována pravidla pro obarvování mnemonik podle aktuální dokumentace instrukční sady.

Balíčkování a distribuce

Editor VSC využívá pro instalaci balíčků s rozšířeními vlastní formát VSIX, což je v podstatě ZIP balík obsahující soubory rozšíření a soubory s metadaty. Pro vytváření těchto balíčků poskytuje Microsoft utilitu `vsce`²⁶. Ta zároveň umožňuje publikovat vytvořené balíčky do katalogu rozšíření *VS Code Extension Marketplace*, který je výchozím katalogem použitým v distribucích editoru, které vydává společnost Microsoft²⁷.

Systém rozšíření editoru umožňuje v katalogu uchovávat různé balíčky pro jednotlivé platformy, tj. kombinace operačního systému a architektury. Při distribuci řešení je tato možnost využita kvůli vloženým nepřenositelným binárním souborům knihoven Unicorn a Capstone, překladače GAS a linkeru `ld`.

Zdrojový kód řešení obsahuje sadu skriptů, které přeloží projekt jazykového serveru a běhové služby, stáhnou aktuální verzi knihovny Unicorn, přeloží ji pro cílové platformy, vše zkopírují do adresáře rozšíření spolu s překladačem GAS a linkerem `ld` a provedou balíčkování rozšíření. Součástí toho je také sloučení a minifikace kódu rozšíření. Skripty jsou určeny pro linuxový shell `bash`, avšak při spuštění v prostředí virtuálního linuxového stroje integrovaného do OS Windows WSL 2 (*Windows Subsystem for Linux*) umožňují provést překlad i pro Windows pomocí překladače MSVC (použití skriptů viz příloha C).

5.3 Poskytování kontextových informací o kódu pomocí jazykového serveru

Jako základ jazykového serveru je využita knihovna OmniSharp C# Language Server Protocol²⁸, která je produktem sdružení .NET Foundation a je dostupná pod licencí MIT. Zabezpečuje řízení komunikace s editorem včetně serializace do formátu popsaného protokolem LSP. Její rozhraní je jednoduché, spočívá především v registraci typů (tříd), které jsou zodpovědné za obsluhu jednotlivých požadavků protokolu.

Pro knihovnu je prostředek využitý k přenosu zpráv mezi serverem a editorem transparentní, programátor jej dodává jako objekt typu proud (*stream*). Editor VSC poskytuje rozhraní pro spouštění procesu jazykového serveru a komunikaci s ním pomocí standardního vstupu a výstupu. V produkční verzi serveru je využita právě tato forma přenosu, neboť

²⁵Repozitář s použitou gramatikou: <https://github.com/dan-c-underwood/vscode-arm>.

²⁶Viz <https://code.visualstudio.com/api/working-with-extensions/publishing-extension>.

²⁷Existují i alternativní, nezávislé katalogy rozšíření, např. Open VSX: <https://open-vsx.org/>.

²⁸Repozitář projektu: <https://github.com/OmniSharp/csharp-language-server-protocol>.

nevyžaduje žádnou konfiguraci a má minimální režii. Pro účely vývoje je možné úpravou konstant překladu využít jako přenosové médium také TCP sockety.

Synchronizace a předzpracování dokumentů

Editor neukládá každou změnu v dokumentech na disk, ale provádí úpravy pouze v paměti. Aby mohl jazykový server analyzovat vždy aktuální verzi zdrojového kódu, poskytuje LSP prostředky pro synchronizaci dokumentů. Po otevření dokumentu editor oznámí serveru, že jej nyní spravuje a server tedy nemá obsah dokumentu číst ze souborového systému. Pro spravované dokumenty zasílá editor změny buď ve formě celého nového obsahu, nebo častěji ve formě inkrementálních změn. Řešení tedy implementuje vlastní službu pro udržování aktuální verze dokumentu a aplikace inkrementálních změn.

Součástí správy dokumentů je také jejich předzpracování (*preprocessing*), které je provedeno po každé úpravě. Předzpracování spočívá v odstranění komentářů a sekvencí bílých znaků (mezer, tabulátorů). Jeho cílem je usnadnit implementaci syntaktického analyzátoru, pro který jsou tyto sekvence irelevantní. Tento přístup je inspirován překladačem GAS [10, kap. 3.1].

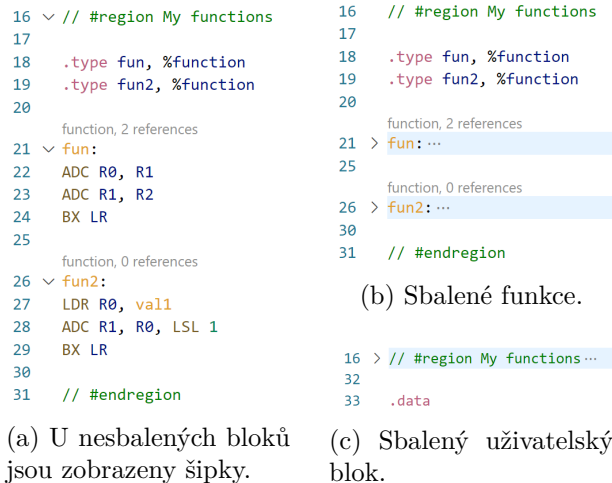
Předzpracování samotné je triviální a je možné jej provést s využitím jednoduchého konečného automatu nebo regulárních výrazů. Další části serveru ovšem potřebují mapovat pozice a rozsahy ²⁹ v textu z původního obsahu na předzpracovaný a naopak (analyzátor např. označuje tokeny v předzpracovaném textu, ale editoru musí být zaslána poloha v původním textu). Spolehlivé provedení tohoto mapování se ukázalo být značně náročnou úlohou.

První verze implementace prováděla předzpracování postupnou analýzou pomocí čtyř regulárních výrazů (pro jednořádkové komentáře, pro víceřádkové komentáře, pro sekvence mezer a pro prázdné řádky). Při každé shodě byla uchována informace o provedené úpravě obsahu (ve smyslu provedených posunů). Pro mapování se pak postupně procházely uložené informace o posunech a upravoval se podle nich vstupní rozsah. To však obnáší poněkud složitější logiku výpočtů pro jednotlivé druhy posunů, ve které se nepodařilo korektně ošetřit všechny možné situace. Tento přístup byl vyhodnocen jako neudržitelný.

V aktuální verzi se provádí předzpracování a mapování po znacích. Algoritmus pracuje se dvěma poli celých čísel – „zdroj na předzpracovaný“ a „předzpracovaný na zdroj“. Do těchto polí jsou uloženy indexy znaků, které odpovídají svým protějškům v opačné variantě dokumentu. Navíc musí být uloženy informace o hranicích sekvencí mezer, které jsou použity při mapování rozsahů, které v takové sekvenci začínají nebo končí. Předzpracování je implementováno jako konečný automat, který při průchodu textem pole plní a zároveň sestavuje předzpracovanou formu textu. Toto řešení je oproti předchozímu výrazně náročnější na paměťové prostředky, ale mapování mezi variantami textu je jednodušší.

Ve fázi předzpracování jsou ukládány informace o použitých **direktivách předzpracování**, což jsou jednořádkové komentáře, jejichž obsah začíná symbolem `#`. Pomocí nich může uživatel definovat vlastní bloky pro skládání (`#region`, viz obr. 5.10), skrýt chybová hlášení analyzátoru (`#suppress` a `#!`) nebo analyzátor úplně deaktivovat (`#ignore` a `#!!`). Rozpoznávání direktiv je implementováno mimo hlavní konečný automat pomocí metod pro práci s řetězci. Automat jej spekulativně spouští vždy po vstupu do jednořádkového komentáře.

²⁹Rozsah (*Range*) je dvojice [počátek, konec] dvojic [řádek, sloupec] označující segment textu.



Obrázek 5.10: Sbalování kódu.

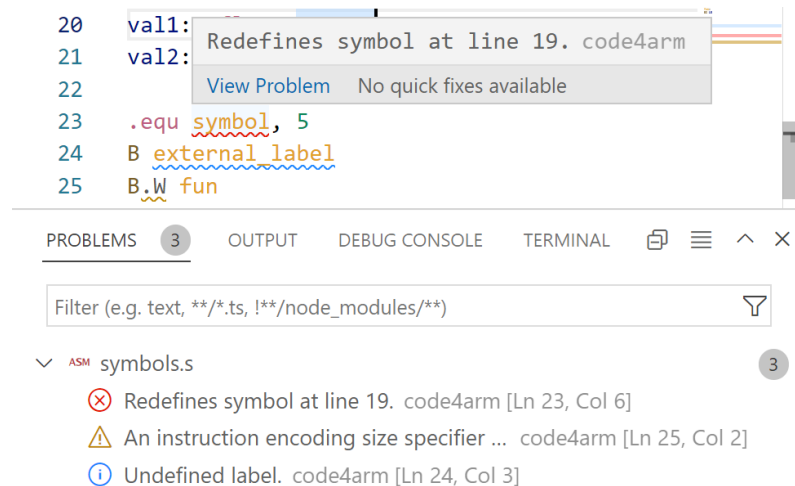
Obsluhy požadavků LSP

Jazykový server obsluhuje požadavky protokolu uvedené v příloze E. Obsluha je implementována ve třídách ve složce `Code4Arm.LanguageServer/Handlers`. Většina těchto obsluh není příliš složitá, v zásadě jen podle požavku uživatele interpretují výsledky jazykové analýzy, která je popsána níže. Komplexnější jsou pouze obsluhy požadavků našeptávání (`CompletionHandler`) a nápovědy k signaturám (`SignatureHelpHandler`), které se snaží na základě analýzy a pozice kurzoru odhadnout, jaké symboly by bylo vhodné na daném místě v kódu použít, resp. jaký operand se zde nachází. Jejich logika je v současném stavu poněkud komplikovaná a ne zcela spolehlivá.

Analýza jazyka a poskytování zpětné vazby

Lexikální a syntaktická analýza jazyka symbolických adres, jakožto proces potvrzení správnosti vstupu a jeho převedení do nějaké strukturované formy vnitřní reprezentace, není obtížná. Uvažujme zjednodušený jazyk bez direktiv překladače, ve kterém bude na každém řádku právě jedna instrukce z instrukční sady a který nebude obsahovat sekvence bílých znaků. Jazyk přijímaný právě na jednom řádku bude konečný: instrukční sada určuje konečný počet mnemonik, každá z nich má konečný počet kombinací operandů, a to včetně konstant, které jsou také ohraničeny.

V kontextu řešení však nejde jen o potvrzení syntaktické správnosti, ale zejména o poskytnutí zpětné vazby uživateli. Cílem je zjistit, jaké všechny podřetězce se na jednotlivých pozicích v řádku mohou vyskytovat a jaký je zde jejich význam, případně také zdůvodnit, proč se na nějakém místě určitý podřetězec vyskytovat nemůže, a odhadnout, jaký správný podřetězec měl uživatel na mysli.

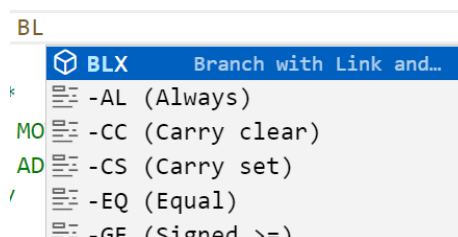


Obrázek 5.11: Různé druhy informací, které na základě analýzy kódu poskytuje jazykový server uživateli. Jsou zvýrazněny v kódu a vypsány v podokně *Problems*.

Analýza mnemonik

Způsob, jakým řešení analyzuje kód, vychází z následujících úvah. V JSA se nenacházejí víceřádkové konstrukce, na jednom řádku může být pouze jeden příkaz³⁰, je tedy možné každý řádek analyzovat zvlášť. Řádek může začínat návěstími, po nich může následovat právě jedna mnemonika, nebo direktiva. U mnemonik je žádoucí rozpoznávat zvlášť mnemoniku samotnou a zvlášť příznaky, které mohou být její součástí³¹, aby mohlo řešení uživateli nabízet jejich našeptávání a speciální obarvování. Až po úspěšném nalezení mnemoniky může následovat část s operandy, které jsou od sebe odděleny čárkami. Syntaxe direktiv je zcela odlišná a je nutné je analyzovat jiným způsobem než instrukce.

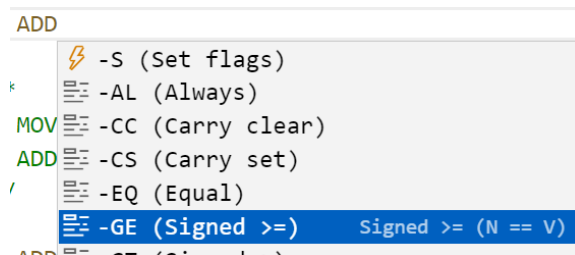
Řešení pro analýzu jazyka využívá techniku v duchu *scannerless parsingu* – jazykové analýzy, ve které není oddělen lexikální a syntaktický analyzátor. Analýzu řídí konečný automat, který zajišťuje čtení návěstí, direktiv, mnemonik a jejich příznaků. Datová struktura, kterou při průchodu vytváří, obsahuje informace o jiných možných mnemonikách a příznacích, případně i o definovaných návěstích nebo direktivách. Informací pak využívá našeptávač v příslušné obsluze požadavku (viz obr. 5.12 a N.1). Po úspěšném přečtení mnemoniky předává automat řízení analýze operandů.



(a) Pro vstup BL nabídne našeptávač jak odlišnou instrukci BLX, tak i podmínkové kódy pro platnou instrukci BL.

³⁰Překladač GAS umožňuje uvést na jednom řádku více příkazů oddělených znakem ;. Tuto možnost řešení zanedbává, ale bylo by možné ji relativně snadně doplnit ve fázi předzpracování.

³¹Podmínkové kódy, sufix -S označující instrukci měnící příznaky procesoru, u SIMD/FP instrukcí specifikátor datového typu, specifikátor šířky kódování instrukce.



(b) Našeptávání u instrukce umožňující nastavování příznaků procesoru (variantu S). Součástí našeptávaných podmínkových kódů je i vysvětlivka s principem dané podmínky.

Obrázek 5.12: Chování našeptávače instrukcí.

Problém analýzy operandů instrukcí

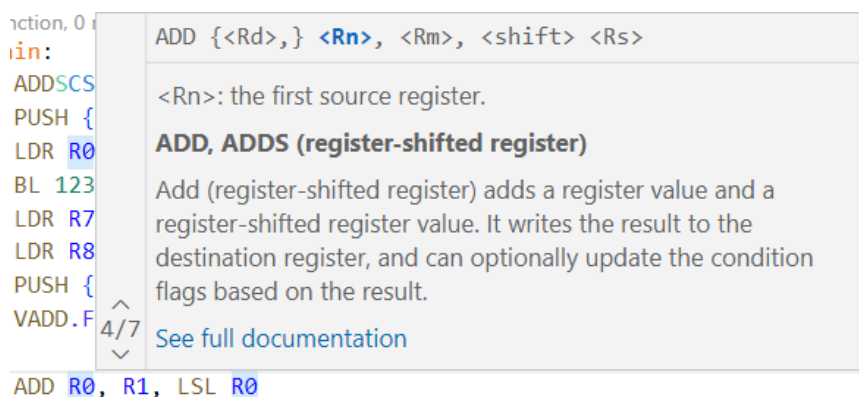
Syntaxe operandů instrukcí je na rozdíl od samotných mnemonik značně dynamická. Například instrukcemi se významněji liší, jedna mnemonika navíc může mít několik druhů sekvencí operandů, z nichž některé mohou být nepovinné, případně některé mohou být speciálními variantami jiných. Příkladem je prostá aritmetická instrukce ADD, pro kterou instrukční sada samostatně specifikuje 9 různých kombinací operandů (složené závorky označují nepovinnost operandu):

- ADD <Rd>, PC, #<const>
- ADD {<Rd>}, <Rn>, #<const>
- ADD {<Rd>}, <Rn>, <Rm>, RRR
- ADD {<Rd>}, <Rn>, <Rm> {, <shift> #<amount> }
- ADD {<Rd>}, <Rn>, <Rm>, <shift> <Rs>
- ADD {<Rd>}, SP, #<const> a předchozí čtyři varianty s SP na pozici <Rn>; tyto varianty se v instrukční sadě/kódování A32 navíc neliší, zvláště jsou definovány pouze pro některá kódování ze sady T32.

Snahou tedy bylo najít řešení, ve kterém nebude příliš obtížné existenci všech těchto různých kombinací definovat a které zároveň bude schopné určit, jaké variantě, resp. jakým variantám odpovídá aktuální stav řádku. To je důležité pro správné našeptávání a poskytnutí odpovídající nápovědy k signatuře (viz obr. 5.13). Proto je také analýza operandů oddělena od samotného konečného automatu, který čte mnemoniky. Bylo by možné ji do něj zahrnout, ale takový konečný automat by měl vcelku extrémní rozsah.

Popis variant instrukcí

Nejsem si vědom způsobu, jakým by bylo možné spolehlivě automatizovaně vytvořit nějakou reprezentaci všech existujících kombinací mnemonik a jejich operandů. Do jisté míry by bylo možné jej získat analýzou strojově čitelné dokumentace (viz část **Dokumentace instrukcí** kap. 5.2), ani ta ovšem neposkytuje dostatečné množství informací, protože podoby instrukce v JSA jsou zde popsány pouze textem v angličtině. Seznam instrukcí a možných kombinací operandů tedy bylo nutné definovat ručně, a to ve vlastním formátu, podle kterého vytvoří jazykový server pravidla pro analýzu.



