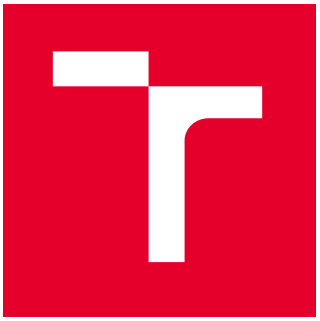


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEPARTMENT OF CONTROL AND INSTRUMENTATION

ROZŠÍŘENÍ PŘEKLADAČE JAZYKA ASSEMBLER O PODPORU NOVÝCH MIKROPROCESORŮ

RETARGETING OF THE ASSEMBLER LANGUAGE COMPILER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Jan Navrátil**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Petr Petyovský, Ph.D.**
SUPERVISOR

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Automatizační a měřicí technika**

Ústav automatizace a měřicí techniky

Student: Jan Navrátil ***ID:*** 221006

Ročník: 3 ***Akademický rok:*** 2021/22

NÁZEV TÉMATU:

Rozšíření překladače jazyka assembler o podporu nových mikroprocesorů

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je navrhnout a realizovat rozšíření překladače jazyka assembler o podporu dalšího embedded mikroprocesoru:

1. Seznamte se s vlastnostmi a architekturou retargetable překladačů jazyka assembler.
2. Nastudujte problematiku definice nového procesoru pro překladač Flat Assembler tak, aby umožňoval generování spustitelného kódu pro nový procesor.
3. Popište vhodnou rodinu embedded mikroprocesorů, kterou překladač nepodporuje. Zvolte pro další práci jeden konkrétní mikroprocesor a diskutujte důvody pro jeho volbu. Zvolte pro mikroprocesor referenční překladač a popište jeho základní vlastnosti.
4. Navrhněte a realizujte vlastní definiční část překladače Flat Assembler pro generování spustitelného kódu pro zvolený mikroprocesor.
5. Zvolte vhodný formát výstupu z backend části překladače tak, aby bylo možné propojit generovaný binární výstup překladu s kódem generovaným z referenčního vývojového prostředí.
6. Důkladně otestujte správnost generovaného spustitelného kódu pro zvolený mikroprocesor.
7. Na zvoleném příkladu ověřte a demonstrujte správnou funkci překladače na reálném HW.
8. Zhodnoťte dosažené výsledky a navrhněte další možná rozšíření.

DOPORUČENÁ LITERATURA:

[1] Knuth E. Donald: Umění programování 1.díl - Základní algoritmy, Computer press 2008, ISBN: 978-80-251-2025-5.

[2] Dvořák V., Drábek V.: Architektura procesorů, Vutium 1999, ISBN: 80-214-1458-8.

Termín zadání: 7.2.2022 ***Termín odevzdání:*** 23.5.2022

Vedoucí práce: Ing. Petr Petyovský, Ph.D.

doc. Ing. Václav Jirsík, CSc.
předseda rady studijního programu

Abstrakt

Bakalářská práce se zabývá implementací modifikace pro retargetable překladač jazyka symbolických adres Flat Assembler G, která umožňuje překlad instrukcí pro rodinu mikroprocesorů HCS08. Práce popisuje stávající řešení tvorby překladačů jazyka assembler, existující obecné překladače a jejich odlišnosti od překladače Flat Assembler G. Dále dokumentuje postup, jak lze pomocí překladače generovat binární výstup ve zvoleném formátu. Poslední část práce se věnuje ověření správnosti implementace a demonstraci na reálném hardware.

Klíčová slova

retargetable assembler, obecný assembler, jazyk symbolických adres, překladač Flat Assembler, fasmg, HCS08, S-record

Abstract

This bachelor thesis describes a design and implementation of modification of retargetable compiler Flat Assembler G that provides a translation of assembly language for HCS08 microprocessor family. It describes current solutions in the design of assembly language compilers, existing retargetable compilers and their differences from Flat Assembler G. Furthermore, it shows process that can generate binary output in selected format with the compiler. Last part of this thesis is dedicated to testing of the correctness of the implementation and demonstration of the correctness on a real hardware.

Keywords

retargetable assembler, assembler language, Flat Assembler compiler, fasmg, HCS08, S-record

Bibliografická citace

NAVRÁTIL, Jan. Rozšíření překladače jazyka assembler o podporu nových mikroprocesorů. Brno, 2022. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/142705. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce Petr Petyovský.

Prohlášení autora o původnosti díla

Jméno a příjmení studenta:	*Jan Navrátil*
VUT ID studenta:	*221006*
Typ práce:	*Bakalářská práce*
Akademický rok:	*2021/22*
Téma závěrečné práce:	Rozšíření překladače jazyka assembler o podporu nových mikroprocesorů

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne: 23. května 2022

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Petrovi Petyovskému Ph. D. za pedagogickou a odbornou pomoc a další cenné rady při konzultování mé bakalářské práce. Dále bych chtěl poděkovat mé rodině, která se mnou měla trpělivost při psaní této práce.

V Brně dne: 23. května 2022

Obsah

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

Úvod

Jazyk symbolických adres (běžně označovaný jako assembler) se používá při psaní programů přímo ovládajících hardware. Typicky se jedná o ovladače zařízení pro operační systémy. Na tyto programy jsou kladeny požadavky na vysokou efektivitu použitého algoritmu. Dále se jazyk symbolických adres používá při programování v kritických aplikacích, kde nesmí docházet k prodlevám, které by mohly být způsobeny například použitím interpretovaného jazyka, případně kde je potřeba z bezpečnostních důvodů mít absolutní kontrolu nad generovaným programem. V neposlední řadě se také používá k programování hardware, pro který neexistuje překladač nebo interpret vyšších programovacích jazyků, případně jeho tvorba je z důvodu specifičnosti cílového hardware složitá.

Standardní překladač jazyka symbolických adres (zkráceně JSA), je určen pro generování strojových instrukcí specifických pro jednu konkrétní rodinu procesorů. Pojem rodina procesorů, případně architektura, zde obvykle označuje soubor pravidel a návrhových řešení (v práci především instrukční sada) společných pro určité procesory. Při vytvoření nové rodiny procesorů je nutné pokaždé vytvořit nový jazyk symbolických adres, přičemž zvolená syntaxe, především pak pseudoinstrukcí, jejichž používání umožňuje efektivnější psaní kódu, je čistě na autorovi a není tudíž jednotná. Zároveň spolu s jazykem je potřeba vytvořit celý překladač. Z tohoto důvodu vznikají tzv. obecné (anglicky retargetable) překladače jazyka assembler, které umožňují používat stejnou syntaxi direktiv ke generování strojového kódu pro různé rodiny procesorů. Navíc jsou navržené jako modulární, takže pro jejich přenos na novou architekturu je potřeba změnit jen architektonicky specifickou část, a nikoliv znovu vytvářet celý překladač.

Tato práce se zabývá právě obecnými překladači jazyka assembler, které umožňují programátorům si snadněji vytvořit překladač pro vlastní architekturu. Mým cílem je srovnat stávající obecné překladače. Dále je mým cílem zdokumentovat na jednom konkrétním obecném překladači proces tvorby rozšíření umožňujícího překlad pro rodinu procesorů, která nebyla dosud pro něj implementována. Nakonec je potřeba otestovat správnost generovaného strojového kódu, a to i na skutečném mikroprocesoru, a porovnat implementované řešení s konvenčním překladačem jazyka assembler.

Práce je členěna do osmi částí takto:

1. V první kapitole jsou rozebrány některé současné obecné překladače jazyka assembler, jejich vlastnosti a vzájemné odlišnosti.
2. Ve druhé kapitole je popsán obecný překladač Flat Assembler G a základy jeho jazyka, který bude dále v práci použit.
3. Třetí kapitola se zabývá popisem rodiny mikroprocesorů, pro kterou bude rozšíření implementováno.

4. Ve čtvrté kapitole jsou rozebrány dva přístupy, jak lze definiční část pro překladač Flat Assembler G implementovat.
5. V páté kapitole je proveden rozbor možných výstupních formátů, kterými lze binární výstup z překladače propojit s kódem z referenčního vývojového prostředí. Poté je zde popsán způsob, jak vytvořit generátor zvoleného formátu pro překladač Flat Assembler G.
6. V šesté kapitole je popsána metodika ověření správnosti překladu do binární i formátované podoby.
7. V sedmé kapitole je popsána demonstrační úloha pro reálný hardware.
8. Osmá kapitola se věnuje testování rychlosti překladu a zhodnocení nedostatků implementovaného řešení.

1. PŘEKLADAČE JAZYKA ASSEMBLER

V této kapitole jsou rozebrány hlavní principy fungování překladačů jazyka assembler. Dále jsou zde představeny vlastnosti a architektura některých současných obecných překladačů.

1.1 Principy překladačů jazyka assembler

Podle knihy Assemblers and Loaders [1] se překladač jazyka assembler dá definovat jako nástroj překládající strojově orientovaný jazyk do strojového jazyka. Tímto se odlišuje od kompilátorů, které jsou určeny k překladu strojově nezávislých vyšších programovacích jazyků orientovaných na řešení problémů do strojového jazyka.

Překladače jazyka assembler ve svých počátcích dokázaly překládat pouze mnemotechnické zkratky neboli operační znaky a adresy do strojového kódu. Překlad probíhal pomocí tabulky s definovanými operačními znaky a mnemotechnickými zkratkami a povolenými typy operandů. Oproti přímému programování ve strojovém kódů tak odpadla nutnost pamatovat si kódy instrukcí. Bylo však potřeba si pamatovat paměťové adresy, se kterými program pracoval, což kladlo velké nároky na programátora a zvyšovalo pravděpodobnost vzniku chyb. To pak vedlo k velké časové náročnosti celé práce.

Tento problém byl vyřešen zavedením návěští, které umožňují dané paměťové místo označit textově. Pro implementaci návěští se začal používat čítač umístění (angl. Location Counter), který každému přeloženému řádku přiřazuje unikátní číslo. Takto pak lze danému řádku instrukce přiřadit textové návěští, jehož název a umístění v kódu se při překladu udržuje v záznamu zvaném tabulka symbolů.

Existuje pak více způsobů, jakými je ošetřeno použití návěští v instrukci předcházející jeho pozdější definici dále v kódu. Ve své práci uvádím dva z nich:

- Prvním způsobem je projít zdrojový kód a pouze identifikovat všechna návěští, zkontrolovat, zda nedošlo k vícenásobné definici a až v momentě, kdy jsou návěštím přiřazeny adresy, kód přeložit. Jelikož je nutno zdrojový kód procházet dvakrát, označují se takové překladače jazyka assembler jako dvouprůchodové.
- Dalším přístupem je při jednom průchodu částečně překládat instrukce, u kterých není předem známa adresa návěští a udržovat jejich místo pomocí lineárního seznamu ukazatelů v kódu. K dokončení překladu takových instrukcí dochází až v momentě, kdy dojde k definici návěští. Takový překladač jazyka assembler se označuje jako jednoprůchodový. Způsob implementace tabulky symbolů je jedním z důležitých faktorů rychlosti překladu. [1]

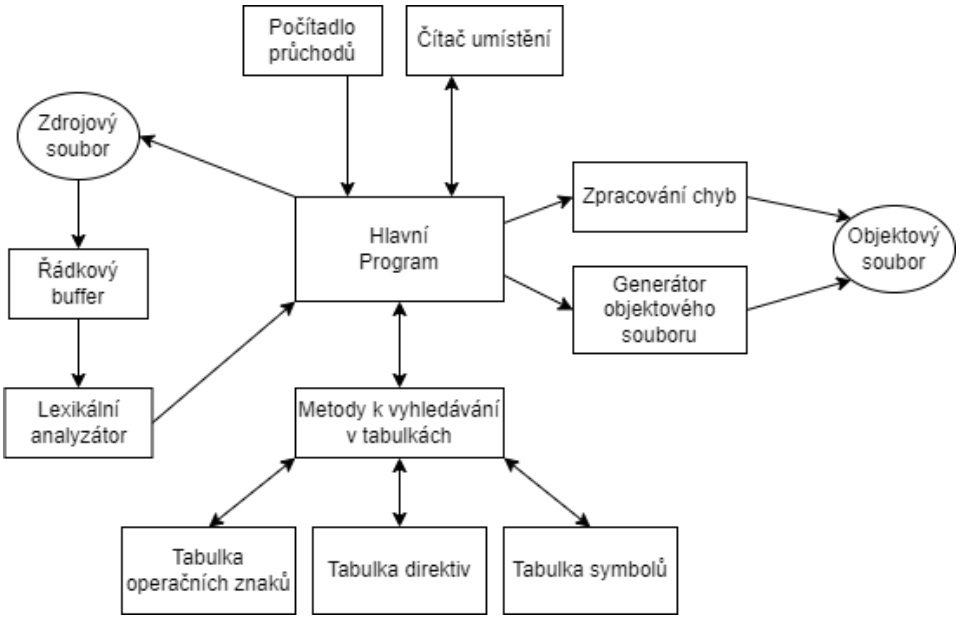
Následně se vývoji překladačů začaly objevovat direktivy (v některých terminologiích také nazývány pseudoinstrukce), které umožňují měnit tok generování výsledného kódu. Mezi nejdůležitější direktivy se řadí definování symbolů, podmíněný

překlad a definování maker. Překladače pracující s direktivami jsou složitější na vytvoření, kde špatná vnitřní implementace může způsobit ve výsledném překladu neočekáváné chování. Jejich přínosem je však daleko efektivnější zápis kódu programátorem.

V neposlední řadě umí moderní překladače společně s linkovacími programy (angl. linkers) optimalizaci strojového kódu, což umožňuje generovat programy s posloupností instrukcí odlišnou od programátorova zápisu. Programy pak provádí stejnou funkci a využívají méně cyklů procesoru nebo paměti. Výhodou těchto překladačů je možnost generování účinnějších programů na daném procesoru bez nutnosti přepisovat zdrojový kód. Umožňují také „tolerovat“ zápis, který může být pro programátora čitelnější, ale nemá přímou podobu v instrukcích strojového kódu. Nevýhodou těchto překladačů je, že programátor vkládá důvěru v autora překladače, že navrhl překladač tak, aby výsledná optimalizace opravdu neovlivnila chování programu.

1.1.1 Relokovatelné překladače jazyka assembler

Mezi další vlastnosti překladačů jazyka assembler patří schopnost relokace kódu. Výstupem relokovatelného překladače může být objektový soubor obsahující určitou reprezentaci přeloženého kódu spolu s dalšími metadaty, který je pak přeložen do spustitelného programu pomocí linkovacího programu. Tím vzniká řešení umožňující vytvořit program z více objektových souborů, jejichž původní zdrojové soubory mohly být napsány v různých jazycích. Vzniklé řešení se například využívá, pokud program píšeme ve vyšším programovacím jazyce, ale některou jeho součást potřebujeme napsat v JSA. Důvodem k takovému postupu může být například nevhodná optimalizace překladu ve vyšším programovacím jazyce.



Obrázek 1.1 Hlavní součásti překladače jazyka Assembler – převzato z [1]

Spustitelný soubor v konkrétním formátu je do paměti nahrán kompatibilním zavaděčem (angl. loader). Absolutní zavaděč pracuje s absolutním objektovým souborem. Jeho účelem je pouze korektně nahrát program do paměti a neprovádí v kódu žádné významné změny. Programátor tak musí brát v zřetel, že všechna paměťová místa, se kterými program pracuje, odpovídají skutečným paměťovým místům. Relokovatelný zavaděč umožňuje nahrát program od určité pozice v neobsazené paměti a zajišťuje přepočet všech adres v kódu tak, aby byly vztaženy k této pozici. [1] [2]

Zde je uveden principiální příklad fungování relokovatelného překladače. Pokud je například ve zdrojovém souboru instrukce pracující s adresovým místem 42 a relokovatelný zavaděč po překladu začne načítat program do paměti na adrese 100, bude instrukce nahraná v paměti počítače ve skutečnosti pracovat s adresovým místem 142.

1.2 Obecné překladače jazyka assembler

Jako obecné překladače jazyka assembler se označují takové překladače JSA, u nichž lze snadno změnit část, která zajišťuje překlad operačních znaků. Tuto část označuji ve zbytku práce jako definiční část. Samotná definice instrukční sady procesoru je v následujících příkladech konkrétních nástrojů implementována různě.

1.2.1 GNU Assembler

GNU assembler (také `gas` nebo jen `as`) je součásti kolekce nástrojů GNU Binutils operujících s binárními soubory. Jeho hlavní použití je pro překlad výstupu z kompilátorů z kolekce `gcc` (GNU Compiler Collection) sloužících k překladu vyšších programovacích jazyků. Ve skutečnosti je `gas` spíše kolekce různých překladačů jazyka assembler, jejichž syntaxe nemusí být vždy stejná.

Samotné jádro `gas` je architektonicky nezávislé a zpracovává výrazy a direktivy společné pro všechny architektury. Zpracovávání instrukcí a některých specifických direktiv umožňuje architektonicky specifická část, která je vytvářena z knihovny operačních kódů, kterou je při přenosu na novou architekturu potřeba vytvořit. Tato knihovna je tvořena soubory `mycpu-dis.c` a `mycpu-opc.c`. První umožňuje implementovat zpětný překlad instrukcí, druhá obsahuje datovou strukturu popisující syntaxi konkrétního jazyka assembler a informace o instrukcích. [3]

Další architektonicky specifická část, se kterou `gas` operuje je knihovna BFD (angl Binary File Description Library), která poskytuje funkce pro manipulaci s objektovými soubory různých formátů (z historického hlediska také zvaných binary file formats). Kvůli své obsáhlosti je náročná na pochopení a následné používání. V článku [4] je popsáno její automatické generování na základě modelu aplikačního binárního rozhraní (angl. Application Binary Interface – ABI), což je model rozhraní umožňující vzájemnou

kompatibilitu součástí programů na úrovni strojového kódu. Tato knihovna je využívána i dalšími nástroji GNU Binutils.

