VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

MOŽNOSTI PROGRAMOVÁNÍ MIKROŘADIČŮ

MICROCONTROLLERS PROGRAMMING POSSIBILITIES

BAKALÁŘSKÁ PRÁCE
BACHELOR THESIS

AUTOR PRÁCE **STANISLAV ŠUBRT**
AUTHOR

VEDOUCÍ PRÁCE **ING. PAVEL HOUŠKA, PH.D.**
SUPERVISOR

BRNO 2011

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Děkanát
Akademický rok: 2010/2011

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

student(ka): Stanislav Šubrt

který/která studuje v **bakalářském studijním programu**

obor: **Strojní inženýrství (2301R016)**

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Možnosti programování mikrořadičů

v anglickém jazyce:

Microcontrollers programming possibilities

Stručná charakteristika problematiky úkolu:

Mikrořadiče jsou základním prvkem automatizačních prvků, používají se v řídicích systémech, snímačích, řídicích jednotkách, informačních terminálech. I přes současný rozvoj v této oblasti, tak je vlastní programování aplikací na mikrořadičích vysoce odborná a časově náročná záležitost. Cílem práce je seznámit se s aktuálními vývojovými prostředky pro mikrořadiče na trhu a porovnat jejich výhody a nevýhody z inženýrského pohledu.

Cíle bakalářské práce:

1. Seznamte se vnitřních strukturou mikrořadičů z pohledu programátora;
2. Seznamte se základy programování mikrořadičů v jazycích asm a C, popište výhody a nevýhody použití těchto jazyků.
3. Seznamte se s programování mikrořadičů pomocí nástrojů vyšší úrovně, popište výhody a nevýhody použití těchto prostředků.
4. Popište přenositelnost kódu vytvořeného v různých vývojových prostředcích mezi různými architekturami a rodinami mikrořadičů.

Seznam odborné literatury:

[1] Pechal S.: Monolitické mikropočítače, BEN - technická literatura, Praha 1998, 2. aktualizované vydání, ISBN 80-86056-30-9
[2] Martin, T: The Insider's Guide To the NXP LPC2300/2400 Based Microcontrollers, Hitex (UK) Ltd., 2007, ISBN: 0-9549988 6
[3] Getting Started with the NI LabVIEW Embedded Module for ARM Microcontrollers, National Instruments 2010, Part Number 375177B-01, dostupné z www.ni.com

Vedoucí bakalářské práce: Ing. Pavel Houška, Ph.D.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2010/2011.

V Brně, dne 22.11.2010

L.S.

	prof. RNDr. Miroslav Doupovec, CSc.
Ředitel ústavu	Děkan fakulty

Abstrakt

Tato práce se zabývá vnitřní strukturou a možnostmi programování mikrořadičů. Cílem práce je popsat vnitřní strukturu mikrořadičů z pohledu programátora, zhodnotit výhody a nevýhody jejich programování v jazyce symbolických adres, jazyce C a nástrojích vyšší úrovně. Dále se práce zabývá přenositelností kódu mezi různými rodinami a architekturami mikrořadičů.

Abstract

This work is focused on inner structure and programming possibilities of microcontrollers. The work goal is to describe inner structure of microcontrollers from the programmer's point of view, find out advantages and disadvantages of their programming in Assembly language, C language and higher-level languages. Code portability between different microcontrollers' families and architectures is discussed further.

Klíčová slova

Mikrořadič, mikrokontrolér, jednočip, procesor, CPU, paměti, periférie, činnost, programování, Assembler, C

Keywords

Microcontroller, processor, CPU, memory, peripherals, programming, Assembler, C

Bibliografická citace

Šubrt, Stanislav. *Možnosti programování mikrořadičů.* Brno : Vysoké Učení Technické, Fakulta Strojního Inženýrství, 2011. Vedoucí bakalářské práce Ing. Pavel Houška, Ph.D.

Prohlášení

Prohlašuji, že jsem bakalářskou práci na téma Možnosti programování mikrořadičů vypracoval samostatně pod vedením Ing. Pavla Houšky, Ph.D. s využitím materiálů uvedených v seznamu použité literatury.

Dne 25. 5. 2011 Stanislav Šubrt

Poděkování

Děkuji svému vedoucímu Ing. Pavlu Houškovi, Ph.D. za odborné vedení, cenné rady a také za jeho drahocenný čas, který mi věnoval při psaní této práce.

Obsah

1 Úvod 15
2 Struktura mikrořadiče 17
2.1 CPU - Centrální procesorová jednotka 17
2.1.1 Části CPU 20
2.2 Paměti 22
2.2.1 Vyrovnávací paměť 23
2.3 Periférie 23
2.3.1 I/O 24
2.3.2 Čítač/časovač 24
2.3.3 ADC/DAC 24
2.3.4 Sběrnice 24
2.4 Popis činnosti CPU 25
2.4.1 Časování CPU 25
2.4.2 Běh programu 26
2.4.3 Přerušovací systém 27
2.4.4 Reset 27
3 Programování mikrořadičů v programovacích jazycích Asm a C 29
3.1 Jazyk symbolických adres 29
3.2 Programovací jazyk C 29
3.3 Práce s proměnnými 29
3.3.1 Základní datové typy 29
3.3.2 Logické datové typy 29
3.3.3 Celočíselné datové typy 29
3.3.4 Datové typy s plovoucí desetinou čárkou 30
3.3.5 Ukazatele 30
3.3.6 Uživatelem definované typy 30
3.3.7 Pole 31
3.3.8 Proměnné v Assembleru 31
3.3.9 Proměnné v C 32
3.4 Funkce 33
3.4.1 Definice funkcí v Assembleru 33
3.4.2 Volání funkcí v Assembleru 33
3.4.3 Definice funkcí v C 34
3.4.4 Volání funkce v C 35
3.5 Obsluha přerušení 35

3.6 Přístup k paměti 36
3.6.1 Typy úložišť 36
3.6.2 Paměťové modely 37
3.6.3 Smíšený paměťový model 37
3.6.4 Direktiva _at 38
4 Programování mikrořadičů pomocí nástrojů vyšší úrovně 39
4.1 Microsoft .NET Micro Framework 39
4.2 LabVIEW 40
5 Možnosti přenositelnosti kódu mezi různými architekturami mikrořadičů 41
6 Závěr 43

1 Úvod

Rychlý vývoj a snižování cen elektroniky způsobily její masivní rozšíření nejen do domácností, ale také do všech odvětví služeb, průmyslu i zemědělství. Málokdo si uvědomuje, že uvnitř většiny těchto zařízení pracuje v nenápadném pouzdře výkonný mikrořadič, který celý obvod významně zjednodušuje a zvedá možnosti obsluhy na mnohem vyšší úroveň. Mobilní telefony, tablety, GPS navigace, kalkulačky, rádia, televize, tiskárny, hudební přehrávače, ledničky, pračky, sporáky, myčky nádobí, měřící zařízení, senzory, automobily, letadla, lodě, obráběcí centra, průmyslové roboty, hydraulické ruce, zemědělské postřikovače, docházkové systémy, automatizační prvky, informační terminály atd. běžně obsahují jeden nebo i více mikrořadičů (v praxi se často také používají označení jednočip, mikrokontrolér, mikročip nebo mikropočítač).

I přes značný pokrok v této oblasti je vývoj aplikací pro mikrořadiče stále vysoce odborná a časově náročná činnost. Programátor musí být dokonale seznámen s danou architekturou a její softwarovou obsluhou. To zahrnuje seznámení se s procesorem, jeho instrukční sadou a registry, dále s přistupováním k paměti a s obsluhou přerušení.

Cílem této bakalářské práce je komplexní náhled do problematiky mikrořadičů od jejich struktury až po „portaci“ kódu mezi architekturami. První část se věnuje mikrořadičům z hardwarového hlediska. Jednotlivé kapitoly, se postupně zabývají vnitřními částmi mikrořadičů tak, aby poskytovaly stručný, ale kompletní přehled základních znalostí z pohledu programátora.

Ve druhé části přechází práce k problematice programování mikrořadičů. Nejprve je popisován strojový kód a dále jazyky nejnižší, nižší a vyšší úrovně. Jako zástupci nejnižších a nižších jazyků byly vybrány jazyky Assembler a C. Velká část práce je věnována programování v těchto jazycích a jejich srovnání. Jako zástupci vyšších programovacích jazyků byly vybrány jazyky C# v prostředí Visual Studio a G v prostředí LabVIEW. Jsou zdůrazněny jejich výhody, nevýhody a základní principy tvorby kódu.

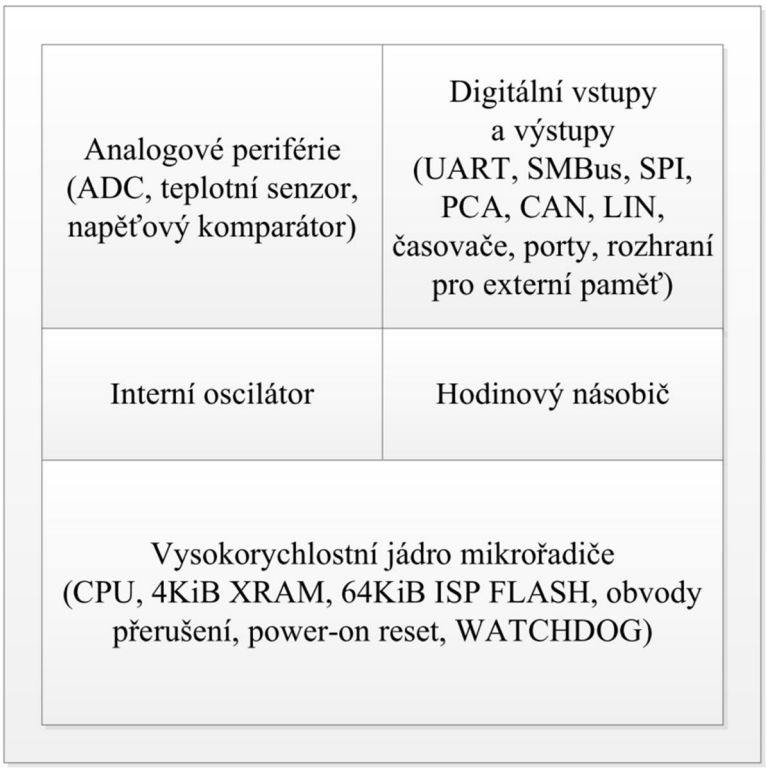
V závěrečné části se práce věnuje problematice přenositelnosti kódu mezi rodinami a architekturami mikrořadičů. Nejprve je vysvětleno, proč je přenositelnost kódu tak důležitá a má smysl se jí zabývat. Dále jsou postupně popsány jazyky assembler, C, C# a G z hlediska přenositelnosti kódu a jaké jsou možnosti a základní principy psaní přenositelného kódu.

2 Struktura mikrořadiče

Mikrořadič je samočinný počítač, který vykonává program uložený v paměti. Realizován je jako integrovaný obvod, který v současnosti převážně integruje všechny potřebné části pro svou samostatnou činnost (k jeho provozu stačí pouze zdroj napájení).

Základní části mikrořadiče jsou centrální procesorová jednotka (CPU), která vykonává program, sběrnice (datové a adresové), paměť typu RAM pro operandy, paměť typu ROM (PROM, OTPROM, EPROM, EEPROM, FLASH) pro program a konstanty, prvky potřebné pro činnost CPU, vstupy/výstupy (digitální/analogové), časovače, čítače, komunikační sériové sběrnice a další.