Obrázek 5.13: Nápopověda k signatuře u konkrétní varianty instrukce ADD. Zobrazen je popis operandu, na kterém je zrovna kurzor, poté základní popis instrukce a odkaz pro zobrazení dokumentaci.

Pro mnemoniky a kombinace operandů jsem navrhl vlastní strukturu uloženou ve formátu JSON, ve které je každá kombinace operandů reprezentována polem řetězců, které obsahují buď literál (neměnnou textovou hodnotu), nebo zástupný symbol reprezentující speciální druh operandu (např. registr nebo číselná konstanta).

Algoritmus pro analýzu operandů

Pro dosažení dostatečné jednoduchosti při definici kombinací operandů byl zvolen postup analýzy spočívající v postupném hledání shod s regulárními výrazy. Pro každou variantu instrukce (mnemonika a jedna konkrétní sekvence operandů) se podle struktury popsané výše vytvoří sada regulárních výrazů. Ty jsou postupně použity na aktuální pozici na řádku. Pokud je nalezena shoda, pokračuje se na další výraz v seznamu. Pokud se pro nějaký z testovaných regulárních výrazů nenajde shoda, může to znamenat buď to, že se na daném řádku testovaná varianta instrukce nenachází, nebo že uživatel přeskočil nepovinný operand. Ve druhém případě je tedy nutné pokusit se analýzu provést od místa poslední chyby znovu, ale s přeskočeným nepovinným operandem.

Tento proces je implementován pomocí „rekurzivního sestupu“. Pro každý operand (respektive deskriptor operandu, což je datová struktura obsahující mimo jiné kolekci regulárních výrazů popisujících daný operand) varianta instrukce je spuštěna pomocná metoda „zkonzumuj operand“, která se pokusí aplikovat regulární výrazy a v případě úspěchu volá sama sebe, aby zpracovala další operand. Tato metoda naplňuje pomocnou datovou strukturu sekvence zanalyzovaných operandů, do které ukládá informace o zanalyzovaných operandech nebo o výskytu chyby. V případě neúspěchu se pak postupně pokouší přeskakovat deskriptory nepovinných operandů. Před analýzou dalšího operandu se duplikuje současný stav sekvence a další sestup se provádí s touto kopií. V případě, že není nalezena žádná úplně shodující se varianta, je tak možné vrátit sekvenci, ve které došlo k největšímu počtu shod (je tedy nejpravděpodobnější, že uživatel zamýšlel právě ji). Součástí procesu je také ošetřování čárek oddělujících operandy a konců řádků. Zjednodušený pseudokód v příloze J nastiňuje celý algoritmus.

Nedostatky algoritmu

Po provedení prvotní implementace se ukázalo, že zvolený postup sám o sobě není pro požadovaný účel dostatečně granulární. Postačoval by, kdyby nebyly operandy dále struk-

turované, ale to nesplňuje například operand instrukce LDR pro načítání z paměti. Arm umožňuje tři režimy adresace, které jsou v JSA zapsány pomocí jedné z nepatrně se lišících sekvencí:

- LDR <Rt>, [<Rn>, <Rm> {, <shift> #<amount> }]
- LDR <Rt>, [<Rn>], <Rm> {, <shift> #<amount> }
- LDR <Rt>, [<Rn>, <Rm> {, <shift> #<amount> }]!

Při počáteční úvaze, že operandy jsou odděleny čárkami, je tedy v prvním a třetím případě za jeden operand považována celá sekvence uvnitř hranatých závorek, ve druhém případě jde o tři samostatné operandy. Pro našeptávání relevantních symbolů pro každou část takto složeného operandu tedy bylo nutné do procesu zanést ještě další vrstvu analýzy operandů.

Deskriptory operandů jsou proto rozděleny ještě na jednodušší, atomické jednotky (tokeny). Tokenem je například název registru nebo typ posunu/rotace. Deskriptory pak neobsahují pouze jeden regulární výraz, ale jejich kolekci, přičemž definují, jak se jednotlivé podvýrazy (*match groups*) uvnitř jednotlivých výrazů mapují na tyto tokeny. Metoda konzumující operand tak postupně aplikuje všechny regulární výrazy a předá výsledek procesu další jednotce (v pseudokódu je to naznačeno voláním funkce **Zanalyzuj_Obsah**), analyzátoru operandu, která na základě výsledků těchto aplikací vyhodnotí, jestli je celý operand platný, nebo ne, případně v jaké jeho části nastala chyba.

Ačkoliv zvolené řešení není v principu špatné, jeho korektní implementace se ukázala být velmi náročná. Komplikuje ji zejména potřeba co nejpřesněji označit rozsah textu, ve kterém došlo k chybě. Realizace tohoto požadavku v kódu není dobře čitelná a spolehlivá. Nedostatkem je také složitá implementace zotavení z chyby, které tak současné řešení neumožňuje. Pokud je v analyzovaném operandu chyba, nelze v analýze pokračovat dál a uživateli tak spolehlivěji napovědět, jak tuto chybu napravit.

Vzhledem k velkému počtu instrukcí jsou definice operandů připraveny pouze pro jednodušší instrukce, se kterými se studenti setkají nejpravděpodobněji. Jejich seznam je uveden v příloze **L**. Pro ostatní instrukce řešení analýzu operandů přeskakuje. Podobně není kompletní také určení správnosti použitých datových typů u vektorových instrukcí, řešení tedy nabízí a akceptuje i specifikátory, které v daném kontextu nejsou platné.

Kapitola 6

Vyhodnocení řešení

V rámci této práce vzniklo komplexní softwarové řešení, které se skládá ze 6 projektů v jazyce C# a jednoho projektu v jazyce TypeScript. Jazykový server má přes 6 000 řádků vlastního kódu (bez komentářů a prázdných řádků). Překladač, běhová a ladicí služba má přes 11 000 řádků vlastního kódu, navíc jsem implementoval také wrapper pro použití knihovny Unicorn v jazyce C#, který má téměř 2 000 řádků kódu. Při návrhu a implementaci jsem dbal na rozšiřitelnost a znovupoužitelnost kódu.

Řešení v současné podobě splňuje základní cíl, a to poskytnout studentům prostředí, ve kterém si mohou vyzkoušet přímou práci s instrukční sadou Armv8 bez nutnosti jakékoliv složité konfigurace. Celé řešení je možné nainstalovat stiskem jednoho tlačítka z katalogu rozšíření editoru VSC na jakékoliv platformě, kterou editor podporuje (mimo webové verze). Ze stávajících nástrojů představených v kapitole 2.4 jej v tomto ohledu předčí pouze simulátor CPUlator, který běží ve webovém prohlížeči.

Část zajišťující simulaci programu a jeho ladění umožňuje spouštění prakticky všech instrukcí architektury, plnohodnotné krokování, umisťování breakpointů i pozastavování chodu při změně dat v registrech či paměti. Nabízí několik možností, jak nahlížet do stavu procesoru a programu, přičemž studentům napomáhá s pochopením různých interpretací dat, se kterými program pracuje. Umožňuje také použití vektorových a FP instrukcí. To vše je integrováno do prostředí populárního editoru, se který bude mít mnoho uživatelů předchozí zkušenosti. Použití řešení tedy považuji za intuitivnější než v případě ostatních představených nástrojů. Co se možností ladicího prostředí týče, mé řešení opět poskytuje více možností než většina z nich. Srovnání nejdůležitějších funkcí jednotlivých nástrojů je uvedeno v tabulce v příloze K.

Část zajišťující podporu při editaci kódu splnila stanovené cíle částečně. Podcenil jsem složitost JSA a pro jeho analýzu jsem zvolil ne zcela vhodný postup. Jeho implementaci a hlavně ladění jsem strávil příliš mnoho času, který jsem tak nemohl použít pro vylepšení interakce s uživatelem a pro realizaci zajímavějších didaktických funkcí. Problematické je především rozpoznávání operandů, kvůli kterému jsou omezeny schopnosti našeptávače a nápovědy k signaturám. V této oblasti je tak stále prostor pro další vývoj. I přesto mé řešení oproti představeným stávajícím nástrojům, které pro programování v JSA neposkytují téměř žádnou podpůrné prostředky (výjimkou je pouze nástroj VisUAL, který nabízí právě zajímavé didaktické pomůcky), přináší užitečné funkce, které programování usnadňují (např. integraci plnohodnotné dokumentace instrukcí nebo nástroje pro práci se symboly).

6.1 Testování řešení

Existuje několik scénářů, ve kterých je nutné řešení otestovat. Bylo by vhodné testovat, zda jednotlivé části kódu vykazují očekávané chování, a to pomocí jednotkových testů (*unit tests*). Dále je nutné testovat, zda se správně chovají celé komponenty řešení (jazykový server, ladicí služba), a zda správně funguje řešení jako celek (tj. jeho instalace, spouštění a konfigurace).

Testování pomocí jednotkových testů typicky obnáší implementaci velkého množství testovacího kódu, na což v průběhu řešení nebyl dostatek času, bylo proto využito pouze

v počáteční fázi implementace jazykového serveru. Důraz byl tedy kladen na manuální testování jednotlivých funkcí řešení a na otestování funkčnosti celého řešení na různých systémech a platformách.

Testování funkčnosti v různých systémech

Pro otestování instalace a použití řešení byly využity následující systémy:

- | | |
|---|--|
| 1. Windows 10 Pro (10.0.19044.1288), arch. x86 (32b), ve virtuálním stroji (VMware Workstation 16 Player) | 5. Fedora Linux 36 (kernel 5.17.13), arch. x86_64, fyzický stroj* |
| 2. Windows 11 Pro (10.0.22000.318), arch. x86_64, ve virtuálním stroji (VMware Workstation 16 Player) | 6. macOS 12 (kernel 21.2.0), arch. x86_64, fyzický stroj* |
| 3. Windows 11 Home (10.0.22000.795), arch. x86_64, fyzický stroj* | 7. Raspbian 11.4 (kernel 5.15.32), arch. aarch64 (64b Armv8), fyzický stroj (Raspberry Pi 400) |
| 4. Ubuntu 22.04 (kernel 5.15.0), arch. x86_64, ve virtuálním stroji (libvirt) | 8. Raspbian 11.4 (kernel 5.15.32), arch. arm (32b Armv8), fyzický stroj (Raspberry Pi 400) |

V konfiguracích, které nejsou označeny hvězdičkou, byla použita čistá instalace dané verze operačního systému, po které byly pouze provedeny aktualizace. Dostupné nebyly počítače se systémy Windows a macOS běžící na arch. Arm, tyto varianty tedy zůstávají netestované. Mimo ně jsou pokryty všechny platformy, na kterých běží editor VSC.

Před provedením testování byly pomocí příložených distribučních skriptů všechny části řešení přeloženy, zabaleny do balíčku rozšíření pro editor VSC a nahrány do katalogu rozšíření Visual Studio Code Marketplace. Na všech systémech pak byly provedeny následující kroky:

- | | |
|---|--|
| 1. Instalace editoru VSC (z balíčku ve verzi 1.69.1 ¹). | 8. Napsání krátkého kódu v JSA, kontrola funkčnosti jazykového serveru. |
| 2. Spuštění editoru a instalace rozšíření z katalogu rozšíření. | 9. Automatické vytvoření konfigurace spouštění. |
| 3. Otevření složky, vytvoření souboru <code>test.s</code> . | 10. Umístění breakpointu, spuštění ladění. |
| 4. Nastavení lokální ladicí služby. | 11. Kontrola úspěšného spuštění lokální ladicí služby a prostředků ladění. |
| 5. Kontrola, zda se úspěšně stáhlo běhové prostředí .NET. | 12. Nastavení vzdálené ladicí služby (spuštěné na jiném počítači). |
| 6. Potvrzení stažení dokumentace instrukcí. | 13. Spuštění ladění, kontrola úspěšného připojení ke vzdálené ladicí službě. |
| 7. Kontrola funkčnosti dokumentace instrukcí (otevření seznamu; vyhledání instrukce příslušným příkazem). | |

¹Instalační balíčky editoru VSC: <https://code.visualstudio.com/download>.

Testování odhalilo problém na OS Raspbian. Knihovna Unicorn byla v testovacím balíčku přeložena křížově s využitím Arm GNU Toolchain ve verzi 11.2-2022.02, a vyžadovala tak dynamické linkování s `glibc`² ve verzi 2.32. OS Debian (a tedy i Raspbian) ovšem ve své aktuální verzi (11.4) stále využívá `glibc` ve verzi 2.31. Řešením bylo přeložit knihovnu Unicorn s využitím této starší verze. Unicorn podle dostupných informací žádnou konkrétní verzi `glibc` nevyžaduje. V době psaní práce některé stále podporované linuxové distribuce využívají i mnohem starší verze `glibc`³ (např. Ubuntu 18.04 LTS využívá verzi 2.27), jednotlivé verze jsou však typicky zpětně kompatibilní, bylo by tedy vhodné distribuovat rozšíření s knihovnou Unicorn přeloženou proti některé ze starších verzí `glibc`.

Kromě uvedeného problému proběhlo toto testování na všech systémech bez větších problémů. Pomohlo odhalit nedostatky v procesu balíčkování a drobné chyby v rozšíření, které byly opraveny. Na základě těchto výsledků považuji cíl zprovoznění simulačního prostředí stiskem jednoho tlačítka za splněný.

Testování chování běhové služby

Pro testování správného chování běhové služby a samotné emulace instrukcí byly využity relevantní příklady z [31]⁴ a také vlastní fragmenty kódu, na kterých bylo testováno chování řešení v různých konkrétních situacích. Vzhledem k využití frameworku Unicorn nebylo cílem ani tak ověřit správné chování jednotlivých instrukcí (v tomto ohledu se předpokládá korektní fungování frameworku, resp. QEMU), ale spíš zjistit, jestli správně reaguje na stavy, které v programu nastávají, a jestli je prezentuje uživateli vhodným způsobem.

Testování bylo zaměřeno konkrétně na chování při skocích mezi různými zdrojovými soubory, při softwarových přerušeních, při chybných přístupech do paměti, při skocích do simulovaných funkcí a při použití textového vstupu a výstupu. Každý testovací kód byl spuštěn bez zastavování, načež proběhlo ověření správnosti koncového stavu. Poté byl kód krokovan po jednotlivých instrukcích a byla ověřována správná odezva ladicích nástrojů.

Testování odhalilo nesprávné chování při krokování a použití simulovaných funkcí pro čtení uživatelského vstupu, které vedlo v kódu běhového jádra k uvážnutí. Tato chyba byla opravena. Dále byly provedeny úpravy ošetřování přerušení, která neměla konzistentní chování, a přidána podpora pro instrukci BKPT (*breakpoint*). Poznatkem z testování je také poněkud nepřehledné chování v situacích, kdy je emulace pozastavena na instrukci, která se nenachází přímo v uživatelském kódu (např. instrukce vygenerované překladačem nebo instrukce zastupující simulované funkce). V těchto případech otevře editor první zdrojový soubor a nevyznačí v něm žádný řádek kódu. V budoucí verzi řešení by bylo vhodné uživatele přenést do okna *Disassembly*, ve kterém je vždy jasně vidět, na které instrukci je simulátor právě zastaven. Další nedostatky nebyly při testování odhaleny.

Uživatelské testování

Bohužel jsem neměl možnost provést uživatelské testování v reálném provozu, neboť se v době psaní práce neotevřel kurz Pokročilé assembly, jehož studenti jsou hlavní cílovou skupinou řešení. Po jeho otevření bude vhodné zajistit sběr zpětné vazby a pokračovat ve vývoji řešení. Uživatelské testování tedy bylo provedeno pouze se dvěma dobrovolníky,

²The GNU C Library, implementace standardní knihovny jazyka C používaná v operačních systémech typu GNU/Linux.

³Viz <https://sourceware.org/glibc/wiki/Release>, cit. 10. 7. 2022.

⁴Dostupné na <https://brucesmith.info/aarch32.html>. Využity nebyly příklady předpokládající běh kódu v jazyce C nebo běh na konkrétním zařízení.

kteří měli základní předchozí znalosti JSA arch. Armv8. Uživatelé měli za úkol rozšíření nainstalovat, naprogramovat jednoduchý kód, který načte dvě čísla, vypíše jejich součet a informaci o tom, zda jsou kladná, či záporná, a tímto kódem krokovat. K dispozici měli manuál uvedený v příloze B.

Uživatelé ocenili jednoduchost instalace, ale zároveň podotýkali, že ve fázi ihned po instalaci, kdy probíhá stahování prostředí .NET, není jasné, že rozšíření provádí vlastní práci na pozadí a že začne být použitelné až po jejím dokončení. Podobně zpětnou vazbu neposkytuje také proces stahování a rozbalování dokumentace instrukcí.

V režimu editace kódu se uživatelé shodli na nedostatečnosti našeptávače a nápovědy k signaturám (operandům) instrukcí. Rušivé pro ně bylo chování našeptávače, který se v některých případech spouští na místě, kde to nedává smysl (např. uvnitř komentářů), a naopak v části operandů často nenabízí symboly, které uživatelé hledají. Za nepraktickou označili také nedostupnost direktiv v našeptávači. Ocenili dostupnost vestavěné dokumentaci instrukcí, podotkli však, že není zjevné, jak v seznamu vyhledávat (na základě tohoto podnětu byl panelu s instrukcemi rozšířen o samostatné tlačítko pro vyhledávání).