Návod, jak GNU Binutils přenést na novou architekturu, je možné nalézt v [5].

Nevýhodou překladače GNU Assembler je tedy náročnost na znalosti práce s knihovnou BFD. Na druhou stranu je GNU assembler velmi používaný nástroj s širokou komunitou. Co se týče přenosu na nové architektury, existují již zdokumentované nástroje, které umožňují potřebné části generovat automaticky, například pomocí jazyků popisujících architekturu (angl. Architecture Description Language – ADL). [6]

1.2.2 CodeX Assembler

CodeX Assembler (původně `XASM` – extensible assembler) byl již skutečně obecný překladač jazyka assembler, který vznikl při vytváření překladače pro mikroarchitekturu SHAP, která operuje s neobvyklou 9bitovou šířkou bajtu. Pokus o adaptování překladače GNU assembler na tuto architekturu selhal, a tak byl vytvořen nový způsob popisu architektury pomocí definování mapy operačních kódů (Opcode maps).
CodeX Assembler je schopen tyto mapy přeložit a používat bez toho, aby se sám musel modifikovat.

V mapách je kromě mnemotechnických zkratek a operačních kódů zapsána specifikace sady registrů, paměťových segmentů, názvy paměťových operandů a adresovacích módů, které platforma používá. Funkcionalitou šetřící čas při rozšiřování tohoto překladače o novou platformu je možnost dědění. Sada instrukcí pro nový procesor z již adaptované rodiny procesorů spolu se všemi jejími rodiči sdílí určitou základní architekturu reprezentovanou kolekcí tříd v jazyce Java, které zajišťují správné zpracování programu v jazyce symbolických adres. Pokud tato základní architektura již existuje, jediné, co je pro definici nového procesoru potřeba, je nová mapa operačních kódů. V případě, že základní architektura nevyhovuje pro daný procesor, je nutné ji nejdříve vytvořit a poté pro ni vytvořit novou mapu.

Výše uvedený přístup umožňuje snadnou definici procesorů s podobnou architekturou. Na druhou stranu tento překladač neomezují konkrétní implementace map, které nemohou být stejné pro příliš rozdílné architektury. [7]

Poslední verze tohoto překladače byla ovšem vydaná v roce 2003 a není od této doby podporována. Pod původním jménem lze v dnešní době nalézt překladač jazyka assembler pro procesor MOS Technology 6502. Navíc definice nové instrukční sady je v porovnání s řešením, které poskytuje překladač Flat Assembler G v kapitole 2.2, pro návrh od základu zbytečně složitá.

1.2.3 HXA Cross Assembler

`HXA Cross Assembler` (také zvaný Hobby Cross Assembler) je soubor volných překladačů jazyka assembler vyvíjený v programovacím jazyku Thompson AWK v letech 2004 až 2013. Každá varianta HXA je určena k použití pro danou rodinu

procesorů. Existuje tak `HXA` pro rodinu procesorů MOS Technology 6502 a jedna určena pro testovací použití. Obě pomocí direktiv umožňují vytvářet výstup do tří formátů, a to do čistého binárního souboru, objektového souboru Intel HEX a záznamů S-record od společnosti Motorola. [8]

Jednotlivé instrukční sady procesoru jsou implementovány pomocí maker, která se pak musí zahrnout do zdrojového souboru. V současnosti jsou napsány instrukční sady pro pět procesorů z rodiny MOS Technology 6502, mikroprocesory Intel 8080/8085 a Zilog Z80.

I když je `HXA Cross Assembler` v současnosti stále udržovaný, nenalezl jsem žádný zdokumentovaný program, který by tento překladač používal.

1.2.4 Překladač Vasm

`Vasm` je volný, přenositelný obecný překladač jazyka assembler vyvíjený od roku 2002. Ve skutečnosti se jedná o kolekci modulů, které si uživatel vybere při kompilaci své varianty `vasm` pomocí souboru `Makefile`. Lze tak zvolit instrukční sadu požadované cílové architektury, ale i preferovanou syntaxi jazyka assembler, ve které bude program psán. V závislosti na zvoleném modulu syntaxe podporuje `vasm` různé množství direktiv jako jsou makra a opakované a podmíněné překlady. Rovněž je překladač schopen provádět optimalizace.

Ve verzi z července roku 2021 lze `vasm` sestavit pro celkem 15 různých rodin cílových procesorů. Jsou podporované 4 různé syntaxe jazyka assembler a 15 různých typů výstupních souborů. Je nutno ovšem poznamenat, že ne všechny tyto kombinace modulů a výstupních souborů je možné sestavit. Soubory `Makefile` jsou dodávané ve variantách pro nejrůznější i pro dnes již zastaralé operační systémy. [9]

I když projekt `vasm` podporuje velké množství procesorů, nejedná se o obecný assembler, jakým je například Flat Assembler G popsaný v kapitole 2.2. Jednotlivé moduly, se kterými pracuje, jsou programované v jazyce C a nikoliv v samotném jazyku assembler. Definice nového procesoru by i v porovnání s překladačem Flat Assembler G byla náročnější.

1.2.5 Macroassembler AS

Macroassembler AS (nezaměňovat s Microsoft Macro Assembler) je přenositelný obecný víceprůchodový překladač jazyka assembler podporující syntaxi pro přibližně 200 různých procesorů. Byl zveřejněn Alfredem Arnoldem v roce 1992, který ho používal jako vlastnoručně napsaný assembler pro mikroprocesor Motorola 68000 a potřeboval ho rozšířit o instrukce mikroprocesoru 68020. [10] Současné verze jsou napsané v jazyce C a jejich zdrojové kódy jsou volně přístupné.

Tento překladač jazyka assembler umožňuje psát zdrojový kód pro různé procesory v jednom souboru. Mezi různými sadami instrukcí se přepíná pomocí příkazu `CPU` následovaným literátem identifikující daný procesor. Překladem zdrojového souboru

pomocí programu `asw.exe` vznikne soubor kódu s příponou `.p`, který je specifický pro tento překladač a používá svou vlastní strukturu uspořádání dat. Tyto soubory lze pak převést do binárního formátu pomocí programu `p2bin.exe`. Macroassembler AS dále také obsahuje program `p2hex.exe`, který umožňuje soubor kódu převést do 9 různých souborových formátů, mezi kterými je například IntelHEX, Morola S-record, případně i pole dat pro zdrojový kód v jazyce C. Překladač neumí vytvářet kód určený k linkování.

I když Macroassembler AS má funkcionality obecného překladače jazyka assembler, nová instrukční sada se definuje úpravou zdrojového kódu. Postup popisující, které části zdrojového kódu je potřeba pro rozšíření překladače o nový procesor upravit je uveden v manuálu k Macroassembler AS v kapitole I.8. [11]

2. POPIS PŘEKLADAČE FLAT ASSEMBLER

V této kapitole je prezentována historie překladače Flat Assembler. Dále je zde představen na něj navazující obecný překladač Flat Assembler G (zkráceně `fasmg`), který je v rámci této práce použit, principy jeho fungování, používaná syntaxe a jeho typy maker umožňující definici nového procesoru.

2.1 Původní překladač Flat Assembler

Flat Assembler (občas nazývaný flat assembler 1) je volný a otevřený víceprůchodový assembler pro architekturu x86, vytvořený Tomaszem Grysztarem. V současnosti je sám napsán v jazyce Flat Assembler a zkompilované verze existují pro operační systémy Windows, DOS a Linux.

Vyznačuje se vysokou rychlostí kompilace. Z jednoho zdrojového souboru dokáže sestavit výstupní soubor bez fáze linkování. Při dostatečném počtu průchodů je schopen optimalizovat velikost výstupního souboru, který může být ve formátu binárního souboru, objektového souboru formátu ELF (Executable and Linkable Format), COFF (Common Object File Format) nebo spustitelného souboru ve formátu MZ, ELF nebo PE (Portable Executable). [12]

Jednou z dalších klíčových vlastností `fasm` je zásada generování stejného výstupu pro stejný zdrojový kód nezávisle na platformě. Samotný překladač operuje při spuštění z příkazové řádky s minimem přepínačů. Formát výstupního souboru je určen direktivami ve zdrojovém kódu, což zabezpečuje snadnější přenositelnost kódu při překladu na různých operačních systémech.

Vývoj Flat Assembleru započal v roce 1998 a první verze byla publikována v roce 2000. Vznikl jako volná alternativa Turbo Assembleru od firmy Borland, od kterého převzal většinu syntaxe. Byl uzpůsoben pro programování operačních systémů. Proto se chování pseudoinstrukce `ORG` odlišuje od Turbo Assembleru tím, že specifikuje číslo řádku následných instrukcí, ale nezmění jejich fyzické umístění ve výstupním souboru a je tak na programátorově zodpovědnosti, že program bude nahrán na specifické místo v paměti, ve které bude pracovat správně. Jelikož je kód takto psán pro ploché rozložení paměti (Model, kdy procesor může přímo adresovat veškerou dostupnou paměť.), byl tento assembler pojmenován flat (angl. plochý). [13]

2.2 Flat Assembler G

Písmeno g v názvu Flat Assembler G znamená označení obecného překladače jazyka assembler (anglicky general). [14] Vývoj překladače `fasmg` započal v roce 2014. V současnosti je dostupný pro operační systémy Windows, Linux a macOS. Je možné ho zkompilovat ze zdrojových souborů pomocí původního překladače `fasm`, ale i přes

současný fasmg (Toto se anglicky nazývá self-hosted compiling). Základní principy a syntaxe byly převzaty z původního překladače fasm.

Jeho hlavní vlastností je, že sám o sobě neumí překládat instrukce pro žádný procesor. Ovšem díky možnostem pokročilejšího definování maker, ve srovnání s původním fasm, je možné požadovanou instrukční sadu pomocí nich snadněji implementovat. [15]

2.2.1 Syntaxe a rozbor nejpoužívanějších příkazů

V této kapitole rozebereme nejzákladnější principy syntaxe fasmg a příkazů, které budou použity při definici nového procesoru. Některé zde uvedené příklady byly převzaty z [16]. Jedná se o kompletní manuál k programování ve fasmg.

Každý příkaz se vyskytuje na jednom řádku. Pro zlepšení čitelnosti zápisu lze příkaz rozepsat na více řádků, pokud je každý řádek zakončen symbolem zpětného lomítka \. Veškerý text, který se na řádku nachází za znakem středníku je považován za komentář a nemá vliv na překlad. Text zdrojového kódu lze rozdělit na jednotlivé symboly. Speciální znaky +/*=<>()[]{}:?!,.|&~#`\- sami o sobě reprezentují symboly. Ostatní souvislé sekvence znaků neoddělené bílými znaky (například mezerou) jsou považovány za jeden symbol, který může reprezentovat identifikátor (někdy také nazýván název), číslo nebo vyhrazené slovo jazyka. Vyhrazená slova zarhnují operátory nebo klíčová slova, kterými je příkaz pojmenován. Pokud je sekvence znaků z obou stran ohraničena uvozovkami, jedná se o řetězec, který může obsahovat i bílé znaky.

Symbol se stává číslem v momentě, kdy začíná jakoukoliv desítkovou číslicí nebo začíná znakem $ a za ním následuje jakákoliv hexadecimální číslice. S čísly lze provádět příslušné matematické nebo binární operace pomocí příslušných operátoru (+, *, /, <=, bswap, xor a další). Zápis je možný v decimální, hexadecimální, binární i oktálové soustavě, která se explicitně specifikuje napsáním písmene d, h, b nebo o bezprostředně za číslo.

Pakliže symbol není číslem ani vyhrazeným slovem jazyka, stává se identifikátorem a může mu být přiřazena hodnota pomocí operátoru =. Dalším způsobem, jak může identifikátor nabýt hodnoty je, pokud se stane návěštím, které se vytváří umístěním identifikátoru na začátek řádku společně s dvojtečkou na jeho konci. Za dvojtečkou pak může následovat příkaz, jenž přítomnost tohoto návěští nijak neovlivňuje.

```
Navesti: deset = 0Ah    ;Komentář: Identifikátor deset má hodnotu 10 v
hexadecimální soustavě. Navesti má hodnotu čítače lokace na daném řádku.
```

Každý identifikátor má svůj vlastní jmenný prostor obsahující potomky, které můžeme definovat přidáním tečky a názvu potomka. Alternativou je použití příkazu namespace následovaným názvem identifikátoru. K jednotlivým potomkům lze v následném bloku příkazů přistupovat bez použití tečky až po konec bloku ukončeným příkazem end namespace. Pokud se v bloku příkazů mezi namespace a end

`namespace` vyskytuje identifikátor, který není součástí aktuálního jmenného prostoru, prohledávají se rodičovské. Výjimkou jsou identifikátory, které jsou v takovém jmenném prostoru definovány, a mají stejný identifikátor jako jejich rodič.

```
global = 0
regional = 1
    namespace regional
            regional = 2            ; regional.regional = 2
            x = global              ; regional.x = 0
            regional.x = regional   ; regional.regional.x = 2
             global.x = global       ; global.x = 0
    end namespace
```

Jediný způsob, jakým `fasmg` zapisuje do výstupního souboru, je pomocí generování sekvencí bajtů. Lze generovat i hodnoty s větší šířkou pomocí odpovídajících příkazů. Například pomocí příkazu `dw` lze generovat 2 bajty v pořadí little-endian. Případně lze generovat více bajtů za sebou pomocí příkazu `db` následovanou jednotlivými hodnotami oddělenými čárkami. Každý vygenerovaný bajt inkrementuje čítač umístění, který je takto roven počtu vygenerovaných bajtů od začátku souboru.

```
db 13,10    ;Generuje 2 bajty s hodnotami 0x0D a 0x0A
```

Pomocí příkazu `org` následovaným číslem adresy lze změnit hodnotu čítače umístění, čímž lze například ovlivnit hodnoty následně definovaných návěští. Samotná generovaná data se ovšem stále zapisují na první volné místo ve výstupním souboru. Samostatně stojící symbol `$$` má pak hodnotu argumentu poslední vykonané instrukce `org`, neboli nabývá hodnoty současné báze adresovacího prostoru. Samostatně stojící symbol `$` nabývá hodnoty ekvivalentní návěští, které by bylo definované na stejném místě.

```
org 2000h
db 'Zdravim!'
size = $ - $$           ; size = 8
```

Příkaz `equ` umožňuje předcházejícímu identifikátoru okamžitě přiřadit následující symbolický výraz složený z více symbolů. Může být i prázdný.

Pokud nechceme obsah symbolického výrazu vyhodnocovat v době jeho definice, použijeme příkaz `define` následovaný identifikátorem a výrazem, který má reprezentovat.

```
a equ 0*
x equ -a
define y -a
a equ 1*
db x 2          ; db -0*2
db y 2          ; db -1*2
```

Použití operátoru `?` bezprostředně za identifikátorem při jeho definici umožňuje takto definovat identifikátor ve variantách nerozlišující velikost písma. Dále lze identifikátor také nadefinovat z již existujících symbolů, které lze spojovat pomocí operátoru `#`.

Tok překladu se dá větvit pomocí příkazů `if`, `else if` a `else`. Větvení musí být ukončeno instrukcí `end if`. `If` je zpravidla následován logickým výrazem, jehož výsledná hodnota určí, zda se následný blok příkazů provede. Logický výraz je pravdivý, pokud má nenulovou hodnotu. Často se logický výraz utváří pomocí logických operátorů porovnávání rovná se =, nerovná se <>, větší >, menší <, větší než >= a menší než <=.

Opakující se blok příkazů je možno uzavřít mezi příkazy `repeat` a `end repeat`. Za příkazem `repeat` je výraz definující kolikrát se daný blok má opakovat. Samotný výraz `%` uvnitř takového bloku má pak hodnotu rovnu aktuálnímu počtu opakování s počáteční hodnotou rovnou 1. Podobně lze opakovat blok příkazů mezi příkazy `while` a `end while`, kde za příkazem `while` je logický výraz, který je každé opakování testován. Blok se opakuje do doby, dokud není výraz nepravdivý.

```
a = 7
while a > 4
    a = a - 2
end while
db a  ;db 3
repeat 8
      db %
end repeat ;vygeneruje 8 bajtů s hodnotami od 1 do 8
```

Příkaz `iterate` se synonymem `irp` umožňuje opakovat blok příkazů po `end iterate`. Po příkazu `iterate` následuje první parametr určující identifikátor použitý uvnitř bloku a následuje různě dlouhý výčet symbolů, kterých bude daný identifikátor při jednom opakování nabývat. Počet opakování je dán počtem symbolů.

```
irp value, 1,2,3
   db value
end irp     ;Vygeneruje bajty s hodnotami 1,2 a 3.
```

Lze také definovat více identifikátorů v jednom opakování současně, pokud jsou oddělené čárkami a uzavřeny do špičatých závorek a následuje je odpovídající počet parametrů.

```
iterate <name,value>, a,1, b,2, c,3
    name = value
end iterate       ;Provede příkazy a = 1, b = 2, c = 3
```

Příkaz `match` umožňuje porovnávat dvě sekvence znaků se speciálními znaky kromě čárky, otazníku a symbolu rovná se, které mají pro tento příkaz zvláštní význam. Druhým

parametrem je sekvence obsahující jakýkoliv text a ta je porovnána se sekvencí v prvním parametru, přičemž pokud speciální znaky jeden po druhém v obou parametrech odpovídají, jsou symboly v prvním parametru vytvořeny podle druhé sekvence a provede se následující blok příkazů ukončený příkazem `end match`. V opačném případě je celý blok přeskočen, nebo se vykonává blok příkazů za příkazem `else`.

```
define a ##042h
match ##value,a
      db value    ;V případě shody se vygeneruje bajt s hodnotou 42
else
      db 00h      ;V případě neshody se vygeneruje bajt s hodnotou 0
end match
```

Použití znaku `=` v sekvenci v prvním parametru umožňuje porovnávat doslovně bezprostředně následující znak. Lze tak porovnávat jak běžné znaky, tak znaky `?` , `,` a `=`.