Co je v mikrořadiči integrováno, závisí na výrobci a době vzniku. Na návrháři aplikace závisí, jaký zvolí typ a tím pádem, které periférie budou dostupné na mikrořadiči a které periférie budou připojeny jako samostatné obvody. [1]



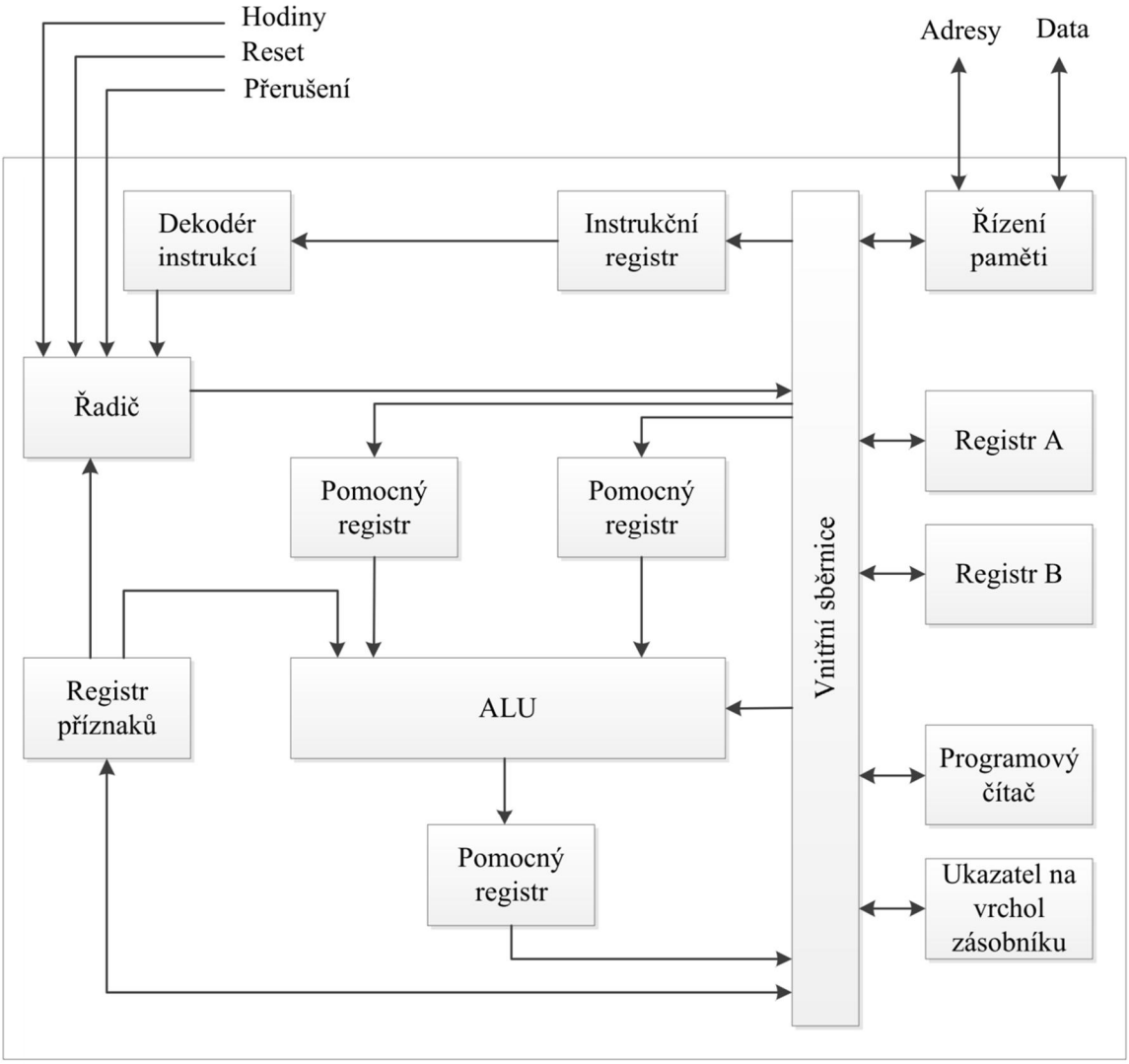
Obr. 1 Blokové schéma mikrořadiče architektury MCS51 od společnosti Silicon Labs.

2.1 CPU - Centrální procesorová jednotka

Centrální procesorová jednotka rozhoduje o výkonu a základních vlastnostech mikrořadiče. Používají se CPU s 4bitovou až 64bitovou architekturou. Čtyřbitové architektury se používají ojediněle ve speciálních aplikacích, jakými jsou např. hodinky. Typicky se v současnosti používají 8 nebo 32bitové architektury.

CPU se skládá z řadiče, instrukčního registru, dekodéru instrukcí, aritmeticko-logické jednotky a programového čítače.

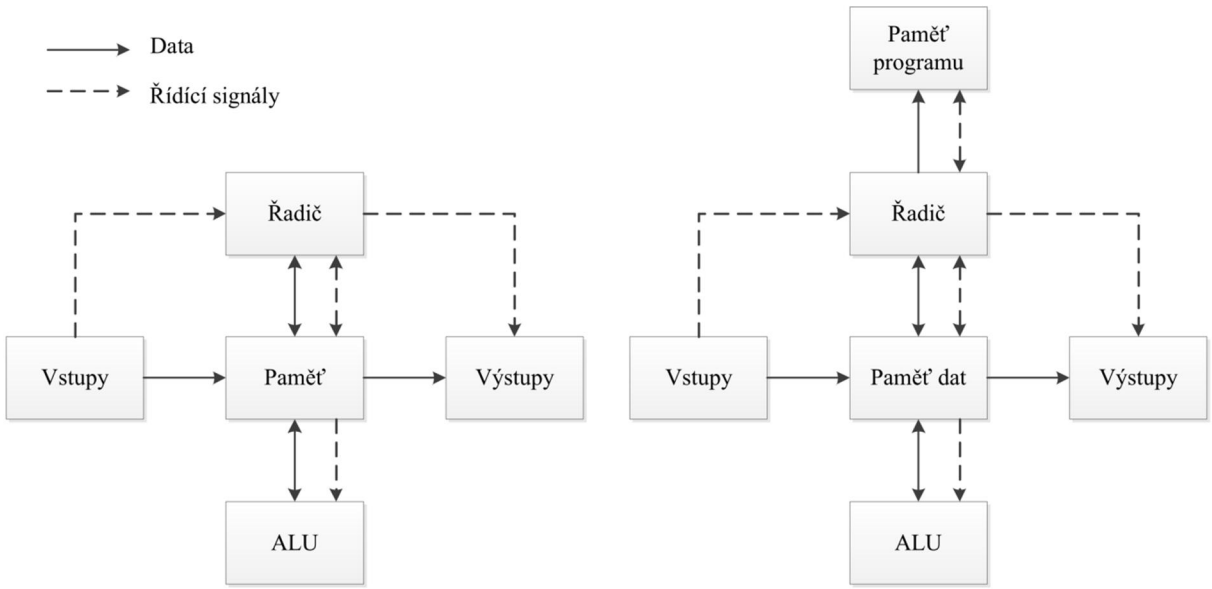
Instrukce je po přečtení z paměti uložena v instrukčním registru a dekodérem instrukcí převedena na logické signály, které zpracovává řadič na jednotlivé řídící signály. Po dokončení zpracování instrukce je načtena další a proces se opakuje. Rozlišují se CPU s kompletní a redukovanou instrukční sadou (tzv. CISC a RISC). Na typu instrukční sady závisí způsob, jakým se bude CPU programovat.



Obr. 2 Blokové schéma jednoduché CPU. [2]

Procesor potřebuje k funkci paměť pro uchování vlastního programu a zpracovávaná data. Von Neumannova architektura používá jediný společný paměťový prostor. Programátor může vytvořit program, který modifikuje sám sebe. Harvardská architektura používá oddělené adresní prostory (sběrnice). Jeden adresní prostor je určen pouze pro program, druhý pouze pro práci s daty. Dochází tak k jednoznačnému oddělení kódu od dat, a proto ke zvýšení bezpečnosti aplikace (obvykle není možné měnit kód programu, čímž se chrání kód před nechtěným přepsáním). Dále tato koncepce přináší dvojnásobný adresní prostor (typicky u CPU s 16bitovou adresní sběrnicí 64kB pro kód a 64kB pro data). Oddělení adresních prostor může dále vést na zvýšení propustnosti, protože jsou kód i data přenášeny paralelně.

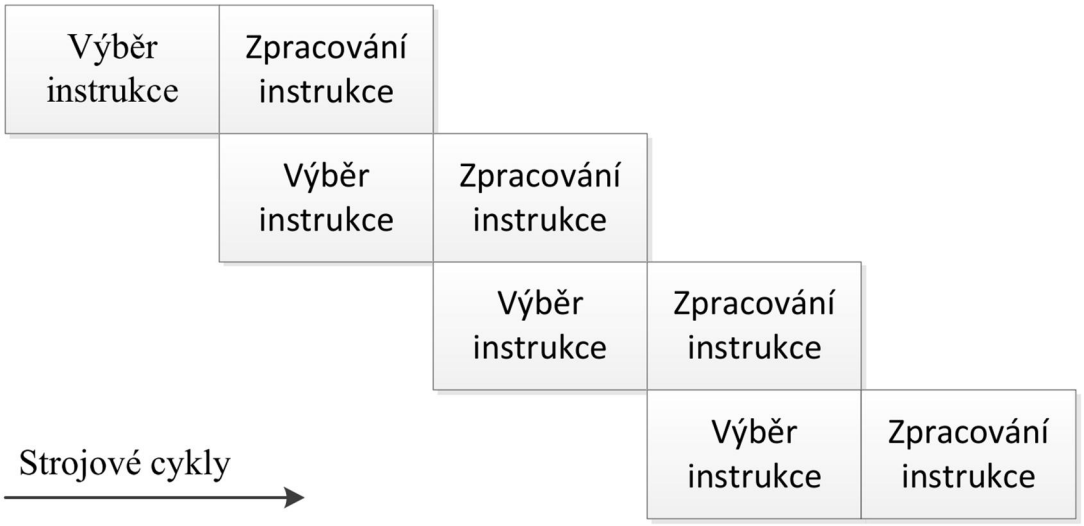
Nelze jednoznačně určit, která architektura je výhodnější. Harvardské architektury se většinou využívá u jednodušších CPU, kde se uplatní uvedené výhody. Von Neumanova architektura se nejčastěji využívá pro složitější CPU (typicky 32bitové), kde se uvedené výhody Harvardské architektury neuplatní. Zástupci Harvardské architektury jsou např. MCS51, AVR, PIC, zástupci von Neumanovy architektury jsou ia32, AMD64, ARM, MIPS.



Obr. 3 Blokové schéma von Neumannovy (nalevo) a Harvardské (napravo) architektury. [3]

Zpracovávaná data jsou ukládána v paměti. Pro vlastní program se používají nejčastěji energeticky nezávislé paměti. Pro data se používají nejčastěji paměti typu RAM, které jsou energeticky závislé.

Pro zvýšení výkonu procesorové části mikrořadiče se využívá často přidání více jader, což je ale náročné na plochu čipu a také jejich vzájemnou komunikaci a spolupráci. Druhý často využívaný způsob je použití zřetězeného zpracování instrukcí (pipelining). To umožňuje procesoru vybírat další instrukci z paměti ve stejný moment, kdy se provádí aktuální instrukce. Díky tomu může procesor ihned po ukončení výkonu aktuální instrukce začít vykonávat další a nemusí čekat na její výběr z paměti. [4] [5]



Obr. 4 Příklad dvoustupňového zřetězení.

