TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: M 2612 – Elektrotechnika a informatika
Studijní obor: 3902T005 – Automatické řízení a inženýrská informatika

Interpret assembleru jako didaktická pomůcka pro předmět Číslicové počítače

Assembler interpret as a didactic tool for school subject Numeric computers

Diplomová práce

Autor: **Jiří Pešek**
Vedoucí práce: Ing. Martin Vlasák
Konzultant: Ing. Tomáš Martinec

V Liberci 6. 5. 2006

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé diplomové práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Diplomovou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum

Podpis

Pešek Jiří DP-2006 Ved. DP: Ing. Martin Vlasák

Interpret assembleru jako didaktická pomůcka pro předmět Číslicové počítače

Anotace:

Cílem diplomové práce je naprogramovat kompilátor a simulátor zjednodušeného assembleru. Pomocí těchto dvou částí byl vytvořen simulátor. Ten bude sloužit studentům prvního ročníku Technické univerzity v Liberci při výuce předmětu Číslicové počítače.

Simulátor bude studentům názorně ukazovat chování teoretického mikroprocesoru, který v sobě sdružuje nejlepší vlastnosti procesorů řady 51 a 86. Pro co možná největší názornost prostředí obsahuje grafické znázornění externích zařízení, které jsou připojeny k portům mikroprocesoru. Jedná se o 8 nastavitelných tlačítek, 16 led diod, generátor signálu a sedmi-segmentový displej.

Jako programovací jazyk pro vývoj programu bylo použito prostředí Microsoft Visual C++ verze 6.

Abstrakt:

The aim of the diploma thesis is pre-set compiler and simulator of simplified assembler. By the help of these two parts was created simulator. That will serve students in first year on Technical university in Liberec with education subject Digital computers.

Simulator will clearly reflect to student behavior of theoretic mictoprocesor, witch combining the best characteristic procesor series 51 and 86. For the biggest clearness enviroment includes graphic illustration of external devices, witch is connect to microprocesor ports. It is concerned about 8 adjustable buttons, 8 led diode, signal generator and 7-segment display.

For program development was used programming environment Microsoft Visual C++ version 6.

Obsah:

Úvod:

Tato diplomová práce se zabývá vývojem softwarového prostředí pro simulaci práce mikroprocesoru vyučovaném v rámci předmětu Číslicové počítače. Studenti si díky této práci budou moci snadněji osvojit své programátorské schopnosti v jazyku assembler. Hlavním důvodem potřeby vzniku tohoto simulátoru byla absence reálného jednoduchého mikroprocesoru na kterém by se studenti mohly naučit požadované znalosti.

V první kapitole je nastíněna historie vzniku mikroprocesorů. Důvod jejich vzniku. Vysvětlení pojmů první,druhá,třetí a čtvrtá generace počítačů.

Dále se kapitola věnuje vlastní funkci mikroprocesoru. Vysvětluje rozdíly, výhody a nevýhody Harvardské architektury od Von Neumannovy. Procesor Harvardské architektury má oddělenou programovou paměť od datové. V programové paměti je uložen program a procesor si z ní bere jednotlivé instrukce, které vykonává jednu po druhé. Oproti tomu Von Neumannova architektura má program uložen v datové paměti a procesor na něj pohlíží stejně jako na data. Jelikož je program uložen v datové paměti je možné ho za běhu modifikovat, což přináší nové možnosti. Nevýhodou je to, že tato architektura potřebuje nějaké paměťové médium, ve kterém by byl program uložen a po zapnutí se nahrál do datové paměti.

Další možné rozdělení mikroprocesorů je na CISC a RISC. Liší se od sebe tím, že CISC potřebují k vykonání jedné instrukce více hodinových cyklů zatímco procesory RISC vykonají instrukce v jediném cyklu.

Na konci kapitoly je popsán model mikroprocesoru, který je vyučován v rámci předmětu Číslicové počítače a který je použit pro vytvoření simulátoru.