Obdobně jako `match` funguje příkaz `rawmatch` se synonymem `rmatch`, který porovnává první argument čistě s textem ve druhém argumentu bez jeho vyhodnocování.

Příkaz `display` a `err` umožňují vypisovat při překladu znaky na standardní výstup. Pokud však překladač zpracuje instrukci `err`, překlad programu se nedokončí.

Příkaz `include` následovaný řetězcem s názvem souboru umožňuje na současném řádku zpracovat zdrojový kód z jinačího souboru. Syntaxe cesty k souboru závisí na použitém operačním systému.

2.2.2 Makra a CALM instrukce

Příkaz `macro` s nejméně jedním argumentem umožňuje definovat nové příkazy, jejichž tělo ukončené příkazem `end macro` se při pozdějším zavolání v textu rozvine. První parametr v definici makra (někdy také nazývaného makroinstrukcí) je identifikátor makra, pod kterým se bude později v kódu volat. Ostatní parametry jsou identifikátory, se kterými makroinstrukce pracuje a přes které jsou ji předávány výrazy při jejím volání. Identifikátoru lze nastavit výchozí hodnotu, která se má za něj dosadit pomocí znaku `:` následovaným výrazem. Tohoto se využije v případě, kdy dojde k volání makra bez uvedení parametrů, se kterými má pracovat. Další způsob je označit při definici makra parametr jako povinný pomocí znaku `*`, kdy volání takové makroinstrukce bez uvedení parametru vyvolá chybu. Uvnitř bloku makra lze pomocí klíčového slova `local` vytvářet symboly unikátní pro každou instanci volání makra.

Ve srovnání s původním překladačem Flat Assembler existují nově tzv. CALM instrukce (angl. Compiled Assembly-like Macro), které jsou navrženy pro definici používané instrukční sady. Nevýhodou klasických maker je, že při překladu pro každou instanci volání makra alokují nový jmenný prostor obsahující lokální symboly, což je činí pomalejšími a náročnějšími na používanou paměť, což se obzvláště projeví s počtem volaných makroinstrukcí v kódu.

CALM instrukce naopak sdílejí své symboly a proměnné napříč všemi svými volanými instancemi a jsou zkompilované již při definici, a to do virtuálního prostoru, který se nezapisuje do výstupního souboru. Kvůli optimalizaci pro rychlost kompilace nepoužívají strukturované příkazy (cykly, větvení) nýbrž skoky. Tímto se podobají spíše samotnému programování v JSA, a odtud získaly i své jméno. [17]

Nevýhodou CALM instrukcí je, že uvnitř svého bloku umožňují používat jen příkazy k tomu uzpůsobené, které se mimo blok CALM instrukce nepoužívají. Rovněž využívají rozdílné programovací paradigma. Výhodou jejich použití je zrychlení a zmenšení paměťové náročnosti překladu.

Následuje srovnání zápisu makra a CALM instrukce:

```
macro octet value*                  ;Parameter s hvězdičkou je povinný.
      db value ;Zapíše do výstupního souboru 2 bajty s hodnotou value.
end macro

octet 042h;Při překladu vytváří nový jmenný prostor se symbolem value.
octet 023h            ;Vytváří další jmenný prostor se symbolem value.
octet   ;Vyvolá chybu kvůli chybějícím argumentům.
```

Příklad zápisu CALM instrukce se stejnou funkcí je v následujícím textu:

```
calminstruction octet value*
      arrange value, =db value      ;Přiřadí text do symbolu value
      assemble value ;Assemble provede příkaz uložený v symbolu value.
end calminstruction

octet 042h;Při překladu vytváří nový jmenný prostor se symbolem value.
octet 023h;Změní hodnotu již existujícího symbolu value a zpracuje ji.
```

Příkaz `arrange` provádí okamžité přiřazení textu do identifikátoru podobně jako příkaz `equ`. Jako první argument má identifikátor, do kterého bude přiřazen text z druhého argumentu. Symboly ve druhém argumentu, které mají být zkopírovány doslovně bez jejich vyhodnocování, musí před sebou mít znak =.

Příkaz `assemble` má vždy jeden argument, a to identifikátor obsahující symbolický výraz, který je okamžitě předán jako příkaz k provedení překladačem. Jedná se tak o jediný způsob, jak provádět běžné příkazy z bloku CALM instrukce.

Další instrukce používané v bloku CALM instrukce jsou popsány na příkladech v kapitole 4.3.

Obecně použití `fasmg` při definování instrukční sady procesoru, ať už pomocí maker nebo CALM instrukcí, je výhodné v tom, že `fasmg` nevyžaduje žádné externí moduly nebo knihovny napsané v jiných jazycích tak, jak tomu je u všech obecných assemblerů popsaných v kapitole 1.2

3. POPIS ARCHITEKTURY HCS08 A VÝVOJOVÉHO PROSTŘEDÍ CODEWARRIOR

V rámci této práce je implementována instrukční sada pro mikrokontroler MC9S08LH64 obsahující mikroprocesor HCS08. Důvodem této volby je, že jsem se již s daným mikrokontrolerem setkal při výuce předmětu Vestavné systémy a mikroprocesory.

V této kapitole budou rozebrány vlastnosti rodiny mikroprocesoru HCS08, popsána specifika mikrokontroleru MC9S08LH64 a také vývojové prostředí CodeWarrior obsahující referenční překladač JSA HCS08/RS08 Assembler. Informace v této kapitole byly mimo jiné převzaty z [18].

3.1 Předchůdci rodiny HCS08

Prvopočátky vývoje rodiny HCS08 lze nalézt již v roce 1979, kdy firma Motorola poprvé uvedla na trh mikroprocesory M6805 a M146805. Obojí byly založené na rodině mikroprocesorů MC6800, což byly jedny z prvních 8bitových mikroprocesorů na trhu, které umožňovaly přímo adresovat až 64 KB paměti, pracovaly na frekvenci 1 MHz a operovaly celkově se 197 instrukcemi. Disponovaly dvěma osmibitovými registry, třemi šestnáctibitovými registry pro uchovávání adres a osmibitovým stavovým registrem. [19]

Pokrok na poli elektrotechnologií umožnil vznik rodiny mikroprocesorů M68HC05, které byly optimalizovány pro digitální regulační úlohy. Oproti MC6800 došlo k odstranění B-registru, čímž bylo uvolněno místo v instrukční sadě pro nové operační znaky, zvláště pak pro operace v nových adresovacích módech. Ty bylo nutné zavést vzhledem k redukci šířky indexového registru X a SP z původních 16 bitů na 8 bitů. Rovněž došlo k odstranění několika instrukcí operujících se zásobníkem a přidány instrukce pro skoky a bitové manipulace. [20]

Navazující rodina mikroprocesorů HC08 byla přizpůsobena pro lepší podporu překladu z jazyka C, a to konkrétně zavedením registrového 8bitového páru H:X a s nimi spojenými adresovacími módy a instrukcemi. Šířka registru SP byla změněna na původní hodnotu 16 bitů.

3.2 Vlastnosti HCS08

Rodina mikroprocesorů HCS08 je osmibitová a navazuje na původní rodinu CISC procesorů HC08, se kterou má kompatibilní instrukční sadu o celkovém počtu 273 instrukcí a používá stejné periferie. Je jednou z rodin mikroprocesorů, z nichž jsou některé používané i v lékařském průmyslu.

Mezi hlavní nové funkcionality patří hardwarová podpora ladění přímo za běhu, která byla představena v rodině šestnáctibitových mikroprocesorů HC12, pomocí vestavěného

ladicího kontroléru (BDC – Background Debug Controller) a modulu (BDM – Background Debug Module). Při propojení s počítačem provádějícím ladění přes programátor připojený na dedikovaný pin mikrokontroleru, lze trasovat instrukce, zachycovat data a stav CPU, vizualizovat data v reálném čase a pozastavit program pomocí softwarových bodů přerušení (angl. breakpoint).

Mezi další vlastnosti rodiny HCS08 patří CPU s pracovní frekvencí až 50 MHz a dvou úrovňovou pipeline, 60 KiB paměti FLASH a až 4 KiB paměti RAM (u některých modelů s jednotkou správy paměti až 4 MiB) a 5 režimů činnosti umožňující snížit spotřebu energie. V závislosti na modelu může mikroprocesor pracovat i při napětích nízkých až 1,8 V. Formát objektového kódu, se kterým pracuje, je kompatibilní s rodinami HC05 a HC08. V porovnání s HC08 operuje HCS08 přibližně třikrát rychleji.

3.3 Programátorský model

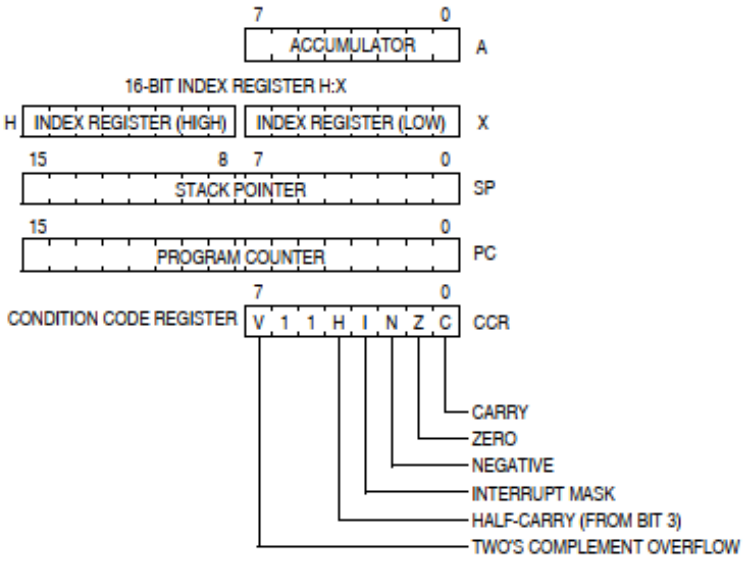
Procesor HCS08 je založen na Von Neumannově architektuře charakteristické použitím adresové, řídicí a datové sběrnice a pamětí společnou pro program a data. Datová sběrnice na HCS08 je osmibitová a je společná pro paměť a periférie.

Periférie jsou paměťově mapované, to znamená, že obsah registrů periferií je namapován na určité místo v paměti, a lze s ním operovat pomocí stejných instrukcí, jakými se přistupuje do zbytku paměti. Jelikož procesor operuje s osmibitovou šířkou dat, hodnoty, které mají větší šířku, jsou v paměti uloženy v pořádí big-endian. To znamená, že hodnoty nejnižších řádů jsou uloženy ve vyšších adresách.

Založení instrukční sady na architektuře CISC znamená, že procesor operuje s mnoha různými operacemi v porovnání s architekturou RISC. Toto umožňuje jednodušší a efektivnější překlad z jazyka C. Na druhou stranu psaní instrukcí v JSA může být náročnější kvůli počtu instrukcí, které programátor může využít. HCS08 zavádí nové adresovací módy pro instrukce `LDHX`, `STHX` a `CPHX` umožňující zmenšit velikost kódu a rychlejší vykonávání.

3.3.1 Registry CPU

Jádro CPU používá celkem 5 registrů. Jejich názornou vizualizaci naleznete na obr. 3.1



Obrázek 3.1 Registry CPU HCS08 [21]

Akumulátor (angl. Accumulator) je 8bitový registr určený zpravidla k načítání operandů z paměti a ukládání výsledků operace ALU.

Programový čítač (angl. Program counter) je šestnáctibitový registr ukazující na paměťové místo následující instrukce, která bude zpracovávána CPU. Jeho délka značí, že umožňuje adresovat až 64 KiB paměťových míst. Při resetu mikrokontroleru se do něj načítá hodnota z paměti uložená na adresách 0xFFFE a 0xFFFF.

Indexový pár H:X je tvořen dvěma 8bitovými registry H a X. Instrukce operující s tímto párem jsou uzpůsobené především pro udržování indexu při procházení polí kdekoliv v paměťovém prostoru. Při provádění některých operací ALU slouží jako pomocný registr pro ukládání hodnot výsledků.

Ukazatel na zásobník (angl. Stack pointer, zkráceně SP) uchovává šestnáctibitovou adresu prvního volného místa před vrcholem zásobníku, což je datová struktura v paměti RAM uchovávající návratové adresy při volání subrutiny nebo uchovává návratovou adresu, obsah programového čítače a registrů A a X při přerušení. Hodnoty jsou uchovávané postupně od nejvyšší adresy k nejnižší a jsou zpětně odebírány z nejnižší k nejvyšší (princip Last-In-First-Out). Hodnota ukazatele se mění automaticky při použití instrukcí typu Push a Pop. Výhodou zásobníku HCS08 oproti některým jiným procesorům RISC je, že umožňují vkládat na zásobník i jiné uživatelské hodnoty, což umožňuje funkcím napsaných v jazyce C předávat hodnoty právě přes zásobník. Kvůli zajištění kompatibility kódu z předchozích procesorů je při resetu mikrokontroleru ukazatel nastaven na adresu 0x00FF, ale může být později nastaven na jinou pomocí instrukce `TXS`. Obsah zásobníku může postupně narůstat až po maximum kapacity RAM.

Proto je v kompetenci programátora nebo vývojového prostředí zabránit programu, aby zásobník narostl do míst s užitečnými daty.

Příznakový registr (angl. Condition Code Register) je osmibitový registr, jehož jednotlivé bity jsou měněny na základě výsledků operací předcházející instrukce a jejich stav při provádění instrukce může rozhodovat o jejím výsledku. Lze takto indikovat přetečení (V), přenos ze třetího do čtvrtého bitu (H), povolení maskovatelných přerušení (I), negativní výsledek (N), výsledek roven nule (Z) a přenos do vyššího bitu (C). Pátý a šestý bit tohoto registru není využit, a proto je jeho logická úroveň vždy 1.

3.3.2 Adresovací módy

Adresovacím módem rozumíme způsob, jakým má daná instrukce zacházet s operandy. V HCS08 existuje celkově 16 adresovacích módů, ale ne všechny jsou dostupné pro všechny instrukce. Při zápisu v JSA různou syntaxí používanou v operandech je specifikován adresovací mód, který má instrukce používat. Ve skutečnosti je však stejná instrukce v závislosti na adresovacím módu překládána na různé operační znaky. Z hlediska hardwaru se tedy jedná o různé instrukce, ovšem z hlediska programátorské abstrakce mají podobnou funkci, a proto používají stejnou mnemotechnickou zkratku. Odlišeny jsou právě adresovacími módy.

V JSA bude následující kód přeložen do strojového kódu 0xA61F, což odpovídá instrukci LDA v **bezprostředním adresovacím módu (angl. Immediate – IMM).** Konkrétně tato instrukce provede načtení hodnoty 31 do registru A.

```
LDA #31
```

Následující zápis bude přeložen do strojového kódu 0xB61F odpovídající instrukci LDA v **přímém adresovacím módu (angl. Direct – DIR)**. Namísto načtení hodnoty v operandu načte tato instrukce do registru A hodnotu paměťového místa 31 (0x001F).

```
LDA 31
```

Nutno poznamenat, že zápis odlišující dané adresovací módy nemusí být u všech instrukcí stejný a naopak, stejný zápis může pro různé instrukce znamenat různé adresovací módy.

Tímto byly představeny bezprostřední a přímý adresovací mód. Následuje stručné vysvětlení dalších adresovacích módů.

Inherentní adresovací mód (angl. Inherent – INH) typicky používá instrukce bez dalších operandů. Chování instrukce je přímo definováno pouze jejím operačním kódem.

Rozšířený adresovací mód (angl. Extended – EXT) na rozdíl od přímého adresovacího módu, který dokáže adresovat jen prvních 256 bajtů paměti (také označovaných jako Direct page), lze načíst hodnotu z kteréhokoliv paměťového místa.

Šířka takové instrukce je v paměti o jeden bajt větší a instrukce je také pomalejší než při použití přímého adresovacího módu.

Relativní adresovací mód (angl. Relative – REL) operuje s adresou danou přičtením hodnoty operandu ve dvojkovém doplňku k aktuální hodnotě programového čítače. Tímto lze tak operovat s adresami v rozsahu -128 až 127 míst od aktuální pozice, přičemž aktuální pozice je adresa určená programovým čítačem po provedení dané instrukce (Čítač již má adresu bezprostředně následující instrukce).