2.1.1 Části CPU

2.1.1.1 Aritmeticko-logická jednotka

Aritmeticko-logická jednotka (ALU - Arithmetic Logic Unit) je jednou ze základních částí CPU, ve které jsou zpracovávány všechny aritmetické (sčítání, odčítání…) a logické operace (logický součet, součin, negace…). Podle výsledků operací jsou nastaveny příznakové bity ve stavovém registru (přetečení registru, nulový výsledek, sudá nebo lichá parita), čehož se využívá například při podmíněném větvení programu nebo operacích s většími datovými typy než CPU podporují. Současný trend je použití více ALU v jednom CPU. Tyto ALU pak mají rozdělené funkce. Jednotlivé ALU fungují nezávisle na sobě i na procesoru, a tak může procesor zpracovat v jednom taktu několik operací. [1] [6]

2.1.1.2 Instrukční sada

Instrukční sada je soubor binárně kódovaných instrukcí obsažených přímo v procesoru, které jsou určeny pro manipulaci s daty a řízení běhu programu. Instrukční sada obsahuje instrukce pro manipulaci s pamětí a daty, aritmetické a logické instrukce a instrukce pro podmíněné a nepodmíněné skoky v programu.

Podle typu instrukční sady se procesory dělí na architektury s kompletní instrukční sadou CISC (Complete Instruction Set Computer) a architektury s redukovanou instrukční sadou RISC (Reduced Instruction Set Computer). Instrukční sadu CISC procesorů tvoří vyšší počet instrukcí různé složitosti. Provádění různých instrukcí trvá různý počet strojových cyklů a také jejich kód (délka) není stejný. To vede na složitější strukturu procesoru, ale zároveň i na vyšší výkon takového procesoru. Naopak procesory RISC mají instrukční sadu redukovanou na typicky 30 jednoduchých instrukcí, jejichž vykonání trvá většinou jeden strojový cyklus. Toto zjednodušení vede na jednodušší strukturu procesoru, ale obvykle i na nižší výkon. Pro mikrořadiče je velmi důležitá spotřeba procesoru, která bývá u jednodušších architektur nižší, proto se v této oblasti velmi často používá RISC procesorových jader.

Instrukční sada tvoří nejnižší úroveň, na které se dá procesor programovat. Každá instrukce obsahuje vždy operační znak, který definuje typ operace, a operandy, se kterými bude pracovat. Psaní programu ve strojovém kódu je velmi obtížné a nepohodlné, a proto vznikl a používá se jazyk symbolických adres, ve kterém jsou kódy strojových instrukcí nahrazeny mnemotechnickými zkratkami a názvy operandů zvolenými jmény. Tento jazyk se poté překládá do strojového kódu pomocí překladače Assembleru. [7]

2.1.1.3 Registry

Registry jsou malé paměti, ke kterým může procesor přistupovat mnohem rychleji než k jiným. Jejich velikost se může lišit, ale nejčastěji bývá stejná jako délka slova procesoru nebo její násobek (4, 8, 16, 32, 64 bitů). Na procesoru jich obvykle bývá velmi omezené množství, a proto se nehodí k dlouhodobějšímu uložení dat. Podle obsahu a funkce rozlišujeme registry:

- datové – slouží k uchování zpracovávaných číselných hodnot. U jednodušších procesorů se používá jeden nebo dva tzv. střadačové registry, až několik set.
- adresové – uchovávají zpracovávané adresy
- univerzální – mohou uchovávat data i adresy
- speciální – slouží obvykle jako registry pro řízení periférií.

- kontrolní a stavové - zaznamenávají chod programu (programový čítač, ukazatel zásobníku, stavový registr)

Z pohledu procesoru mohou být registry umístěny v datovém adresovém prostoru nebo v samostatném prostoru. Datové a univerzální registry se převážně považují za prvky, které nemají adresu. U speciálních registrů, které nejsou svázány s architekturou procesoru, ale s integrovanými periferiemi se převážně používá jejich mapování do adresového prostoru CPU, velmi často se speciálním přístupem (samostatný adresní prostor). [8]

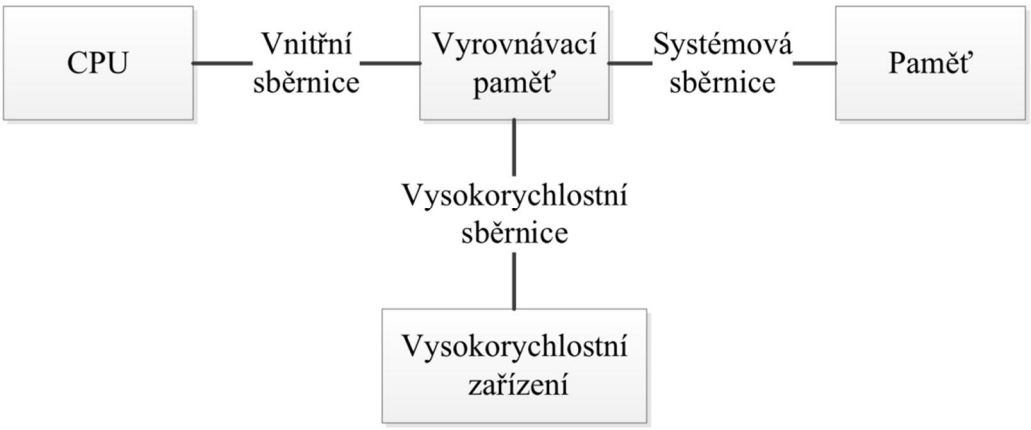
2.1.1.4 Sběrnice a jejich architektura

Sběrnice zabezpečují přenos dat mezi jednotlivými částmi procesoru a také komunikaci s okolím. Jejich vlastnosti přímo ovlivňují rychlost zpracování dat procesorem. Sběrnice fungují na principu paralelního připojení všech zařízení a následném řízení jejich komunikace (určení vysílacího/přijímacího zařízení a vyhrazení času pro jejich komunikaci). Komunikaci nejčastěji řídí procesor. V procesoru se obvykle nachází několik typů sběrnic:

- datová sběrnice – slouží pro přenos dat (paměť - procesor). Pro správnou funkci sběrnice je třeba zajistit, aby vždy vysílalo pouze jedno zařízení, jinak by mohlo dojít k přečtení neurčitého stavu nebo destrukci. Proto jsou připojená zařízení vždy vybavena třístavovým budičem sběrnice, který toto zabezpečuje. Šířka datové sběrnice je velmi důležitým parametrem, který udává, kolik bitů je sběrnice schopna přenést najednou (platí, že čím širší sběrnice, tím větší rychlost přenosu, až do šířky slova procesoru).
- instrukční sběrnice - je komunikační a řídící kanál mezi mikroprocesorem a ostatními bloky. Slouží mimo jiné pro přenos signálů MR/MW (Memory Read/Write řídí časování čtení a zápisu dat), READY nebo jiné signály potřebné pro funkci jednotlivých bloků.
- adresová sběrnice – umožňuje procesoru označit místo čtení nebo zápisu dat. Šířka sběrnice ovlivňuje maximální velikost adresovatelné paměti. Protože procesory pracují s celými bajty, umožňuje adresovat sběrnice o šířce 32bitů maximálně $2^{32} = 4GiB$ a sběrnice o šířce 64bitů $2^{64} = 16EiB$ paměti.

Sběrnice mohou mít různé uspořádání (architekturu). V moderních procesorech se používá architektura se třemi druhy sběrnic:

- vnitřní sběrnice – je dána architekturou mikroprocesoru a probíhá po ní komunikace mezi mikroprocesorem a jeho vyrovnávací pamětí.
- systémová sběrnice – propojuje vyrovnávací paměť mikroprocesoru s operační pamětí a vysokorychlostní sběrnicí.
- vysokorychlostní sběrnice – je přes můstek připojena k systémové a vnitřní sběrnici mikroprocesoru. Připojují se k ní vysokorychlostní zařízení (video, HDD, LAN…) a je také možno připojit rozšiřující sběrnici pro pomalejší zařízení (myš, klávesnice, sériový port). [9] [10]



Obr. 5 Blokové schéma sběrnic v mikrořadiči.

2.1.1.5 Adresní prostory

Adresní prostor je velikost paměti, kterou je procesor schopen přímo adresovat. Rozlišujeme tzv. logický adresní prostor a fyzický adresní prostor. Logický vychází z délky adresy, se kterou procesor pracuje (tzn. procesor s 32bitovou adresou je schopen adresovat $2^{32} = 4GiB$). Fyzický adresní prostor je množství paměti, které je k procesoru fyzicky připojené (z ekonomických a technologických důvodů bývá většinou menší než logický adresní prostor).

2.1.1.6 Koprocesory

Koprocesor obecně označuje samostatný procesor, který je specializován na určitou činnost. FPU (floating-point unit) je speciální typ aritmeticko-logické jednotky, která pracuje s čísly s plovoucí desetinnou čárkou (typické operace jsou sčítání, odčítání, násobení, dělení, mocniny, logaritmy, trigonometrie…). Má vlastní procesor se speciální instrukční sadou a často i několik ALU. Výhodou koprocesoru je, že dokáže pracovat nezávisle na procesoru, který může současně zpracovávat jiné instrukce, a tím významně ušetřit strojový čas. [11]

2.2 Paměti

Paměti jsou zařízení sloužící k uchování binárních dat. V mikrořadičích můžeme najít dva základní typy pamětí: paměti energeticky závislé typu RAM a paměti energeticky nezávislé typu EEPROM, flash-EEPROM, EPROM, PROM, ROM a OTP. Energeticky závislé paměti ztrácí při odpojení napájecího napětí nebo jeho poklesu pod kritickou mez všechny uložené informace. Energeticky nezávislé paměti si dokážou uchovat svůj obsah až 40 let bez napájení a používají se proto nejčastěji k uložení programového kódu.

- RAM (Random Access Memory) – energeticky závislá paměť, která se nejčastěji používá pro uchování zpracovávaných dat a spuštěného programového kódu.
- ROM (Read Only Memory) – paměť pouze pro čtení. Data se zapisují při výrobním procesu a poté je téměř nelze změnit. Dříve byly ekonomicky výhodné, ale dnes se téměř nepoužívají.

- EPROM (Erasable Programmable Read-Only Memory) – programovatelná paměť s možností vymazání obsahu ultrafialovým světlem. Velmi nákladné pouzdro s oknem pro UV záření předurčilo tento typ paměti pouze pro vývojové účely. Dnes je tato technologie již zastaralá a nepoužívá se.
- OTP (One Time Programmable) – programovatelná paměť bez možnosti smazání obsahu.
- EEPROM (Electrically Erasable Programmable Read-Only Memory) – programovatelná a elektricky smazatelná paměť. Má omezený počet zápisů, který se pohybuje od 10^3 do 10^7 podle provedení, a proto se nepoužívá pro zápis dočasných dat. Na rozdíl od paměti EPROM není třeba mazat celý obsah paměti.
- Flash-EEPROM – Ve srovnání s EEPROM je levnější, rychlejší a dosahuje větších kapacit. Mazání probíhá po blocích. [12]

2.2.1 Vyrovnávací paměť

Vyrovnávací paměť (cache) slouží jako zařízení vyrovnávající rychlost komunikace mezi rozdílnými systémy. Funguje na principu rozdělení sebe i paměti na stejně velké bloky a načtení nejčastěji používaných bloků z paměti do cache. V případě požadavku procesoru na čtení paměti se nejprve ověří, jestli není daná adresa v cache. Pokud ano, přečte se z ní, pokud ne, načte se z paměti do ní (na místo nejdéle nepoužitého bloku) a poté přečte.