V režimu ladění komentovali uživatelé pouze chování při pozastavení na instrukci mimo zdrojový kód, které jsem popsal výše. Jeden z uživatelů také podotkl, že interpretace registrů v různých datových typech zobrazené ve výchozím nastavení jsou poněkud rušivé. Vzhledem ke snadné nastavitelnosti zobrazení proměnných v konfiguračním GUI editoru to však nepovažuji za relevantní. Uživatelům scházelo také snadné procházení obsahu paměti, ačkoliv se shodli, že možnosti přístupu k paměti pomocí výrazů to dostatečně kompenzují. Možnosti, které k tomu poskytuje editor VSC, jim přišly matoucí a nevyhovující.

6.2 Možnosti vylepšení stávajícího řešení

V předchozím textu byly naznačeny některé nedostatky stávající implementace řešení. V jazykovém serveru to je především způsob analýzy jazyka, v běhové službě např. způsob implementace proměnných (viz příloha M), omezené možnosti zpětného krokování nebo nepřívětivé chování při zastavení na kódu mimo zdrojový text. V rozšíření samotném by pak bylo vhodné upravit způsob distribuce jednotlivých součástí a zlepšit zpětnou vazbu uživateli.

Úprava jazykového serveru

Za prioritní úkol pro další vývoj nástroje považuji implementaci nové verze jazykového serveru, jejímž cílem bude využít nějaké stávající řešení pro provedení jazykové analýzy. Existuje řada nástrojů, které na základě dodané gramatiky vygenerují kompletní lexikální a syntaktický analyzátor v cílovém jazyce. Potenciálně vhodnými by mohly být např. generátory ANTLR⁵ nebo Hime⁶. Před finálním výběrem bude nutné ověřit, jak zvládají nástroje interpretovat chybné vstupy a zotavovat se z nich.

Rozšíření možností zpětného krokování

Běhové jádro aktuálně umožňuje pouze základní zpětné krokování, které je dosaženo ukládáním kontextu procesoru při každém požadavku *Next*. Není tedy možné vracet se do stavu před prvním krokem. Součástí uloženého kontextu je pouze stav registrů, a ne paměti. Bylo

⁵ANTLR: <https://www.antlr.org/>.

⁶Hime: <https://cenotelie.fr/projects/hime>.

by vhodné rozšířit jádro o možnosti ukládání po každé provedené instrukci, volitelně i se zachytáváním změn paměti.

Distribuce komponent řešení

V současné verzi je využit způsob distribuce, kdy je v balíčku s rozšířením přibaleno také jazykový server a běhová služba, a to včetně nepřenositelných knihoven a nástrojů přeložených pro konkrétní cílovou platformu. Kvůli nim je nutné při každé aktualizaci řešení vytvořit pro všechny platformy balíčky zvlášť. To vytváří prostor pro chyby a neočekávaná chování, zároveň to zvyšuje velikost celého balíčku. Také by nebylo možné v případě potřeby vydat verzi přizpůsobenou pro konkrétní systémovou konfiguraci.

Vhodným řešením by byla implementace vlastního systému balíčků s komponentami. Rozšíření by si tak potřebné závislosti pro konkrétní systém stahovalo až v případě, že by je potřebovalo využít.

Dynamické alokování paměti

Řešení poskytuje triviální implementaci simulované funkce `malloc`, pomocí které může program žádat o přidělení paměti. Využívá jednoduchý postup na základě lineárního seznamu přidělených bloků. Při časté manipulaci s menšími úseky paměti tak může docházet k fragmentaci. Navíc pro tuto paměť není implementován striktní režim omezující přístup mimo aktuálně přidělený prostor. Bylo by vhodné implementovat důmyslnější způsob dynamického přidělování paměti, který by více odpovídal chování knihovny `glibc`. Druhou možností je zavedení skutečné standardní knihovny jazyka C.

6.3 Možnosti dalšího rozšíření

Zvolená architektura celého řešení a způsob implementace jeho jednotlivých komponent poskytují široký prostor pro možná rozšíření. Identifikoval jsem několik směrů, ve kterých by rozšíření byla vhodná.

Implementace běhové služby jako vzdáleného serveru nabízí různé možnosti interakce učitele se studenty. Nabízí se např. zadávání úloh s doplňováním kódu a automatickým vyhodnocením. Vyučující by mohl mít přístup ke kódu v reálném čase, aby mohl studentovi pomáhat vyřešit některé problémy, kód komentovat atp. Možnosti rozšíření v tomto směru jsou široké, bylo by tedy především nutné určit, které z nich a v jaké konkrétní podobě by byly skutečně využívány.

Z hlediska didaktických funkcí by bylo prospěšné přidání různých dalších vizualizací činnosti instrukcí (v tomto ohledu považuji za dobrou inspiraci nástroj **VisUAL**, viz kap. 2.4.1). Vhodné by bylo také přidat možnosti interakce uživatele s programem pomocí simulovaných periférií (prvky v okně editoru, které reprezentují například spínače, LED diody, displeje atp.). Z kódu by byly jejich hodnoty zpřístupněny na konkrétních adresách v paměťovém prostoru (technika MMIO).

Funkce standardní knihovny jazyka C, simulace linuxového jádra

Při výuce programování v JSA je žádoucí, aby se studenti naučili interagovat s existujícím kódem a knihovnami, typicky s funkcemi jazyka C. Většinu z nich je možné implementovat pomocí simulovaných funkcí. Problém nastává u těch, které volají uživatelský kód – aktuální implementace běhového jádra nepočítá s možností spuštění dalšího kroku emulace z kódu

simulované funkce. Pokrýt celou standardní knihovnu jazyka C by vyžadovalo mnoho práce, neboť obsahuje stovky funkcí⁷. Nejde navíc jen o funkce, ale i o některé globální proměnné, například `errno` nebo `stdin`.

Vhodným rozšířením by byla simulace některých systémových volání linuxového jádra. To by umožnilo zavést do paměti emulovaného programu existující implementaci standardní knihovny (např. `uClibc-ng`: <https://uclibc-ng.org/>). Systémových volání je, podle manuálové stránky, ve verzi jádra 5.11 přibližně 425, ale pro potřeby výukového řešení je významný jen zlomek z nich, např. volání pro vstup a výstup, pro základní práci se soubory a pro práci s pamětí.

Hlavní výhodou tohoto přístupu je, že chování simulovaného programu bude velmi přesně odpovídat tomu, co může programátor očekávat při běhu na skutečném systému. Další výhodou je, že student může takto krokovat kódem knihovny a získat povědomí o její implementaci a složitosti. Také by to umožnilo do překladu zahrnout i kód v jazyce C, ačkoliv by bylo zcela mimo rozsah nástroje pro něj poskytnout ladění na úrovni zdrojového kódu.

Využití ve webové verzi editoru

Editor VSC je implementován pomocí webových technologií, přirozeně se tak nabízí možnost spustit jej i přímo ve webovém prohlížeči. Microsoft poskytuje webovou verzi editoru na adrese <https://vscode.dev>. Při běhu v prohlížeči má editor k dispozici pouze omezené prostředky (ve smyslu rozhraní pro interakci s operačním systémem), pro využití simulátoru ve webové verzi editoru by tak bylo nutné provést několik úprav.

Webová verze neumožňuje spouštění procesů mimo prohlížeč, což je překážkou v použití jazykového serveru. Existují však možnosti překladu a spouštění C# (.NET) kódu přímo v prohlížeči prostřednictvím technologie WebAssembly. S úpravami⁸ by tak bylo možné jazykový server spustit v prohlížeči a připojit se k němu i z webové verze editoru.

Běhová služba může fungovat ve vzdáleném režimu, na straně klienta (editoru) pak není potřebný žádný kód, který by nebylo možné ve webové verzi spustit. Problémem je, že např. zmíněná instance webové verze editoru má všechny funkce ladění vypnuté, bylo by tedy nutné nasadit vlastní upravenou instanci. To by však neměl být, vzhledem k otevřenému zdrojovému kódu editoru, větší problém.

S jistým úsilím by bylo možné ve webovém prostředí zprovoznit i lokální verzi běhové služby. Samotnou službu by bylo možné přeložit pro běh v prohlížeči pomocí technologie WebAssembly, totéž lze s úpravami provést pro knihovnu Unicorn⁹. Dále by bylo nutné v prohlížeči provádět překlad a linkování JSA. Existující projekty naznačují, že to je možné s pomocí překladače `clang`, který lze také spustit v prostředí WebAssembly¹⁰.

V rozšíření pro editor VSC by bylo nutné upravit manifest rozšíření, přidat speciální aktivační kód pro webovou verzi a upravit funkce pracující se soubory dokumentace instrukcí (nyní využívají rozhraní pro práci se souborovým systémem, která nejsou při běhu ve webovém prohlížeči dostupná).

⁷Tento seznam uvádí 280 funkcí: <https://www.ibm.com/docs/en/i/7.1?topic=extensions-standard-c-library-functions-table-by-name>. Implementace glibc ve verzi 2.35 obsahuje přes 1 400 globálních symbolů.

⁸Diskuze k použití knihovny `OmniSharp C# LSP` v prostředí (Blazor) WebAssembly: <https://github.com/OmniSharp/csharp-language-server-protocol/issues/456>.

⁹Překlad frameworku Unicorn pro použití ve webovém prostředí implementuje projekt `unicorn.js`: <https://github.com/AlexAltea/unicorn.js>.

¹⁰Viz projekty <https://github.com/tbfleming/cib> a <https://github.com/binji/wasm-clang>.

Použití řešení pro simulaci architektury x86

Úprava řešení pro simulaci 32bitové verze architektury x86 by si našla využití v dalším předmětu na FIT VUT, Programování na strojové úrovni (ISU). Pokud by se takto upravené řešení navíc povedlo integrovat do webové verze editoru, vznikl by mimořádně jednoduchý nástroj, který by měl výrazně větší dosah (zmíněný předmět je povinný a každoročně jej absolvují stovky studentů, přičemž v posledních letech přibývají studenti, kteří používají počítač s procesorem arch. Arm, a tak potřebují naopak emulovat procesor arch. x86).

Z hlediska použití frameworku Unicorn je taková úprava triviální, stačí změnit parametry při vytváření jeho instance. Pro použití s arch. Arm je však navržena velká část kódu běhového jádra, který by bylo nutné změnit. Jde především o způsob manipulace s registry – jejich inicializace i jejich zpřístupňování pomocí proměnných. Na několika místech kód běhového jádra předpokládá fixní délku instrukce 4 B, zde by tedy bylo nutné najít alternativní řešení umožňující použití s instrukcemi variabilní délky. Jiné potenciální problémy jsem neidentifikoval.

Jazyková podpora poskytovaná jazykovým serverem, který v rámci řešení vznikl, je specifická pro JSA arch. Arm a pro použití s JSA arch. x86 je prakticky nepoužitelná. Pokud by však došlo k přepracování způsobu analýzy jazyka, předpokládám, že by bylo možné část řešení použít i pro analýzu JSA jiných architektur. V rozšíření by pak bylo nutné upravit způsob získávání dokumentace instrukcí a provést drobné změny v podokně pro zobrazování podmínkových příznaků.

Kapitola 7

Závěr

Cílem práce bylo vytvořit nástroj, který usnadní výuku programování procesorů Arm na strojové úrovni, v jazyce symbolických adres (assembleru). Práce se zaměřuje na 32bitový režim AArch32 architektury Armv8-A, konkrétně na instrukční sadu A32. Dosaženým výstupem práce je samostatný server, který umožňuje simulaci této instrukční sady a poskytuje prostředky pro ladění spouštěného kódu. Dalším výstupem je rozšíření pro editor zdrojového kódu Visual Studio Code, které zpřístupňuje služby simulačního serveru uživateli. Navíc rozšíření poskytuje prostředky pro usnadnění vývoje, např. zvýraznění syntaxe, našeptávač či integrovanou dokumentaci instrukční sady.

V první části práce jsem představil některé významné aspekty skupiny procesorových architektur Arm. Provedl jsem analýzu stávajících nástrojů pro simulaci činnosti těchto architektur a vysvětlil jsem, proč je považuji za nedostatečné. Nastudoval jsem způsob použití emulační knihovny Unicorn a možnosti rozšiřování jazykové podpory v editoru Visual Studio Code. Na základě zjištěných poznatků jsem navrhl řešení, které považuji za vhodné pro využití ve výuce.

Navržené řešení, které se koncepčně skládá ze tří částí (rozšíření pro editor, jazykový server a služba zajišťující překlad, běh a ladění), jsem implementoval způsobem, který umožňuje snadnou instalaci stiskem jednoho tlačítka na všech platformách, kde je dostupný zmíněný editor (mimo webovou verzi). Rozsah kódu je cca 20 000 řádků především v jazycích C# a TypeScript. Při řešení práce jsem přispěl také do kódu projektu Unicorn a implementoval jsem podpůrnou knihovnu, která umožňuje jeho využití v jazyce C# (resp. prostředí .NET). Zveřejnil jsem ji také samostatně pod svobodnou licencí GNU LGPL.

Pro dosažení cílů práce jsem se musel v dostatečné míře seznámit především s principy a specifiky arch. Arm, s jejím aplikačním binárním rozhraním, způsobem překladu, linkování a zavádění programů, s rozhraním knihovny Unicorn, s rozhraním editoru Visual Studio Code a s podpůrnými protokoly *Language Server Protocol* a *Debug Adapter Protocol*. Využil jsem také jazyk TypeScript nebo technologii SignalR, se kterými jsem měl dříve jen minimální zkušenosti. Řešení této práce také významně prohloubilo mé znalosti regulárních výrazů a prostředků pro interakci mezi kódem v jazyce C# a nativním kódem v jazyce C.

Funkčnost řešení jsem úspěšně ověřil na systémech Windows (32b i 64b verze arch. x86), GNU/Linux (x86_64, 32b i 64b verze arch. Armv8) a macOS (x86_64). Uživatelské testování proběhlo pouze v omezené míře, neboť v čase vzniku práce nebyl otevřen kurz Pokročilé assembly na FIT VUT v Brně, pro jehož studenty je nástroj primárně určen.

Na základě provedených testů považuji simulační část implementovaného nástroje za plně vyhovující pro účely výuky. Řešení pokrývá prakticky celou instrukční sadu a nabízí uživatelům vhodné prostředky pro ladění a pochopení činnosti jednotlivých instrukcí. V části poskytující jazykovou podporu jsem zvolil méně vhodný způsob analýzy zdrojového textu, je v ní tedy prostor pro zlepšení, ale i přesto si myslím, že ve srovnání s dosavadními nástroji poskytuje více prostředků usnadňujících uživateli vývoj.

Nástroj nabízí široký potenciál k dalšímu vylepšování. V krátkodobém horizontu plánuji upravit jazykový server tak, aby využíval spolehlivější algoritmus analýzy vstupního textu. Také hodlám vylepšit vnitřní chování jedné části simulační služby způsobem, který zvýší její flexibilitu. Tyto změny pak umožní relativně jednoduše doplnit v řešení podporu

instrukční sady x86, díky čemuž by mohlo být použito i ve výuce jiných předmětů na FIT, případně také 64b varianty instrukční sady Armv8. Výhledově bych také v budoucnu rád pracoval na implementaci simulace většího množství systémových volání linuxového jádra s cílem umožnit zavedení standardní knihovny jazyka C (a tedy i jednodušších programů v tomto jazyce). Vhodné by bylo doplnění dalších didakticky zaměřených funkcí, např. vizualizace řetězeného zpracování instrukcí. Použitá architektura řešení umožňuje také budoucí rozšíření o různé možnosti interakce mezi studentem a učitelem, a to i v reálném čase.

Literatura

- [1] ALCORN, P. *AMD's CPU Market Share and Revenue Jump as Apple's M1 Arm Chips Rise in PCs* [online]. Srpen 2021 [cit. 2022-03-25]. Dostupné z: <https://www.tomshardware.com/news/amds-cpu-market-share-and-revenue-jump-as-apples-m1-arm-chips-rise>.
- [2] ARM LIMITED. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* [online]. Verze C.d. Cambridge (UK): Arm Limited, březen 2018 [cit. 2022-06-14]. 2720 s. Dostupné z: <https://developer.arm.com/documentation/ddi0406/cd>.
- [3] ARM LIMITED. *Arm Architecture Reference Manual Armv8, for A-profile architecture* [online]. Verze G.b. Cambridge (UK): Arm Limited, červenec 2021 [cit. 2021-12-08]. 8696 s. Dostupné z: <https://developer.arm.com/documentation/ddi0487/gb>.
- [4] ARM LIMITED. *Arm Architecture Reference Manual Supplement Armv8, for R-profile AArch64 architecture* [online]. Verze A.d. Cambridge (UK): Arm Limited, prosinec 2021 [cit. 2022-06-14]. 570 s. Dostupné z: <https://developer.arm.com/documentation/ddi0600/ad>.
- [5] ARM LIMITED. *What is Instruction Set Architecture (ISA)?* [online]. 2021 [cit. 2021-12-08]. Dostupné z: <https://www.arm.com/glossary/isa>.
- [6] ARM LIMITED. *Armv8-M Architecture Reference Manual* [online]. Verze B.s. Cambridge (UK): Arm Limited, duben 2022 [cit. 2022-06-14]. 2147 s. Dostupné z: <https://developer.arm.com/documentation/ddi0553/bs>.
- [7] ARM LIMITED. *CPU Architecture* [online]. 2022 [cit. 2022-03-26]. Dostupné z: <https://www.arm.com/architecture/cpu>.
- [8] ARM LIMITED. *Using the Arm Trademark* [online]. 2022 [cit. 2022-03-26]. Dostupné z: <https://www.arm.com/company/policies/trademarks/arm-trademark-list/arm-trademark>.
- [9] BIRRELL, A. D. a NELSON, B. J. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*. New York, NY, USA: Association for Computing Machinery. Únor 1984, sv. 2, č. 1, s. 39–59. DOI: 10.1145/2080.357392. ISSN 0734-2071. Dostupné z: <https://doi.org/10.1145/2080.357392>.
- [10] ELSNER, D. a FENLASON, J. *Using as* [online]. Verze 2.38. Boston (MA): Free Software Foundation, únor 2022 [cit. 2022-03-25]. 428 s. Dostupné z: <https://sourceware.org/binutils/docs-2.38/as.pdf>.
- [11] GAMMA, E. e. a. *Design patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995. Addison-Wesley professional computing series. ISBN 0-201-63361-2.
- [12] HENDERSON, H. *Encyclopedia of computer science and technology*. Revised ed. New York: Facts on file, 2009. Facts on file science library. ISBN 978-0-8160-6382-6.