Indexovaný adresovací mód bez offsetu (angl. Indexed, no offset – IX) načítá hodnotu s kterou bude pracovat z paměťového místa určeného aktuální hodnotou registrového páru H:X.

Indexovaný adresovací mód bez offsetu s inkrementací (angl. Indexed, no offset with post-increment – IX+) pracuje podobně jako předchozí mód, ale obsah registrového páru H:X je v poslední kroku vykonávání instrukce inkrementován – zvýšen o 1. Tento mód je využit pouze instrukcí `CBEQ`.

Indexovaný adresovací mód s 8bitovým offsetem (angl. Indexed, with 8-bit offset – IX1) přičte hodnotu operandu k obsahu registrového páru H:X a tím získá výslednou adresu, s jejíž hodnotou bude dále operovat. Obsah H:X zůstává nezměněn. Tento mód je užitečný při procházení pole, kdy 8bitový offset je adresa začátku pole a obsah registrového páru H:X obsahuje číslo prvku, který je z pole čten.

Indexovaný adresovací mód s 8bitovým offsetem a inkrementací (angl. Indexed, with 8-bit offset and post-increment – IX1+) pracuje stejně jako předchozí mód, ale posledním krokem instrukce je inkrementování registrového páru H:X. Tento adresovací mód používá pouze instrukce `CBEQ`.

Indexovaný adresovací mód s 16bitovým offsetem (angl. Indexed, with 16-bit offset – IX2) pracuje stejně jako mód IX1, ovšem umožňuje používat 16bitovou hodnotu operandu. Lze tak procházet polem s bází umístěnou mimo direct page.

Adresovací mód relativní k zásobníku s 8bitovým offsetem (angl. SP-relative, with 8-bit offset – SP1) pracuje obdobně jako mód IX1, ovšem namísto indexového páru H:X je použita hodnota ukazatele na zásobník. Obsah ukazatele na zásobník není změněn. Toto umožňuje adresovat hodnoty až 255 míst pod vrcholem zásobníku, čehož také využívají překladače jazyka C.

Adresovací mód relativní k zásobníku s 16bitovým offsetem (angl. SP-relative, with 16-bit offset – SP2) pracuje obdobně jako předchozí mód, ale umožňuje takto adresovat hodnoty hlouběji než 255 míst pod zásobníkem.

Následující adresové módy jsou používané pouze instrukcí `MOV`, která kopíruje data přímo do paměťových míst nebo mezi dvěma paměťovými místy.

Mód z bezprostřední hodnoty do přímé adresy (angl. Immediate to direct – IMM/DIR) inicializuje hodnotu adresy v direct page hodnotou v prvním operandu. Tato operace je rychlejší a úspornější než kopírování hodnoty přes akumulátor A pomocí instrukcí `LDA` a `STA`.

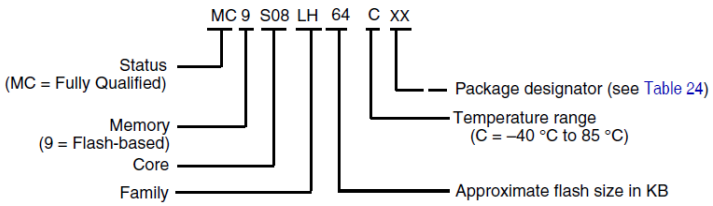
Mód z přímé adresy do přímé adresy (angl. Direct to direct – DIR/DIR) zkopíruje data z jedné adresy direct page do druhé.

Mód z indexované adresy do přímé s inkrementací (angl. Indexed to direct with post-increment – IX+/DIR) je používaný k přenosu dat pole. Zdrojová adresa je určená indexovým párem H:X a cílová je určena operandem ukazujícím do direct page. Po každém přesunu je obsah H:X inkrementován.

Mód z přímé do indexované adresy s inkrementací (angl. Direct to indexed with post-increment – DIR/IX+) operuje stejně jako předchozí příklad, ale data jsou přenášena opačným směrem.

3.4 Vlastnosti MC9S08LH64

Pro určení vlastností konkrétního modelu mikrokontroleru zavedla firma Freescale značení představené na obrázku 3.2



Obrázek 3.2 Systém značení mikrokontrolerů Freescale [21]

Označením MC9S08LH64 tedy rozumíme mikrokontroler s pamětí FLASH, jádrem S08, řadičem pro ovládání LCD panelu a 64 KiB celkové paměti. [21]

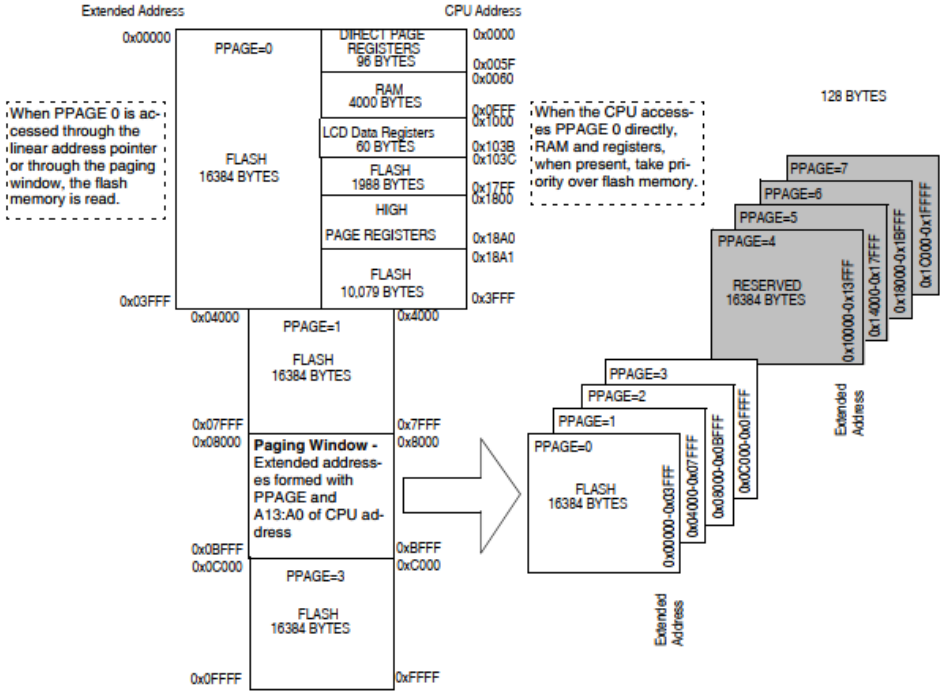
3.4.1 Rozložení paměti

Při vytváření definiční části pro tento mikrokontroler by ze specifik mohlo být užitečné zmínit jeho rozložení paměti, které vizualizuje obrázek 3.3.

V rozsahu adres od 0x0000 do 0x005F jsou namapované registry, u nichž se předpokládá časté využívání, a jsou proto obslužitelné instrukcemi s přímým adresováním. Následujících 4000 bajtů do adresy 0x0FFF je volná paměť RAM. Na konci RAM od adres 0x1000 do adresy 0x103B se nacházejí datové registry ovládající LCD panel. Následující adresový prostor již operuje s pamětí FLASH, ve které se v oblastí od 0x1800 až 0x18A0 nacházejí High-page registry, s jejichž hodnotou se neoperuje tak často jako s direct-page registry, čímž se šetří místo pro častěji používané přímo adresovatelné proměnné v RAM.

Poslední rezervovanou oblastí jsou nevolatilní registry na adresách 0xFFB0 až 0xFFBF obsahující počáteční hodnoty některých registrů, do kterých jsou tyto hodnoty po resetu mikrokontroleru kopírovány. Také se v této oblasti nachází 8bajtový klíč umožňující uživateli kontrolovaný přístup k zabezpečené paměti. Místo pro tabulku

vektorů přerušení se nachází v oblasti 0xFFD2 až 0xFFFF, pokud vektory daný program využívá. [21]



Obrázek 3.3 Rozložení paměti pro MC9S08LH64 [21]

3.5 Vývojové prostředí CodeWarrior

Jako referenční překladač pro daný procesor bude v této semestrální práci použit překladač JSA HC(S)08/RS08 z vývojového prostředí CodeWarrior, které je vydávané firmou NXP.

První verze prostředí CodeWarrior byla vydána již v roce 1993. Toto vývojové prostředí podporuje programovací jazyky C, C++, Java a JSA pro většinu procesorů. Mezi ně patří běžné rodiny desktopových a embedded procesorů, například x86, PowerPC a MIPS. Kód zapsaný v JSA je možno propojit s kódem napsaným ve vyšších programovacích jazycích, a to i přímo ve zdrojovém kódu (angl. inline assembler).

Mezi nástroje, které CodeWarrior poskytuje, je správce projektu, editor, kompilátor, linker, debugger a profiler. Další funkcionality lze rozšířit pomocí pluginů. Ze zmíněných nástrojů při testování referenčního kódu budou pro nás důležité schopnosti debuggeru.

Program lze ladit jak v simulátoru, tak přímo na mikrokontroleru připojeném přes rozhraní PEmicro‘s USB Multilink k počítači. Při ladění lze krokovat, sledovat hodnoty proměnných, zobrazovat a měnit obsah cache, registrů a paměti včetně programového čítače, importovat nebo exportovat paměťová místa v různých formátech a prohlížet instrukce v paměti pomocí disassembleru, který je převede na čitelnější zápis v JSA. Strojový kód vygenerovaný ve `fasmg` lze tak nahrát do paměti při spuštěném

ladění a rovnou analyzovat, zda instrukce odpovídají očekávanému výstupu. Detailní popis prostředí CodeWarrior lze nalézt v [23].

3.5.1 Integrovaný HC(S)08/RS08 Assembler

Při programování pro rodinu HCS08 lze použít zdrojový kód v jazyce C, případně i v kombinaci s relokovatelným jazykem assembler, nebo lze použít absolutní jazyk assembler. V případě volby jazyka assembler používá CodeWarrior vlastní integrovaný překladač HC(S)08/RS08 Assembler a linkování provádí SmartLinker.

Ve zmíněném jazyce assembler lze programovat pro rodiny mikroprocesorů uvedené v jeho názvu. Překladač podporuje makra. Výstupní formát souborů může být ve formátu objektového souboru s příponou `.obj` nebo absolutního souboru s příponou `.abs`. V případě generování absolutního ELF souboru je současně generován soubor ve formátu S-Record (zkráceně S-rec), který může být přímo nahrán do FLASH paměti mikrokontroleru. Je možné generovat i soubor výpisu, který ukazuje, z kterých prvků zdrojového kódu vznikly konkrétní instrukce, což umožňuje programátorovi lépe interpretovat výstupní soubor.

Uživatelský manuál k integrovanému překladači HC(S)08/RS08 si lze přečíst v [24].

4. IMPLEMENTACE DEFINIČNÍ ČÁSTI PŘEKLADAČE

V této kapitole je demonstrován způsob implementace definiční části překladače pro nový procesor na úryvcích jejího zdrojového kódu. Definice kompletní instrukční sady se nachází v příloze A v souboru `hcs08.inc`. Definice pomocí CALM instrukcí, ve které bylo z časových důvodů implementováno 40 instrukcí z instrukční sady se nachází v příloze A v souboru `hcs08_calm.inc`.

Při návrhu definiční části překladače je nutné si nejdříve určit, jakou syntaxi budou dané mnemotechnické zkratky mít. Byla zvolena stejná syntaxe, kterou používá překladač HCS08/RS08 Assembler integrovaný v prostředí CodeWarrior, jelikož takto bude možné programy původně pro něj napsané snadněji přepsat pro překlad ve `fasmg`. Popis syntaxe jednotlivých instrukcí lze nalézt v [24]. Zde se také nachází mapa operačních znaků, která usnadňuje přehled v systému číslování operačních znaků.

4.1 Příklad definice makra adresovacích módů

Adresovací mód instrukce, pro které existuje více adresovacích módů, se volí podle syntaxe v operandech. Toto již bylo rozebráno v kapitole 3.3.2. Pokud očekáváme v operandech jakýkoliv číselný výraz, je nutné provést kontroly, zda dané operandy jsou v očekávaném rozsahu. Jelikož tyto kontroly bývají pro většinu operandů stejné, nabízí se provést je pomocí maker.

V následujícím příkladu je definice makra pro bezprostřední adresovací mód (IMM):

```
macro HCS.imm avalue,areg,aopcode
      rmatch ,areg
            if    0d <= avalue & avalue <= 255d
                  db aopcode,avalue
            else if -128d <= avalue & avalue <= 127d
                  local signed
                   signed = 0FFh + avalue + 1
                   db aopcode,signed
            else
                  err "Constant value not in range"
            end if
      else
            err "Invalid addressing mode - IMM has only one parameter"
      end rmatch
end macro
```

Toto makro má jako vstupní parametry `avalue` nesoucí hodnotu operandu instrukce, `areg` nesoucí hodnotu druhého parametru instrukce a `aopcode` nesoucí hodnotu operačního znaku pro danou instrukci.

Pomocí příkazu `rmatch ,areg` je kontrolováno, zda druhý operand instrukce je skutečně prázdný. V případě, kdy tomu tak není, je generována chyba překladu. Následně je hodnota `avalue` zkontrolována, zda leží v rozsahu 0x00 až 0xFF. V případě, že je

předávaná hodnota záporná, zkontroluje se, zda leží v rozsahu hodnot, které mají šířku jednoho bajtu, a následně se provede její přepočet do druhého doplňku. Nakonec jsou do výstupního souboru generovány hodnoty operačního znaku a zpracované bezprostřední hodnoty.

V dalším příkladu je definice makra pro adresovací mód relativní k zásobníku s 8bitovým offsetem (SP1).

```
macro HCS.sp1 avalue,aopcode
if    0d <= avalue & avalue <= 255d ;SP1
            dw (aopcode) bswap 2
            db avalue
      else
            err "Address value not in range"
      end if
end macro
```

Na tomto příkladu lze demonstrovat jednu z vlastností `fasmg`, kterou je nutné brát na zřetel. V případě, kdy operační znak má délku větší než 1 bajt, bude v paměti mikrokontroleru HCS08 umístěn v pořadí big-endian (nejvýznamnější bajt je v paměti uložen na nižší adrese). Flat Assembler je ovšem navržen pro platformu x86, která používá řazení little-endian a tato vlastnost byla přenesena i do `fasmg`. Při použití direktivy `dw` je tedy nutno vždy použít operátor `bswap`, který umožňuje jeho levostrannou hodnotu po bajtech posunout o počet určený jeho pravostrannou hodnotou. Jedině takto mohou být bajty ve výstupním souboru při použití `dw` ve správném pořadí.

4.2 Příklad definice instrukce pomocí maker

Jednotlivé definice instrukcí jsou v souboru `hcs08.inc` přibližně v pořadí, v jakém se vyskytují v mapě operačních znaků. Jelikož jednotlivé instrukcí používají většinou stejnou syntaxi, lze jejich makra jednoduše nadefinovat pomocí iterací nebo výčtů.

V následujícím příkladu je definice velké části instrukcí ze skupiny pro instrukce provádějící v jednom cyklu procesoru načtení, modifikaci a zpětný zápis (Read-Modify-Write).

```
irp <mnem,opcode>, NEG, 030h, COM, 033h, LSR, 034h, ROR, 036h, ASR, \
037h, LSL, 038h, ROL, 039h, DEC, 03Ah, DBNZ, 03Bh, INC, 03Ch, TST, \
03Dh, CLR, 03Fh
macro mnem? operand,reg
      local value
      rmatch ,operand ;IX
            ix reg,opcode+040h
      else  rmatch value,operand
                  rmatch ,reg
                        HCS.dir value, opcode
                  else rmatch =X?,reg
                        HCS.ix1 value, opcode+030h
                  else rmatch =SP?,reg
```

```
                        HCS.sp1 value, opcode+09E30h
                  else
                        err "Invalid addressing mode - Unknown operand"
                  end rmatch
      end rmatch
end macro

macro mnem#A?
      db opcode + 010h
end macro

macro mnem#X?
      db opcode + 020h
end macro
end irp
```

Při překladu tohoto kódu je při každé iteraci definováno makro instrukce spolu s odpovídající hodnotou operačního znaku.

Pokud programátor některou z těchto instrukcí použije, zavolá se příslušné makro, kde se zkontrolují jeho použité parametry, a následně se zavolá makro příslušného adresovacího módu, jehož příklad je uveden v předchozí kapitole. Jelikož rozmístění v mapě operačních znaků je pro všechny tyto instrukce stejné vzhledem k instrukci s přímým adresovacím módem (DIR), lze hodnotu výsledného operačního znaku získat přičtením posunu vzhledem k této instrukci (V kódu je toto zapsané například jako `opcode+010h`).

Dále jsou při každé iteraci definovány makra instrukcí pracující s akumulátorem A nebo registrem X (například `NEGA` a `NEGX`). Samotné tělo těchto maker je jednoduché, jelikož se jedná o instrukce s inherentním adresovacím módem, který je v paměti reprezentován pouhým operačním znakem a není tudíž nutné pro tento adresovací mód psát makra.

Makra pro adresovací módy nejsou rovněž psaná u instrukcí, které používají v porovnání s ostatními instrukcemi unikátní zápis operandů nebo jejich rozložení v mapě je neobvyklé. Typicky se jedná o instrukce `STHX`, `LDHX`, `CPHX` nebo například instrukce MOV.