Ve druhé kapitole je vysvětlen pojem překladač. Jeho rozdělení na kompilační a interpretační. Překladač převádí zdrojový kód programu do ekvivalentního programu ve strojovém kódu. Jeho výhodou je, že analýza překladu se provádí jen jednou. Jeho nevýhodou je někdy dosti obtížné hledání chyb ve zdrojovém programu. Oproti tomu interprety vykonávají příkazy zdrojového jazyka tak, jak jsou napsané, a přímo provádějí odpovídající akce. Používají se zejména pro účely výuky jazyků nebo pro malé mikropočítače.

Následuje vysvětlení fází překladu obecného překladače, jeho rozdělení na přední a zadní část. Kde přední část se skládá z částí, které obvykle závisejí na zdrojovém jazyku a jdou nezávislé na cílovém počítači. Zahrnují části lexikální analýzu, syntaktickou analýzu, vytváření tabulek symbolů, sémantickou analýzu a generování intermediálního kódu. Zadní část provádí optimalizaci mezikódu, generování a optimalizaci cílového kódu.

Třetí kapitola popisuje vlastní způsob řešení překladače. Postup vytvoření vizuální podoby aplikace a popis jeho částí. U každé části je vysvětlen její účel a činnost kterou provádí. Poté je nastíněna realizace modelu mikroprocesoru vytvořeným objektem CuProcesor a vysvětlením některých jeho proměnných a funkcí. Další části popisují způsob realizace analýzy instrukcí. Vytvoření tabulky skoků. A na konci jak funguje vlastní interpretace jednotlivých instrukcí.

Poslední kapitola slouží jako pomůcka při orientaci v programovém kódu simulátoru. Popisuje rozdělení programu do jednotlivých souborů a zviditelňuje jejich nejdůležitější funkce.

1. Procesor

1.1 Historie vzniku

Na počátku 19. století vytvořil Joseph Marie Jacquard tkalcovský stroj, který vytvářel tkaný vzor pomocí děrných štítků. Jacqardův stroj se ve světě prosadil, v roce 1801 ho představil průmyslníkům v Paříži a podobná technika tkaní se používá dodnes. Požadavek ke zrychlování počítání vedl ke vzniku dalších počítacích strojů. V USA vypsali soutěž na počítací stroj pro potřeby sčítání lidu. Tuto soutěž vyhrál se svým strojem H. Hollerith, který je považován za jednoho z otců IBM. Jeho stroj se jmenoval tabelátor. Jednalo se o stroj pracující s děrnými štítky, na která zaznamenával data. Z tohoto štítku mohl tato data později kdykoli opět vyvolat. Dalším člověkem, jehož myšlenky ovlivnili konstrukce počítačů byl Alan M. Turing, který v roce 1937 publikoval svoji myšlenku Turingových strojů.

Významným počítačem ve vývoji, který byl zkonstruován byl v roce 1944 počítač MARK 1. Základním prvkem tohoto počítače byla elektromechanická součástka relé. Počítač MARK 1 nebyl prvním počítačem založeným na něm, předcházel mu počítač Z1 a v roce 1942 počítač Z2. Počítač MARK 1 vznikl za spolupráce s firmou IBM, skládal se z 9000 relé, 497 mil drátů a celý vážil 5 tun. Jeho výpočetní výkon byl tři operace sčítání za sekundu a jeho nasazení bylo jasné – výpočet atomové bomby. K dalšímu vývoji přispělo několik faktorů, především rozvoj elektroniky, konkrétně použití elektronek místo relé a vynález tranzistoru v roce 1947, dále především myšlenka, kterou prezentoval John von Neumann v roce 1947. Zabýval se logickým návrhem výpočetních strojů. Uvažoval o práci s jedničkami a nulami – binárním systému, který tvoří základ dnešních počítačů. Další jeho myšlenkou bylo , že se počítač skládal z centrální výpočetní jednotky, která prováděla výpočty a paměti, ve které byly uloženy data a vlastní program. Zda jsou v paměti uložena data nebo program určovala pouze interpretace těchto dat procesorem – s programem se dalo zacházet jako s daty. Toto vedlo ke vzniku pojmu von Neumannovská architektura, na níž je založena drtivá většina současných počítačů.