Vyrovnávací paměť lze realizovat softwarově, nebo hardwarově. Softwarová cache je typická pro komunikaci mezi procesorem a pomalým pevným diskem. Hardwarová realizace je častější u velkých přenosových rychlostí. Typickým příkladem je komunikace procesoru s operační pamětí, která probíhá právě přes hardwarovou cache. Ta umožňuje přenášení dat z operační paměti v blocích, což je u moderních typů rychlejší než náhodné čtení. Procesor pak pracuje plnou rychlostí, když se naplní cache. Z důvodu vysoké ceny cache přímo na procesoru se může používat víceúrovňová. Jedná se o sériově řazené cache označené nejčastěji L1, L2, L3… mezi procesorem a operační pamětí. L1 bývá obvykle přímo na procesoru. Má tedy malou kapacitu a je velmi drahá. L2, L3… bývají externí a tudíž i pomalejší, ale díky nižší ceně mají několikanásobně větší kapacitu. Jedná se tedy o kompromis mezi rychlostí a cenou. [13]

2.3 Periférie

Periférie jsou obvody, které propojují mikrořadič s okolím. Jejich množství a kvalita významně ovlivňují vlastnosti mikrořadiče. Periferních obvodů existuje mnoho a záleží na výrobci, které z nich do mikrořadiče implementuje. Při návrhu může vhodná volba mikrořadiče nejen zjednodušit a zrychlit obslužný program, ale také zjednodušit celé zařízení. Mezi nejčastěji používané periférie patří obvody vstupu a výstupu, čítače, časovače, a AD/DA převodníky. Nepoužívané periférie může programátor vypnout a tím snížit proudový odběr. [1] [8] [14] [15]

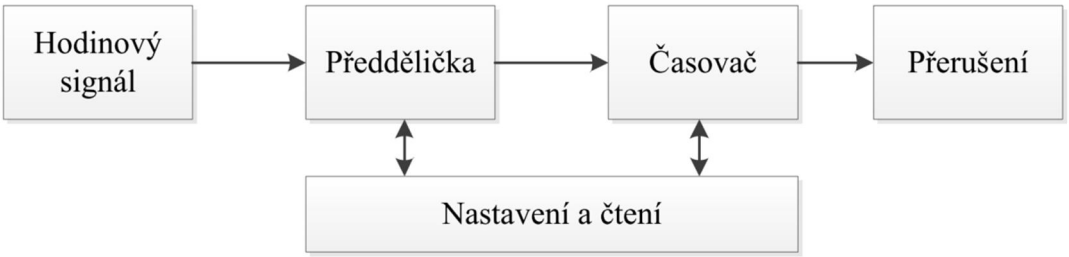
2.3.1 I/O

I/O obvody jsou nejjednodušší a nejčastěji používaná rozhraní pro vstup nebo výstup. Obvykle se jedná o čtyři nebo osm logických vývodů, které tvoří tzv. port. U většiny portů lze nastavit, jestli budou vývody přijímat nebo vysílat a instrukční soubor s nimi může pracovat jednotlivě nebo jako s celkem. Jednotlivé výstupy mohou sloužit pro čtení logických hodnot, ovládání výkonových přepínačů nebo mohou tvořit rozhraní pro sériovou nebo paralelní komunikaci.

2.3.2 Čítač/časovač

Čítač je periferie, která při detekci události na vstupu (náběžná nebo sestupná hrana) inkrementuje hodnotu čítače (registr). Při dosažení svého maxima přeteče a vyvolá přerušení. Počáteční hodnota čítačového registru je nastavitelná.

Časovač pracuje obdobně jako čítač s tím rozdílem, že na jeho vstup přivádíme přesně definovaný kmitočet z generátoru. Díky tomu lze přesně stanovit čas potřebný k přetečení registru časovače a pomocí přerušení reagovat. Časovače i čítače bývají obvykle vybaveny tzv. předděličkou signálu, která umožňuje celočíselně dělit vstupní kmitočet.



Obr. 6 Blokové schéma časovače.

2.3.3 ADC/DAC

V reálném prostředí mají fyzikální veličiny většinou spojitý charakter. Pro jejich měření se používají analogově-digitální převodníky (ADC – Analog-Digital Converter), které spojitou veličinu převedou do digitální podoby zpracovatelné mikrořadičem. Fungují nejčastěji na principu porovnávání měřeného napětí s referenční hodnotou, která je kvantována. Dalším způsobem měření spojitých veličin je měření pomocí RC článků. Měří se doba nabíjení kondenzátoru na určitou úroveň. Odpor nebo kapacita se pak mění v závislosti na měřené veličině, která se dopočítává.

Digitálně-analogový převodník (ADC – Digital-Analog converter) převádí digitální signál na analogovou veličinu.

2.3.4 Sběrnice

Sběrnice slouží jako rozhraní pro přenos dat na základě dodržovaných standardů. Podle šířky datové sběrnice může jít o sériový nebo paralelní přenos. U paralelního přenosu dat se při každém taktu přenese více bitů najednou. Dosahuje teoreticky větších rychlostí přenosu, ale v praxi má několik problémů (mnoho signálových spojů, které výrazně zvyšují náklady a jejichž vzájemná kapacita a indukčnost mohou způsobit chyby přenosu).

V současné době se stále častěji používá sériový přenos, který probíhá po jediném vodiči. Nevýhodou jsou vysoké nároky na dodržování přenosového protokolu. Vzhledem k velmi častému použití sériového přenosu vzniklo mnoho standardů. Pro přenos dat na delší vzdálenosti mezi různými zařízeními se dříve používaly standardy RS232 nebo RS485, dnes je nejznámější sběrnice USB (Universal Serial Bus), která je neustále vyvíjena. Pro komunikaci uvnitř zařízení a v průmyslu se využívají sériové sběrnice CAN, I2C aj.

2.4 Popis činnosti CPU

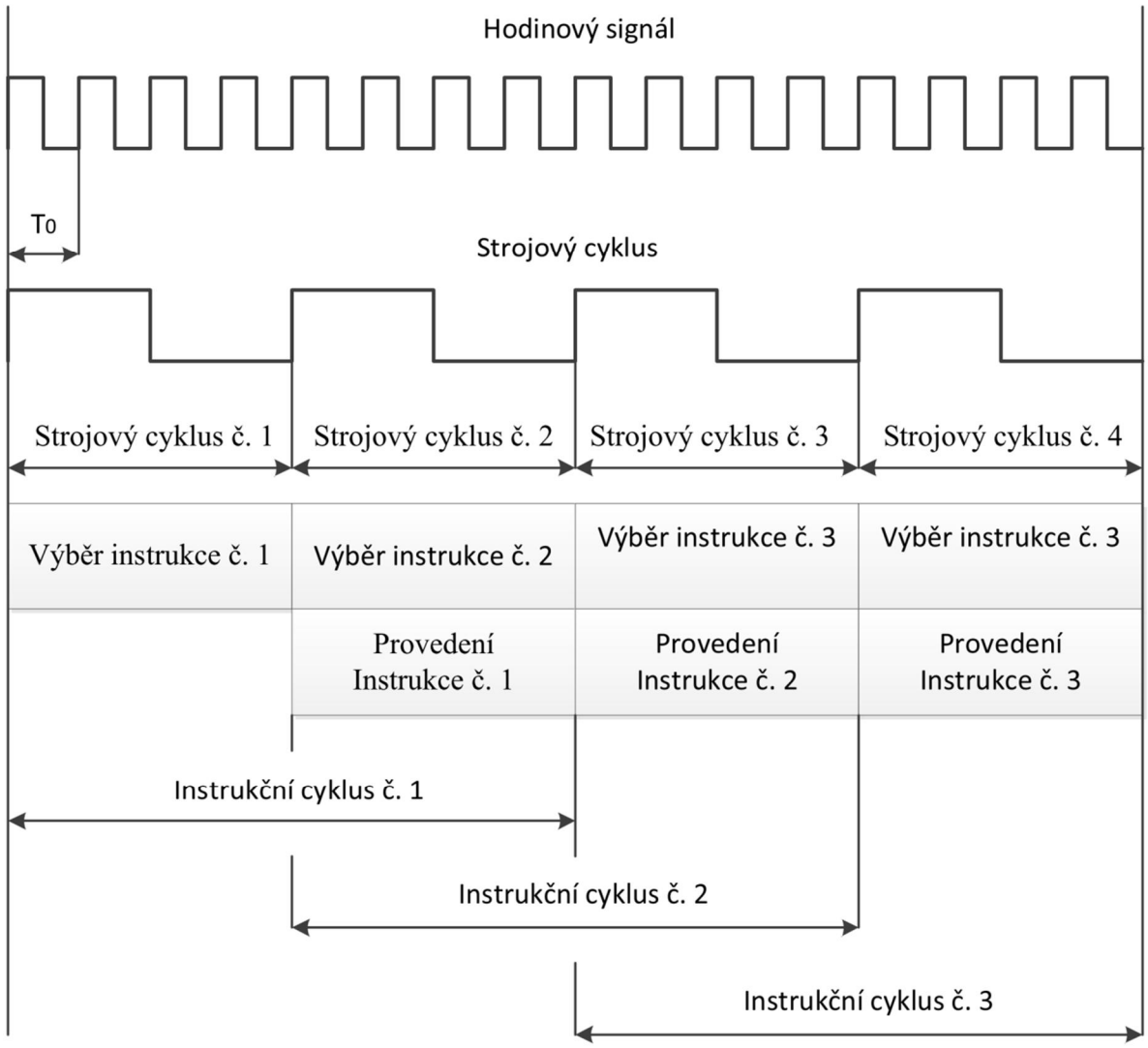
2.4.1 Časování CPU

Procesor potřebuje pro svůj chod zdroj taktovacích (hodinových) impulzů tzv. zdroj synchronizace. Ten vytváří časovou základnu pro provádění všech operací. Může být integrován přímo v procesoru, ale takový nemá dobrou stabilitu a nehodí se pro vykonávání operací, kde je důležité zajistit přesnou délku trvání jedné instrukce. Častěji se využívá externích zdrojů synchronizace:

- Krystal
- Keramický rezonátor
- LC obvod
- RC obvod
- Specializovaný oscilátor

Krystaly a keramické rezonátory dosahují nejlepší stability, ale jsou velmi drahé. RC obvody jsou nejlevnější, ale také méně přesné a závislé na napájecím napětí. LC obvody se používají tam, kde je nutné synchronizovat více zařízení.

Frekvence hodinového signálu má výrazný vliv na spotřebu zařízení. Čím je nižší, tím je menší spotřeba. Některé moderní procesory obsahují zařízení, které umožňuje měnit kmitočet hodinového signálu. Jedná se o smyčku fázového závěsu PLL, která dokáže celočíselně dělit kmitočet oscilátoru a tím zmenšit proudový odběr zařízení. Procesory mají obvykle několik úsporných režimů V těchto režimech se snižuje hodinový kmitočet nebo se úplně zastaví. Z taktovacích impulzů je odvozena délka trvání strojového cyklu.



Obr. 7 Příklad časových závislostí v počítači. [8]

2.4.2 Běh programu

Základní funkcí každého procesoru je schopnost vykonávat program. Ten je reprezentován posloupností instrukcí, zapsaných v paměti. Pro úspěšné vykonání instrukce je zapotřebí čtyř kroků: výběr instrukce, dekódování instrukce, zpracování instrukce a zápis výsledku.