- [13] HENNESSY, J. a PATTERSON, D. *Computer Architecture: A Quantitative Approach*. 5. vyd. Waltham: Morgan Kaufmann, 2011. ISBN 978-0-12-383872-8.
- [14] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*. 2019. DOI: 10.1109/IEEESTD.2019.8766229.
- [15] JETBRAINS. *The State of Developer Ecosystem in 2021: C Programming* [online]. 2021 [cit. 2022-06-12]. Dostupné z: https://www.jetbrains.com/lp/devecosystem-2021/c/#C_which-compilers-do-you-regularly-use.
- [16] JOHNSON, B. *Visual Studio Code: End-to-End Editing and Debugging Tools for Web Developers*. 1. vyd. Indianapolis: Wiley, 2019. ISBN 978-1-119-58818-4.
- [17] KUNDOJJALA, S. *Arm-based Notebook PC Processor Shipments Jump Five-fold* [online]. Srpen 2021 [cit. 2022-03-25]. Dostupné z: <https://www.strategyanalytics.com/strategy-analytics/blogs/components/handset-components/handset-components/2021/08/12/arm-based-notebook-pc-processor-shipments-jump-five-fold>.
- [18] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A. et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. Červen 2005, sv. 40, č. 6, s. 190–200. DOI: 10.1145/1064978.1065034. ISSN 0362-1340. Dostupné z: <https://doi.org/10.1145/1064978.1065034>.
- [19] MICROSOFT. *Language Server Protocol: Overview* [online]. 2021 [cit. 2022-01-17]. Dostupné z: <https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/>.
- [20] MICROSOFT. *Visual Studio Code Extension API: Extensions Capabilities Overview* [online]. Srpen 2021 [cit. 2022-01-16]. Dostupné z: <https://code.visualstudio.com/api/extension-capabilities/overview>.
- [21] MICROSOFT. *Visual Studio Code Extension API: Language Server Extension Guide* [online]. Srpen 2021 [cit. 2022-01-16]. Dostupné z: <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>.
- [22] MICROSOFT. *Debug Adapter Protocol: Tools supporting the DAP* [online]. Únor 2022 [cit. 2022-06-03]. Dostupné z: <https://microsoft.github.io/debug-adapter-protocol/implementors/tools/>.
- [23] MICROSOFT. *Language Server Protocol: Tools supporting the LSP* [online]. Leden 2022 [cit. 2022-01-17]. Dostupné z: <https://microsoft.github.io/language-server-protocol/implementors/tools/>.
- [24] MICROSOFT. *Visual Studio Code Extension API: Debugger Extension* [online]. Květen 2022 [cit. 2022-06-03]. Dostupné z: <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>.
- [25] NGUYEN, A. Q. *Unicorn & QEMU* [online]. 2022 [cit. 2022-06-03]. Dostupné z: https://www.unicorn-engine.org/docs/beyond_qemu.html.

- [26] NGUYEN, A. Q. a DANG, H. V. *Unicorn: Next Generation CPU Emulation Framework* [online]. Srpen 2015 [cit. 2022-01-03]. Dostupné z: <https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf>.
- [27] PETRÁČKOVÁ, V. *Akademický slovník cizích slov. I. díl, A-K*. 1. vyd. Praha: Academia, 1995. ISBN 80-200-0523-4.
- [28] RALSTON, A. a MEEK, C. L. *Encyclopedia of computer science*. 1st ed. New York: Van Nostrand Reinhold, 1979. ISBN 0-442-80321-4.
- [29] SALOMON, D. *Assemblers and Loaders*. Chichester (UK): Ellis Horwood, 1992. ISBN 0130525642.
- [30] SARKAR, A. The impact of syntax colouring on program comprehension. In: COLES, M. a OLLIS, G., ed. *Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2015, Bournemouth, UK, July 15-17, 2015*. Psychology of Programming Interest Group, 2015. Dostupné z: <https://ppig.org/papers/2015-ppig-26th-sarkar1/>.
- [31] SMITH, B. *ARM A32 Assembly Language: Guide to Programming 32-bit ARM, Thumb, Floating Point and Neon*. 1. vyd. Great Britain: BSB, 2017. ISBN 978-0-9923916-9-0.
- [32] STACK OVERFLOW. *2021 Developer Survey: Integrated development environment* [online]. 2021 [cit. 2022-01-12]. Dostupné z: <https://insights.stackoverflow.com/survey/2021>.
- [33] TIS COMMITTEE. *Executable and Linking Format (ELF) Specification* [online]. Version 1.2. TIS Committee, květen 1995 [cit. 2022-06-06]. Dostupné z: <https://refspecs.linuxfoundation.org/elf/elf.pdf>.

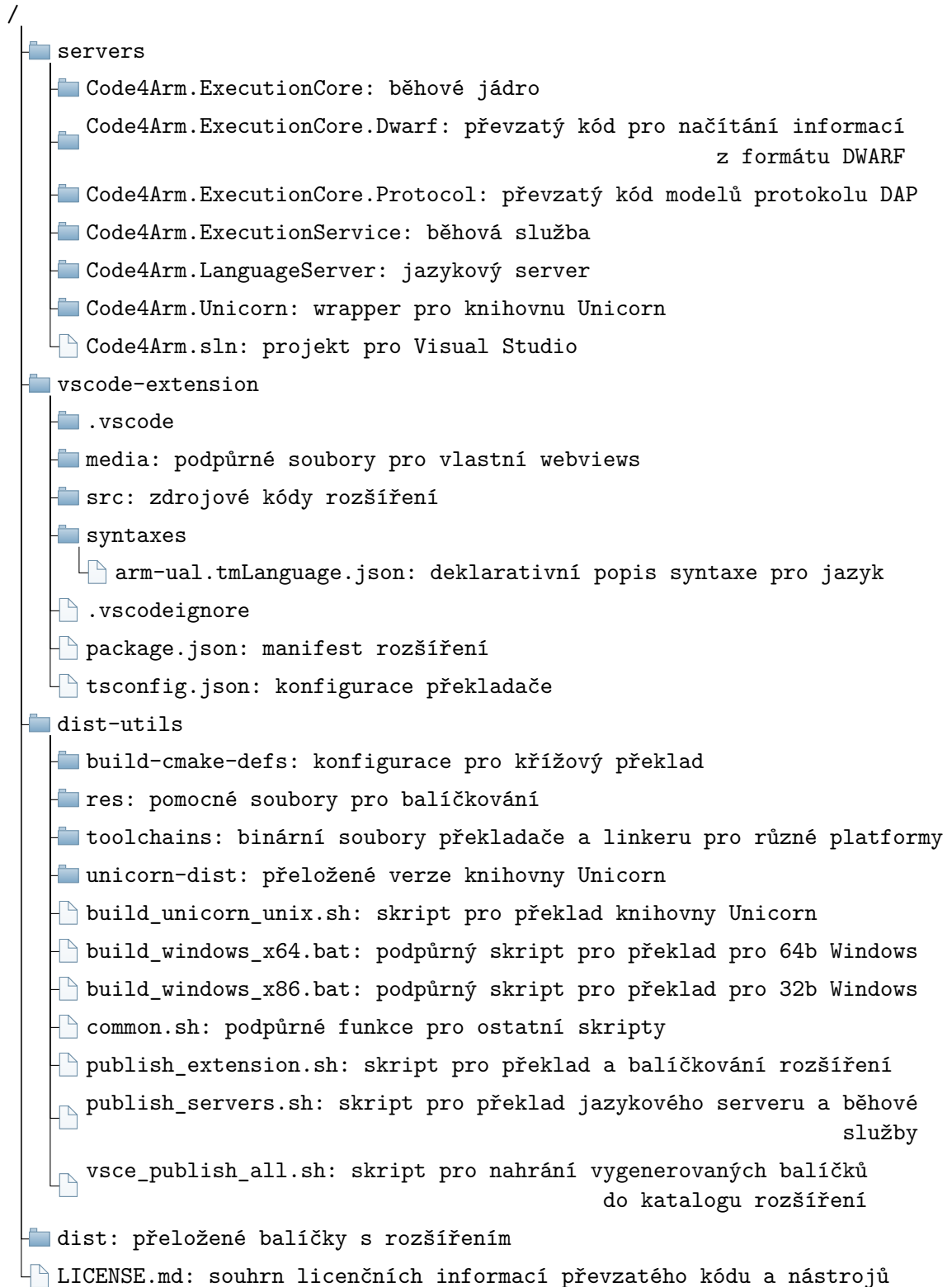
Přílohy

Seznam příloh

A	Obsah paměťového média	86
B	Uživatelský manuál	87
C	Překlad a spuštění ze zdrojových kódů	98
D	Gramatika sledovatelných výrazů	101
E	Podpora požadavků protokolů DAP a LSP	102
F	Překládová služba (diagram tříd)	104
G	Emulační a ladicí služba (diagram tříd)	106
H	Rámce a proměnné (sekvenční diagram)	107
I	Emulační služba (stavový diagram)	108
J	Analýza operandů (pseudokód)	109
K	Srovnání možností ladění	111
L	Podpora instrukcí jazykovým serverem	114
M	Možnosti úpravy systému proměnných	116
N	Ukázky řešení	118
	N.1 Zobrazení proměnných v režimu ladění	119

Příloha A

Obsah paměťového média



Příloha B

Uživatelský manuál

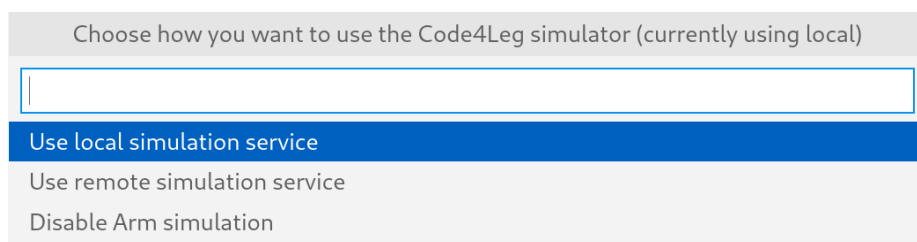
Instalace a prvotní konfigurace

Rozšíření je možné nainstalovat z katalogu rozšíření pro editor Visual Studio Code: <https://marketplace.visualstudio.com/items?itemName=ondryaso.code4arm>. Na této stránce je možné pomocí odkazu *Download extension* stáhnout také VSIX balíček pro instalaci v distribucích editoru, které nepoužívají tento katalog rozšíření. Instalační balíčky pro jednotlivé platformy jsou dostupné také na přiloženém paměťovém médiu.

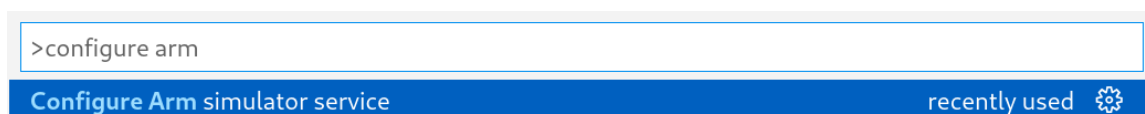
Po prvním spuštění dojde ke stažení prostředí .NET, které rozšíření vyžaduje. Proces stahování je indikován ve stavové liště ve spodní části editoru:



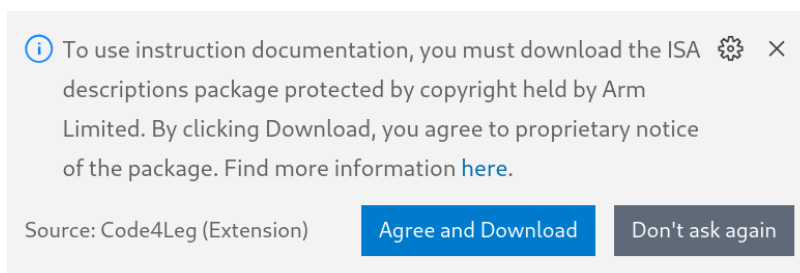
Po stažení prostředí .NET vyzve rozšíření v horní části obrazovky k nastavení služby pro simulaci a ladění:



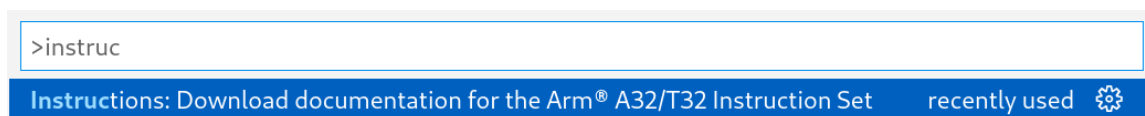
První možnost nastaví použití lokální simulační služby běžící na vašem počítači. Tato možnost nemusí být na některých systémech dostupná. Druhá možnost nastaví použití vzdálené simulační služby. Po jejím zvolení je nutné zadat adresu služby ve formátu adresy URI včetně protokolu: `http://sluzba.xyz` nebo `https://sluzba.xyz`. Třetí možnost deaktivuje možnosti simulace. Pro pozdější změnu nastavení služby je možné použít příkaz v příkazovém okně editoru (Ctrl+Shift+P):



Po nastavení služby vyzve rozšíření také ke stažení souborů s dokumentací instrukční sady:



Pokud stažení souborů odmítnete, je možné výzvu později znovu zobrazit použitím příkazu v příkazovém okně editoru (Ctrl+Shift+P):



Stažením souborů s dokumentací je rozšíření připraveno k použití.

Použití jazykové podpory

Jazyková podpora pro assembler je automaticky dostupná po otevření souboru s příponou `.s/.S`. Je možné ji nastavit v konfiguračním rozhraní editoru (Ctrl+,) v sekci *Extensions* → *Code4Leg* → *Editor*. Jednotlivé konfigurační položky jsou zde podrobně popsány.

Pomocné příkazy pro rozšíření v kódu

Pomocí komentářů se speciálními příkazy je možné ovlivnit chování některých jazykových funkcí. Tyto příkazy se zadávají pomocí jednořádkového komentáře `//` následovaného symbolem `#`:

`//#region` začne vlastní blok pro skládání kódu.

`//#endregion` ukončuje poslední započatý blok pro skládání kódu.

`//#suppress` začne blok, ve kterém nebudou hlášeny syntaktické chyby.

`//#endsuppress` ukončuje všechny započaté předchozí bloky.

`//#!` zamezí hlášení chyb na aktuálním řádku.

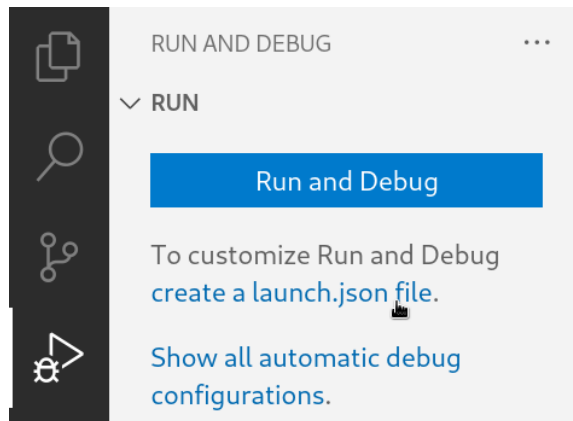
`//#ignore` začne blok, pro který nebude prováděna jazyková analýza.

`//#endignore` ukončuje všechny započaté předchozí bloky.

`//#!` přeskočí při jazykové analýze aktuální řádek.

Simulace a ladění

Jeden právě otevřený soubor je možné přeložit a spustit v simulátoru bez použití konfigurace ladění (souboru `launch.json`) pomocí menu *Run* → *Start Debugging* (F5). Konfiguraci ladění je možné použít pro zahrnutí několika souborů do překladu nebo určení konkrétních nastavení simulační služby. Soubor `launch.json` se všemi nalezenými zdrojovými soubory v assembleru je možné vytvořit v panelu *Run and Debug* (Ctrl+Shift+D):



Konfigurace ladění

Typem konfigurace ladění pro spuštění simulátoru je `code4arm-runtime`. Povinnou vlastností je buď pole `sourceFiles` obsahující absolutní cesty ke zdrojovým souborům, nebo textová vlastnost `sourceDirectory` obsahující absolutní cestu k adresáři, ze kterého budou použity všechny soubory `.s/.S`. Na OS Windows musí být použity správné oddělovače cesty (zpětná lomítka).

Součástí konfigurace ladění mohou být libovolné konfigurační položky simulační služby (viz níže). Hodnota položky v konfiguraci ladění má přednost před hodnotou stejné položky nastavené v konfiguraci editoru.