Aby byla historie počítačů přehlednější, začal se vývoj počítačů rozdělovat na jednotlivé etapy – generace počítačů. Každá etapa je charakteristická nejen použitým hardwarem, ale i způsobem obsluhy, programováním počítače, vlastním softwarem, atd.

1.1.1 První generace

První počítačová generace je charakteristická tím, že počítače byly především vyvíjeny školami a nebo na základě grantů od vlády. Většinou se samotní tvůrci počítačů stávali její obsluhou. Co se týče konstrukce, tak počítače první generace byly založené na elektronkách, které nahradily dříve používané relé. Elektronka měla podobnou funkci, jakou má tranzistor, jejich problém byl, že byly oproti tranzistorům značně poruchové. Počítače měly často výpadky i přesto, že počítače byly v klimatizovaném prostředí. Vstup a výstup se na těchto počítačích realizoval pomocí děrných štítků a měly magneticko bubnovou paměť. Typicky tyto počítače obsluhovali velké týmy operátorů, programování probíhalo ve strojovém kódu. Významným jevem této etapy je i pozvolné pronikání počítačů i do komerční sféry. Typickým zástupcem této éry je počítač ENIAC (Electronic Numerical Integrator And Computer) sestrojený v Bellových laboratořích. Počítač se skládal ze 17 000 elektronek, 1500 relé, 70 000 odporů a 10 000 kondensátorů. K jeho pospojování bylo použito více než 500 000 letovaných spojů. Násobení na něm trvalo 2,8 ms.

1.1.2 Druhá generace

Druhá počítačová generace začalo kolem roku 1959 a její hlavní charakteristikou je použití tranzistorů na místo elektronek. Tranzistory byly menší, rychlejší, levnější a spolehlivější. Nejen díky tomu vzrostla výpočetní rychlost až na 230 000 operací za sekundu. Dalším rysem bylo zmenšování počítačů a zlevnění jejich výroby. Přesto tyto počítače měli velice omezené možnosti, na které úlohy se daly používat. Hlavní charakteristikou u programů je dávkové zpracování dat. Paměti se začali používat feritové, na ukládání se používala magnetická páska a začalo se na vývoji magnetických disků. V programování se začali objevovat programovací jazyky, což zpřístupnilo práci s počítačem více lidem. Začali se objevovat i skutečné operační systémy, jak je chápeme z dnešního hlediska. Typickým zástupcem této éry byl počítač IBM 650, který byl první počítačem, vyráběným hromadně, celkem se ho prodalo 1500 kusů.

1.1.3 Třetí generace

Třetí počítačová generace spadá do let 1964 až 1970. Jejím hlavním znakem je použití integrovaného obvodu, který byl vynalezen roku 1959. Integrovaný obvod se skládal z tisíců tranzistorů umístěných na ploše pár centimetrů čtverečních. Opět se snížila cena a spotřeba počítačů a vzrostla rychlost a spolehlivost. Počítače dosahovali rychlosti 2 500 000 operací za sekundu. K ukládání dat se používal magnetický disk. Z hlediska programů byl výrazný rozvoj operačních systémů a hlavně možnost práce více uživatelů na jednom počítači zároveň.