Při testování popsaném v kapitole 6.1 jsem objevil instrukce, které nejsou zapsané v tabulce operačních znaků. Jedná se konkrétně o instrukce `ASL`, `BHS` a `BLO`. Jsou to pouhé duplicitní mnemotechnické zkratky, které označují instrukce `LSL`, `BCC` a `BCS` provádějící stejnou činnost. Stačí je nadefinovat jako další makra, jejichž obsah je totožný.

V manuálu k HCS08 [24] existuje také nesrovnalost se syntaxí instrukce `CBEQ` v adresovacím módu IX+, která má podle něj mít tvar:

```
CBEQ ,X+,rel      ;rel je výraz udavající posun vůči současné adrese.
```

Překladač ve vývojovém prostředí CodeWarrior by však tento zápis přeložil jako instrukci CBEQ v adresovacím módu IX+1. Správný zápis této instrukce je tedy:

```
CBEQ    X+,rel
```

Jelikož má syntax odpovídat té, s kterou operuje překladač HC(S)08/RS08 Assembler, zvolil jsem druhou variantu.

4.3 Příklad definice pomocí CALM instrukcí

V této podkapitole jsou rozebrány další příklady příkazů používaných uvnitř bloků CALM instrukcí, které jsem využil při tvoření definiční části překladače.

Následující CALM instrukce používá stejné rozhraní a plní stejnou funkci jako makroinstrukce pro bezprostřední adresovací mód (IMM) popsaná v kapitole 4.1.

```
calminstruction HCS.imm avalue,areg,aopcode*
      match ,areg
      jno ParsErr
      check 0d <= avalue & avalue <= 255d
      jno negat
      arrange     cmd,=db aopcode,avalue
      jump ending
negat: check -128d <= avalue & avalue <= 127d
      jno rangeErr
      compute signed,0FFh + avalue + 1
      arrange cmd,=db aopcode,signed
      jump ending
rangeErr:   arrange cmd,=err "Constant value not in range"
      jump ending
ParsErr:    arrange cmd,=err "Invalid addressing mode - HCS.imm has
only one parameter"
ending: assemble cmd
end calminstruction
```

Jako první se v příkladu vyskytuje příkaz `match`, který má stejné použití jako mimo blok CALM instrukce. Princip, na základě kterého porovnává, byl již popsán v kapitole 2.2.1. Pakliže je porovnávání sekvencí úspěšné, nastavuje se v překladači příznak o této skutečnosti. Stejný příznak se nastavuje i po vyhodnocení výsledku příkazu `check`, který má v bloku CALM instrukce stejnou funkci jako instrukce `if`.

Podmíněné skoky v programu se provádí pomocí příkazů `jyes` a `jno` následovaných názvem návěští ukazujícím na místo v kódu, kam má být skok proveden. Podmínka je určena na základě hodnoty výsledku předcházející instrukce `match` nebo `check`. Například pokud byl těmito příkazy nastaven příznak do logické 1, instrukce `jyes` provede podmíněný skok. Nepodmíněný skok se provádí pomocí instrukce `jump`.

V dalším příkladu je příklad definice instrukcí `AIS` a `AIX` pomocí CALM instrukcí.

```
irp <mnem,opcode>, AIS?, 0A7h, AIX?, 0AFh
calminstruction mnem operand,reg
      match #value,operand ;HCS.imm
      jno ParsErr
      compute aopcode,opcode
      call HCS.imm,value,reg,aopcode
      exit
ParsErr:    arrange cmd,=err "Invalid addressing mode"
assemble cmd
end calminstruction
end irp
```

Na tomto příkladu lze demonstrovat volání CALM instrukce z bloku CALM instrukce. Provádí se pomocí příkazu `call`, kde prvním argumentem je název volané CALM instrukce a v následujících jsou argumenty, se kterými je volána. Tyto argumenty musí být identifikátory proměnných. Argumentem tedy nemůže být například číselný výraz.

Identifikátoru se přiřazuje hodnota pomocí příkazu `compute`, kde jeho prvním argumentem je název identifikátoru a druhým je výraz jehož hodnota má být identifikátoru přiřazena.

Pomocí příkazu `exit` lze ukončit provádění bloku CALM instrukce před dosažením jeho konce.

5. FORMÁT VÝSTUPNÍHO SOUBORU

Jelikož samotný binární výstup není pro další práci použitelný, je nutné zvolit konkrétní souborový formát, který umožní přeložený program zapouzdřit tak, aby s ním mohlo být dále operováno dalšími nástroji.

V této kapitole jsou rozebrány uvažované formáty výstupních souborů. Dále je zde popsán způsob, jak lze implementovat zpracovávání překladu do zvoleného formátu ve `fasmg`. Část, která implementuje tuto funkci, nazývám ve zbytku své práce generátorem.

5.1 Souborový formát ELF

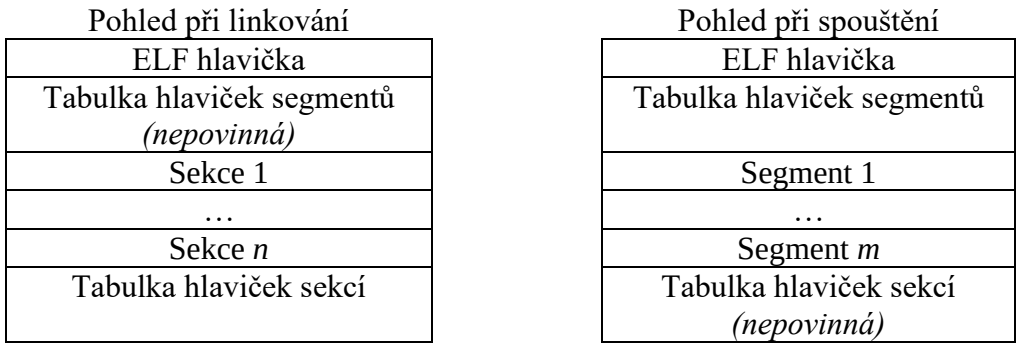
Executable and Linking Format (zkráceně ELF) je souborový formát pro ukládání programů v binární reprezentaci pro přímé vykonávání procesorem. Byl vyvinut v laboratořích UNIX System Laboratories jako součást aplikačního binárního rozhraní pro UNIX System V. I když byl vyvíjen pro architekturu i386, je navržen jako přenositelný na různé architektury i operační systémy.

Existují 3 hlavní typy objektových souborů, se kterými tento formát pracuje: relokovatelný, spustitelný a sdílený objektový soubor (neboli sdílená knihovna).

Relokovatelný soubor obsahuje kód a data v jednotlivých sekcích. Může vzniknout překladem nebo linkováním z více původních objektových souborů. Je určen k pozdějšímu statickému linkování do nového objektového souboru a vytvoření tak nového relokovatelného, sdíleného nebo spustitelného objektového souboru.

Spustitelný soubor obsahuje program s daty v jednotlivých segmentech, které jsou na rozdíl od sekcí při spuštění souboru nahrány na definovaná místa v paměti. Pokud se jedná o spustitelný soubor s dynamický linkováním, obsahuje spustitelný soubor informace o sdíleném objektovém souboru, jehož segmenty mají být při spuštění spolu s programem nahrány.

Obecně ELF soubor může obsahovat jak segmenty, tak sekce. Jelikož obojí jsou definovány pouze svou adresou v ELF souboru a velikostí, může segment zahrnovat více sekcí. Způsob, jakým se bude na daná data v ELF souboru pohlížet, je tedy dán skutečností, zda se bude soubor spouštět nebo linkovat. Tento pohled je znázorněný na obrázku č. 5.1.



Obrázek 5.1 Pohledy na strukturu ELF souboru – převzato z [26]

Řídící data vlastní ELF souboru (data v hlavičkách) jsou na rozdíl od dat v jednotlivých segmentech nebo sekcích nezávislá na architektuře. Používají znamínkové i neznamínkové celočíselné datové typy o délce 1 až 8 bajtů (v případě 32bitových architektur pouze 1 až 4). Z důvodů kompatibility nejsou použita bitová pole.

Hlavička ELF souboru obsahuje magické číslo sloužící k identifikaci ELF souboru a hodnoty identifikující mimo jiné typ ELF souboru, cílovou architekturu, vstupní bod programu, umístění tabulek hlaviček sekcí, segmentů a tabulky řetězců, jejich velikost a počet.

Tabulka hlaviček segmentů nebo sekcí je indexované pole jednotlivých hlaviček. Každá sekce má právě jednu hlavičku a zaujímá jednu souvislou nebo i žádnou sekvenci bajtů v ELF souboru, které se nesmí překrývat. V hlavičce sekcí lze nalézt jméno sekce (odkaz do tabulky řetězců), typ sekce, umístění v ELF souboru, umístění v paměti programu nebo odkaz na další sekce. Sekce neobsahují pouze data samotného programu, ale i data, která využívá zavaděč programu. Typicky se jedná například o tabulku symbolů, tabulku řetězců, tabulku relokací a poznámky.

V hlavičce segmentů lze mimo jiné nalézt údaje o typu segmentu (Samotný segment může ukazovat na spustitelné instrukce nebo jiný typ dat.), pozici segmentu v ELF souboru, pozici, na kterou má být nahrán v paměti, přístupová práva segmentu v paměti (čtení, zápis, vykonávání) a způsob, jakým má být v paměti zarovnán.

Podrobnější informace lze nalézt v původní dokumentaci formátu ELF [26].

Při hledání informací, se kterými parametry budovat ELF soubor pro architekturu HCS08 jsem objevil pouze specifikaci M8/M16 EABI (Embedded Application Binary Interface) [27], která specifikuje parametry pro mikrokontrolery M68HC05, M68HC08, M68HC11, M68HC12 a M68HC16. Pro tyto mikrokontrolery se nemůže v ELF souboru vyskytovat dynamické linkování, sdílené knihovny a relokace.

Formát ELF je využíván při programování v prostředí CodeWarrior, kdy je vytvořený program do tohoto formátu přeložen a ve výchozím nastavení se ukládá v adresáři projektu do složky `FLASH` s příponou `.abs` nebo `.obj`. V současnosti existují hlavičkové soubory pro `fasmg`, které umí vytvářet soubory ve formátu ELF. Jsou dostupné při stažení `fasmg` v adresáři `.\examples\x86`. Komplexnost těchto souborů ale i problematika formátu ELF je natolik složitá, že v této práci nebude dále rozvedena.

5.2 Motorola S-record

Obsah této kapitoly je převzat z přílohy dokumentace k procesoru Motorola 68000. [28]

Formát S-record (nebo také Motorola S-record) byl vytvořen v 70. letech 20. století k ukládání programů a dat v lidsky čitelném formátu, aby je šlo snadno editovat v textovém editoru. [29] Dociluje toho tak, že binární data kóduje jako ASCII znaky, respektive každý nibble (polovina bajtu) je uložen jako čitelná hexadecimální číslice. Dvojice těchto číslic pak reprezentuje původní hodnotu bajtu. Formát slouží pro přenos dat mezi počítačem a programátorem paměti mikrokontroleru. Při překladu programu ve vývojovém prostředí CodeWarrior pro mikroprocesor HCS08 vzniká v adresáři projektu ve složce FLASH soubor s příponou `.sx`, který obsahuje program přeložený do tohoto formátu.

V textové podobě obsahuje soubor S-record jednotlivé řádky zvané záznamy. Struktura jednoho řádku je znázorněna na obrázku č. 5.2. Každý záznam je započat znakem S a číslicí identifikující typ záznamu (v rozsahu 0-9). Následující dva znaky obsahují počet bajtů uložených v záznamu mimo znaku S, typu a délky záznamu. Část s adresou je v závislosti na typu záznamu dlouhá 2, 3, nebo 4 bajty a většinou reprezentuje adresu v paměti, na kterou mají být data záznamu uložena. Samotná část dat může být dlouhá 0-2*n bajtů. V závislosti na aplikaci, která bude S-record používat může být délka této části z důvodu kompatibility omezena tak, aby se záznam vešel na obrazovku. Kvůli omezení většiny současných textových editorů doporučuji, aby délka záznamu nepřekročila 1024 znaků.

Poslední část v záznamu o délce 1 bajtu je kontrolní součet, který je obdržen jako součet hodnot jednotlivých bajtů v záznamu s výjimkou znaku S a typu. Tento součet je pak ořezán na nejméně významný bajt a z této hodnoty je pak spočítán první doplněk. Tento výpočet lze také zrealizovat jako aritmetický součet všech hodnot, následovaný logickým součinem s maskou 0FFh, logickou negací a přičtením 1. Kontrolní součet a délka záznamu slouží aplikacím operujícím s tímto formátem k ověření, že daný záznam nebyl při přenosu měněn.

Začátek záznamu	Typ	Počet	Adresa	Data	Kontrolní součet
Znak S	1 nibble (0-9)	1 bajt	2,3 nebo 4 bajty	0-2*n bajtů	1 bajt

Obrázek 5.2 Struktura záznamu S-record – překresleno z [28]

Celkem existuje 10 typů záznamů. V případě architektury HCS08 budou pro nás důležité záznamy typu S0, S1 a S9, ze kterých je složena varianta formátu S-record kompatibilní s touto architekturou zvaná S19. Název je odvozen právě od čísel typů záznamů, které tato varianta používá.

Typ S0 slouží jako hlavička pro následující záznamy v souboru. Část s adresou je typicky vyplněna nulami. Část s daty obsahuje jakákoliv identifikující data, která nejsou

určena k nahrávání. Soubory ve formátu S-record generované prostředím CodeWarrior zde typicky obsahují název konfiguračního souboru `.prm`, ze kterého byly vygenerovány.

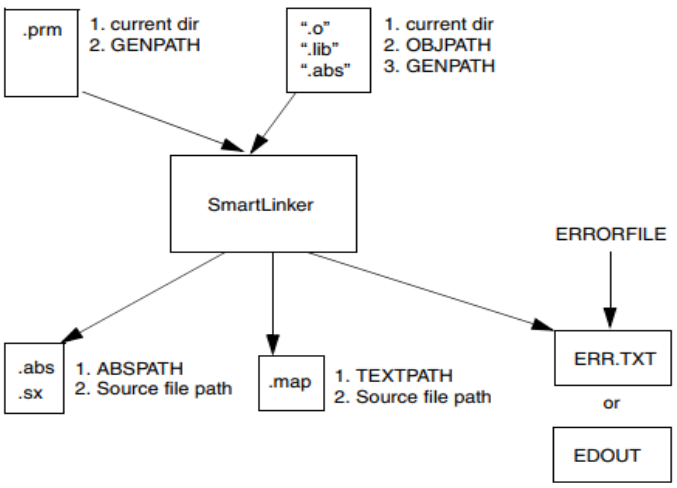
Typ S1 slouží k samotnému uložení dat programu. Délka části s adresou má 2 bajty. Tím se odlišuje od záznamů typu S2 a S3, které používají rozdílné délky této části.

Typ S9 slouží k ukončení několika po sobě jdoucích bloků typu S1. Část s adresou může obsahovat adresu instrukce, ze které má být program spouštěn. Může však také být vyplněna nulami, pokud s touto informací aplikace nepracuje. Při použití překladače HC(S)08/RS08 Assembler při nastavení, kdy je přímo generován absolutní objektový soubor, je v současně vytvořeném souboru S-record skutečně adresa, od které je program spouštěn. Část s daty se v tomto typu záznamu nevyskytuje.

Jak již bylo zmíněno, překladač HC(S)08/RS08 Assembler umí generovat soubory ve formátu S-record. Lze je tedy používat jako referenční pro otestování správnosti generování výstupu překladu z `fasmg`. Dále jsem tento formát zvolil kvůli své jednoduchosti a kvůli lepší věrohodnosti testování rychlosti překladu popsaného v kapitole 8. Implementovaný generátor formátu S-record je v příloze A pod názvem `srec.inc`.

5.3 Formát propojitelný s CodeWarrior

Jelikož samotný soubor ve formátu S-record je ve vývojovém prostředí CodeWarrior vytvořen programem SmartLinker až po dokončení linkování [30] a neumožňuje již další spojení s jiným kódem, bylo potřeba vymyslet alternativní vhodný výstup překladu z `fasmg` tak, aby se dal opět propojit s programem ve vývojovém prostředí CodeWarrior. Zde by byl program přeložen a mohl by být nahrán na skutečný mikrokontroler. Jedna z možností je program přenést na textovou sekvenci pseudoinstrukcí ORG a DC.B



Obrázek 5.3 Diagram procesu linkování [30]

s hodnotami z binárního výstupu oddělenými čárkami, které budou uložené v paměti. Jelikož samotný program nebude za běhu měněn, může být takto zapsán do paměti. Příklad, jak lze generátor takového výstupu napsat ve `fasmg` je v následující podkapitole.