Příklad konfigurací ladění:

```

1 {
2   "configurations": [
3     {
4       "type": "code4arm-runtime",
5       "request": "launch",
6       "name": "Code4Arm: Files",
7       "sourceFiles": [ "${workspaceFolder}/fileA.s",
8         "${workspaceFolder}/fileB.s" ],
9       "debuggerOptions": {
10        "variableNumberFormat": "Hex"
11      }
12    },
13    {
14      "type": "code4arm-runtime",
15      "request": "launch",
16      "name": "Code4Arm: Directory",
17      "sourceDirectory": "${workspaceFolder}",
18      "executionOptions": {
19        "timeout": 10000
20      }
21    }
22  ]
23 }
```

Nastavení simulační služby

Simulační službu je možné nastavovat v konfiguračním rozhraní editoru (Ctrl+,) v sekcích *Extensions* → *Code4Leg* → *Runtime* (nastavení pro běh programu), *Debugger* (nastavení pro ladici rozhraní v editoru), *Assembling and linking* (nastavení překladač a linkování). Jednotlivé konfigurační položky jsou zde podrobně popsány.

V sekci *Runtime* je možné nastavit především maximální čas běhu, možnosti umisťování a počátečních hodnot v zásobníku, striktní režim (viz níže) a možnosti počátečních hodnot registrů. Adresy je nutné zadat v desítkové soustavě. Při použití vzdálené simulační služby mohou být některé hodnoty omezeny (např. maximální čas běhu nebo velikost zásobníku).

V sekci *Debugger* je možné nastavit zejména obsah okna *Variables*: zobrazovaná data, soustavu pro zobrazování čísel, dělení registrů na jednotky menších datových typů.

V sekci *Assembling and linking* je možné nastavit parametry příkazové řádky při spouštění překladače a linkeru. Při použití vzdálené simulační služby mohou být dostupné parametry omezeny.

Nastavení editoru je možné přepsat na úrovni jednotlivých konfigurací ladění pomocí stejně pojmenovaných vlastností. Název konfiguračního klíče je možné zjistit označením položky v konfiguračním UI, kliknutím na symbol ozubeného kola a zvolením *Copy Setting ID*. Klíče jsou mapovány na vlastnosti v konf. ladění takto:

- `code4arm.runtime` → `executionOptions`;
- `code4arm.debugger` → `debuggerOptions`;
- `code4arm.build` → přímo vlastnosti objektu konfigurace ladění.

Vstupní bod programu

Simulační služba vyžaduje definici symbolu `main`. Do programu je vložen pomocný kód, který obsahuje skok na adresu tohoto symbolu a po skoku zpět (po dokončení funkce `main`) provede ukončovací sekvenci. Pomocný kód navíc obsahuje definici datového symbolu `errno`, který využívají některé simulované funkce. Vhodnou základní kostrou programu je tedy např.:

```
.global main
main:
    PUSH {LR}
    // Kód
    POP {PC}
```

Alternativní možností je definovat symbol `_start`. V takovém případě není do programu pomocný kód vložen. Program by měl zajistit ukončení pomocí softwarového přerušení (viz níže).

Ukončení programu

Vhodným způsobem ukončení programu je použití softwarového přerušení, jehož parametrem v registru R7 je 1:

```
MOV R7, 1
SVC 0
```

Pokud je vstupním bodem programu návěští `main`, preferovanou možností ukončení je skok na adresu, která se při vstupu do programu nachází v registru `LR`.

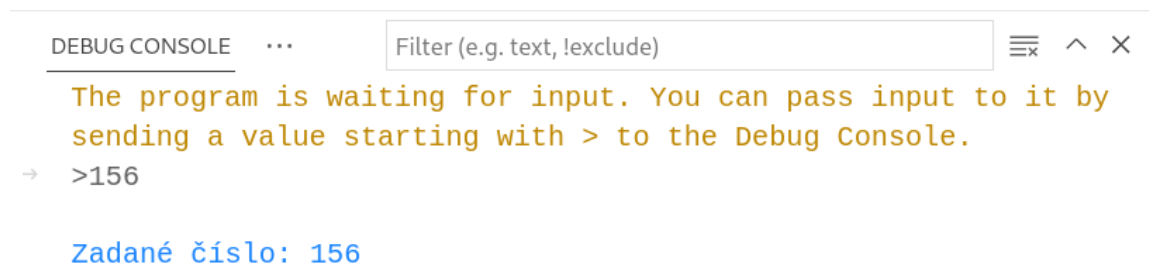
Simulátor ukončí chod programu i po dosažení poslední instrukce v programovém segmentu, to však není korektní způsob ukončení programu.

Vstup a výstup

Textový vstup a výstup může program realizovat voláním simulovaných funkcí. V kódu programu je to realizováno skokem s využitím symbolu, názvu simulované funkce, stejně jako u běžných funkcí. Kompletní popis simulovaných funkcí je uveden v části **Simulované funkce** níže. Příklad volání funkce:

```
BL ReadInt32
MOV R1, R0
LDR R0, =text
BL printf
.data
text: .asciz "Zadané číslo: %d\n"
```

Interakce s programem probíhá v okně *Debug Console* (Ctrl+Shift+Y). Barevně jsou odlišeny zprávy simulační služby a výstup programu. Služba po jisté době nečinnosti upozorňuje, že program čeká na vstup. Protože konzole slouží také pro vyhodnocování výrazů, je nutné vstup pro program uvozovat znakem `>`:



Na konec předávaného vstupu je automaticky připojen znak nového řádku `\n`. Toto chování je možné upravit počtem znaků `>` na začátku vstupu. Programu bude předána vyznačená část následujících vstupů:

- `>text\n`
- `>>text\n`
- `>>>text\n`
- `>>>>text\n`

Všechn zaslaný vstup je ukládán do mezipaměti (bufferu), který čtou příslušné funkce. Funkce nemusí mezipaměť vyprázdnit vždy celou. Vstup je možné zaslat, i když to program zrovna neočekává.

Výrazy

Pomocí okna *Watch* je možné komplexně sledovat a měnit obsah registrů a paměti. Sledovatelný výraz má obecně tvar (složené závorky vyjadřují nepovinnost):

`{{(datový typ)}} výraz {:modifikátor zobrazení}`

K dispozici jsou tři typy výrazů: paměťové, registrové a výrazy hodnoty symbolu. Datový typ a modifikátor zobrazení ovlivňují způsob, jakým bude daný výraz přistupovat k datům a jak je bude zobrazovat.

Dostupné jsou následující datové typy. Celočíselné datové typy mají vždy dvojici bezznaménkový, znaménkový; některé typy mají více názvů.

- 8 b: byte, sbyte
- 16 b: ushort/uhword, short/hword
- 32 b: uint/uword, int/word
- 64 b: ulong/udword, long/dword
- 32b s plovoucí ř. č.: float/single
- 64b s plovoucí ř. č.: double
- textový řetězec: string

Změnu hodnoty výrazu je možné provést v kontextovém menu výrazu (klik pravým tlačítkem → *Set Value*).

Paměťové výrazy

Paměťové výrazy jsou vždy uzavřeny v hranatých závorkách, povinně obsahují zdroj adresy a nepovinně offset a jeho posun/rotace. Tato syntaxe je modifikovanou formou adresování v assembleru Arm.

`[ zdroj, ± zdroj_offsetu, posun konstanta ]`

Zdrojem adresy může být obecný registr, číselná konstanta nebo symbol. Zdrojem offsetu může být obecný registr nebo číselná konstanta. Posun/rotace je řetězec LSL, LSR, ASR, ROR nebo ROL následovaný číselnou konstantou. Konstanty jsou decimální nebo hexadecimální (uvozeny 0x). Příklady validních paměťových výrazů:

- `[ R0 ]` – paměťová položka na adrese R0.
- `[ R0, R1 ]` – paměťová položka na adrese R0 + R1.
- `[ R0, -R1 ]` – paměťová položka na adrese R0 - R1.
- `[ 0x100, R1, LSL 2 ]` – paměťová položka na adrese 0x100 + (R1 << 2).
- `[ symb, 10 ]` – paměťová položka na adrese symb + 10.
- `[ symb, -16, LSR 0xA ]` – paměťová položka na adrese symb + (-16 >> 10).

Počet bajtů přečtených z paměti vždy určuje použitý datový typ. Pokud není žádný typ určený, použije se typ bezznaménkový bajt.

- `[ R0 ]` – čte 1 B.

- `(int)[ R0 ]` – čte 4 B, vrací znaménkové celé číslo.
- `(string)[ R0, -R1 ]` – čte, dokud nenarazí na nulový bajt, vrací data jako textový řetězec.

Při použití celočíselných datových typů mohou být použity modifikátory `:d` (decimální zobrazení), `:x` (hexadecimální zobrazení) nebo `:b` (binární zobrazení).

Při použití datových typů `float` nebo `double` může být použit modifikátor `:ieee`, který do odpovědi přidá rozbor IEEE 754 kódování čísla:

```

▼ (float)[res_pos]:ieee: 5.69824E-13
  sign: 0 (+)
  exponent: -41 (01010110)
  mantissa: 2122789 (01000000110010000100101)

```

Registrové výrazy

Registrové výrazy mají formu názvu cílového registru. Je možné použít registry R0–R15, LR, PC, SP, Q0–Q15, D0–D31, S0–S15. Pokud je určen menší datový typ, je navraceno pole hodnot. Výraz umožňuje vrátit konkrétní hodnotu z pole (např. 3. bajt v registru) pomocí indexování:

- R0 – hodnota R0.
- `(byte)R0` – hodnota R0, seznam jednotlivých bajtů.
- `(byte)R0[2]` – hodnota třetího bajtu v registru R0.

U registrů není možné použít datový typ `(string)`. V současné chvíli není u registrů Q možné použít jiné typy než `(float)` a `(double)`.

Výrazy hodnoty symbolu

Tyto výrazy pouze vrátí hodnotu definovaného symbolu (tou je obvykle adresa). Hodnoty není možné měnit. Není možné s nimi používat datové typy a modifikátor `:ieee`. Jsou uvozeny znakem `&`, např. `&errno`.

Vyhodnocování v ladicí konzoli

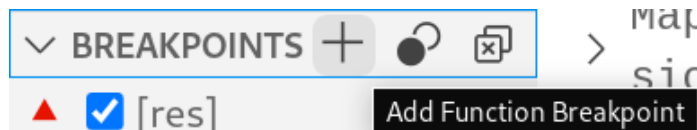
Výrazy je možné také jednorázově vyhodnotit zadáním do vstupního řádku okna *Debug Console* (Ctrl+Shift+Y). Formát výrazů zůstává stejný. Je zde možné také provést přiřazení:

```
[ R0 ] = 0x10
```

Zastavování při změně dat

Datové breakpointy je možné přidat na jednotlivých proměnných v jejich kontextovém menu (klik pravým tlačítkem → *Break on Value Change*).

Editor v současné době neumožňuje přidat datový breakpoint na výraz definovaný v okně *Watch*. Rozšíření umožňuje sledovat změnu hodnoty přidáním výrazu jako funkčního breakpointu:



Syntaxe výrazů je zde stejná jako v okně *Watch*.

Simulované funkce

Chování simulovaných funkcí odpovídá standardu volání procedur architektury Arm:

<https://github.com/ARM-software/abi-aa/blob/main/aapcs32/aapcs32.rst>.

Návratové hodnoty jsou uloženy do registru R0 (a případně dalších). Při volání je důležité dbát na způsob, jakým jsou argumenty ukládány do registrů. Pokud není řečeno jinak, funkce napodobují chování standardní knihovny jazyka C. Datové typy jsou definovány stejně jako v části o výrazech, v závorce jsou uvedeny odpovídající typy jazyka C, pokud se liší.

Vstup a výstup

getchar

Přečte jeden znak z mezipaměti uživatelského vstupu.

Parametry: –

Návratová hodnota: `int`

printf

Vypíše formátovaný řetězec.

Parametry:

1. `uint (const char *)`: ukazatel na formátovací řetězec
2. `...`: jednotlivé položky pro dosazení do řetězce

Návratová hodnota: –

putchar

Vypíše jeden znak. Oproti jazyku C jsou podporovány také 16b UTF-16 hodnoty (v obou případech se argumenty rozšiřují na 32 b, funkci je tedy možné používat v souladu s chováním jazyka C).

Parametry:

1. `ushort (char)`: znak k vypsání.

Návratová hodnota: –

puts

Vypíše textový řetězec uložený ve formátu C, tedy zakončený nulovým bajtem. Obsah paměti je interpretován v kódování určeném konfigurací služby (*Debugger: C String Encoding*), ve výchozím nastavení UTF-8.

Parametry:

1. `uint (const char *)`: ukazatel na řetězec

Návratová hodnota: –

ungetc

Vrátí jeden znak zpět do mezipaměti uživatelského vstupu.

Parametry:

1. `ushort (int)`: navracený znak

Návratová hodnota: `ushort (int)`: navracený znak

Načítání čísel

Následující funkce nejsou standardními funkcemi jazyka C. Pokud je v programu definován symbol `errno`, v případě neúspěchu jedné z těchto funkcí je nastavena hodnota v odkazované paměti na 22 (`EINVAL`).

ReadInt32

Načte řádek (ukončený `\n`) z mezipaměti uživatelského vstupu a načte z něj 32b decimální znaménkové celé číslo. Tolerovány jsou bílé znaky na začátku a na konci vstupu.

Chování odpovídá metodě [Int32.TryParse](#) prostředí .NET.

Parametry: –

Návratová hodnota: `int`, v případě neúspěchu -1

ReadUInt32

Načte řádek (ukončený `\n`) z mezipaměti uživatelského vstupu a načte z něj 32b bezznaménkové celé číslo. Tolerovány jsou bílé znaky na začátku a na konci vstupu.

Číslo může začínat sekvencí `0x` nebo končit znakem `h`, v takovém případě je načteno jako hexadecimální. Číslo může začínat sekvencí `0b` nebo končit znakem `b` (pokud nezačíná `0x`), v takovém případě je načteno jako binární.

Parametry: –

Návratová hodnota: `uint`, v případě neúspěchu 4294967295 (`0xffffffff`)

ReadFloat32

Načte řádek (ukončený `\n`) z mezipaměti uživatelského vstupu a načte z něj 32b číslo s plovoucí řádovou čárkou. Tolerovány jsou bílé znaky na začátku a na konci vstupu. Oddělovačem desetinné části je výhradně tečka.

Chování odpovídá metodě [Single.TryParse](#) prostředí .NET (využívá `InvariantCulture`).

Parametry: –

Návratová hodnota: `float`, v případě neúspěchu `NaN`

ReadInt64

Načte řádek (ukončený `\n`) z mezipaměti uživatelského vstupu a načte z něj 64b decimální znaménkové celé číslo. Tolerovány jsou bílé znaky na začátku a na konci vstupu.

Chování odpovídá metodě [Int64.TryParse](#) prostředí .NET.

Parametry: –

Návratová hodnota: `long`, v případě neúspěchu `-1`

ReadUInt64

Načte řádek (ukončený `\n`) z mezipaměti uživatelského vstupu a načte z něj 64b bezznaménkové celé číslo. Tolerovány jsou bílé znaky na začátku a na konci vstupu.

Číslo může začínat sekvencí `0x` nebo končit znakem `h`, v takovém případě je načteno jako hexadecimální. Číslo může začínat sekvencí `0b` nebo končit znakem `b` (pokud nezačíná `0x`), v takovém případě je načteno jako binární.

Parametry: –

Návratová hodnota: `ulong`, v případě neúspěchu `0xffffffffffffffff`

ReadFloat64

Načte řádek (ukončený `\n`) z mezipaměti uživatelského vstupu a načte z něj 64b číslo s plovoucí řádovou čárkou. Tolerovány jsou bílé znaky na začátku a na konci vstupu. Oddělovačem desetinné části je výhradně tečka.

Chování odpovídá metodě [Double.TryParse](#) prostředí .NET (využívá `InvariantCulture`).

Parametry: –

Návratová hodnota: `double`, v případě neúspěchu `NaN`

Správa dynamické paměti

calloc

Alokuje blok paměti a vynuluje jej.

Parametry:

1. `uint (size_t)`: počet prvků k alokaci
2. `uint (size_t)`: velikost jednoho alokovaného prvku

Návratová hodnota: `uint (void *)`: ukazatel na alokovanou paměť nebo 0, pokud operace selhala

free

Uvolní blok paměti dříve alokovaný pomocí `malloc`, `calloc` nebo `realloc`.

Parametry:

1. `uint (void *)`: ukazatel na alokovanou paměť

Návratová hodnota: –

malloc

Alokuje blok paměti.

Parametry:

1. `uint (size_t)`: velikost paměti k alokaci

Návratová hodnota: `uint (void *)`: ukazatel na alokovanou paměť nebo 0, pokud operace selhala

realloc

Změní velikost alokovaného bloku paměti. Funkce může přesunout blok paměti na jiné místo.

Parametry:

1. `uint (void *)`: ukazatel na původní blok paměti. Pokud je 0, chování je stejné jako by byla volána funkce **malloc**.
2. `uint (size_t)`: nová velikost bloku paměti

Návratová hodnota: `uint (void *)`: ukazatel na alokovanou paměť nebo 0, pokud operace selhala a původní blok paměti nebyl změněn.