1.1.4 Čtvrtá generace

Čtvrtá generace počítačů, která probíhá od roku 1970 až dodnes by se dala charakterizovat neustálou miniaturizací integrovaných obvodů, v komerční oblasti hlavně rozšiřování i do oblastí, kde se dříve počítače vůbec nepoužívali. Jako začátek této éry se uvádí vyrobení prvního mikroprocesoru firmou Intel. Jednalo se o typ 4004 a měl se původně používat do kalkulaček. Jednalo se o integrovaný obvod, který obsahoval 2300 tranzistorů a jeho cena byla kolem 100 USD. Celý obvod prováděl matematické operace a instrukce, které dostával z dalších obvodů. Ke zkompletování počítače stačilo připojit paměť a vstupy/výstupy. Ukázalo se, že to je vhodné nejen do kalkulaček, ale i do mnoha jiných zařízení. Obvod 4004 je považování za vůbec první mikroprocesor, který kdy vzniknul. Tímto krokem byl v podstatě zvolen směr, jakým se počítače dnes ubírají.

1.2 Funkce procesoru

Procesor je v podstatě polovodičová součástka tvořená především křemíkovou destičkou s několika příměsemi, která vykonává velmi jednoduché operace s datovou pamětí a to velmi rychle. Jedná se například. o přesun hodnoty z jedné paměťové buňky do jiné nebo sčítání obsahu dvou buněk. To co však dělá procesor procesorem nejsou tyto jednoduché operace (říká se jim instrukce procesoru) sami o sobě ale to, že můžeme naprogramovat které a v jakém pořadí se mají vykonávat. Tím můžeme ovlivňovat to, jak se procesor bude chovat navenek.

Jednotlivé instrukce si procesor bere z tzv. programové paměti a vykonává je jednu po druhé. Tato programová paměť může být zcela oddělená od datové paměti a dokonce může být určena jen pro čtení, protože obvykle není potřeba měnit program za běhu (tzv. Harvardská architektura). Druhou možností je, že programová paměť je součástí datové paměti (tzv. Von Neumannova architektura). Oba dva tyto přístupy mají své výhody a nevýhody.

Harvardská architektura se používá zejména v malých systémech (například tzv. jednočipové mikroprocesory), kde je paměť RAM velmi cenná. Program je totiž uložen v paměti typu ROM (Read Only Memory) nebo EPROM (Erasable Programable Read Only Memory) a z této paměti ho procesor za běhu čte a hned ho vykonává. Pokud chceme program změnit, je potřeba tuto paměť přeprogramovat. Není ale nutné po každém spuštění program kopírovat do cenné datové paměti RAM (Random Access Memory), která tak zůstává volná pro data.

Oproti tomu Von Neumannova architektura je mnohem univerzálnější. Program je uložen v datové paměti a procesor tak na něj pohlíží stejně jako na data. Je možné ho za běhu modifikovat, což přináší mnohé nové možnosti. Nevýhodou je to, že tato architektura potřebuje nějaké paměťové médium, ve kterém by byl program uložen a po zapnutí se nahrál do datové paměti. Datová paměť je totiž obvykle typu RAM a po vypnutí napájení se její obsah ztratí. Proto se tato architektura používá zejména tam, kde je k procesoru připojeno jednak dostatečné množství paměti typu RAM a také nějaké paměťové médium, kde je program uložen před jeho nahráním do paměti a spuštěním. Proto asi nikoho nepřekvapí, že počítače IBM-PC používají práv_ tuto architekturu.

Další možné rozdělení procesorů je např. podle doby vykonávání jednotlivých instrukcí. Procesor je totiž ve své podstatě synchronní stroj (jeho chod je řízen hodinovým signálem), přivedeným zvenčí. Většinou je tento signál generován

krystalovým oscilátorem a jedné jeho periodě říkáme hodinový cyklus. Doba trvání jednoho hodinového cyklu se pohybuje v řádu ns až μs.