Při spuštění ukáže programový čítač (ukazuje vždy na aktuální pozici v programu) na první instrukci v programu a dojde k jejímu přečtení. Při tom musí rychlý procesor přečíst instrukci z pomalé paměti. Z tohoto důvodu se v moderních procesorech často využívá vyrovnávací paměti a zřetězeného zpracování instrukcí. Po přečtení je instrukce dekódována v dekodéru instrukcí a rozdělena na důležité části (operační kód a operandy). Operační kód definuje operaci a operandy slouží jako její parametry. Následuje vykonání instrukce, nastavení příznaků, zápis výsledků do registru nebo paměti (s možností využití vyrovnávací paměti) a inkrementace programového čítače. Přečte se nová instrukce z adresy, na kterou ukazuje programový čítač a vše se opakuje. [1] [15]

2.4.3 Přerušovací systém

Přerušovací systém umožňuje programu reagovat na asynchronní události při běhu programu. Asynchronní děj je takový, který nebyl v daném okamžiku očekáván (stisknutí tlačítka, příjem signálu po sériové lince). Používá se z důvodu ušetření strojového času, protože periodické testování všech periferií by procesor velmi zatěžovalo. Dříve se objevily procesory bez přerušovacího systému, které spoléhaly na svou rychlost, ale velmi rychle se ale ukázala nesprávnost takového řešení.

Přerušení znamená dočasné zastavení běžícího programu, vykonání obslužného programu a pokračování v zastaveném programu, přičemž běžící program se o přerušení vůbec nedozví. Přerušení může být doplněno systémem priorit, což umožňuje rozhodovat, které přerušení bude zpracováno přednostně. Při zpracování přerušení může dojít k tzv. vnořování (zpracování prioritního přerušení v době, kdy probíhá přerušení). Jednodušší procesory vnořování nemusí vůbec zvládnout, moderní mohou obsluhovat až desítky úrovní přerušení.

Důležitou vlastností přerušení je možnost jeho zakázání. Toho se využívá především jako zábrana proti vnořování. V případě, že je vnořování zakázáno, zařadí se ostatní přerušení do fronty podle jejich priority. [1]

2.4.4 Reset

Vzhledem k tomu, že je mikroprocesor sekvenční logický obvod, závisí jeho chování nejen na strojových instrukcích a jejich argumentech, ale i na předcházejících stavech. Pro správné spuštění programu musí být přesně definován počáteční stav. Ten je výrobci označován jako RESET.

Pro správné provedení RESETu je zapotřebí signál dostatečně dlouhý na to, aby procesor stihl vrátit všechny hodnoty registrů do počátečního stavu. Vnější zdroje RESETu jsou obvykle tlačítka s RC členem, vnitřní zdroje jsou řízeny většinou některým z monitorovacích systémů (monitorování poklesu napájecího napětí, WATCHDOG). RESET se používá nejen ke správnému spuštění (při spouštění se musí ustálit napájecí napětí a frekvence krystalu), ale také k odstranění neurčitých stavů při poklesu napájecího napětí. [1]

3 Programování mikrořadičů v programovacích jazycích Asm a C

Programování je proces vytváření strojového kódu pro procesor. Jelikož je psaní programu ve strojovém kódu velice obtížné, nepřehledné a hledání chyb téměř nemožné, byly vytvořeny programovací jazyky, které se snažily tento proces zjednodušit. Podle míry nahrazení a zjednodušení strojového kódu dělíme jazyky na nejnižší, nižší a vyšší.

3.1 Jazyk symbolických adres

Patří k nejnižším programovacím jazykům. Základem je nahrazení strojového kódu mnemotechnickými zkratkami, které vyjadřují činnost dané instrukce. Zkratky jsou definovány vývojáři architektury procesoru, a proto je kód velmi obtížně přenositelný na jiné architektury (tzv. strojově závislý). Pro jazyk symbolických adres se často používá označení Assembler (JSA je anglicky assembly language).

Překladače jazyka symbolických adres do strojového kódu obvykle obsahují přepínač, který ovládá citlivost na velikost písmen použitých v kódu. Tento přepínač je důležitý zejména v součinnosti s překladači jazyka C, které velikost písmen rozlišují vždy.

3.2 Programovací jazyk C

Programovací jazyk C patří k velmi rozšířeným programovacím jazykům, pro programování mikrořadičů je možno C označit jako nejrozšířenější. Jedná se o jazyk nižší úrovně, který může být libovolně doplňován inline Assemblerem. Zdrojový kód je o poznání čitelnější a přenositelnější na jiné platformy (tzv. strojově nezávislý). Překladače C jsou citlivé na velikost písmen a dodržují standard ISO/IEC 9899:1990, který bývá také nazýván C90. V současné době již existuje standard C99, který zatím není všemi překladači plně podporován. [16] [17]

3.3 Práce s proměnnými

3.3.1 Základní datové typy

Základní datové typy představují reprezentaci dat pro procesor. Z pohledu použití je dělíme na logické, celočíselné a číselné s plovoucí desetinou čárkou. Ostatní datové typy jsou poskládané z těchto základních typů. Velmi záleží na architektuře procesoru, které datové typy podporuje. Nepodporované obvykle bývají emulovány softwarově.

3.3.2 Logické datové typy

Opět záleží na architektuře procesoru, jestli jako logický datový typ používá některý z celočíselných, nebo má speciální datový typ. Logický datový typ je používán pro logické operace a je výsledkem porovnání.

3.3.3 Celočíselné datové typy

Délka	Označení v Asm	Označení v C
1B	BYTE (DB)	char
2B	WORD (DW)	short; na 8bit a 16bit CPU také int
4B	DWORD (DD)	long; na 32bit a 64bit CPUtaké int
8B	QWORD (DQ)	long long

3.3.4 Datové typy s plovoucí desetinou čárkou

Datové typy s plovoucí desetinou čárkou jsou definovány standardem IEEE 754, který většina výrobců mikroprocesorů i překladačů dodržuje.

Délka	Označení v Asm	Označení v C
4B	REAL4 nebo DD	float
8B	REAL8 nebo DQ	double

3.3.5 Ukazatele

Ukazatel představuje proměnnou, uchovávající adresu v paměti na konkrétní datový typ. Pro odlišení klasické proměnné od ukazatele slouží tzv. dereferenční operátor. Pro získání adresy proměnné slouží tzv. referenční operátor. Jednou z velmi užitečných vlastností ukazatelů je, že umožňují volání parametrů funkce odkazem. To znamená, že se funkci nepředává hodnota proměnné, ale její adresa. [18]

3.3.6 Uživatelem definované typy

Uživatelské datové typy jsou rozšířením základních datových typů. Jejich hlavní význam je podpora moderních stylů programování. [19]

3.3.6.1 Výčty

Výčty se používají pro nahrazení číselných hodnot smysluplně pojmenovanými identifikátory a ke sdružení identifikátorů, které spolu souvisí, kdy se můžou navzájem vylučovat nebo doplňovat. Základní datový typ pro výčet je typ odpovídající šířce slova procesoru, ale lze vybrat libovolný celočíselný datový typ podporovaný procesorem. [20]

Definice výčtu v C:

```
enum jmeno_vyctu {
  prvni_prvek = hodnota_1,
  druha_prvek,
  treti_prvek  = hodnota_X
};
```

3.3.6.2 Struktury

Struktury slouží pro sdružování dat různých typů do jednoho datového typu. Je vhodné, aby data ve struktuře spolu souvisela, tím je možné držet data, co k sobě patří pohromadě a snáze s nimi manipulovat. Vzniká tak „balík“ dat, se kterým lze ve většině případů manipulovat jako s jednoduchou proměnou. Vhodné používání struktur vede na tzv. strukturované programování.

Definice struktury v Assembleru:

```
nazev_struktury    STRUCT
  nazev_polozky    datovy_typ
nazev_struktury    ENDS
```

Definice struktury v C:

```
struct nazev_struktury {
  datovy_typ nazev_polozky;
};
```

3.3.7 Pole

Pole je datová struktura, ve které jsou seřazeny za sebou proměnné stejného typu. [20]

Definice pole v Assembleru má syntaxi:

```
nazev_pole   datovy_typ pocet_prvku     DUP(0)
nazev_pole   datovy_typ hodnota1, hodnota2, hodnota3
```

V případě vícerozměrného pole je syntaxe následující:

```
nazev_pole   datovy_typ hodnota11, hodnota12, hodnota13
             datovy_typ hodnota21, hodnota22, hodnota23
             datovy_typ hodnota31, hodnota32, hodnota33
```

Definice pole má v jazyce C syntaxi:

```
datovy_typ jmeno_pole [velikost];
datovy_typ jmeno_pole [velikost] = {hodnoty};
```

Pole mohou být také vícerozměrná. Deklarace a inicializace má pak syntaxi:

```
datovy_typ jmeno_pole [vel1][vel2] = {{hodnoty},{hodnoty}};
```

3.3.8 Proměnné v Assembleru

Základní syntaxe definice proměnné v Assembleru má tento tvar:

```
nazev_promenne datovy_typ inicializace
```

`nazev_promenne` je název proměnné, který splňuje standardy pro tvorbu názvu, `datovy_typ` specifikuje datový typ deklarované proměnné. `inicializace` představuje počáteční hodnotu proměnné. V případě, kdy na počáteční hodnotě nezáleží, lze použít operátor `?`, který nechá proměnnou nezinicializovanou (bude obsahovat to, co bylo v paměti před jejím vytvořením).

Při deklaraci proměnných v Assembleru je možné specifikovat oblast paměti, ve které má být proměnná vytvořena, tj. pomocí direktivy SEGMENT lze specifikovat požadovaný segment paměti, ve kterém má být proměnná alokována. Syntaxe pro použití direktivy segment je následující:

```
typ_pameti   SEGMENT
  promenna1  datový_typ1
  promenna2  datový_typ2
typ_pameti   ENDS
```

Direktiva segment je velice užitečná, neboť umožňuje kontrolu nad využitím paměťového prostoru (ukládání proměnných do napěťově nezávislé paměti, vyčlenění velkého bloku v paměti atp.). [21] [22]

Většina instrukcí dokáže zpracovat data pouze z registrů. Proto se do nich příslušné proměnné musí nejprve přesunout. K tomu slouží příkaz `MOV`:

```
MOV registr, nazev_promenne
```

Příkaz `mov` nemůže přenášet hodnotu z jedné do druhé proměnné, ale vždy z proměnné do registru nebo obráceně. Pokud je třeba přiřadit jedné proměnné hodnotu druhé, musí se použít následující zápis:

```
MOV ax, promenna2 ;přesune hodnotu z promenna2 do registru
MOV promenna1, ax ;přesune hodnotu z registru do promenna1
```

3.3.9 Proměnné v C

Základní syntaxe pro definici proměnné je:

```
datovy_typ   nazev_promenne;
```

Syntaxe pro definici pole je:

```
datovy_typ   nazev_pole[dimenze_pole];
```

Syntaxe pro definici proměnné s inicializací je:

```
datovy_typ jmeno = hodnota;
```

Pro pole a složené datové typy musí být hodnota uzavřena ve složených závorkách a jednotlivé prvky musí být odděleny čárkou. Použitím výrazu { 0 } dojde k vynulování celé proměnné bez ohledu na datový typ. [23] [24] [25]

3.3.9.1 Paměťové třídy

Direktivy paměťových tříd určují v jaké paměti a jak dlouho bude proměnná uložena. [24] V jazyce C jsou rozlišeny třídy:

- auto – definuje lokální proměnnou, která má být uložena do paměti RAM. Existuje od vstupu do bloku a zaniká při výstupu. Při vstupu navíc nabývá náhodné hodnoty. Výchozí třída pro lokální proměnné.
- register – definuje lokální proměnnou, která má být uložena do registru.
- extern – definuje globální proměnnou, která je viditelná pro všechny zdrojové soubory. Používá se při odděleném překladu souborů. Výchozí třída pro globální proměnné.
- static – definuje „lokální“ proměnnou, která existuje od vstupu do bloku a zaniká při výstupu. Na rozdíl od třídy auto si proměnná udržuje hodnotu i mezi voláními funkce. Paměť je alokována po celou dobu běhu programu a uvolněna po skončení. Mimo blok sice není viditelná, ale stále s ní lze manipulovat pomocí ukazatelů.