Příloha C

Překlad a spuštění ze zdrojových kódů

Potřebné nástroje

- Jazykový server a běhová služba: .NET SDK ve verzi 6¹.
- Rozšíření pro editor VSC: Node.js² (vývoj probíhal na verzi 16.16.0 LTS); editor Visual Studio Code (vývoj probíhal na verzi 1.69.1).
- Překlad knihovny Unicorn: CMake (alespoň ve verzi 3.1); při použití dodaného skriptu verzovací nástroj git;
 - na unixových systémech: pkg-config; pro křížový překlad z x86_64 na arm/arm64 také příslušný překladač³;
 - na Windows: vývojářské nástroje pro jazyk C/C++ z vývojového prostředí Visual Studio 2022⁴; při použití dodaného skriptu prostředí WSL 2.
- Pro balíčkování rozšíření: nástroj vsce⁵ dostupný ve správci balíčků npm⁶.

Jazykový server

Jazykový server je možné samostatně spustit pomocí .NET CLI:

```
cd src/Code4Arm.LanguageServer
dotnet run
```

V konfiguraci *Debug* je ve výchozím nastavení použito nastavení umožňující připojení k serveru pomocí TCP socketů. To je možné deaktivovat odstraněním konstanty `USE_SOCKETS` v souboru projektu `.csproj`. Pak server komunikuje pomocí standardního vstupu a výstupu, což je také výchozí nastavení v konfiguraci *Release*.

Běhová služba

Běhovou službu je možné samostatně spustit pomocí .NET CLI:

```
cd src/Code4Arm.ExecutionService
dotnet run
```

¹.NET SDK: <https://dotnet.microsoft.com/en-us/download/dotnet/6.0>.

²Node.js: <https://nodejs.org/en/download/>.

³Dostupný v repozitářích distribucí nebo na <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/downloads>.

⁴V instalátoru Visual Studio položky „MSVC v143 - VS 2022 C++ x64/x86 build tools (Latest)“

⁵vsce: <https://code.visualstudio.com/api/working-with-extensions/publishing-extension>.

⁶`npm install -g vsce`

Ve výchozím nastavení je služba spuštěna v lokálním módu (tj. pro jednoho klienta, bez navazování sezení nástroje). Pro přepnutí do vzdáleného módu je nutné pro příslušnou konfiguraci definovat konstantu překladu `REMOTE` (např. úpravou souboru projektu).

Rozšíření

Před laděním rozšíření je nutné stáhnout balíčky:

```
cd vscode-extension
npm install
```

Pak je možné rozšíření otevřít v editoru Visual Studio Code. Pro ladění je vhodné upravit soubor `vscode-extension/src/dev_consts.ts`, kde je možné nastavit vývojový mód, ve kterém se rozšíření připojuje k externě spuštěným instancím jazykového serveru a běhové služby. Ty je tak možné spustit např. v jiném IDE v režimu ladění.

Mimo vývojový mód rozšíření spouští vlastní instance serveru a služby, které musí být publikovány v adresáři `vscode-extension/servers`. To je možné zajistit přiloženými skripty, viz níže.

Použití překladových a balíčkovacích skriptů

V adresáři `dist-utils` je sada skriptů pro překlad knihovny Unicorn, publikování jazykového a běhového serveru, překlad a zabalení rozšíření a pro nahrání rozšíření do katalogu.

Překlad knihovny Unicorn

K překladu knihovny Unicorn je určen skript `build_unicorn_unix.sh`. Ten stáhne nejnovější verzi knihovny a přeloží ji pro všechny nastavené platformy. Skript poskytuje následující kombinace platforem pro křížový překlad:

- Z `linux-x86_64` pro:
 - `linux-x86_64`
 - `linux-arm64`
 - `linux-arm`
 - `linux-i386`
 - `windows-x86_64` (pouze ve WSL 2!)
 - `windows-i386` (pouze ve WSL 2!)
- Z `darwin-x86_64` pro:
 - `darwin-x86_64`

Pro křížový překlad pro `arm` nebo `aarch64` je nutné v souboru `common.sh` nastavit proměnné `BINUTILS_` a `ROOT_` na cesty k adresářům obsahujícím překladač, resp. knihovny⁷. Cílové platformy je nutné nastavit úpravou skriptu `build_unicorn_unix.sh` (úplně na konci). Skript je nutné spustit z adresáře `dist-utils`.

⁷Viz https://cmake.org/cmake/help/latest/variable/CMAKE_FIND_ROOT_PATH.html.

Publikace serverů

Produkční verzi jazykového serveru a běhové služby je možné přeložit pomocí skriptu `publish_servers.sh` spuštěného z adresáře `dist-utils`. Servery budou uloženy do adresářů `publish/execution-service-local` a `publish/language-server`.

Překlad a zabalení rozšíření

Rozšíření je možné přeložit a zabalit skriptem `publish_extension.sh`. Před jeho spuštěním je **nutné** provést publikaci serverů a **možné** provést překlad knihovny Unicorn (viz výše). Skript automaticky použije buď vlastní přeložené verze knihovny, nebo přibalené verze z adresáře `unicorn-dist`. Pokud není dostupná knihovna Unicorn nebo toolchain (překladač a linker), balíček pro danou platformu bude vytvořen se zakázaným lokálním režimem běhové služby.

Na konci skriptu je možné upravit cílové platformy. Balíčky jsou uloženy do adresáře `publish/extension`. Skriptem `vsce_publish_all.sh` je možné je nahrát do katalogu rozšíření (pro více informací viz <https://code.visualstudio.com/api/working-with-extensions/publishing-extension>).

Příloha D

Gramatika sledovatelných výrazů

Gramatika je popsána pomocí rozvinuté Backusovy–Naurovy formy (EBNF) podle standardu ISO/IEC 14977 s vlastními úpravami popsanými dále. Upozorňuji na následující jev: notace { *x* }– značí opakování neterminálu *x*, ale s alespoň jedním výskytem.

Výraz použitelný v okně *Watch* je označen neterminálem *in_expr*, výraz použitelný v okně *Debug Console* je označen neterminálem *in_expr_repl*. Neterminál *string* označuje libovolnou sekvenci UTF-8 znaků, neterminál *digit* označuje číslici (0–9). Symbolický zápis se třemi tečkami u neterminálů *reg_name*, *s_reg_name*, *d_reg_name* a *q_reg_name* označuje vyjmenování dostupných názvů příslušného typu registrů.

```
in_expr_repl = [ type ], expr, [ format ], [ "=", string ]
              | inp, string | "*", [ type ], expr ;
in_expr      = [ type ], expr, [ format ] | direct_reference_expr ;
inp          = ">" | ">>" | ">>>" | ">>>>" ;

type         = "(", type_name, ")" ;
type_name    = "float" | "single" | "double" | "byte" | "short"
              | "hword" | "int" | "word" | "long" | "dword"
              | "sbyte" | "ushort" | "uhword" | "uint" | "uword"
              | "ulong" | "udword" | "string" ;
format       = ":", format_type ;
format_type  = "x" | "b" | "ieee" | "d" ;

direct_reference_expr = "!!!", { digit }–, ".", string ;
expr = addressing_expr | register_expr | variable_path | symbol_addr ;

addressing_expr = "[", address_expr, [ ",", expr_offset ], "]" ;
address_expr    = reg_name | imm | symbol ;
expr_offset     = sign, offset_expr, [ ",", shift " " imm ] ;
offset_expr     = reg_name | imm ;
sign            = "+" | "-" | "" ;
reg_name        = "R0" | "R1" | ... | "R15" | "PC" | "LR" | "SP" ;
shift           = "LSL" | "LSR" | "ASR" | "ROR" | "ROL" ;
imm             = [ "0x" ], { digit }– ;
symbol          = string ;

variable_path = string, { ".", string }, [ "." ] ;
symbol_addr   = "&", string ;
register_expr = reg_name, [ indexer ] | s_reg_name, [ indexer ]
              | d_reg_name, [ indexer ] | q_reg_name, [ indexer ] ;

s_reg_name    = "S0" | "S1" | ... | "S15" ;
d_reg_name    = "D0" | "D1" | ... | "D31" ;
q_reg_name    = "Q0" | "Q1" | ... | "Q15" ;
indexer       = "[", { digit }–, "]" ;
```

Příloha E

Podpora požadavků protokolů DAP a LSP

Řešení využívá protokol Debug Adapter Protocol ve verzi 1.49.x, jehož specifikace je dostupná na <https://microsoft.github.io/debug-adapter-protocol/specification>.

Podporuje jeho následující schopnosti (*capabilities*):

- DataBreakpoints
- DisassembleRequest (zpětný převod do JSA)
- InstructionBreakpoints (nastavování breakpointů v okně *Disassembly*)
- LogPoints
- RestartRequest (řízený restart programu)
- SetExpression (změny hodnot výrazů)
- SetVariable (změny hodnot proměnných)
- StepBack (zpětné krokování)
- TerminateRequest (vynucené ukončení programu)
- BreakpointLocationsRequest (získání možných poloh breakpointů)
- ConfigurationDoneRequest (oznámení dokončení konfigurace)
- EvaluateForHovers (vyhodnocování výrazů v kódu při najetí kurzorem myši)
- ExceptionInfoRequest (informace o výjimce)
- GotoTargetRequest (získání možných cílů skoku)
- LoadedSourcesRequest (získání seznamu zdrojových souborů)
- ReadMemoryRequest (čtení paměti procesu)
- ValueFormattingOptions (vyžádání konkrétního formátu čísla)
- WriteMemoryRequest (zápis do paměti procesu)

Řešení nepodporuje následující schopnosti:

- CancelRequest
- ClipboardContext
- CompletionsRequest
- ConditionalBreakpoints
- ExceptionOptions
- FunctionBreakpoints
- ModulesRequest
- RestartFrame
- SteppingGranularity
- SuspendDebuggee
- TerminateDebuggee
- ExceptionFilterOptions
- HitConditionalBreakpoints
- TerminateThreadsRequest
- DelayedStackTraceLoading
- StepInTargetsRequest

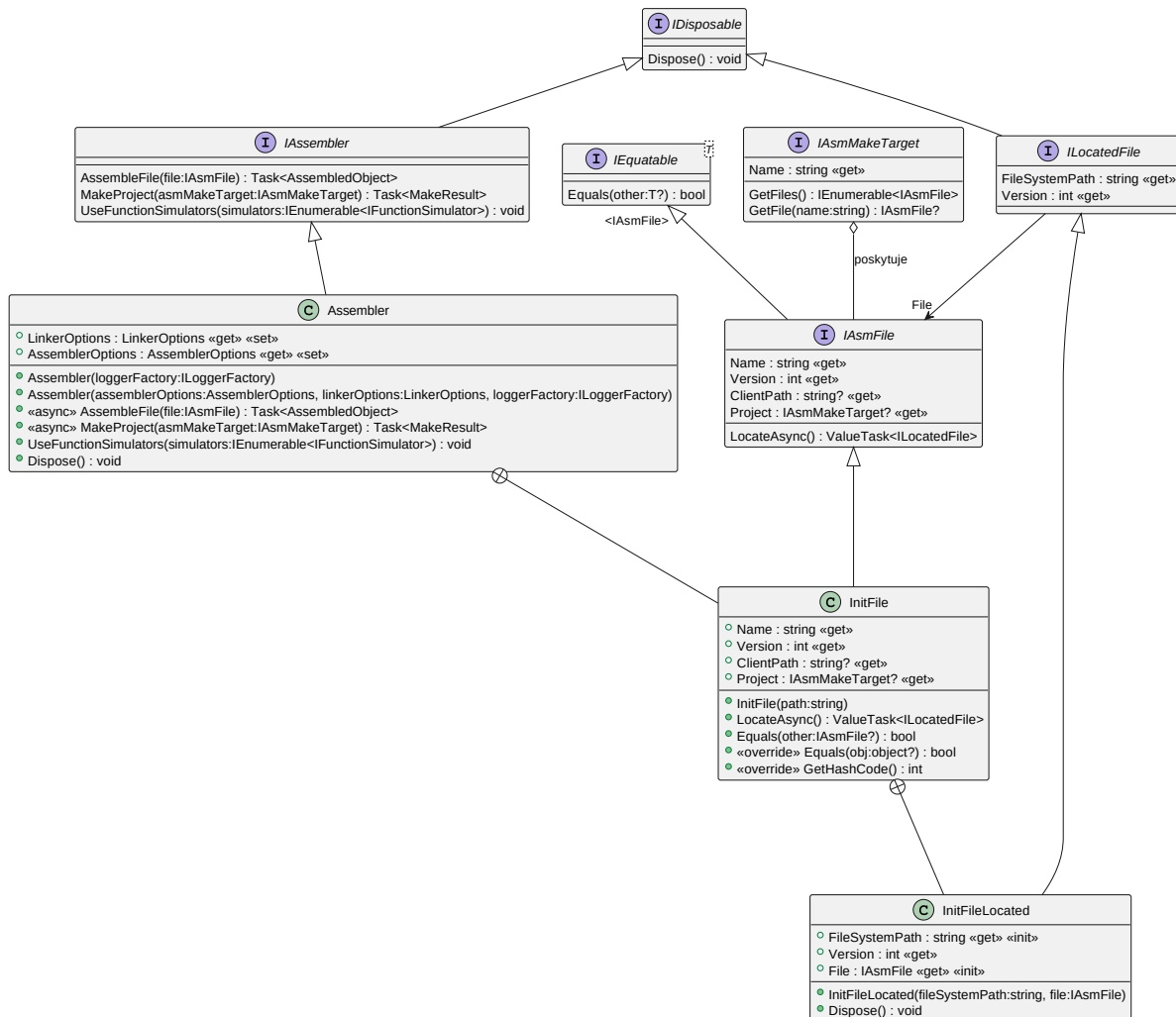
Řešení využívá protokol Language Server Protocol ve verzi 3.16, jehož specifikace je dostupná na <https://microsoft.github.io/language-server-protocol/specifications/specification-3-16/>. Podporuje následující požadavky:

- didOpen, didChange, didClose (synchronizace dokumentů)
- completion (našeptávání)
- signatureHelp (náповěda k signaturám)
- definition (vyhledávání místa definice symbolu)
- references (vyhledávání míst použití symbolu)
- documentSymbol (seznam symbolů)
- codeLens (doplňkové informace k symbolům)
- foldingRange (skládání bloků kódu)
- semanticTokens (tokenizace)

Příloha F

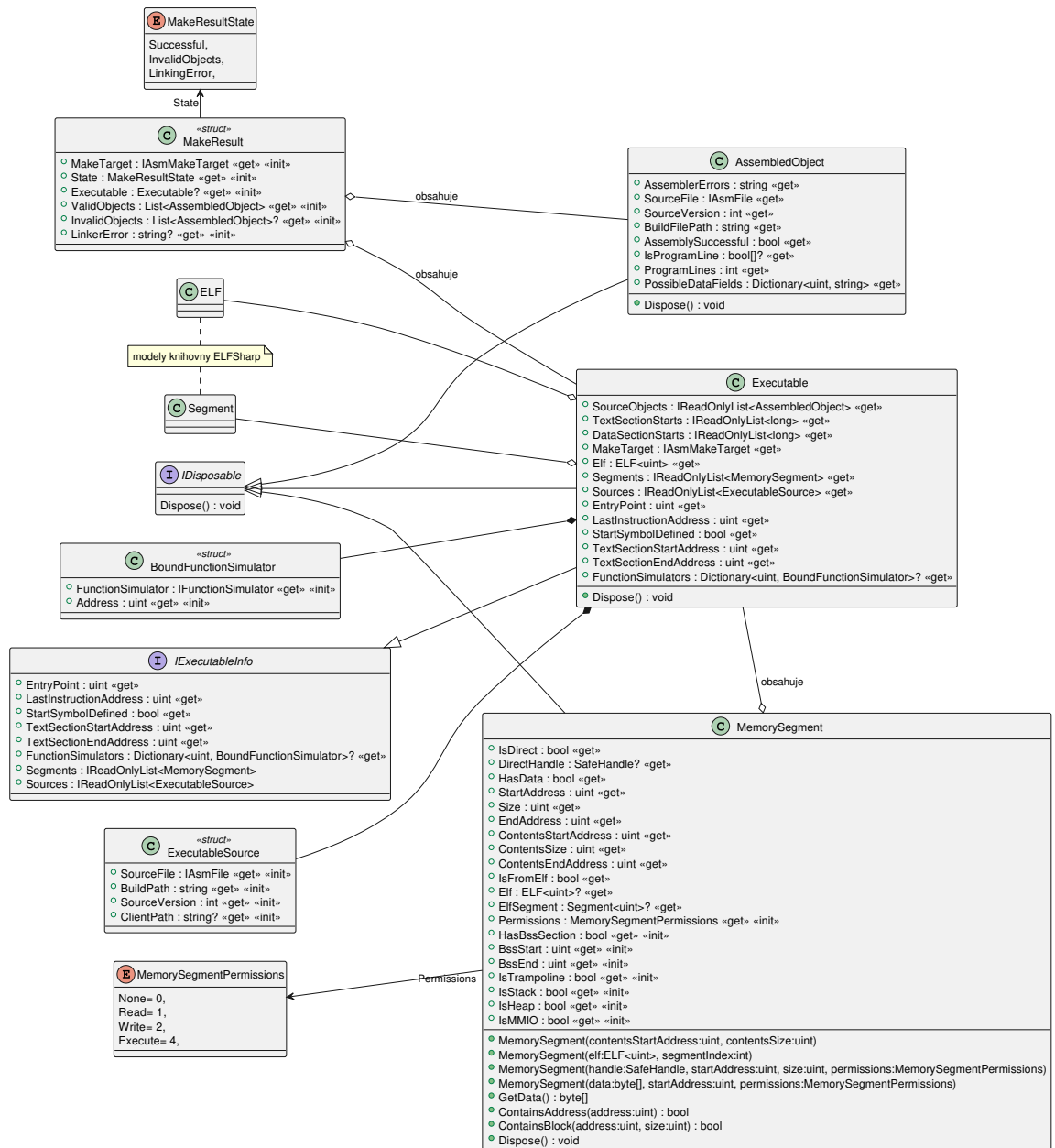
Překladačová služba (diagram tříd)

Následující diagram tříd obsahuje veřejné rozhraní překladačové části běhového jádra, a to včetně rozhraní poskytující abstrakci souborového systému a jeho konkrétní implementaci. Diagram tříd s modely překladačové služby, které reprezentují výstup překladače, je k dispozici na následující straně.