Starší generace procesorů potřebovali pro vykonání jedné instrukce a všech souvisejících operací (tzv. instrukční cyklus) několik hodinových cyklů. Procesor totiž musí nejprve dekódovat instrukci, aby zjistil co bude pro její vykonání potřebovat. Poté z paměti načte požadovaná data, provede požadovanou operaci a uloží výsledek do paměti. Takže například procesorům Intel 8051 trval jeden instrukční cyklus dvanáct hodinových cyklů. Některé instrukce vyžadovali dokonce pro svoje vykonání více instrukčních cyklů (násobení trvalo 6 instrukčních cyklů, tzn. 72 hodinových). Procesory tohoto typu se nazývají obecně CISC (Complete Instruction Set Computer). Mají velké množství instrukcí (což komplikuje fázi dekódování instrukce) a některé instrukce jsou i poměrně složité. Jejich vykonávání je časově náročné a každá instrukce může zabrat jiné množství času.

Proto se v poslední době velmi prosazují procesory typu RISC (Reduced Instruction Set Computer). Tyto procesory mají menší množství instrukcí a jejich instrukce jsou jednodušší. To umožňuje jejich vykonávání během jednoho hodinového cyklu. Proto procesory RISC vykonávají mnohem více instrukcí než procesory CISC na stejné hodinové frekvenci. Na druhou stranu je to vykoupeno tím, že na některé složitější operace (jako je násobení) nemají přímo instrukce a tak je potřeba tyto operace naprogramovat softwarově.

Kromě datové a programové paměti ještě procesor ke své práci potřebuje jeden druh paměti, a to vnitřní registry procesoru. Tato sada registrů se používá pro ukládání stavu procesoru (adresa právě vykonávané instrukce, ukazatel do datové paměti, ukazatel na konec zásobníku apod.) a obsahuje i univerzální registry pro ukládání mezivýsledků výpočtů.

Registry jsou vždy součástí procesoru a proto je přístup k nim velmi rychlý (rychlejší než k normální datové paměti). Některé procesory dokonce ani neumožňují provádět některé operace s daty jinde, než v těchto registrech. Data je potřeba do nich nahrát, provést požadovanou operaci a uložit výsledek z registru do datové paměti.