3.3.9.2 Direktiva const

Direktiva `const` upozorňuje kompilátor, že v definovaném objektu nesmí být po inicializaci změněna hodnota a může být pouze čten. V případě pokusu o změnu hodnoty kompilátor hlásí chybu. `const` se využívá zpravidla při definici formálních parametrů funkce nebo jako zabezpečení důležité hodnoty před nechtěnou změnou. [18]

```
const float pi = 3.1415;
```

3.3.9.3 Direktiva volatile

Během překladu kódu obvykle dochází k jeho optimalizaci, která má za úkol zrychlit běh programu a snížit objem generovaného kódu. Direktiva `volatile` slouží k informování překladače, že nemá danou proměnnou optimalizovat, protože může změnit hodnotu nezávisle na chodu programu (například při obsluze přerušení). [25]

```
void fce (void) {
   volatile int var=0;
   while (var != 255);
}
```

```
void fce (void) {
     while (1);
}
```

Nalevo je ukázka kódu po optimalizaci s direktivou volatile, napravo je ukázka kódu po optimalizaci bez direktivy volatile (proměnná var je překladačem eliminována).

3.4 Funkce

Funkce je část kódu programu, kterou je možné opakovaně používat. Obsahuje část kódu, který může mít parametry (argumenty) a který může vracet výsledek. Funkce lze v kódu programu téměř libovolně používat (volat) a jejich počet volání není omezený. Umožňují tak zjednodušení programu i práce programátora tím, že je možné opakovaně používaný kód (činnost) pouze voláním funkce ať s parametry nebo bez. K dalším výhodám patří také lépe čitelný kód. Nevýhodou může být zvýšená zátěž procesoru při volání a návratu z obsluhy funkce.

Funkci v základu tvoří prolog, tělo funkce a epilog. V prologu je obvykle alokováno místo pro lokální proměnné a schovány hodnoty registrů, jejich obsah funkce bude měnit. Epilog obvykle obsahuje ošetření návratové hodnoty, odstranění lokálních proměnných, obnovení uschovaných registrů a instrukci pro návrat z podprogramu/funkce. Prolog a epilog je závislý na volacích konvencích a na architektuře procesoru.

Volací konvence je způsob, jakým jsou funkci předávány parametry a navraceny hodnoty. Dále určuje, jestli se o obnovení zásobníku postará funkce, nebo následující instrukce.[27]

3.4.1 Definice funkcí v Assembleru

V Assembleru je podpora pro definici funkce pomocí direktivy PROC:

```
jmeno_funkce PROC parametr1 :datovy_typ,
                  parametr2 :datovy_typ
  ;telo funkce
  ret
jmeno_funkce ENDP
```

`jmeno_funkce` i `datovy_typ` se řídí stejnými pravidly jako u proměnných. [28]

3.4.2 Volání funkcí v Assembleru

Pro volání funkcí, které nevrací hodnotu, se používá typicky instrukce CALL, její název se liší podle instrukční sady

```
CALL jmeno_funkce
```

Ta uloží do zásobníku návratovou adresu (adresa další instrukce za funkcí) a provede skok na adresu funkce. Pro návrat se obvykle používá instrukce RET, její název zase záleží na instrukční sadě procesoru.

Při volání funkcí, které vrací hodnotu, je situace trochu složitější. Parametry se funkci mohou předat buď v registrech, na zásobníku nebo jejich kombinací. Zde pak musí programátor zajistit buď naplnění registrů požadovanými hodnotami, nebo uložení těchto hodnot na zásobník. Některé překladače assembleru podporují direktivu INVOKE, která je schopná zajistit podle volacích konvencí předání parametrů funkci. Syntaxe je následující:

```
INVOKE parametr1, parametr2
```

Návratová hodnota funkce je vracena buď ve vybraných registrech, nebo přes zásobník. Direktiva INVOKE se postará i o „úklid“ po volání funkce, v případě použití instrukce pro volání funkce/podprogramu se musí o „úklid“ dle volacích konvencí postarat programátor.

3.4.3 Definice funkcí v C

Syntaxe v C vypadá následovně:

```
datovy_typ jmeno_funkce (datovy_typ jmeno_parametru) {
  //tělo funkce
  return hodnota;
}
```

Před jménem funkce je třeba nejprve definovat, jakého datového typu bude její návratová hodnota (viz tabulka datových typů u proměnných v C). V případě, že funkce hodnotu nevrací, napíše se místo datového typu klíčové slovo `void`. Za jménem funkce následuje v závorce výpis všech parametrů s datovým typem oddělených čárkou. Provádění funkce je ukončeno, když kód dorazí k uzavírací složené závorce, nebo provedením příkazu return.

```
void nazev_funkce (int i) {
  //tělo funkce
}
```

V případě, že funkce vrací návratovou hodnotu, musí obsahovat příkaz return s návratovou hodnotou. [28]

```
int nazev_funkce (int a, int b, int c) {
  //tělo funkce
  return a + b + c;
}
```

Kromě definice funkce se možné deklarovat její prototyp. Tento případ se používá, pokud je volaná funkce definována až za místem prvního volání, nebo je definována v jiném souboru. Syntaxe prototypu je následující:

```
datovy_typ nazev_funkce (datove_typy_parametru);
```

Jde tedy o první řádek definice funkce s tím rozdílem, že není povinné vypsat jména parametrů, ale stačí pouze datové typy.

3.4.4 Volání funkce v C

Syntaxe volání funkce:

```
nazev_funkce (parametry);
```

Při volání funkce lze její návratovou hodnotu nechat zapsat do konkrétní proměnné:

```
vysledek = nazev_funkce (param1, param2);
```

3.4.4.1 Direktiva inline

Pokud je tělo funkce velmi krátké (režie volání je delší než tělo funkce), je možné použít direktivu `inline` která překladači sdělí, aby nedefinoval funkci, ale přímo tělo funkce vložil do místa volání. Syntaxe je následující:

```
inline int nazev_funkce (int param) {
  //tělo funkce
}
```

Použitím inline funkcí lze zrychlit běh programu, velkým problémem je, že to jestli bude inline funkce řešena vložením nebo voláním záleží většinou na překladači a jeho optimalizačním nastavení.

3.5 Obsluha přerušení

Obsluhy přerušení jsou speciálním typem funkcí, které se od standardních funkcí liší tak, že jsou volány asynchronně řadičem přerušení, mívají předepsány adresy, kde se musí nacházet, nemívají parametry, většinou jsou prováděny na jiné úrovni než standardní funkce a ukončují se instrukcí návratu z přerušení (obvykle IRET nebo RETI).

Z pohledu programování v assembleru jsou to funkce, které jsou ukončeny instrukcí návrat z přerušení a mívají předepsanou adresu, na které se musí nacházet.

V jazyce C je pro definici přerušení používáno klíčové slovo interrupt, které podle implementace může obsahovat adresu nebo číslo obsluhy přerušení.

```
interrupt (1) void obsluha_preruseni (void) {
        return;
}
```

V případě přerušení dojde k uložení všech hodnot z registrů, které by mohly být použity obsluhou přerušení, do zásobníku. Po návratu z obsluhy přerušení jsou obnoveny původní hodnoty. U některých procesorů lze nastavit obsluze přerušení novou banku registru direktivou `_using (bank)`. Tím lze snížit odezvu přerušení, neboť není potřeba obsloužit zápis a čtení dat ze zásobníku. Příklad syntaxe:

```
interrupt (1) using (2) void obsluha_preruseni (void) {}
```

Při volání jiné funkce z funkce obsluhy přerušení je potřeba, aby využívala stejnou banku registru. Příklad syntaxe:

```
using (2) int jina_funkce (void) {}
```

Pro obsluhu přerušení jsou doporučené paměťové modely small, large nebo aux. V případě použití modelu reentrant může dojít k problémům s přepínáním bank. Je proto lepší nevyužívat direktivy `_using` a nechat na překladači jejich ukládání a přepínání.

3.6 Přístup k paměti

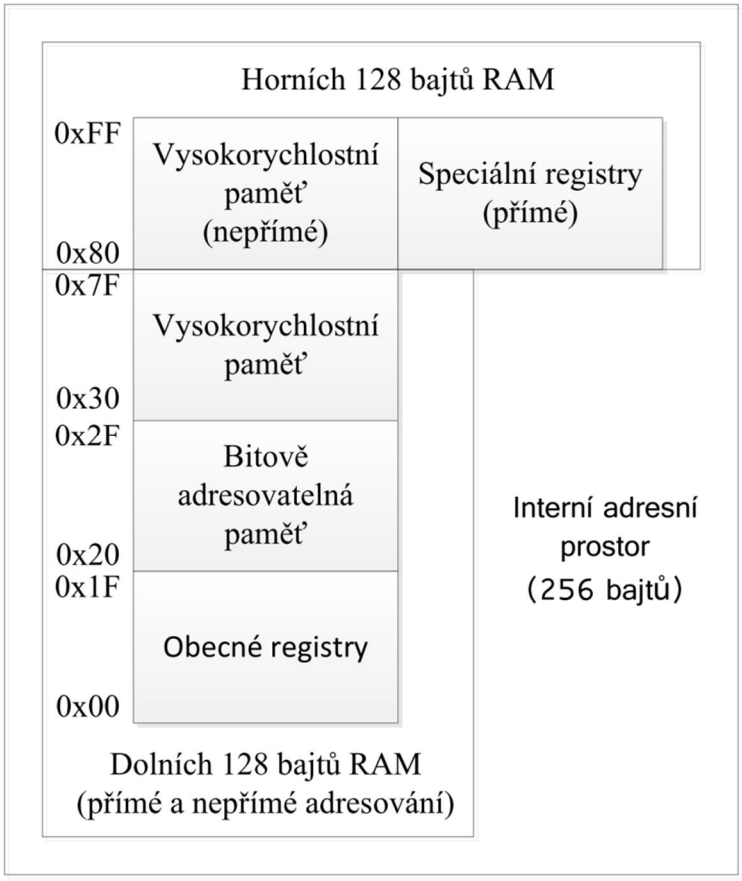
Před návrhem obslužného programu mikroprocesoru musí programátor nejprve dokonale zvládnout práci s jeho pamětí. Špatně zvolený paměťový model nebo typ úložiště může mít za následek výrazné zpomalení nebo úplnou nefunkčnost programu. [15] [30]

3.6.1 Typy úložišť

Jednotlivé architektury mají více adresovacích způsobů, které buď vedou na efektivnější (rychlejší) přístup k paměťové oblasti, nebo zpřístupňují paměťové oblasti jinak nedostupné. Překladač musí proto podporovat rozlišení těchto adresovacích módů.

Jako příklad budou dále popsány direktivy adresních prostorů architektury MCS51, která která z tohoto pohledu patří mezi nejsložitější a nejkomplikovanější. MCS51 je postavena na Harvardské architektuře. Má definovaný adresní prostor pro kód „rom“ (64KiB), pro interní RAM (256B) a externí RAM (64KiB). Interní RAM je rozdělena na oblasti „data“ adresy 0x00-0x7F, pouze nepřímo adresovatelná data „idata“ adresy 0x80-0xFF, speciální funkční registry „sfr“ adresy 0x80-0xFF pouze s přímým přístupem a bitově adresovatelnou oblast „bdata“ adresy 0x20-0x2F. K externí paměti lze přistupovat jako k celku „xdata“, nebo po 256B stránkách „pdata“.