Modely překladačové služby

Následující diagram tříd obsahuje datové modely, které slouží jako nosiče informací o přeloženém programu – jsou tedy výstupem překladačové služby a vstupem emulační služby.



Příloha G

Emulační a ladicí služba (diagram tříd)

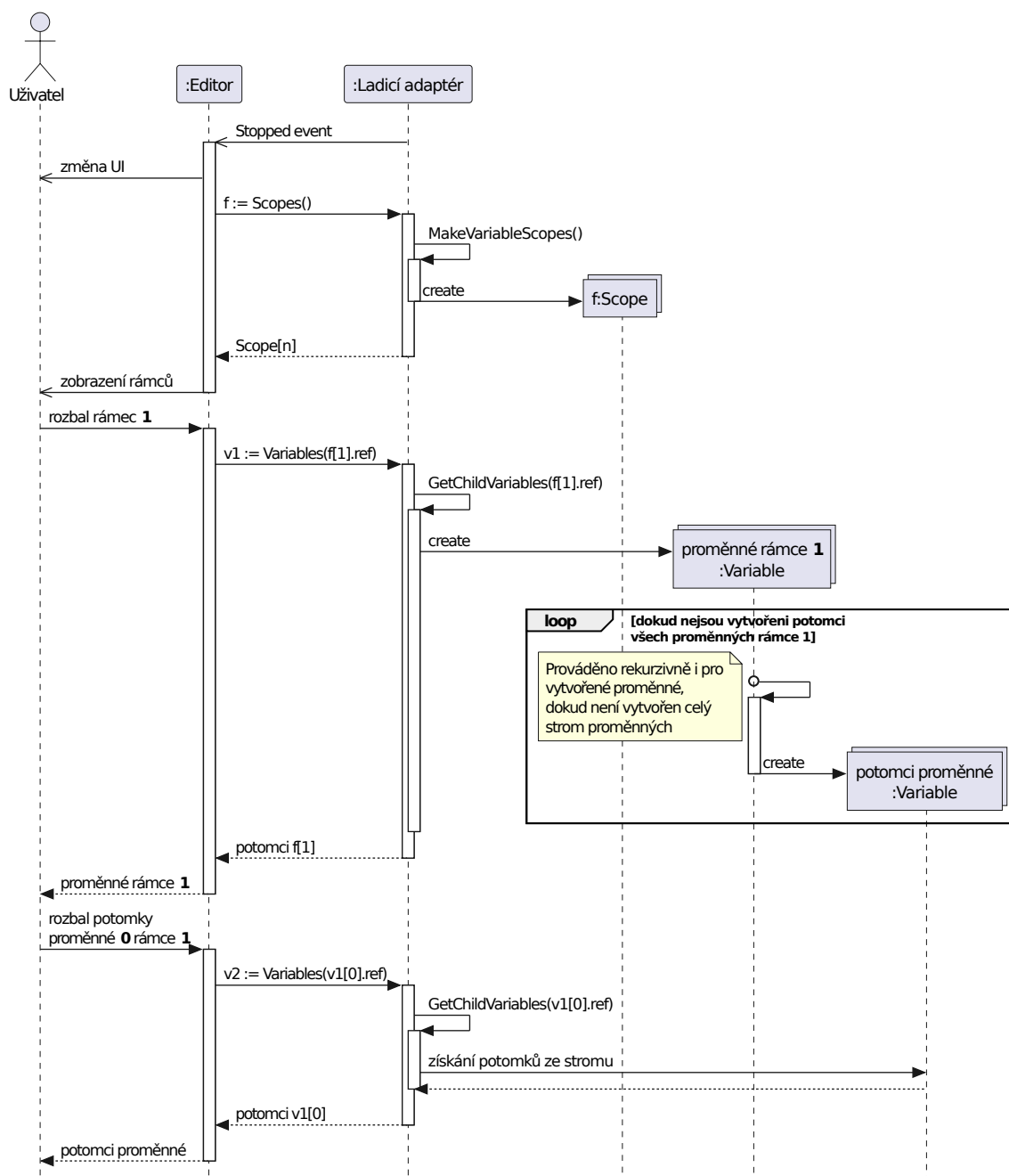
Následující diagram tříd představuje pouze veřejné rozhraní emulační a ladicí části běhového jádra. Záměrně v něm nejsou zahrnuty třídy představující jednotlivé typy zpráv protokolu DAP. Metody implementující konkrétní požadavek protokolu DAP jsou označeny stereotypem «DAP».



Příloha H

Rámce a proměnné (sekvenční diagram)

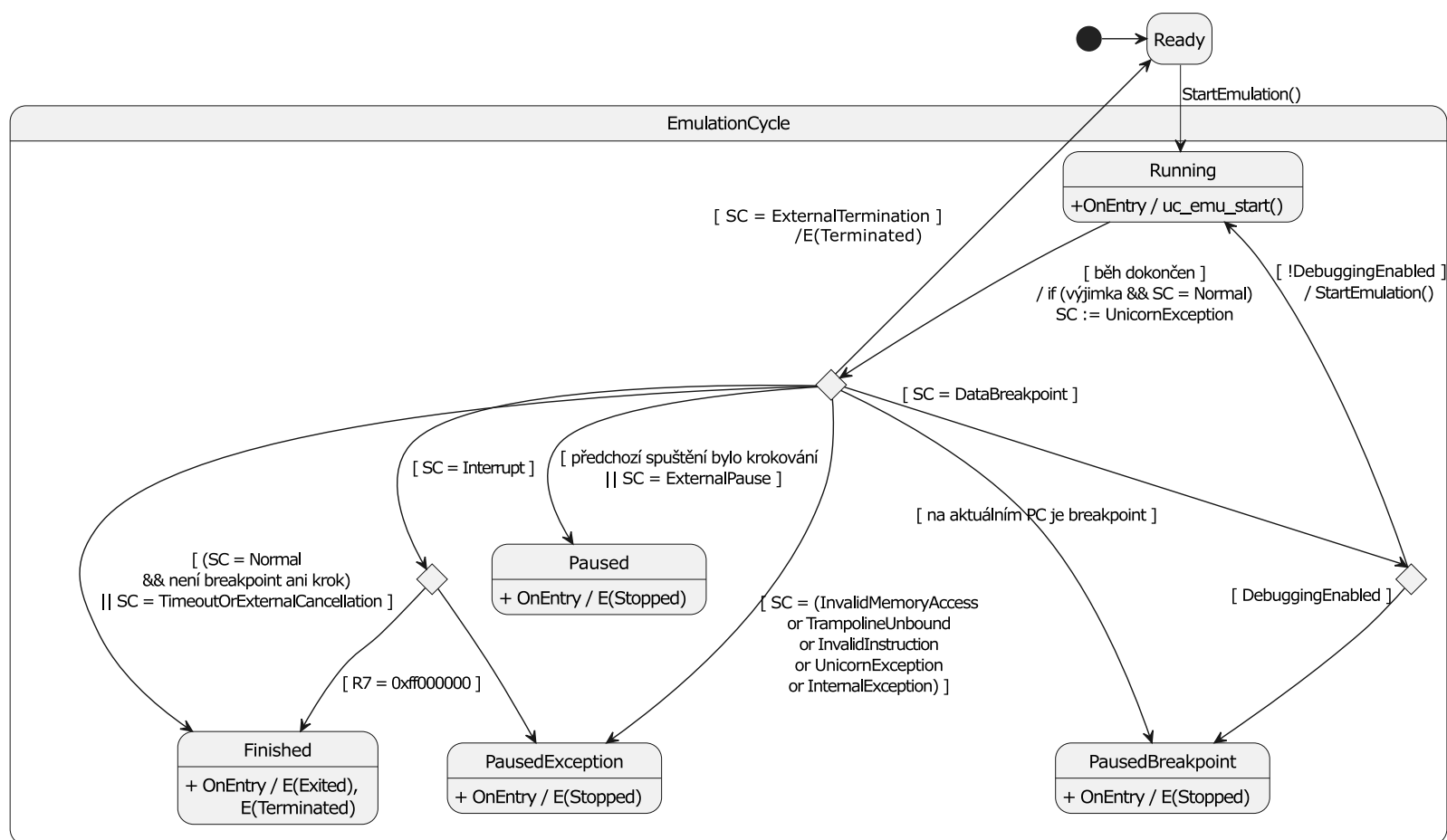
Následující diagram zobrazuje možnou komunikaci editoru a ladicího adaptéru (resp. ladicí služby) po pozastavení kódu (například po dosažení brekapointu).



Příloha I

Emulační služba (stavový diagram)

Symbol SC označuje příznak příčiny konce. Symbol E() označuje zaslání notifikace protokolu DAP. Symbol R7 označuje hodnotu registru.



Příloha J

Analýza operandů (pseudokód)

```
sekv = new sekvence_zanalyzovaných_operandů()
nejdelší_sekv = sekv

for i in (0, počet_operandů_varianty):
    deskriptor = operandy_varianty[i]
    výsledek = Zkonzumuj_Operand(deskriptor, &sekv)

    if !výsledek && deskriptor.nepovinný:
        if sekv.počet >= nejdelší_sekv.počet:
            nejdelší_sekv = sekv

    sekv = new sekvence_zanalyzovaných_operandů()
    continue

// Pokud byl i-tý operand povinný, je výsledek
// analýzy začínající tímto operandem konečný.
break

Zkonzumuj_Operand(deskriptor, &sekv):
    if sekv.stav == 'Konzumace':
        shody = deskriptor.regulární_výrazy.postupně_aplikuj()
        zanalyzovaný_obsah = Zanalyzuj_Obsah(deskriptor, shody)
        sekv.zanalyzované.add(zanalyzovaný_obsah)

    if zanalyzovaný_obsah.neplatný:
        return false
    if deskriptor.poslední:
        sekv.stav = 'Poslední_Zkonzumován'
    else:
        sekv.stav = 'Zkonzumován'

    return Zkonzumuj_Operand(deskriptor.následující, &sekv)
else if sekv.stav == 'Zkonzumován':
    if následuje_čárka && deskriptor.je_poslední:
        // Za operandem nalezena čárka, analyzuj další operand
        // podobně jako výše (s uložením stavu sekvence pro
        // případ nepovinného operandu).
        sekv.stav = 'Konzumace'
        nejdelší_sekv = sekv
```

```

// pokračování těla podmínky z předchozí strany
for i in (deskriptor.index, počet_operandů_variandy):
    možné_pokračování_sekv = Duplikuj(sekv)
    další_deskriptor = operandy_variandy[i]
    výsledek = Zkonzumuj_Operand(další_deskriptor,
                                &možné_pokračování_sekv)

    if !výsledek && další_deskriptor.nepovinný:
        if možné_pokračování_sekv.počet >=
            nejdelší_sekv.počet:
                nejdelší_sekv = možné_pokračování_sekv

        continue

    sekv = možné_pokračování_sekv
    return výsledek

if konec_řádku && deskriptor.poslední:
    return true

// Nenalezena čárka nebo je za posledním operandem
// neočekávaný text.
return false

else if sekv.stav == 'Poslední_Zkonzumován':
    if konec_řádku:
        return true

return false

```

Příloha K

Srovnání možností ladění

	VisUAL 1	ARMStrong	CPUlator	QEMU ¹	Mé řešení
Pokrytí instrukční sady	Minimální	Kompletní ²	Pouze Armv7+FP	Kompletní ³	Kompletní ⁴
Zobrazení hodnot registrů	Ano	Ano	Ano	Ano	Ano
Zobrazení hodnot SIMD/FP registrů	Ne	Ne	Pouze S a D	Ano	Ano (celočíselně i FP)
Úprava hodnot registrů	Ne	Ano	Ano	Pouze v okně Watch	Ano (také po částech a v různých soustavách)
Interpretace hodnot registrů různými datovými typy	Ne	Ne	Ne	Ne přímo ⁵	Ano, plně nastavitelné
Nastavitelné formy zobrazení hodnot registrů	Decimální (sig.), hexadec. (unsig.), binární	Decimální (sig., unsig.), hexadec. (unsig.)	Decimální (sig., unsig.), hexadec. (unsig.)	Pouze jednorázově použitím příkazu gdb	Decimální a hexadec. (sig., unsig.); binární; IEEE 754 FP
Zobrazení podmínkových příznaků NZCV	Ano	Ano	Ano	gdb výraz v okně Watch	Ano
Zobrazení dalších příznakových registrů s popisy příznaků	Ne	Ne	Ne	Ne	Ano
Zobrazení seznamu symbolů	Ano	Ne	Ano	Použitím příkazu gdb	Ano

²Použité jádro umožňuje emulaci prakticky všech instrukcí (viz poznámka níže), ale při spuštění není aktivována jednotka SIMD/FP.

³Nepodařilo se mi zjistit, jaké konkrétní instrukce QEMU podporuje, ale předpokládám, že pokrytí bude téměř úplné.

⁴Viz poznámka výše (využívá QEMU).

⁵Do okna Watch je možné přidat gdb výraz, který bude mít podobnou funkci; může však být poněkud komplexnější, je nutné přidat jej pro každý sledovaný registr zvlášť.

	VisUAL 1	ARMStrong	CPUlator	QEMU	Mé řešení
Zobrazení hodnot na adresách označených symboly	Ano	Ne	Ne	gdb výraz v okně Watch	Ano (automaticky/ručně s určením datového typu)
Zobrazení hodnot na zásobníku	Ano	Ne (pouze pomocí prostředků zobrazení paměti)	Ne (pouze pomocí prostředků zobrazení paměti)	Ne (pouze pomocí prostředků zobrazení paměti)	Ano
Vyhodnocování výrazů pro přístup k paměti a registrům	Ne	Ne	Ne	Podle možností gdb	Ano, registry i paměť, rozsáhlé možnosti adresace
Souhrnné zobrazení a úprava obsahu paměti	Ano	Ano	Ano	Omezené ⁶	Omezené ⁷
Manuální skákání v programu	Ne	Ne	Přepisem PC	Ano	Ano
Zpětné krokování	Omezené	Ne	Ne	Ne	Omezené ⁸
Upozornění na problémové operace v kódu (při běhu)	Detekce nekonečných smyček	Ne	Mnoho možností	Ne ⁹	Omezené ¹⁰
Breakpointy	Ano	Ne	Ano	Ano	Ano
Logpointy	Ne	Ne	Ne	Ano	Ano
Datové breakpointy	Ne	Ne	Pouze paměť	Pouze paměť	Ano, registry nebo jejich části i paměť ¹¹
Podmíněné breakpointy	Ne	Ne	Ano	Ano	Ne

⁶Požadavky DAP implementovány, podpora v editoru ale není příliš intuitivní.

⁷Jako v poznámce výše.

⁸V současnosti zachovává pouze stav registrů, kód je připraven na rozšíření možností zpětného krokování.

⁹V rozsahu poskytovaném nástrojem gdb.

¹⁰Možnost striktně sledovat neplatné přístupy do paměti; možnost omezit dobu běhu.

¹¹Pro nastavování datových breakpointů na paměťových výrazech nenabízí současná verze editoru VSC uživatelské rozhraní, je možné jej vytvořit příkazem v konzoli.

	VisUAL 1	ARMStrong	CPUlator	QEMU	Mé řešení
Vyskakování z funkcí (step out)	Ne	Ne	Ano	Ano	Ne
Sledování volání (call stack)	Ne	Ne	Ano	Ano	Ne
Statistické informace o provádění kódu (counters)	Počet provedených instrukcí	Ne	Ano ¹²	Ne	Ne
Simulace funkcí std. knihovny C	Ne	Ne ¹³	Ne	Ano (plnohodnotná systémová emulace)	Pouze několik jednotek
Simulace systémových volání	Ne	Pouze několik jednotek	Ne ¹⁴	Ano (plnohodnotná systémová emulace)	Ne (v plánu pro další verze)
Textový vstup a výstup	Ne	Ano (vlastní simulované funkce)	Ne ¹⁵	Ano (plnohodnotná systémová emulace)	Ano (simulované funkce std. knih. jazyka C)
Další simulované periferie / MMIO	Ne	Ano (diody, spínače, 7segm. displej)	Ano (mnoho periférií v režimu DE1-SoC)	Ne ¹⁶	Ne (v plánu pro další verze)
Disassembling	Ne	Ne	Ano	Ano	Ano
Randomizace paměti a registrů	Ne	Pouze paměť	Ne	Ne	Ano
Více zdrojových souborů	Ne	Ne	Ne	Ano	Ano

¹²Počet provedených instrukcí, paměťové operace, čas simulace, MIPS.

¹³Nástroj nabízí několik vlastních simulovaných funkcí, jejich použití však není zdokumentováno.

¹⁴Nástroj neumožňuje simulaci systémových volání jádra Linux, ale umožňuje řízení některých simulovaných periférií pomocí softwarových přerušení.

¹⁵V režimu DE1-SoC možnost použít simulovaný UART pomocí MMIO.