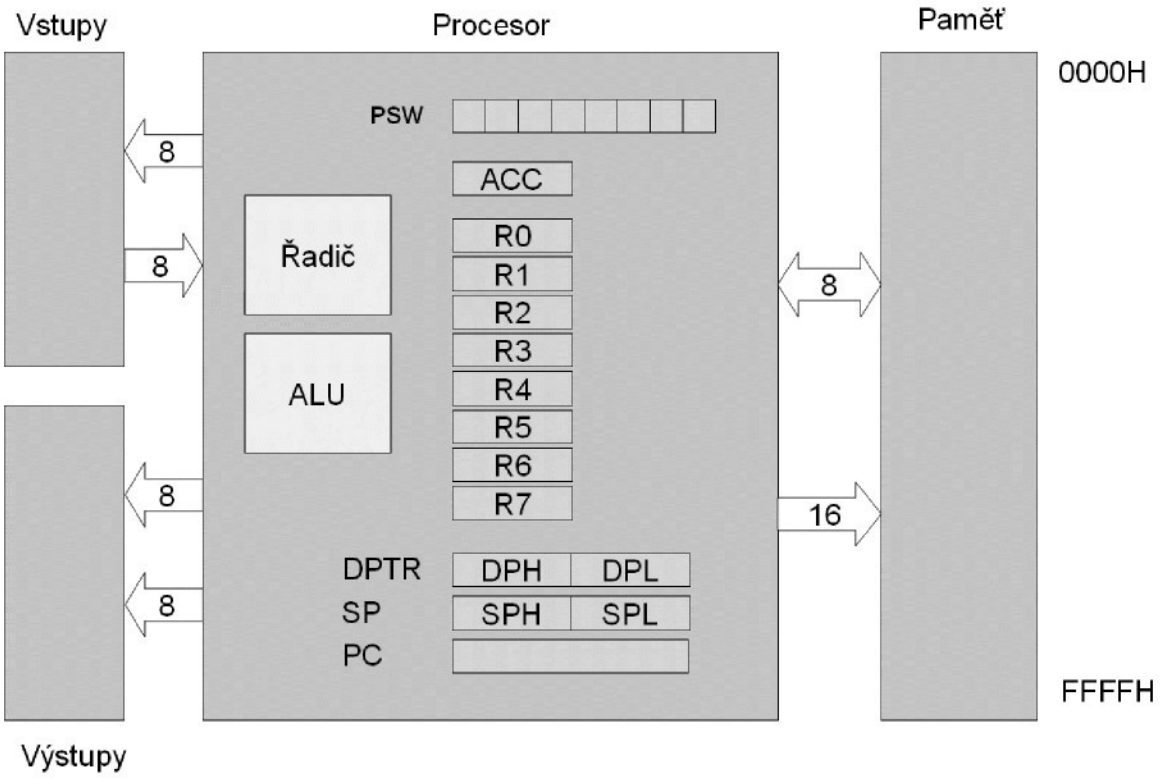
1.3 Model procesoru pro simulátor

V rámci předmětu Číslicové počítače se studenti učí programovat procesor, který je velmi jednoduchý a je do velké míry inspirován procesory řady x51.

Tento virtuální mikroprocesor bude používat harvardskou architekturu, to znamená bude mít oddělenou paměť pro program a paměť pro data. Programovou paměť v tomto modelu zanedbáme. Budeme předpokládat, že je v ní program, který se vykonává. Pro práci s procesorem o ní nic dalšího vědět nepotřebujeme.

Pro zjednodušení budeme předpokládat, že je to procesor typu RISC a tak vykonání každé instrukce zabere právě jeden hodinový cyklus. Tento cyklus bude mít periodu 1 μs, tzn. Že tento náš procesor vykoná milión instrukcí za vteřinu.

Celý procesor je osmibitový a proto všechny datové sběrnice a registry budou mít osm bitů. Adresová sběrnice k datové paměti bude šestnáctibitová. To znamená, že do paměti se vejde 2^{16} = 65536 bytů dat.



Obr 1.3: Model mikroprocesoru

Aby procesor mohl komunikovat s okolním světem, potřebuje i nějaké vstupy a výstupy. Ty budou také osmibitové a pracovat se s nimi bude obdobně jako s datovou pamětí. Adresová sběrnice pro vstupy i výstupy bude osmibitová. To znamená, že vstupů a výstupů může být až $2^8 = 256$ bytů.

Procesor se skládá z:

- Řadiče instrukcí – dekóduje instrukci na vstupu a provede potřebné operace
- ALU – Aritmeticko-logická jednotka, ta provádí vlastní výpočty
- Univerzální použitelné registry R0..R7 – slouží jako rychlá paměť pro výpočty
- Akumulátor = registr A – speciální registr, jediný možný operand některých instrukcí
- Registr PSW (Program State Word) – tento registr obsahuje stav procesoru, v našem modelu budeme používat z tohoto registru pouze dva bity, a to bit C (Carry) který je v jedničce pokud poslední matematická operace skončila přetečením a bit Z (Zero) který je v jedničce pokud aktuální obsah akumulátoru je 0
- Registr DPTR (Data Pointer) – slouží pro nepřímé adresování v datové paměti, je šestnáctibitový ale přistupujeme k němu osmibitově pomocí registrů DPL a DPH
- Registr SP (Stack Pointer) – slouží jako ukazatel na vrchol zásobníku, viz kapitola o zásobníku, podobně jako DPTR je šestnáctibitový ale pracujeme s ním osmibitově pomocí registrů SPL a SPH
- Registr PC (Program Counter) – tento registr obsahuje adresu aktuální vykonávané instrukce v programové paměti. Je šestnáctibitový, takže program může mít maximálně 65536 instrukcí

1.4 Instrukce procesoru

Jednotlivé instrukce jsou uloženy za sebou v programové paměti. Každá instrukce se v paměti skládá z operačního kódu a operandů.