5.3.1 Implementace generátoru výstupu z binárních dat

Pro plnohodnotný popis je zde potřeba uvést celý zdrojový kód uložený v příloze A v souboru `cw.inc`. Tento hlavičkový soubor je vždy potřeba zahrnout pomocí instrukce `include` v souboru před samotným zdrojovým kódem programu.

```
define CW
format binary as 'txt'

virtual at 0
      CW.digits:: db '0123456789ABCDEF'
end virtual

macro CW.byte value
      local digit
      load digit:byte from CW.digits:(value) shr 4
      db digit
      load digit:byte from CW.digits:(value) and 0Fh
      db digit
end macro

macro CW.seg address:0
      virtual at address
end macro

macro CW.endseg
      local code,address,size,bajt
      code:: address = $$
                  size = $-$$
      end virtual
      if address = 0 & size <> 0
            db 9,'DC.B '
      end if
      while size > 246d
            repeat 246d
                  db '$'
                  load bajt:byte from code:address
                  CW.byte bajt
                  if 246 <> %
                        db ','
                  else
                        db 13,10
                  end if
                  address=address+1
            end repeat
            db 9,'DC.B '
            size = size - 246d
      end while
      repeat size
            db '$'
            load bajt:byte from code:address
            CW.byte bajt
```

```
            if size <> %
                  db ','
            else
                  db 13,10
            end if
            address=address+1
      end repeat
end macro

macro ORG? address
      if $ <> address
            CW.endseg
            local low,high
            low = address and 0FFh
            high = address shr 8
            db 9,'ORG $'
            CW.byte high
            CW.byte low
            db 13,10,9,'DC.B '
            CW.seg address
      end if
end macro

CW.seg

postpone
            CW.endseg
end postpone
```

Pro úplnost zde rozeberu příkazy, které jsou v této ukázce použity. Informace o těchto příkazech byly opět převzaty z manuálu k `fasmg` [16].

Příkaz `format` je určen k volání makroinstrukce vyskytující se v jeho jmenném prostoru, která má měnit vlastnosti výstupního souboru. Jediná vestavěná makroinstrukce má název `binary` a má být následována klíčovým slovem `as` a řetězcem, který obsahuje název přípony výstupního souboru, pokud není specifikována z příkazové řádky.

Příkaz `virtual` vytváří novou virtuální paměťovou oblast, do které jsou zapisovány data pomocí příkazů `db`, `dw` a dalších. Jejím argumentem je klíčové slovo `at` a číslo adresy, které má být použito jako báze této paměťové oblasti. Pakliže tento argument není použit, je bázová adresa rovna hodnotě čítači umístění, při které je příkaz `virtual` použit. Virtuální oblast se uzavírá příkazem `end virtual`. Hodnota čítače umístění mimo virtuální oblast zůstává stejná jako před příkazem `virtual`.

Příkaz `load` slouží k definici proměnné, které je přiřazena hodnota z paměťové oblasti již vygenerovaných dat. Jejími argumenty jsou identifikátor proměnné následovaný znakem dvojtečka a číslem určující délku načítaných dat. Následuje klíčové slovo `from` následované adresou již vygenerovaného paměťového místa.

Existuje další varianta příkazu `load`, která se vyskytuje v ukázce, kde místo adresy se nachází název speciálního návěští pro tento příkaz následovaný dvojtečkou a číslem určující posun od báze paměťové oblasti, ve které je návěští defniované. Tato varianta umí vyčítat data i z právě nepoužívaných virtuálních oblastí. Návěští pro tento příkaz se

od obyčejného odlišuje použitím dvou dvojteček místo jedné a samo o sobě neukazuje na konkrétní místo v paměťové oblasti. Slouží jen jako odkaz, pomocí kterého lze k virtuální paměťové oblasti přistupovat.

Příkaz `postpone` umožňuje označit blok příkazů, jehož obsah má být proveden až poté, co byly provedeny všechny ostatní příkazy následující po příkazu `end postpone`. Toto umožňuje implementovat zpracovávání vytvořených dat v jednom hlavičkovém souboru. Příkazy, které se mají provést až na konci zpracovávání zdrojového souboru lze takto vložit do bloku příkazů `postpone` a není je nutné ručně vkládat na konec například pomocí příkazu `include`. Pokud mezi příkazy následujícími `end postpone` existuje další příkaz `postpone`, jsou jednotlivé bloky provedeny v opačném pořadí jejich uvedení. Tedy blok `postpone`, který je v kódu uveden jako poslední, se provede jako první.

`Fasmg` umožňuje redefinování klíčových slov. V této ukázce redefinujeme příkaz `org` vlastním makrem, které takto nahradí funkcionalitu původního vestavěného příkazu. Dřívější definice obecně jakéhokoliv symbolu může být provedena instrukcí `restore` následovanou daným symbolem.

Při implementaci generátoru jsem vycházel z návrhu generátoru objektového formátu Intel HEX, který je při stažení `fasmg` dostupný v umístění `.\examples\8051\hex.inc`. Generátor začíná příkazem `format`, určujícím příponu generovaného souboru. Dále jsou definovány jednotlivé makroinstrukce sloužící ke zpracovávání samotného souboru. Nakonec je volána makroinstrukce `CW.seg`, která způsobí otevření virtuálního adresového prostoru. To způsobí, že celý zdrojový kód, který bude tento hlavičkový soubor obsahovat se bude překládat do virtuální paměťové oblasti Jakmile je v kódu dosaženo makroinstrukce `org`, dojde k uzavření virtuální paměťové oblasti, data v ní vygenerovaná se zpracují a upravená se zapíšou do skutečného výstupního souboru. Následně je otevřena nová virtuální oblast na bázi určené parametrem volané makroinstrukce `org` a generovaný výstup ze zdrojového kódu je opět zapisován do ní. Jakmile je dosaženo konce zdrojového souboru, vyvolá se makroinstrukce `CW.endseg` z bloku příkazu `postpone`, která zbývající data opět zpracuje a uloží.

Při úpravě do formátovaného výstupu bylo potřeba zohlednit limitaci překladače HC(S)/RS08 Assembler, který dokáže přeložit řádek o maximální délce 1024 znaků. Jelikož odsazení a direktiva DC.B zabere 10 znaků a hodnota každého bajtu spolu s dělící čárkou je vyjádřena 4 znaky, je potřeba sekvence s více jak 246 bajty rozdělit na více řádků.

6. TESTOVÁNÍ SPRÁVNOSTI GENEROVANÉHO KÓDU

Pro úplné testování správnosti generování binárního výstupu je třeba vytvořit syntetický test reprezentovaný zdrojovým kódem obsahujícím všechny instrukce používané procesorem HCS08 včetně všech adresovacích módů. Pro toto testování postačuje, že zdrojový kód nebude reprezentovat konkrétní fungující program. Pro ověření funkčnosti formátu výstupního souboru zvoleného v kapitole 5 je vhodné testování i na fungujícím programu. Tato kapitola popisuje metodu, jakou probíhalo testování správnosti generování jak binárního, tak formátovaného výstupu.

6.1 Testování binárního výstupu

Snahou této práce je vytvořit překladač jazyka assembler, který bude operovat se stejnou instrukční sadou jako již existující nástroje. Proto je zbytečné vytvářet testovací zdrojové soubory znovu. V souborech dodávaných k překladači Macroassembler AS, popsaném v kapitole 1.2.5, existuji testovací soubory pro všechny instrukční sady, které tento překladač podporuje. Mezi nimi lze nalézt i testovací soubor k mikroprocesoru HCS08.

Jednotlivé testovací soubory jsou ve složce `.\tests` rozmístěny ve složkách podle názvu architektury. Tyto složky obsahují soubor se zdrojovým kódem, doprovodný komentář v souboru `.doc` a soubor s příponou `.ori` obsahující obraz výsledného překladu v binární podobě, který je porovnáván s překladem pomocí Macroassembler AS. Po nahlédnutí do souboru `.doc` ve složce `t_hcs08` lze zjistit, že v testovacím souboru jsou zahrnuty pouze rozšiřující instrukce k instrukční sadě mikroprocesoru 68HC08. Proto je třeba nahlédnout do složky `t_6808`, kde jsou testovány instrukce rozšiřující instrukční sadu mikroprocesoru 68HC05. Toto dává smysl vzhledem k tomu, že instrukční sada HCS08 je zpětně kompatibilní s 68HC05 a bylo by proto zbytečné vícekrát testovat překlad stejných instrukcí. Pro účely testování instrukční sady HCS08 bylo potřeba spojit obsah souborů `t_6805.asm`, `t_6808.asm` a `t_hcs08.asm` do jednoho souboru `test_hcs08.asm` uvedeného v příloze A. Tento soubor jsem přeložil pomocí překladače Macroassembler AS do souboru kódu s příponou `.p` a tento soubor převedl do binárního formátu pomocí příkazu `p2bin.exe`.

Stejným postupem jako v předchozím odstavci vznikl testovací soubor `testg_hcs08.asm` pro překladač `fasmg` uložený v příloze A. Namísto direktivy `CPU 68HCS08` na začátku souboru je použit příkaz `include` s názvem souboru obsahujícím definici instrukční sady. Zásadní problém je použitá syntaxe instrukcí `BRSET`, `BRCLR`, `BSET` a `BCLR` provádějících manipulaci s bity. Tyto instrukce mají různé operační kódy v závislosti na prvním parametru, kterým je číslo v rozsahu od 0 do 7, specifikujícím konkrétní bit, kterého se operace týká. Macroassembler AS používá syntaxi převzatou z operační tabulky znaků v manuálu k HCS08 [25], kde lze nabýt dojmu, že tento

parametr je součástí mnemotechnické zkratky, a proto je tato instrukce v testovacích souborech zapsána:

```
        brset0  $12,*
```

Jelikož byla pro definiční část ve `fasmg` převzata syntaxe z vývojového prostředí CodeWarrior, je potřeba zápis u těchto instrukcí mírně pozměnit na správný zápis:

```
        brset   0,$20,*
```

Překlad pomocí `fasmg` vygeneruje soubor s daty v binárním formátu. Takto vygenerovaný soubor by se měl shodovat s binárním souborem vygenerovaným překladačem Macroassembler AS. To lze v systému Windows zkontrolovat například pomocí příkazu `fc`.

Pro validaci správně generovaného výstupu byl rovněž použit referenční překladač z prostředí CodeWarrior. Ve vytvořeném projektu s absolutním překladačem JSA byl do zdrojového souboru do části pro kód vložen obsah souboru `testg_hcs08.asm`. Po přeložení a spuštění programu v simulátoru bylo možné exportovat obsah paměti s kódem v binární podobě do souboru `test_hcs08_dcbcw.bin` a tento soubor porovnat s výsledkem překladu z `fasmg`.

6.2 Testování formátovaného výstupu

Správnost formátovaného výstupu byla otestována pro všechny instrukce pomocí zdrojového souboru popsaného v předchozí podkapitole. Po přeložení překladačem `fasmg` byl vytvořen soubor s instrukcemi v textové podobě a jeho obsah byl vložen do zdrojového souboru v projektu prostředí CodeWarrior. Po přeložení byla paměťová oblast s instrukcemi exportována do binárního souboru a ten byl porovnán s binárním výstupem překladu z prostředí CodeWarrior při použití běžného zápisu instrukcí.

Pro otestování na spustitelném kódu byl vytvořen demonstrativní program, který byl vypracován jako samostatná úloha v rámci bakalářského předmětu Vestavné systémy a mikroprocesory. Zadáním této úlohy je napsat program v absolutním jazyce assembler, který provede vyčíslení výrazu

$$res = \frac{4*c1-c2}{5}, \tag{6.1}$$

kde proměnné *res*, *c1*, *c2* o délce 2 bajtů jsou definovány jako globální v paměti RAM.

Původní úloha vytvořená v prostředí CodeWarrior musela být uzpůsobena pro překladač `fasmg`. Konkrétně bylo potřeba upravit původní hlavičkový soubor `derivative.inc` obsahující definici makra pro resetování časovače watchdog a `mc9s08lh64.inc` obsahující klíčová slova specifická pro daný mikrokontroler.

Do `derivative.inc` byla rovněž nadefinovány makroinstrukce `DC.B`, `DC.W`, `DS.B` a `DS.W`, tak, aby jejich funkce odpovídala příkazům `db`, respektive `dw` používaným překladačem `fasmg`. Takto lze v původním zdrojovém souboru odstranit jen direktivy `XDEF` a `ABSENTRY` a soubor je přeložitelný překladačem `fasmg`.

Přeložený výstup byl vložen do nového projektu v prostředí CodeWarrior. Ve zdrojovém souboru je potřeba zanechat původní direktivu `XDEF _Startup`, která exportuje návěští do souboru určeného k linkování a `ABSENTRY _Startup`, která specifikuje, na kterém návěští má program začít. Rovněž je samozřejmě nutné zachovat návěští `_Startup` na své původní pozici vůči kódu. Přeložením a spuštěním programu v simulátoru lze demonstrovat, že formátovaný výstup je spustitelný a provádí stejnou činnost, jako původní úloha napsaná pomocí běžných instrukcí.

6.3 Test generování souboru S-record

Pro testování správného generování souboru ve formátu S-record byl použit soubor obsahující všechny instrukce, který vznikl přidáním generátoru do souboru z kapitoly 6.1. Dále je použit upravený spustitelný program obsahující aritmetické operace z kapitoly 6.2 uložený jako `example_abs.asm` v příloze A.

Do záznamu typu S9 je potřeba uložit adresu, na které program začíná. Jelikož toto zajišťuje u překladače HC(S)08/RS08 Assembler direktiva `ABSENTRY`, nabízí se v generátoru nadefinovat stejnojmennou makroinstrukcí, která plní stejný účel a použít ji ve všech testovacích souborech.

Výstupem obou překladačů jsou soubory ve formátu S-record, které se shodují s výjimkou prvního záznamu typu S0. V případě překladu pomocí `fasmg` zde je uložen název zdrojového souboru a nikoliv název souboru `.prm`, jelikož s ním tento překladač nepracuje.

7. DEMONSTRACE NA REÁLNÉM HARDWARE

Pro demonstrování funkčnosti generovaného spustitelného kódu jsem vytvořil dva programy, které jsem pak nahrával na vývojovou desku TWR-S08LH64, na které je osazen mikrokontroler MC9S08LH64. Tato kapitola krátce popisuje vlastnosti desky a demonstrované programy.

7.1 Vývojová deska TWR-S08LH64 a displej GD-5360P

Deska TWR-S08LH64 je dostupná ve výukových laboratořích na Ústavu automatizace a měřicí techniky. Napájení a programování je uskutečněno přes mini-B USB konektor. Dále je osazena čtyřmi tlačítky, čtyřmi LED diodami, odporovým potenciometrem, akcelerometrem, piezo-bzučákem, světelným senzorem, tlačítkem RESET umožňujícím restart posledně spuštěné úlohy, konektorem pro komunikaci po rozhraní RS-232 a LCD displejem typu GD-5360P. [31]

Na displeji se vyskytuje celkem 55 ovládatelných segmentů. Je určený hlavně pro zobrazování času ve 12hodinovém formátu. Ovládá se pomocí řadiče integrovaného v mikrokontroleru MC9S08LH64, který se ovládá změnou hodnot v příslušných paměťově mapovaných registrech. K ovládání lze využít knihovnu `lcd_gd-5360p` vytvořenou panem Miroslavem Štibraným v rámci bakalářské práce, která je napsaná v jazyce C. [32]

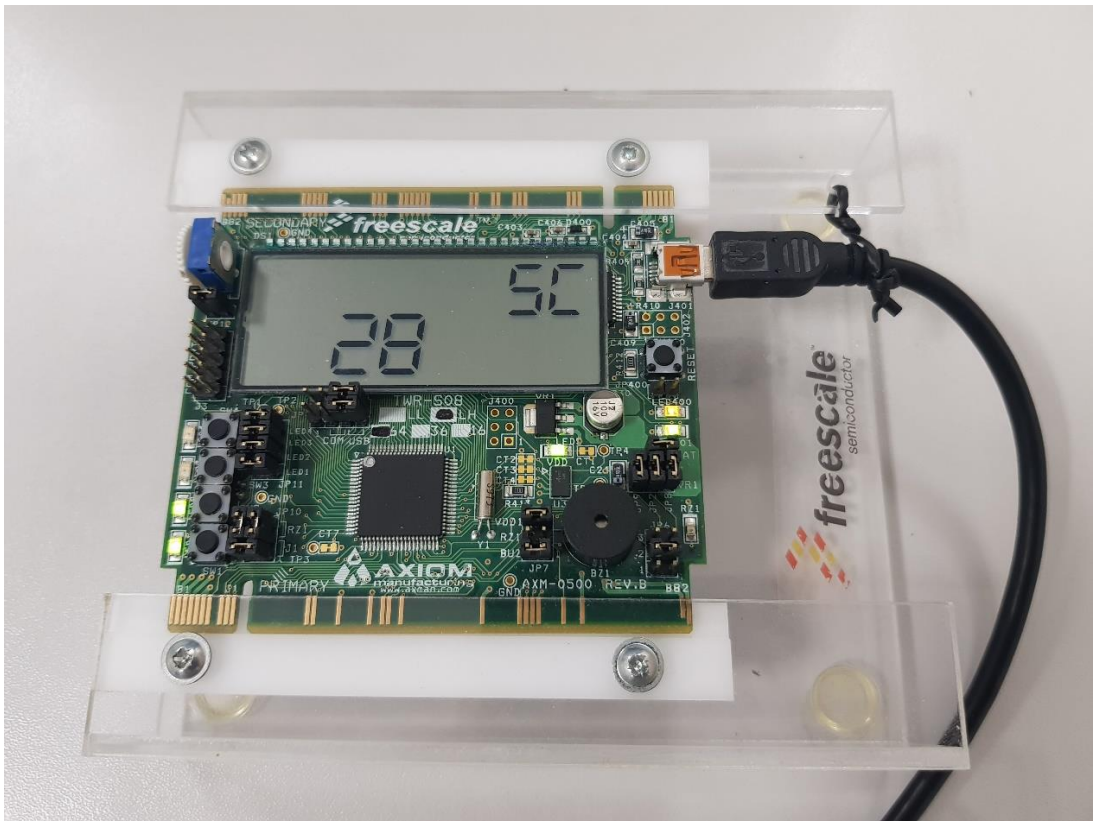
7.2 Adaptace úlohy s aritmetickými operacemi

První program se svou funkcí shoduje s programem z kapitoly 6.2, kde byl napsán v absolutním jazyce assembler. Pro ovládání displeje pomocí knihovních funkcí je ovšem potřeba použít jazyk C. Proto musí být program pro demonstraci napsán v jazyce C a relativním jazyce assembler.