¹⁶QEMU poskytuje rozsáhlé možnosti implementace různých vlastních periférií, ty jsou pro didaktické účely ale příliš komplexní a chybí jejich vizuální reprezentace v editoru.

Příloha L

Podpora instrukcí jazykovým serverem

Instrukce jsou v jazykovém serveru rozděleny do tří skupin: základní, rozšířené a pokročilé. Našeptávání operandů je v současné verzi podporováno pouze u základních instrukcí. Varianty se sufixem `-S` jsou vždy podporovány, ale nejsou v seznamu uvedeny. Symbol `*` značí, že jsou zahrnuty všechny instrukce s daným prefixem, viz [3], kap. F5.1. V seznamu nejsou uvedeny SIMD/FP instrukce, pro ty není našeptávání operandů dostupné.

Základní instrukce:

- | | | | | | |
|--------|--------|---------|--------|---------|---------|
| • ADC | • BL | • LDRD | • MVN | • RSC | • STRH |
| • ADD | • CBNZ | • LDRH | • ORN | • SBC | • SUB |
| • ADR | • CBZ | • LDRSB | • ORR | • SDIV | • SVC |
| • AND | • CMN | • LDRSH | • POP | • SMLAL | • TEQ |
| • ASR | • CMP | • LSL | • PUSH | • SMULL | • TST |
| • B | • EOR | • LSR | • RBIT | • STM | • UDIV |
| • BFC | • LDM | • MLA | • ROR | • STR | • UMLAL |
| • BFI | • LDR | • MOV | • RRX | • STRB | • UMULL |
| • BKPT | • LDRB | • MUL | • RSB | • STRD | |

Rozšířené instrukce:

- | | | | | | |
|--------|---------|----------|---------|----------|----------|
| • BLX | • QASX | • SEL | • SMUA* | • UASX | • UQSAX |
| • BX | • QDADD | • SHADD* | • SMUL* | • UBFX | • UQSUB* |
| • CLZ | • QDSUB | • SHASX | • SMUS* | • UHADD* | • USAD* |
| • MLS | • QSAX | • SHSAX | • SSAT* | • UHASX | • USAT* |
| • MOVT | • QSUB* | • SHSUB* | • SSAX | • UHSAX | • USAX |
| • MRS | • REV* | • SMLA* | • SSUB* | • UHSUB* | • USUB* |
| • MSR | • SADD* | • SMLS* | • STC | • UMAAL | • UXT* |
| • PKH* | • SASX | • SMML* | • SXT* | • UQADD* | |
| • QADD | • SBFX | • SMMUL* | • UADD* | • UQASX | |

Pokročilé instrukce:

• BX	• DSB	• LDRBT	• PLD*	• SRS*	• TSB*
• BXJ	• ERET	• LDREX*	• PLI	• SSBB	• UDF
• CLREX	• ESB	• LDRHT	• PSSBB	• STL*	• WFE
• CPS*	• HLT	• LDRSHT	• RFE*	• STRBT	• WFI
• CRC32*	• HVC	• LDRT	• SB	• STREX*	• YIELD
• CSDB	• ISB	• MCR	• SETEND	• STRHT	
• DBG	• IT	• MCRR	• SETPAN	• STRT	
• DCPS*	• LDA*	• MRC	• SEV*	• TBB	
• DMB	• LDC	• MRRC	• SMC	• TBH	

Příloha M

Možnosti úpravy systému proměnných

Tato příloha podrobněji rozebírá nedostatky implementace proměnných (ve smyslu protokolu DAP) a možnosti jejich řešení v budoucí verzi nástroje. Základním problémem je neflexibilita v sestavování stromu proměnných (tedy tvoření jejich potomků) způsobená návrhem, ve kterém:

- objekt proměnné vytváří své potomky sám, ve svém konstruktoru,
- kolekce potomků je zvnějšku neměnná,
- objekt proměnné si sám vytváří svůj identifikátor (RP, reference proměnné, *variables reference*) podle svého typu a svého vnitřního stavu,
- identifikátor je tvořen n -ticí hodnot, které nesou typ proměnné a nějakou identifikující hodnotu (např. číslo registru), ale ne celý vnitřní stav proměnné, identifikátor je jedinečný.

Zvolený návrh neumožňuje dynamické vytváření podstromů, což je překážkou při použití systému proměnných v kontextu vyhodnocovatelných výrazů. Lépe jej demonstruje příklad: v odpovědi na požadavek *Scopes* (pro získání rámců proměnných) je položka { **Název:** "Registers", **RP:** [TypKontejneru.Registry] }. Editor provede požadavek *Variables* pro tuto RP. Server podle pole typ kontejneru v RP určí, že má vytvořit objekty **RegisterVariable**. Přitom se ihned tvoří i objekty reprezentující jejich potomky, které mají dvě úrovně (potomek 32b registru je buď 32b interpretace, nebo kolekce menších hodnot). Proměnné bez potomků mají RP vždy nulovou. Pro jeden registr se tak v paměti vytvoří, v závislosti na konfiguraci, např. následující struktura:

- **RegisterVariable** – název: "R1", RP: [TypKontejneru.PodtypyRegistru, R1]
 - **SubtypeVariable** – název: "unsigned shorts", RP: [TypKontejneru.HodnotyPodtypuRegistru, R1, unsigned bytes]
 - * **SubtypeAtomicVariable** – název: "[0]", RP: 0
 - * **SubtypeAtomicVariable** – název: "[1]", RP: 0
 - **SubtypeVariable** – název: "signed integer", RP: 0
 - **SubtypeVariable** – název: "float", RP: [TypKontejneru.HodnotyPodtypuRegistru, R1, float]
 - * **SinglePrecisionIeeeSegmentVariable** – název: "sign", RP: 0
 - * **SinglePrecisionIeeeSegmentVariable** – název: "exponent", RP: 0
 - * **SinglePrecisionIeeeSegmentVariable** – název: "mantissa", RP: 0

Problém nastane, když uživatel zadá k vyhodnocení výraz (float)R1:ieee. Očekávaným chováním by bylo vytvoření dočasného objektu proměnné, při jehož vyhodnocení je vrácena hodnota registru vyjádřená jako FP číslo, jeho přímými potomky by pak byly objekty **SinglePrecisionIeeeSegmentVariable** reprezentující jednotlivé části FP čísla v kódování IEEE 754. Toho však není možné dosáhnout, protože vznikem nového objektu

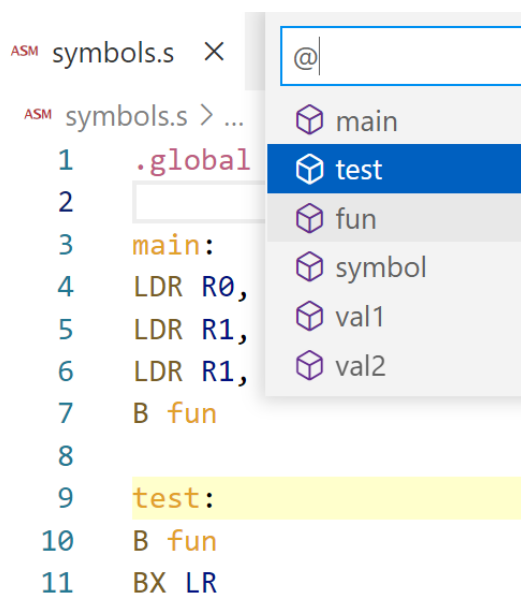
`RegisterVariable` by byl přepsán ten stávající, neboť RP nese informaci pouze o typu proměnné a číslu registru. Změnila by se tak i proměnná v okně *Variables*. Rozšiřování RP o další hodnoty přesněji identifikující jeden objekt proměnné není z obecného pohledu udržitelné. Souvisí s tím také problém nastavení způsobu, jakým jsou hodnoty převáděny na text. Zde je nutné vycházet z uživatelského nastavení, kde je možné přepínat mezi dekadickým, hexadecimálním a binárním zobrazením), z konkrétního použití (u výrazů je možné typ zobrazení nastavovat modifikátorem) a také z kontextu (při použití typu `float` musí být číslo vždy zobrazeno jako reálné číslo v dekadické soustavě). Ten je v současné implementaci poněkud nekonzistentní.

Řešením by byla výraznější úprava systému proměnných, ve kterých budou RP přidělovány dynamicky a neponesou informaci o typu proměnné. Správa RP a jejich přiřazování jednotlivým položkám stromu bude plně v režii nosné třídy `DebugProvider`, příp. by bylo vhodné celou správu proměnných z této třídy vydělit ven, neboť jde o celkem rozsáhlý kód. Co se týče tvorby potomků, nabízí se dva přístupy: buď ji ponechat v režii rodičovských objektů proměnných a přesunout ji z konstruktoru do metod či skupin metod, které budou nabízet dostatečně podrobné možnosti nastavení tvorby, nebo umožnit přiřazování potomků zvnějšku. V tomto případě by ale bylo nutné výrazněji abstrahovat jednotlivé typy proměnných jakožto zdroje dat, aby bylo možné objekty potomkových proměnných implementovat způsobem, který umožní jejich přiřazení k libovolnému rodiči. Toto je v současné implementaci provedeno jen částečně. Druhý přístup nabízí vyšší flexibilitu, přináší ale riziko duplikace kódu a zanesení třídy `DebugProvider` velkým množstvím kódu pro přípravu jednotlivých typů proměnných. Ideální by tedy bylo tyto přístupy zkombinovat, nesmí to ale být na úkor přehlednosti kódu.

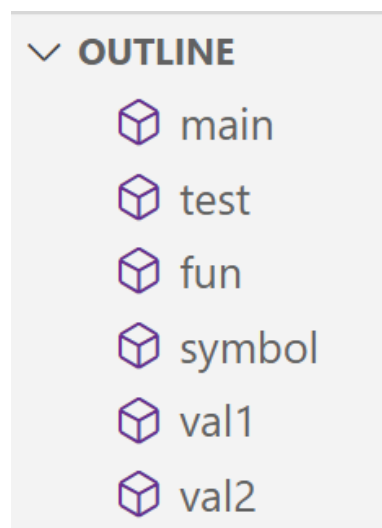
Dalším námětem k vylepšení je využití funkcí frameworku Unicorn pro hromadné čtení hodnot několika registrů najednou, což by vedlo k mírnému zvýšení rychlosti řešení, které by nemuselo provádět volání nativní funkce pro každý registr zvlášť. Otázkou zůstává, jakým způsobem je do celého návrhu zaintegrovat.

Příloha N

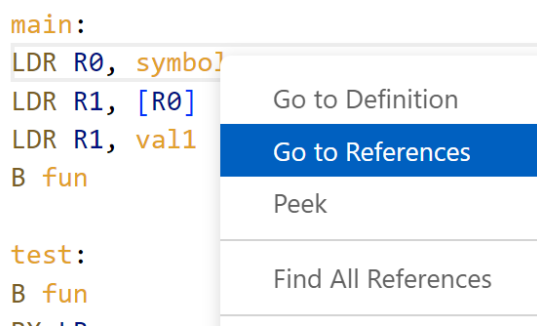
Ukázky řešení



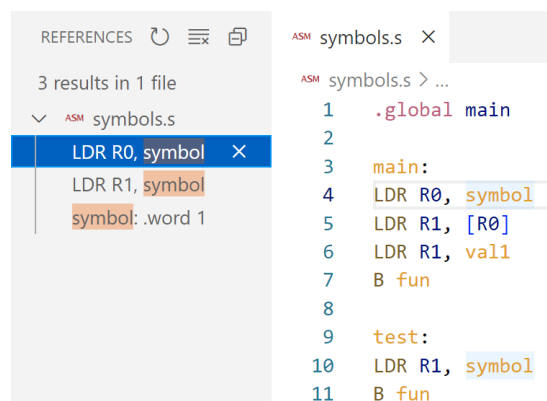
(a) Hledání symbolu pomocí funkce *Go to symbol in editor*.



(b) Seznam symbolů v souboru v podokně *Outline*.

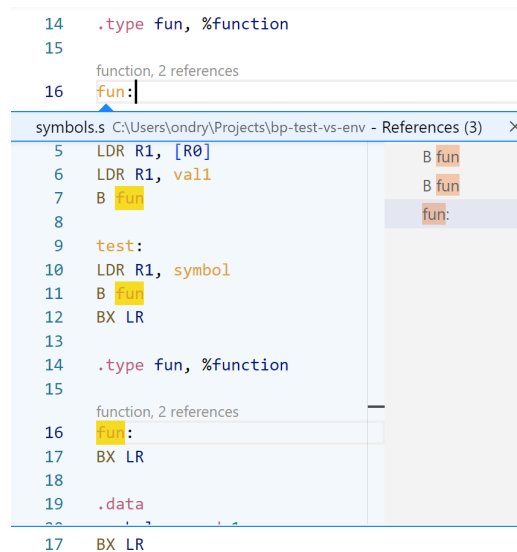


(c) Kontextové menu symbolu v kódu umožňuje přejít na definici nebo hledat všechny výskyty.



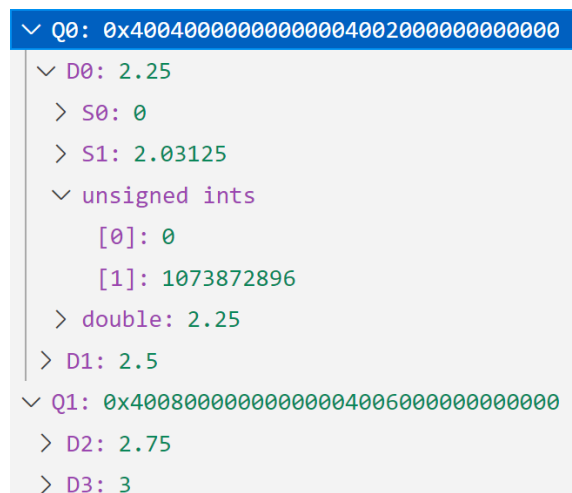
(d) Hledání výskytů symbolu.

Obrázek N.1: Funkce pro práci se symboly v kódu.

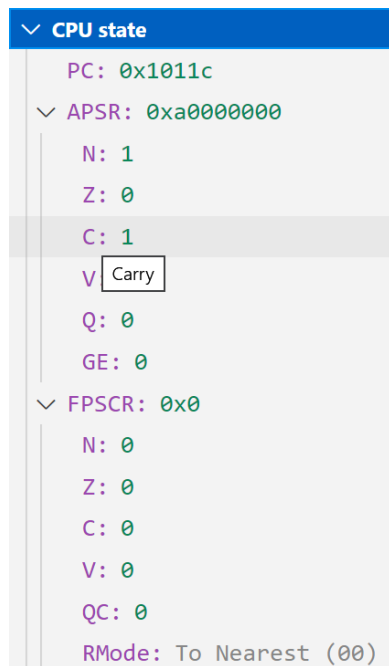


Obrázek N.2: Kontextová informace *CodeLens* zobrazená nad symbolem `fun` označeným na ř. 14 direktivou jako funkční. Po kliknutí na text *CodeLens* se zobrazí okno s náhledem na výskyty tohoto symbolu v kódu.

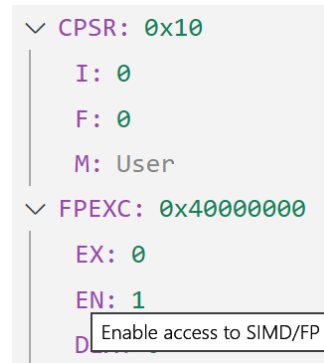
N.1 Zobrazení proměnných v režimu ladění



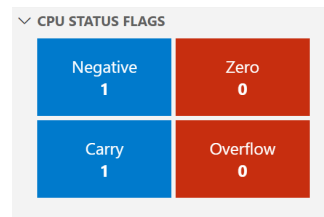
Obrázek N.3: Zobrazení SIMD/FP registrů. Registry nižší třídy jsou zobrazeny jako potomci registrů vyšší třídy, což odpovídá principu jejich fungování (registr nižší třídy je pohledem na registr vyšší třídy). Zobrazeny mohou být také interpretace hodnoty registru jako n -tice menších datových typů, na obrázku je registr D0 rozdělen na dvě 32b celočíselné hodnoty bez znaménka.



(a) Základní zobrazení příznakových registrů (a PC).



(b) Některé příznaky v rozšířeném zobrazení.



(c) Vlastní podokno zobrazující podmínkové příznaky.

Obrázek N.4: Zobrazení příznakových registrů. Na obr. a) a b) jsou vidět některá pole, která je možné nastavovat buď číselnou hodnotou, nebo odpovídajícím názvem možnosti.



Obrázek N.5: Reprezentace dat na zásobníku. Zobrazen je offset od SP, adresa dané položky v paměti, pomyslný index položky na zásobníku a její hodnota v různých datových typech.

▼ Symbols	56	MOV R0, #0
▼ value1: float32 at 0x20148	57	MOV R7, #1
[value1]: 1.25	58	SWI 0
▼ value2: unsigned int32 at 0x2014c	59	
[value2]: 0x29a	60	.data
▼ value3: unsigned short16 at 0x20150	61	value1: .float 1.25
[value3]: 0x4e	62	value2: .word 666
▼ value4: double64 at 0x20152	63	value3: .hword 78
[value4]: 3.141592	64	value4: .double 3.141592
▼ message: string at 0x2015a	65	message:
[message]: This is a string.	66	.asciz "This is a string."
	67	
	68	

Obrázek N.6: Zobrazení odhadnutých typů, adres a hodnot symbolů v datové sekci.