V reálném mikroprocesoru může zápis instrukce v paměti zabírat různé množství paměťových buněk. Instrukce, která nemá žádné operandy, zabere pouze jednu buňku. Naopak instrukce, která například ukládá konstantu do nějakého registru, zabere tři buňky. Registr PC (Program Counter) obsahuje adresu aktuální vykonávané instrukce a po startu procesoru je v něm hodnota 0. Po každém vykonání instrukce přičte řadič instrukcí do registru PC velikost instrukce v paměti a tak PC ukazuje na další instrukci, která se má vykonat. Jedinou výjimkou jsou instrukce skoku, po kterých se může vykonávat i jiná instrukce, než která následuje za touto instrukcí.

Operační kód instrukce je v paměti samozřejmě uložen jako číslo. Instrukcí sice procesor nemá mnoho (jednočipové mikroprocesory mají řádově okolo 100 různých instrukcí), ale stejně je pro člověka nepředstavitelné, že by znal všechny jejich operační kódy zpaměti a pamatoval si jejich význam. Navíc by si musel pamatovat i kódy jednotlivých operandů a v případě skoků by musel při každé změně v programu přepočítat adresy instrukcí v paměti. Takový program by se nejen obtížně psal, ale i četl a ladil. Proto se procesory neprogramují přímo ve strojovém kódu ale v takzvaném jazyce symbolických adres (JSA), který se někdy označuje jako assembler. V tomto jazyce je každé instrukci přiřazena tzv. mnemonická zkratka. Stejně tak jsou zavedeny jednoduchá jména pro registry procesoru místo jejich čísel. Také je možné používat v takovém programu symbolická návěstí pro označení místa, kam ukazuje instrukce skoku. Program napsaný v assembleru je pak nutné přeložit pomocí překladače assembleru do strojového kódu. Překladač nahradí jména instrukcí, registrů a návěstí konkrétními čísly a poskládá instrukce za sebe do paměti procesoru. Toto je triviální úloha, která nám ale přináší obrovský komfort při psaní programů na úrovni jednotlivých instrukcí.