Program je spouštěn z rutiny `main` psané v jazyce C, která obsluhuje počáteční nastavení periferií včetně LCD displeje. Dále je zde prováděno volání rutiny `example_flat_asm` napsané v jazyce assembler. Výsledek této funkce je zobrazen na segmentech displeje v hexadecimální podobě a hodnoty nejnižších čtyř bitů z výsledku jsou indikovány zhasnutím jednotlivých LED diod.

Samotný soubor pro překladač relativního jazyka assembler obsahuje, jak kód napsaný pomocí běžného zápisu v JSA, tak pomocí formátovaného výstupu z `fasmg`. Dále obsahuje definici proměnných pro tuto rutinu v segmentu zvaném Z_RAM. Přesné umístění počátku tohoto segmentu v paměti lze vyčíst z konfiguračního souboru pro linker `Project.prm`. Hodnoty, se kterými má rutina pracovat, nejsou předávány jako parametr funkce, nýbrž jsou definované přímo v kódu, protože jejich předávání by bylo komplikovanější na programování.

Program z kapitoly 6.2 bylo potřeba upravit tak, aby výsledek výpočtu byl načten do registrového páru H:X, kterým je předávána návratová hodnota celého 2bajtového čísla [33], a místo nekonečného opakování došlo k návratu z rutiny. Při vytvoření zdrojového souboru pro překladač `fasmg` pak bylo potřeba opět správně nadefinovat hodnoty proměnných a pomocí pseudoinstrukce `ORG` je přiřadit na správnou adresu. Pakliže nebude obsah souboru Project.prm měněn, bude tato hodnota rovna 0x0060. Z přeloženého souboru jsem zkopíroval pouze část obsahující spustitelný kód a vložil ho do souboru v prostředí CodeWarrior za návěští `example_flat_asm`. Toto návěští spolu s návěštím `example_asm`, které ukazuje na původní kód, jsem pak exportoval pro linkování pomocí pseudoinstrukce `XDEF`. V hlavičkovém souboru `example_asm.h` je pak potřeba deklarovat funkce v jazyce C se stejnými názvy. Tyto funkce lze pak volat z rutiny `main`. Jednoduchým přepsáním programu lze pak ověřit, že obě funkce provádí stejnou činnost.



Obrázek 7.1 Zobrazený výsledek výrazu (6.1). Nejnižší čtyři bity výsledného čísla jsou 1100, což je kvůli obrácené logice výstupu indikováno zhasnutím vrchních dvou LED diod.

7.3 Program implementující LFSR typu Galois

Druhý demonstrační program provádí výpočet stavu 16bitového LFSR čítače typu Galois s polynomem $x^{15}+x^{13}+x^{12}+x^{10}+1$. Tento čítač se často používá jako generátor pseudonáhodných čísel. Pokud známe jeho polynom, délku čítače a počáteční hodnotu, můžeme vypočíst jeho následující hodnotu. [34]

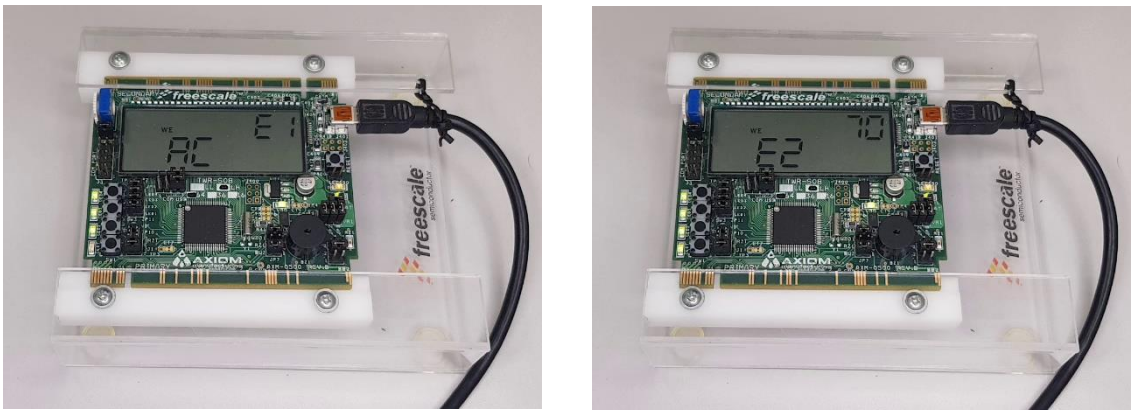
Část programu napsaná v jazyce C postupně zobrazuje na displeji hodnoty 3 různých proměnných. První proměnná, jejíž zobrazení je indikováno zapnutým segmentem s textem MO, má hodnotu výsledku výpočtu pomocí funkce napsané v jazyce C. Druhá proměnná indikovaná segmentem TU má hodnotu výsledku výpočtu pomocí funkce napsané v jazyce assembler. Třetí hodnota indikovaná segmentem WE je vypočtena funkcí zapsanou formátovaným výstupem z překladače `fasmg`. Nejnižší 4 bity z právě zobrazené proměnné jsou rovněž indikovány zhasnutím příslušných LED diod. Program dále aktivně čeká na zmáčknutí tlačítka SW1 připojeného na port PTA6 a při jeho stisknutí provede výpočet následující hodnoty LFSR čítače pomocí všech tří zmíněných funkcí. Pakliže se výsledky shodují, je algoritmus zapsaný v jazyce assembler shodný s výstupem z překladače `fasmg` a také s funkcí zapsanou v jazyce C.

Zdrojový kód v jazyce assembler byl vytvořen stejně jako u předcházejícího příkladu s rozdílem, že neobsahuje lokální proměnné. Jeho tvorba tak byla jednodušší.

Videonahrávka zobrazující jednotlivé hodnoty LFSR čítače na displeji vývojové desky je uložena v příloze C.

Tabulka 7.1 Sekvence hodnot LFSR čítače pro počáteční hodnotu 0xACE1 vypočtená pomocí referenční funkce v jazyce C

0xACE1	**0xE270**	**0x7138**	**0x389C**	**0x1C4E**	**0x0E27**	**0xB313**	**0xED89**	**0xC2C4**



Obrázek 7.2 Výsledek výpočtu dvou po sobě jdoucích hodnot LFSR čítače pomocí funkce zapsané formátovaným výstupem z `fasmg`

8. ZHODNOCENÍ IMPLEMENTOVANÉHO ŘEŠENÍ

V této kapitole jsou uvedena omezení implementovaného řešení, na které je potřeba při použití brát zřetel. Dále se kapitola zaměřuje především na měřením rychlostí překladu. Je zde srovnána rychlost překladu pomocí překladače HC(S)08/RS08 Assembler a `fasmg` s instrukční sadou definovanou pomocí makroinstrukcí a pomocí CALM instrukcí. Jelikož HC(S)08/RS08 assembler umožňuje přímo překládat pouze do formátu ELF, ze kterého je pak automaticky generován soubor S-record [33], testovací soubory pro překladač `fasmg` rovněž obsahují generátor tohoto formátu.

8.1 Omezení implementovaného řešení

Jedním z omezení maker a CALM instrukcí, je, že neumožňují limitovat přístup k jejich používání, jako je tomu například u metod a proměnných třídy v objektově orientovaných programovacích jazycích. Lze tak ze zdrojového kódu volat i makra, která zajišťují funkci implementovaného řešení, ale nejsou určeny k přímému používání programátorem. Snahou, jak alespoň nechtěnému použití těchto maker předejít, bylo jejich začlenění do jejich vlastního jmenného podprostoru.

Dále při definici instrukční sady jsou implementovány pouze kontroly, zda jsou operandy, se kterými instrukce pracuje, ve správném rozsahu a formátu. Kromě daných kontrol jsem nesledoval další krajní situace zápisu kódu, při kterých by překlad proběhl úspěšně, ovšem nesprávně. Tyto problémy bych řešil po otestování překladače na komplexnějších úlohách. Pakliže se ale bude programátor držet syntaxe používané překladačem HC(S)08/RS08, překlad proběhne správně, jak již bylo testováno v kapitolách 6 a 7.

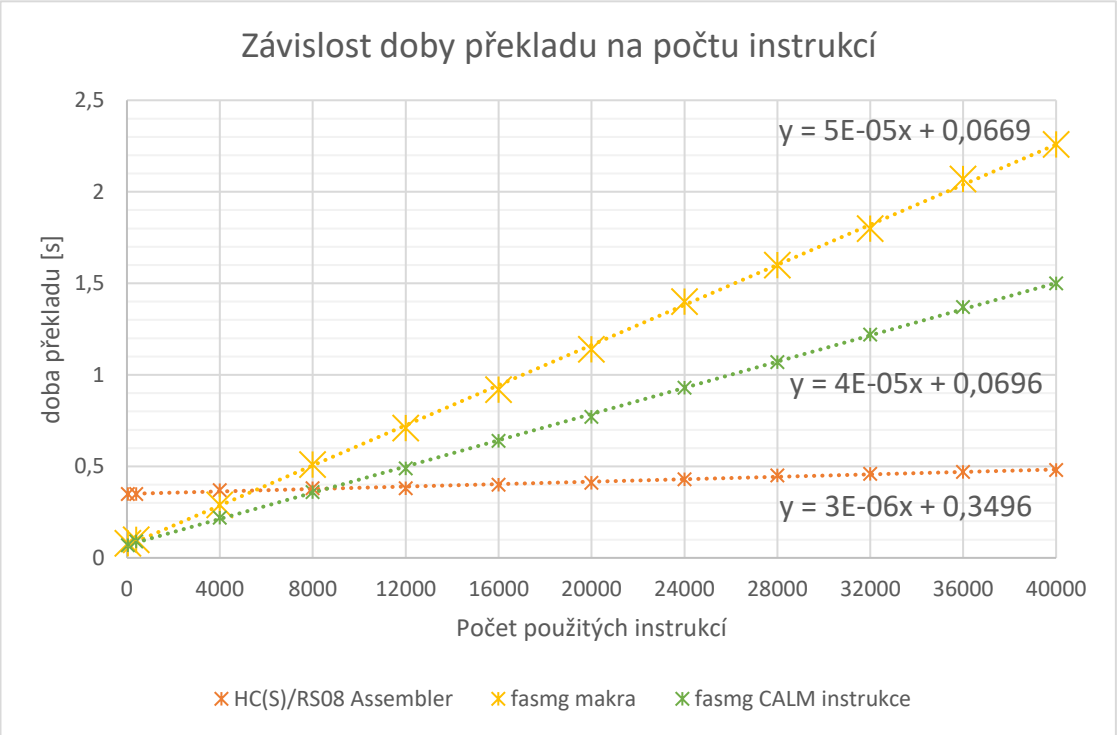
8.2 Postup měření

Měření času je realizováno skriptem `timecmd.bat` spouštěného z příkazové řádky, který spočítá rozdíl času mezi dokončením a spuštěním příkazu, který ho následuje, a výsledek zobrazí na obrazovce. V mém případě se jedná o příkaz volající překladač.

Samotný překladač HC(S)08/RS08 Assembler, který lze spouštět z příkazové řádky, je uložen v podsložce s instalací prostředí CodeWarrior pod jménem `ahc08.exe`. Tento program se spouští s parametry `-Cs08 -Compat=b -FA2` a s názvem souboru, který má přeložit. Parametr `-Cs08` specifikuje použití instrukční sady architektury HCS08, `-FA2` nastaví formát výstupu jako absolutní soubor formátu ELF a `-Compat=b` povoluje použití pseudoinstrukce `FOR` a `ENDFOR`. Pokud je potřeba zaznamenat informace, které překladač generuje, do souboru, lze tak učinit přidáním znaku > následovaným názvem souboru za spouštěný skript. Překladač `fasmg`, je z příkazové řádky spouštěn s názvem

zdrojového souboru spolu s parametrem `-n`, který potlačí výpis informací o počtu průchodů a generovaných bajtů.

Rychlost překladu je testována na souborech `performance_test.asm` a `performance_test_calm.asm`, které obojí obsahují stejnou sekvenci 40 instrukcí, která je ovšem uzavřena do bloku příkazu `repeat`, aby se počet volaných instrukcí mohl snadno měnit. První soubor obsahuje definiční část napsanou pomocí maker, druhý pomocí CALM instrukcí. Záměrně jsou tedy použity jen ty instrukce, které jsem pomocí CALM instrukcí měl v době testování implementované. Ekvivalentní zdrojový soubor byl vytvořen i ve vývojovém prostředí CodeWarrior, kde místo blokového příkazu `repeat` jsou použity pseudoinstrukce `FOR` a `ENDFOR`, jejichž použití je nutno povolit v nastavení překladače. Všechny tři soubory obsahují mimo opakovaných instrukcí také direktivy a části kódu, které jsou nutné k úspěšnému překladu. Rychlost překladu takto byla proměřena pro 40 až 40 000 přeložených instrukcí. Naměřená data jsou uložena v tabulkovém souboru v příloze C.



Obrázek 8.1 Graf s vynesenou závislostí doby překladu na počtu instrukcí

8.3 Zhodnocení výsledků měření

Z naměřených dat lze dojít k závěru, že pro zdrojové soubory s malým počtem instrukcí je překlad pomocí překladače `fasmg` rychlejší. V případě delších kódů s více jak 8 000 instrukcemi je pak 16krát rychlejší referenční překladač HC(S)08/RS08 Assembler v porovnání s `fasmg` s instrukční sadou nadefinovanou pomocí maker. Pokud k definici

instrukční sady použijeme CALM instrukce, je doba překladu 1,25krát rychlejší než při použití maker, čímž bylo prokázáno tvrzení z kapitoly 2.2.2.

Výše uvedená tvrzení je však potřeba zohlednit vzhledem ke skutečnosti, že generátor výstupního formátu je sám napsán pomocí maker a dal by se zrychlit použitím CALM instrukcí. Dále také HC(S)08/RS08 Assembler překládá nejprve kód do objektového souboru ELF, ze kterého teprve vytváří soubor ve formátu S-record, přičemž oba soubory jsou ukládány na disk.

9. ZÁVĚR

Cílem této bakalářské práce bylo zachytit postup tvorby definiční části retargetable překladače Flat Assembler G a demonstrovat funkčnost jeho správného překladu. Rovněž demonstruje jeho možnosti použití při vytváření části překladače zpracovávající čistě binární výstup do různých výstupních textových formátů.

V první kapitole této práce byly představeny principy fungování překladačů jazyka assembler. Dále byl proveden průzkum překladačů, které se označují jako retargetable (v této práci označované také jako obecné). Byl proveden stručný rozbor jejich architektury a způsoby, jakými je lze adaptovat na nové rodiny procesorů. Bylo zjištěno, že každý z uvedených překladačů svou obecnost implementuje jinak. Většinou je rozšíření o podporu nového mikroprocesoru realizováno úpravou zdrojových souborů, čímž se překladač Flat Assembler G od nich odlišuje.

V druhé kapitole byl představen Flat Assembler a jeho obecná varianta Flat Assembler G. Obě varianty jsou navrhnuty pro překládání programů pro ploché rozložení paměti a odtud získaly svůj anglický název. Bylo potřeba nastudovat především syntaxi určenou pro definici instrukční sady nové rodiny procesorů. Jako hlavní nástroj zde byly popsány makra a CALM instrukce (z angl. Compiled Assembly-like Macro), které jsou na rozdíl od maker kompilované již při své definici a jsou pro účel definice instrukční sady přímo vytvořené. Tato kapitola ani celá práce se však nevěnuje všem příkazům, které lze při programování pro překladač Flat Assembler G použít.

V třetí kapitole bylo provedeno seznámení s vlastnostmi rodiny mikroprocesorů HCS08, především s vlastnostmi jeho instrukční sady a s ní souvisejících pojmů, které je nutné pro psaní programu v JSA nastudovat. Dále zde bylo představeno vývojové prostředí CodeWarrior obsahující překladač HC(S)08/RS08 Assembler, který byl v práci použit jako referenční.

Ve čtvrté kapitole byl s využitím ukázek kódu prezentován způsob, jakým byla definována instrukční sada rodiny HCS08 jak pomocí maker, tak s využitím CALM instrukcí. V rámci této práce byla implementována popsaným způsobem pomocí maker celá instrukční sada architektury HCS08. Pomocí CALM instrukcí bylo z časových důvodů implementováno pouze 40 instrukcí.

V páté kapitole byly rozebrány možné formáty výstupního souboru, které by se daly propojit s kódem pro referenční překladač. Byly zde rozebrány existující formáty ELF a S-record. Pro další práci byl zvolen výstupní textový formát obsahující posloupnost pseudoinstrukcí `ORG` a `DC.B` následovaných konstantami reprezentujícími program ve strojovém kódu. Následně jsou zde rozebrané instrukce, kterými lze naimplementovat generátor tohoto formátu pro překladač Flat Assembler G

V rámci šesté kapitoly jsem vytvořil syntetický test pomocí zdrojového souboru obsahujícího všechny podporované instrukce dané rodiny procesoru a ověřil jsem validitu

přeloženého programu oproti referenčnímu překladači. Dále jsem otestoval správnost generování zvoleného výstupního formátu z předchozí kapitoly a formátu S-record.