Správné používání vhodných pamětí vede na kratší a efektivnější, proto se v programu musí řešit problematika adresních prostorů



Obr. 8 Blokové schéma interní paměti RAM architektury MCS51.

Programátor může u proměnných řídit umístnění v adresním prostoru zvolením globálního paměťového modelu, lokálního paměťového modelu, nebo specifikací pomocí direktiv adresních prostorů (data, bdat, idat, pdat, xdat, rom).

Typ paměti	Popis
_data / data	Přímo adresovatelná interní RAM
_bdat	Přímo adresovatelná interní RAM s přístupem k jednotlivým bitům
_idat / idat	Nepřímo adresovatelná interní RAM
_pdat / pdat	Externí RAM do 256B na jedné straně
_xdat / xdat	Externí RAM
_rom / rom	Interní / externí ROM

Přistupování k interní paměti je výrazně rychlejší než k externí. Proto je výhodné přiřazovat nejčastěji využívané proměnné do interních a velké, málo používané proměnné do externích pamětí. Příklady deklarací proměnných:

```
_data  char  c;
_xdat  int   array[10][4];
_idat  long  l;
```

3.6.2 Paměťové modely

Paměťové modely definují různá výchozí úložiště pro automatické proměnné, deklarace bez určení typu úložiště a předávané parametry.

Paměťový model	Parametry / automatické proměnné	Ostatní proměnné	Maximální velikost RAM	Výchozí typ úložiště
small	_data	_data	128B	_data
auxpage	_pdat	_pdat	256B	_pdat
large	_xdat	_xdat	64KB	_xdat
reentrant	virtuální zásobník v _xdat	_xdat	64KB	_xdat

Model small se nejvíce hodí pro aplikace, kde je kladen důraz na vysokou rychlost, kdežto model reentrant umožňuje předávání parametrů funkcí přes virtuální zásobník v externí RAM a hodí se spíše pro rozsáhlé a pomalejší aplikace.

3.6.3 Smíšený paměťový model

Vzhledem k tomu, že jednotlivé modely mají své výhody a nevýhody, lze je pro každou funkci specifikovat zvlášť.

Klíčové slovo	Výchozí model
_small	small model
_aux	auxpage model
_large	large model
_reentrant	reentrant model

Příklad:

```
// Výchozí model je large
int rozdil (int prvni, int druhy) {          // large model
  return prvni - druhy;
}

int _small soucet(int prvni,int druhy) {     //small model
  return prvni + druhy;
}
```

3.6.4 Direktiva _at

Direktiva `_at` slouží k umístění proměnné na konkrétní adresu. Syntaxe pro definici proměnné Z v externí paměti na adrese 0x2000:

```
_xdat unsigned char Z _at(0x2000);
```

Při umisťování proměnných pomocí direktivy `_at` platí mnohá omezení a vzniká nebezpečí vzájemného překrytí více proměnných. Protože je v mnoha případech překrytí žádoucí, nekontrolují překladače tento stav.

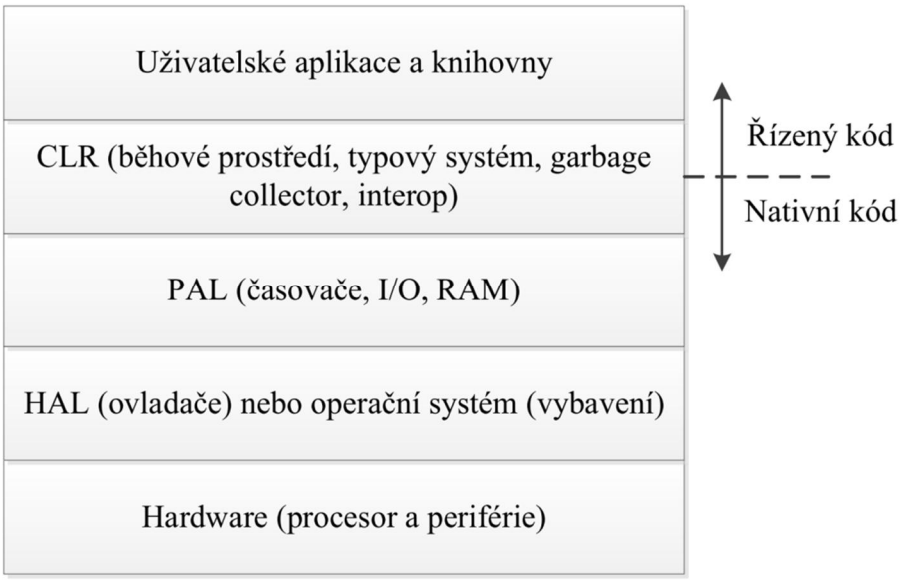
4 Programování mikrořadičů pomocí nástrojů vyšší úrovně

4.1 Microsoft .NET Micro Framework

Microsoft .NET Micro Framework je výkonné a nenáročné .NET prostředí používané pro spuštění řízeného kódu na zařízeních s omezenými hardware prostředky (mikrořadiče atp.). Umožňuje programování v jazycích vyšší úrovně, aktuálně je podporován jazyk C#. Dále je v prostředí MS Visual Studio podporováno použití emulátoru hardware pro rychlý návrh a odladění kódu.

MS .NET Micro Framework je vyvíjen cíleně pro 32bitové mikrořadiče s větším množstvím paměti RAM. Aktuální verze .NET Micro Framework 4.1 obsahuje základní běhové prostředí Tiny Common Language Runtime Framework (Tiny CLR), které potřebuje ke svému běhu minimálně 300 KiB paměti RAM. Proto lze prostředí .NET Micro Framework používat pouze na poměrně výkonných a paměťově vybavených mikrořadičích. V současnosti je zatím podporována pouze architektura ARM, která se v poslední době stává jednou z nejrozšířenějších.

Velkou výhodou .NET Micro Framework je jeho nezávislost na hardware. O kód, který je specifický pro hardware se stará tzv. Hardware Abstraction Layer (HAL) a Platform Abstraction layer (PAL). Procesu vytváření HAL a PAL se říká portování a stará se o něj nejčastěji výrobce hardware. Nevýhodou tohoto přístupu vyšší režie přístup k periferiím a tím k částečné ztrátě výkonu aplikace.



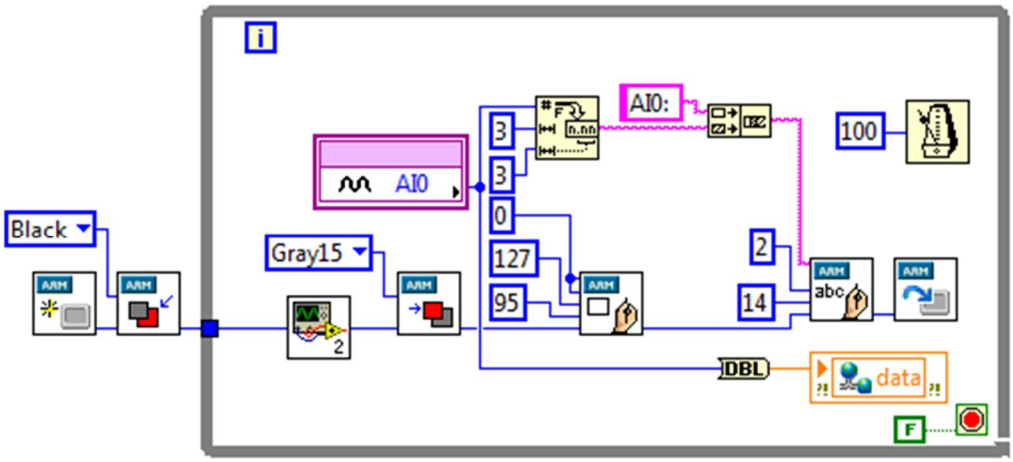
Obr. 9 Schéma architektury Microsoft .NET Micro Framework.

Po napsání kódu dojde automaticky k jeho kompilaci a odeslání do mikrořadiče. Veškerý chod lze monitorovat a zobrazovat jak na PC, tak i na vývojové desce. Další významnou výhodou tohoto prostředí je použití moderního jazyka vyšší úrovně C#, který umožňuje při dodržení jednoduchých pravidel poměrně dobrou přenositelnost zdrojových kódu mezi podporovanými mikrořadiči. [31] [32]

4.2 LabVIEW

LabVIEW je grafické programovací prostředí specializované především na oblast měření a řízení (sběr, analýza a prezentace dat, komunikace, řízení), které slouží k vývoji systémů pomocí blokových diagramů za pomoci grafického programovacího jazyka G (Graphic Language). Takto naprogramované bloky jsou v případě mikrořadičů převedeny na zdrojové soubory jazyka C a tyto jsou následně s kódem knihoven kompilovány překladačem jazyka C do výsledné binární aplikace. Velkou výhodou LabVIEW je velmi dobrá podpora pro real-time aplikace a kompatibilita na úrovni podporovaných knihoven. Dále je to podpora pro sdílení dat (proměnných) po síti – tzv. „sdílená síťová proměnná“, která výrazně zjednodušuje výměnu dat mezi aplikacemi.

Embedded Module for ARM Microcontrollers je modul, který přináší možnost grafického programování v prostředí LabVIEW i pro 32bitové mikrořadiče ARM. Pro maximálně snadný vývoj jsou k dispozici vývojové desky s integrovaným mikrořadičem ARM, ovládacími prvky, displejem a množstvím dalšího vybavení, které se liší v závislosti na typu desky.



Obr. 10 Snímek pracovní plochy v Labview při programování jednoduché aplikace, která s frekvencí 10Hz čte hodnotu z analogového vstupu, zobrazuje ji na displeji a odesílá ji přes ethernetové rozhraní.

Na obr. 10 je příklad jednoduché aplikace pro práci s displejem, analogovým vstupem a sdílenou síťovou proměnnou, která zpřístupňuje změřená data na síti. Při požadavku na spuštění aplikace zajistí LabVIEW její překlad, zavede kód aplikace do mikrořadiče a spustí ho. Veškerý chod aplikace lze monitorovat a zobrazovat jak na PC, tak i na vývojové desce. [33] [34]

5 Možnosti přenositelnosti kódu mezi různými architekturami mikrořadičů

U aplikací pro mikrořadiče je omezujícím faktorem přenositelnost kódu z jednoho typu mikrořadiče na jiný. K potřebě „portace“ kódu na jiný typ mikrořadiče vede několik faktorů. Prvním z nich je ten, že výrobce přestane konkrétní typ mikrořadiče dočasně vyrábět, nebo dokonce výrobce zastaví výrobu kompletně. Dalším faktorem je „růst“ aplikace, kdy se vývojáři dostanou mimo parametry mikrořadiče a musí dojít k přechodu na jiný typ nebo architekturu.

Zde se začne projevovat nekompatibilita mikrořadičů nejen při přechodu na jinou architekturu, ale i mezi jednotlivými typy. Naráží se zde na problém jiných typů periférií, odlišných způsobů jejich ovládání atd. Přechod obvykle bývá velmi drahý a problematický, a proto se různí výrobci mikrořadičů i vývojových nástrojů snaží tento přechod usnadnit.

Při programování v Assembleru má programátor na jednu stranu absolutní kontrolu nad programem, ale takový kód je velice strojově závislý a přenositelný v rámci jedné rodiny mikrořadičů (v nejlepším případě vývojové větve). Při použití Assembleru musí programátor opakovaně řešit stejné nebo podobné problémy a vývoj aplikací probíhá velmi pomalu. Jedná se i o tak jednoduché záležitosti, jako jsou matematické operace s reálnými čísly na procesoru bez koprocesoru, předávání parametrů funkcím, složitější větvení programu a další. V neposlední řadě musí být programátor důkladně seznámen se strukturou mikrořadiče a periférií. Za další nevýhody lze považovat horší čitelnost a orientaci ve zdrojových kódech tak náchylnost ke vzniku elementárních chyb v kódu. Jediným důvodem pro tvorbu části kódu v Assembleru jsou časově kritické části, které nezvládne překladač vyšších jazyků přeložit dostatečně efektivně.