V sedmé kapitole byla správnost generovaného formátu otestována na dvou úlohách spouštěných na reálné vývojové desce. První úloha provádí aritmetické operace odečítání, násobení a dělení, druhá výpočet hodnoty LFSR čítače typu Galois. Tímto byla otestována správnost mnou napsané definice instrukční sady pro překladač Flat Assembler G a správná funkce generátoru formátovaného textového výstupu.

V osmé kapitole jsou uvedena omezení použití implementovaného řešení. Dále byly provedeny srovnávací testy rychlosti překladače HC(S)08/RS08 Assembler a překladače Flat Assembler G s definovanou instrukční sadou pomocí maker a také s instrukční sadou definovanou CALM instrukcemi. Testy naplnily předpoklad, že definice instrukční sady pomocí CALM instrukcí vede k zvýšení rychlosti překladu o čtvrtinu oproti definici pomocí maker. Dále testy ukazují, že se zvyšujícím se počtem překládaných instrukcí narůstá doba překladu pomocí překladače Flat Assembler G 16krát rychleji, než je tomu u referenčního překladače HC(S)08/RS08 Assembler.

Výstupem této práce je tedy otestovaný soubor s kompletní definicí všech instrukcí používaných rodinou mikroprocesorů HCS08, který po začlenění do zdrojového kódu umožňuje psát programy v JSA pro překladač Flat Assembler G se stejnou syntaxí instrukcí, jakou používá referenční překladač HC(S)08/RS08 Assembler. Dále byly pro překladač Flat Assembler G vytvořeny dva generátory výstupních formátů, které zpracovávají binární výstup překladu. První byl použit pro zpětné propojení s referenčním překladačem, druhý umožňuje generovat soubory ve formátu S-record ve variantě S19. Celkově jsem psaním práce, zdrojových souborů a testováním správnosti překladu věnoval přibližně 550 hodin.

Hlavním přínosem práce bylo zdokumentovat tvorbu instrukční sady pro překladač Flat Assembler G. Tento nástroj lze využít, pokud potřebujeme vytvořit překladač jazyka assembler pro nový, například v FPGA vytvořený, procesor a neklademe velké požadavky na rychlost překladu. Použití tohoto překladače je výhodné v tom, že pro vytvoření nového překladače nemusíme znovu vytvářet nástroje provádějící lexikální analýzu nad zdrojovým souborem, algoritmy operující se symboly, návěštími a direktivami a další nástroje spojené s tvorbou moderního překladače. Navíc při definování instrukční sady pro překladač Flat Assembler G si lze osvojit použití bohaté škály příkazů, kterou tento překladač poskytuje, a znovu ji využít při psaní programů pro konkrétní procesor.

Další možným rozšířením práce by mohlo být kompletní přepsání definice instrukční sady a také generátoru výstupního formátu pomocí CALM instrukcí, čímž by se zvýšila rychlost překladu. Dále by mohl být současný generátor souborů formátu ELF, který je distribuovaný spolu s překladačem Flat Assembler G, přeprgramován na generování ELF souborů pro architekturu HCS08. Problematika formátu ELF však svou náročností převyšuje rámec bakalářské práce.

LITERATURA

[1] BARRON, D. W. (David William). Assemblers and Loaders. New York, 1971, 61 s.

[2] Linkers and Loaders. SciTech Book News [online]. Portland: Ringgold, 2000, 24(4), 28 [cit. 2021-12-28]. ISSN 0196-6006.

[3] ABBASPOUR, M a Jianwen ZHU. Retargetable binary utilities. In: Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324) [online]. IEEE, 2002, s. 331-336 [cit. 2021-12-21]. ISBN 1581134614. ISSN 0738-100X. Dostupné z: doi:10.1109/DAC.2002.1012645

[4] ABBASPOUR, Maghsoud a Jianwen ZHU. Automatic Porting of Binary File Descriptor Library. BADAWY, Wael a Graham JULLIEN, ed. System-on-Chip for Real-Time Applications [online]. Boston, MA: Springer US, 2003, s. 193-202 [cit. 2021-12-31]. The Kluwer International Series in Engineering and Computer Science. ISBN 978-1-4613-5034-7. Dostupné z: doi:10.1007/978-1-4615-0351-4_18

[5] Binutils Porting Guide To A New Target Architecture. Sourceware.org [online]. [cit. 2022-01-31]. Dostupné z: https://sourceware.org/binutils/binutils-porting-guide.txt

[6] HUSÁR Adam: Implementace obecného assembler. Brno, 2007, diplomová práce, FIT VUT v Brně.

[7] KAUFMANN, Marco. The official CodeX Assembler Homepage. Page of Marco [online]. Marco Kaufmann (C) 2003, 2021 [cit. 2021-12-22]. Dostupné z: https://web.archive.org/web/20210124071007/http://pageofmarco.de/codex/codexdnl.php

[8] ANTON, Treunfels. HXA_Cross_Assembler. GitHub [online]. Fridley MN 55421 [cit. 2021-12-31]. Dostupné z: https://github.com/AntonTreuenfels/HXA_Cross_Assembler

[9] BARTHELMANN, Vollker. Vasm portable and retargetable assembler [online]. [cit. 2021-12-31]. Dostupné z: http://sun.hasenbraten.de/vasm/index.php?view=main

[10] ARNOLD, Alfred. The Beginning of the AS Project. The Macroassembler AS [online]. Aachen, 1998 [cit. 2022-05-05]. Dostupné z: http://john.ccac.rwth-aachen.de:8000/as/beginning.html

[11] ARNOLD, Alfred. User's Manual for Macro Assembler AS. *The Macroassembler AS* [online]. Aachen, 2022 [cit. 2022-05-05]. Dostupné z: http://john.ccac.rwth-aachen.de:8000/as/as_EN.html

[12] GRYZSTAR, Tomasz. Flat assembler 1.73: Programmer's Manual. Flat assembler [online]. 2020 [cit. 2021-12-23]. Dostupné z: https://flatassembler.net/docs.php?article=manual

[13] GRYZSTAR, Tomasz. Design Principles. Flat assembler [online]. 2009 [cit. 2021-12-23]. Dostupné z: https://flatassembler.net/docs.php?article=design
[14] GRYZSTAR, Tomasz. CALM extension of fasmg. Https://board.flatassembler.net/ [online]. [cit. 2021-12-26]. Dostupné z: https://board.flatassembler.net/topic.php?t=17328
[15] GRYZSTAR, Tomasz. Flat Assembler G: Introduction and Overview. Https://flatassembler.net/docs.php [online]. 2021 [cit. 2021-12-26]. Dostupné z: https://flatassembler.net/docs.php
[16] GRYZSTAR, Tomasz. Flat Assembler G: User Manual. Https://flatassembler.net [online]. 2021 [cit. 2021-12-26]. Dostupné z: https://flatassembler.net/docs.php?article=fasmg_manual
[17] GRYZSTAR, Tomasz. CALM extension of fasmg. Https://board.flatassembler.net/ [online]. [cit. 2021-12-26]. Dostupné z: https://board.flatassembler.net/topic.php?p=212226#212226
[18] PEREIRA, Fábio. HCS08 unleashed: designer´s guide to the HCS08 Microcontrollers. 2. Charleston: BookSurge Publishing], 2008, 411 S. ISBN 1-4196-8592-9.
[19] DANIELS, R. Gary a William C. BRUCE. Built-In Self-Test Trends in Motorola Microprocessors. IEEE Design & Test of Computers [online]. **2**(2), 64-71 [cit. 2021-12-31]. ISSN 0740-7475. Dostupné z: doi:10.1109/MDT.1985.294865
[20] MOTOROLA INC. M6805 M146805 Family Microcomputer/Microprocessor Users Manual [online]. [cit. 2021-12-27] 49-51. Dostupné také z: https://archive.org/embed/Motorola-SeminarsandApplicationBooksM6805M146805UsersManualOCR
[21] NXP, MC9S08LH64 Series Data Sheet [online]. Rev. 6.1, 08/2012, [cit. 31. 12. 2021]. Dostupné z URL: https://www.nxp.com/docs/en/data-sheet/MC9S08LH64.pdf
[22] NXP, MC9S08LH64 Reference Manual [online], Rev. 5.1, 05/2012, [cit. 30. 12. 2021]. Dostupné z URL: https://www.nxp.com/docs/en/reference-manual/MC9S08LH64RM.pdf
[23] FREESCALE SEMICONDUCTOR, INC. CodeWarrior Development Studio Common Features Guide [online]. 10.x, 02/2014., 2014 [cit. 2022-01-02]. Dostupné z: https://community.nxp.com/pwmxy87654/attachments/pwmxy87654/cw-development-tools/5128/1/CodeWarrior%20Common%20Features%20Guide.pdf
[24] FREESCALE SEMICONDUCTOR, INC. HC(S)08/RS08 Assembler Manual for Microcontrollers [online]. 2007[cit. 2022-01-02]. Dostupné z: https://www.nxp.com/docs/en/reference-manual/HC08ASMRM.pdf

[25] FREESCALE SEMICONDUCTOR, INC. HCS08 Family Reference Manual [online]. 2., 2007 [cit. 2022-01-02]. Dostupné z: https://www.nxp.com/files-static/microcontrollers/doc/ref_manual/HCS08RMV1.pdf
[26] Dokumentace formátu ELF binárních souborů [online]. In: . Prentice Hall (UNIX Press), 1999 [cit. 2022-03-15]. Dostupné z: http://www.skyfree.org/linux/references/ELF_Format.pdf
[27] Motorola 8- and 16-bit Embedded Application Binary Interface (M8/16EABI): SYSTEM V APPLICATION BINARY INTERFACE Motorola M68HC05, M68HC08, M68HC11, M68HC12, and M68HC16 Processors Supplement. In: Uclibc.org [online]. Motorola, Inc.: Transportation Systems Group, Microcontroller Division, 2012, 5. 5. 2012 [cit. 2022-05-03]. Dostupné z: https://uclibc.org/docs/psABI-m8-16.pdf
[28] MC68000 FAMILY PROGRAMMER'S REFERENCE MANUAL [online]. In: . Motorola, 1992, Příloha C [cit. 2022-05-03]. ISBN 978-0-13723289-5. 978-0-13723289-5. Dostupné z: https://www.nxp.com/files-static/archives/doc/ref_manual/M68000PRM.pdf
[29] KŘENEK, P. USB zavaděč pro 8/32 bitové mikrokontroléry. Brno, 2009. 80 s. Vedoucí diplomové práce Ing. Pavel Lajšner. Freescale Polovodiče ČR
[30] HC(S)08/RS08 and S12(X) Build Tools Utilities Manual [online]. Austin, TX 78735 U.S.A.: Freescale Semiconductor, 2010 [cit. 2022-05-12]. Dostupné z: https://www.nxp.com/docs/en/reference-manual/HCS-RS08-Build_Tools_Utilities.pdf
[31] AXIOM MANUFACTURING: TWR-S08 User Guide, Rev. F [online].9/2009 updated 4/2010. [cit. 2022-05-13] Dostupné na URL: https://www.nxp.com/docs/en/user-guide/TWRS08LH64UG.pdf
[32] ŠTIBRANÝ, M. Knihovna pro řízení LCD displeje GD-5360P. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2014. 55 s.Vedoucí bakalářské práce Ing. Tomáš Macho, Ph.D..
[33] *HC(S)08 Compiler Manual* [online]. Austin, TX 78735 U.S.A.: Freescale Semiconductor, 2005 [cit. 2022-05-13]. Dostupné z: https://ece-classes.usc.edu/ee459/library/documents/Compiler_HC08.pdf
[34] KOLOUCH, Jaromír. *Programovatelné logické obvody: přednášky*. Vyd. 3., dopl. Brno: VUT FEKT, ústav radioelektroniky, 2006, 76 s. : il. ISBN 80-214-3270-5.

SEZNAM SYMBOLŮ A ZKRATEK

Zkratky:

ABI	Aplikační binární rozhraní (angl. Application Binary Interface)
ADL	Jazyky popisující architektruru (angl. Architecture Description Language)
ALU	Aritmeticko-logická jednotka (angl. Arithmetic logic unit)
BFD	angl. Binary File Descriptor
BDM	angl. Background Debug Module
CISC	Architektura s komplexní instrukční sadou (angl. Complex Instruction Set Computer)
CPU	Centr. procesorová jednotka (angl. Central Processing Unit)
CALM	angl. compiled Assembly-like macro
ELF	spustitelný a linkovatelný formát (angl. Executable and Linking Format)
EABI	Aplikační binární rozhraní pro vestavěné systémy (angl. Embedded Application Binary Interface)
Fasm	Flat Assembler
Fasmg	Flat Assembler G
FLASH	Nevolatilní elektronicky programovatelná a mazatelná paměť
GCC	Sada překladačů vytvořených v rámci projektu GNU (angl. GNU Compiler Collection)
JSA	Jazyk symbolických adres
LED	Elektroluminiscenční dioda (angl. Light-Emitting Diode)
LCD	Displej z tekutých krystalů (angl. Liquid Crystal Display)
RAM	Paměť s náhodným přístupem (angl. Random Access Memory)
RISC	Architektura s redukovanou komplexní instrukční sadou (angl. Reduced Instruction Set Computer)
USB	Univerzální sériová sběrnice (angl. Universal Serial Bus)

SEZNAM PŘÍLOH

Příloha A - Zdrojové kódy a přeložené soubory

Příloha obsahuje zdrojové soubory pro překladač Flat Assembler G a Macroassembler AS. Dále také obsahuje soubory vzniklé jejich překladem.

```
│   cw.inc.....generátor souborů ve formátu propojitelným s CodeWarrior
│   derivative.inc.....hlavičkový soubor s makry používanými CodeWarrior
│   hcs08.inc.....definice instrukční sady pomocí maker
│   hcs08_calm.inc.....definice instrukční sady pomocí CALM instrukcí
│   MC9S08LH64.inc..... hlavičkový soubor pro překladač fasmg mapující periferie
│   srec.inc.....generátor souborů ve formátu S-record
│
├─binary_test..... složka se syntetickými testy všech instrukcí
│   calm_testing.asm.....test pro definici pomocí CALM instrukcí
│   calm_testing.bin.....přeložený test definice pomocí CALM instrukcí
│   testg_hcs08.asm..... test pro definici pomocí maker
│   testg_hcs08.bin..... přeložený test definice pomocí maker
│   testg_hcs08_cw.asm..... test ve formátu propojitelném s CodeWarrior
│   testg_hcs08_cw.txt ..... přeložený test ve formátu propojitelném s CodeWarrior
│   testg_hcs08_srec.asm.....test ve formátu S-record
│   test_hcs08.asm.....test definice pro Macroassembler AS
│   test_hcs08.bin..... přeložený test definice pro Macroassembler AS
│   test_hcs08.p .....soubor kódu pro Macroassembler AS
│   test_hcs08_cw.bin ..... export paměti programu testing_all z CodeWarrior
│   test_hcs08_dcbcw.bin..... export paměti programu testing_all_g z CodeWarrior
│
├─out_test ..... složka s testovacími úlohami
│   example_abs.asm.....zdrojový kód programu s aritmetickými operacemi
│   example_abs.s19.....program s aritmetickými operacemi ve formátu S-record
│   example_abs.txt.....program s aritmetickými operacemi pro CodeWarrior
│   example_asm_cw.asm .....zdrojový kód funkce s aritmetickými operacemi
│   example_asm_cw.txt .....přeložená funkce do formátu pro CodeWarrior
│   lfsr.asm ..... zdrojový kód funkce počítající stavy LFSR čítače
│   lfsr.txt .....přeložená funkce LFSR čítače do formátu pro Codewarrior
│
└─perf_test ..... složka s testy rychlosti překladu
    performance_test.asm.....test s definicí pomocí maker
    performance_test.s19 přeložený test s definicí pomocí maker ve formátu S-rec
    performance_test_calm.asm..... test s definicí pomocí CALM instrukcí
    performance_test_calm.s19.přeložený test s definicí pomocí CALM instrukcí
```

Příloha B - Projekty pro CodeWarrior

Příloha B obsahuje v jednotlivých složkách exportované projekty z vývojového prostředí CodeWarrior.

├─`arithmetic_test` ..... Program s aritmetickými operacemi v rel. JSA a jazyce C
├─`arithmetic_test_abs` ..... Program s aritmetickými operacemi v abs. JSA
├─`artihmetic_test_abs_DCB_HW` .... Program s aritm. op. ve formátovaném výstupu
├─`galois_lfsr` ..... Program počítající stavy LFSR čítače v rel. JSA a jazyce C
├─`performance_test` ..... Projekt k testování rychlosti překladu
├─`testing_all` ..... Projekt k testování správnosti definice instrukční sady
└─`testing_all_g` ..... Projekt k testování správnosti formátovaného výstupu z `fasmg`

Příloha C - Videonahrávka a naměřená data

`demonstrace LFSR.mp4` .....nahrávka demonstrující program s LFSR čítačem

`měření.xlsx` .....naměřené hodnoty dob překladu

`timecmd.bat` ..... skript k měření času