Programovací jazyk C patří do skupiny nižších programovacích jazyků. V případě dodržení programátorských zásad je kód velice snadno čitelný a v případě důsledného oddělení kódu programu od ovladačů je i dobře přenositelný. Přenositelnost je také podmíněna existencí knihoven k cílové architektuře. Programátor má stále velmi dobrou kontrolu nad chodem programu s výhodou výrazně vyšší rychlosti vývoje aplikací a kontrolou elementárních chyb překladačem. Z pohledu přenositelnosti je ale nutné oddělit kód aplikace na kód na hardware závislý a nezávislý. Dále je nutné se vypořádat s nekompatibilitou jednotlivých překladačů jazyka C, jak na úrovni rozšíření odvozených od architektury (direktivy pro paměťové modely, pevné adresy a přerušení), nedodržením standardů jazyka C, tak absencí různých knihoven funkcí a další problematiky.

Využití vývojového prostředí Microsoft .NET Micro Framework přináší výhody spojené s programováním ve vyšším jazyce C#, který poskytuje vysokou míru abstrakce a je snadno čitelný. MS .NET Micro Framework poskytuje komunikační mezivrstvu mezi mikrořadičem a programem, která zajišťuje dobrou přenositelnost mezi různými mikrořadiči od různých výrobců (mikrořadič musí být podporován), avšak za cenu zvýšených hardwarových nároků.

Grafické programovací prostředí LabVIEW patří také mezi vyšší programovací jazyky s vysokou mírou abstrakce. Přenositelnost mezi podporovanými mikrořadiči je téměř zaručena. S použitím vývojových desek se jedná o ideální prostředí pro tvorbu měřících, komunikačních a zobrazovacích zařízení.

Obecně lze říci, že čím je vyšší míra abstrakce od strojového kódu, tím snadnější je přenos kódu mezi jednotlivými rodinami a architekturami mikrořadičů v případě, že existují knihovny ovladačů, které dané typy podporují.

6 Závěr

Cílem první části této bakalářské práce bylo popsat strukturu mikrořadičů a jejich jednotlivých částí jako jsou registry, instrukční sada, sběrnice, adresní prostory, koprocesory, paměti, cache, periférie a dále s přerušovacím systémem. Výsledkem je přehledné seznámení se strukturou a činností mikrořadičů z pohledu programátora.

Další části této práce jsou již věnovány programování mikrořadičů a jeho zvláštnostem. Nejprve je uvedena problematika vytváření strojového kódu a seznámení s nižšími programovacími jazyky Assemblerem a C. Poté následuje seznámení s jejich základními a uživatelsky definovanými datovými typy, definicemi proměnných a funkcí, obsluhou přerušení a prací s pamětí.

Dále se v práci nachází část o programování mikrořadičů ve vyšších programovacích jazycích, konkrétně v jazycích C# a G v prostředích Visual Studio a LabVIEW. Jedná se o jazyky s vysokou mírou abstrakce, které se velmi dobře hodí k vytváření dobře čitelného a přenositelného kódu, ale jsou velmi závislé na existenci knihoven pro podporované typy mikrořadičů. Zejména LabVIEW s ARM modulem se ukázalo jako velice výkonné a jednoduché prostředí pro programování pokročilých měřících, řídících a komunikačních aplikací.

Poslední část se věnuje problematice přenositelnosti kódu mezi rodinami a architekturami mikrořadičů, což je velice důležité zejména při nutnosti „portovat“ kód ať už kvůli „růstu“ programu nebo jiným důvodům. Dobré přenositelnosti lze dosáhnout v jazyce C v případě pečlivého oddělení kódu závislého a nezávislého na hardware. Výbornou přenositelnost pak nabízí programovací jazyky vyšší úrovně, ale pouze v rámci podporovaných zařízení.

Psaní této bakalářské práce pro mě bylo velice náročné, protože jsem neměl žádné předchozí vzdělání v oblasti mikrořadičů a programování. Přesto jsem velice rád, že jsem si vybral právě toto zadání a podařilo se mi ho dokončit. V průběhu psaní jsem se velice dobře seznámil s různými architekturami mikrořadičů např. ARM, MCS51… a získal základy programování v assembleru, jazyce C a všeobecný přehled o možnostech jejich programování v jazycích vyšší úrovně. Dále jsem měl možnost být osobně při vývoji několika aplikací pro mikrořadiče, čímž se pro mě tato rešeršní práce stala do určité míry také prací praktickou. Tato práce mě také natolik inspirovala, že jsem si v průběhu psaní vymyslel námět hned na několik projektů s mikrořadiči, které bych chtěl do budoucna uskutečnit.

Seznam použité literatury

1. **Pechal, Stanislav.** *Monolitické mikropočítače.* 3. Praha : BEN - technická literatura, 1990. 80-86056-30-9.
2. **Tišnovský, Pavel.** Princip činnosti řadiče v mikroprocesorech. *root.cz.* [Online] 26. 04 2011. [Citace: 19. 05 2011.] http://www.root.cz/clanky/princip-cinnosti-radice-v-mikroprocesorech/.
3. **Corri.** Architektura počítačů. [Online] 25. 06 2009. [Citace: 19. 05 2011.] http://corri.blog.cz/0906/architektura-pocitacu.
4. **Martin, Trevor.** *ARM7 CPU core.* [PDF dokument] Coventry : Hitex (UK) Ltd., 01. 02 2007. 0-9549988 6.
5. **Wikipedia contributors.** Instruction pipeline. *Wikipedia.* [Online] 19. 05 2011. [Citace: 19. 05 2011.] http://en.wikipedia.org/w/index.php?title=Instruction_pipeline&oldid=425081992.
6. **Nebojsa, Matic.** PIC Microcontrollers for beginners. *Ioan Slavici University Timisoara.* [Online] 2003. [Citace: 19. 05 2011.] http://www.islavici.ro/cursuri/conducere%20sist%20cu%20calculatorul/PICbook/0_Uvod.htm.
7. **Alexandridis, N.** *Computer Systems Concepts and Processor Architecture.*
8. **Olivka, Petr.** *Monolitické počítače.* Ostrava : katedra informatiky FEI VŠB-TU, 2010.
9. **Kotásek, Z.** Texty přednášek v elektronické podobě k předmětu IPZ. *Fakulta Informačních Technologií.* [Online] 05. 12 2008. [Citace: 18. 05 2011.] http://www.fit.vutbr.cz/study/courses/IPZ/public/.
10. **Olivka, Petr.** *Komunikace s periferiemi.* Ostrava : Katedra informatiky FEI VŠB-TU, 2011. Studijní materiál pro předmět Architektury počítačů.
11. **Wikipedia contributors.** Coprocessor. [Online] Wikipedia, The Free Encyclopedia, 05. 05 2011. [Citace: 19. 05 2011.] http://en.wikipedia.org/w/index.php?title=Coprocessor&oldid=427596137.
12. **Tišňovský, Pavel.** Nevolatilní paměti. *root.cz.* [Online] 18. 09 2008. [Citace: 19. 05 2011.] http://www.root.cz/clanky/nevolatilni-pameti/.
13. —. Vyrovnávací paměti cache. *root.cz.* [Online] 03. 07 2008. [Citace: 19. 05 2011.] http://www.root.cz/clanky/vyrovnavaci-pameti-cache/.
14. **Microchip Technology Inc.** *PIC18F2455/2550/4455/4550 datasheet.* [PDF] 2009. DS39632E.
15. *C8051F50x/F51x.* **Silicon Laboratories Inc.** Austin : autor neznámý, 2011, Sv. 1.2. C8051F500/1/2/3/4/5/6/7/8/9-F510/1.
16. **Wikipedia contributors.** C (programming language). *Wikipedia.* [Online] Wikipedia, The Free Encyclopedia. , 16. 05 2011. [Citace: 19. 05 2011.]
17. **Bryant, Randal a O'Hallaron, David.** *Computer systems - A programmer's perspective.* New Jersey : Prentice Hall, 2003. 0-13-034074-x.
18. **Herout, Pavel.** *Učebnice jazyka C.* III. Příbram : KOPP, 1994. 80-85828-21-9.
19. Stručný úvod do programovacího jazyka C. *Fakulta informatiky.* [Online] Masarykova univerzita Brno, 2005. [Citace: 24. 05 2011.] http://www.fi.muni.cz/usr/jkucera/pb071/uvodc.htm.
20. **Fabián, David.** Úvod do C. *Poznámky k jazyku C.* [Online] 2010. [Citace: 19. 05 2011.] http://kmlinux.fjfi.cvut.cz/~fabiadav/cecko/poznamky-k-jazyku-c/struktury-a-vycty.
21. **Joehanes, Roby.** Variables in Assembly. *PC Assembly Tutorial.* [Online] 15. 05 2011. [Citace: 18. 05 2011.] http://www.etfos.hr/~jbognar/pozadine/temp/asembly%20tutorial/x86asm/asml1002.html.
22. **Microsoft.** Directives reference. *MSDN.* [Online] [Citace: 20. 05 2011.]

http://msdn.microsoft.com/en-us/library/8t163bt0.aspx.
23. C variables. *C programming.* [Online] 2007. [Citace: 19. 05 2011.] http://cprogramminglanguage.net/c-variables.aspx.
24. **Wikipedia contributors.** C variable types and declarations. *Wikipedia.* [Online] 17. 05 2011. [Citace: 20. 05 2011.] http://en.wikipedia.org/w/index.php?title=C_variable_types_and_declarations&oldid=429568663.
25. **Microsoft.** Declarators and variable declarations. *MSDN.* [Online] [Citace: 20. 05 2011.] http://msdn.microsoft.com/en-us/library/e5ace6tf.aspx.
26. **Wikipedia contributors.** Volatile variable. *Wikipedia.* [Online] 03. 05 2011. [Citace: 20. 05 2011.] http://en.wikipedia.org/w/index.php?title=Volatile_variable&oldid=427181520.
27. **Roupec, Jan.** *Programování v Assembleru.* [PDF] Brno : Ústav automatizace a informatiky FSI VUT, 2004.
28. **Zezula, Ladislav.** Procedury a funkce. *Výuka Assembleru.* [Online] 2004. [Citace: 19. 05 2011.] http://www.zezula.net/cz/teach/assembler_07.html.
29. **Microsoft.** Function declarations. *MSDN.* [Online] 2011. [Citace: 19. 05 2011.] http://msdn.microsoft.com/en-us/library/9w92dxk3.aspx.
30. **Tasking.** *C Cross-Compiler user's manual.* [PDF] 2006. MA008-002-00-00.
31. **Kučera, Jan.** *Microsoft .NET Micro Framework Tools & Resources.* [Online] [Citace: 25. 05 2011.] http://informatix.miloush.net/microframework/Tutorials/WhatIs.aspx.
32. **Thompson, Donald a Miller, Colin.** *Introducing the .NET Micro Framework.* [Document] místo neznámé : Microsoft, 2007.
33. **National Instruments.** NI LabVIEW Embedded Module for ARM Microcontrollers. *NI LabVIEW.* [Online] National Instruments, 2011. [Citace: 25. 05 2011.] http://www.ni.com/labview/arm/.
34. —. Write String to Display VI. *LabVIEW for the ARM microcontroller LM3S8962.* [Online] National Instruments, 01. 09 2009. [Citace: 25. 05 2011.] https://decibel.ni.com/content/docs/DOC-6373.