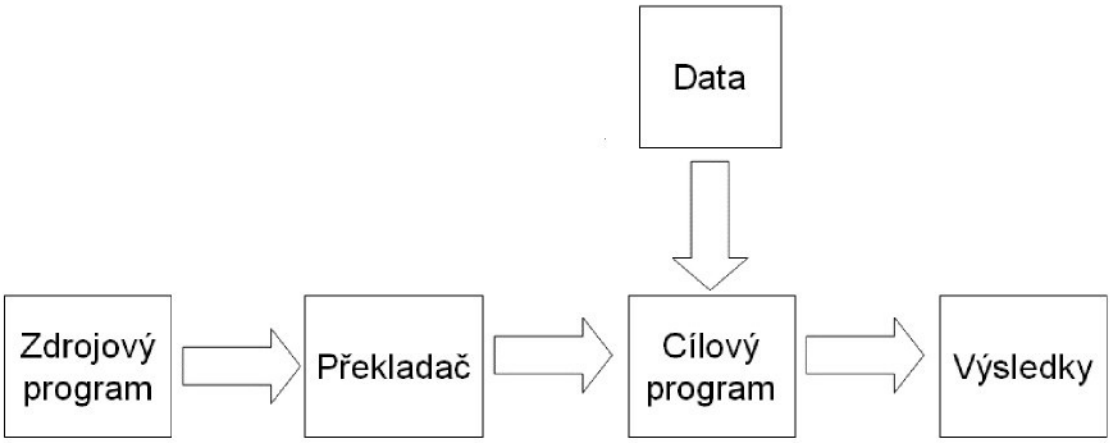
2. Překladač

Pojem překladač se používá od začátku 50. let, kdy se začaly vyvíjet uživatelsky orientované programovací jazyky vyšší úrovně, podstatně méně závislé na strojovém kódu konkrétního počítače. V tu dobu však ještě vládla všeobecná skepse nad použitelností automatického programování, jak se tehdy programování ve vyšších jazycích nazývalo. První jazyky tohoto typu (např. FORTRAN) a autokódy, ze kterých se vyvinuly, však byly silně poznamenány tehdy existujícími instrukčními soubory počítačů. Například FORTRAN IV umožňoval práci pouze s trojrozměrnými poli, neboť jeho první implementace byla provedena na počítači IBM 709, který měl pouze tři indexové registry.

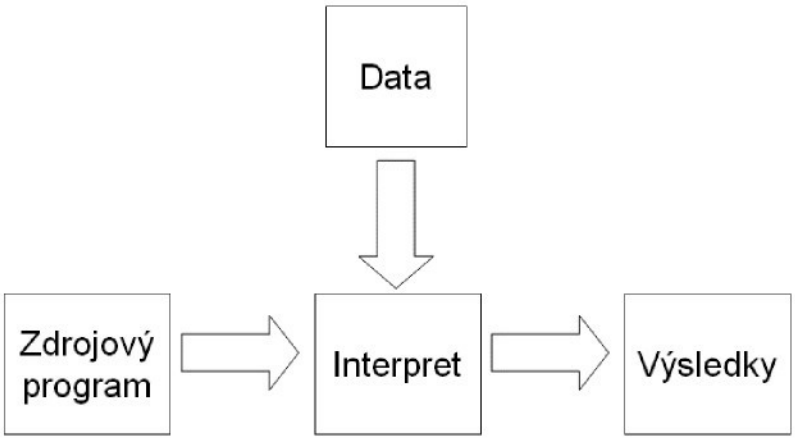
Moderní jazyky vysoké úrovně svým obvykle stručným zápisem umožňují zvýšit produktivitu práce programátora, poskytují různá sémantická omezení (např. typovou kontrolu), kterými se dají redukovat logické chyby v programech, a zjednodušují ladění programů. Další velmi významnou vlastností současných programovacích jazyků je možnost vytváření strojově nezávislých programů, které se dají přenášet i mezi principiálně různými architekturami počítačů. Jejich nevýhodou je rychlost překladu (typicky 2 až 10krát nižší než u ručně psaných programů v jazyce asembleru) a velikost, jak překladače, tak přeloženého kódu. Tyto nevýhody jsou však redukovány s rozvojem moderních počítačových architektur. V oblasti návrhu a implementace jazyků se nyní často dostáváme do zcela opačné situace, než jaká byla na počátku vývoje jazyků, kdy jsou navrhovány procesory již s ohledem na překlad konkrétních jazyků.

Máme-li program napsaný v některém vyšším programovacím jazyce, existuje několik možných přístupů k jeho spuštění. Buď můžeme program převést do ekvivalentního programu ve strojovém kódu počítače. Překladače tohoto typu se označují názvem kompilátory nebo kompilační překladače, nebo můžeme napsat program, který bude interpretovat příkazy zdrojového jazyka tak, jak jsou napsané, a přímo provádět odpovídající akce. Programy realizující druhý přístup se nazývají interprety nebo interpretační překladače. Obrázky 2.1 a 2.2 představují schémata činnosti obou typů překladačů. Výhodou kompilace je, že analýza zdrojového programu a jeho překlad se provádějí jen jednou, i když může jít o časově dosti náročný proces. Dále již spouštíme pouze ekvivalentní program ve strojovém kódu, který je výsledkem

překladu. Nevýhodou je někdy dosti obtížné hledání chyb ve zdrojovém programu, pokud máme pouze informace o místu chyby vyjádřené v pojmech strojového jazyka (adresy, výpisy obrazu paměti).



Obr 2-1: Kompilační překladač

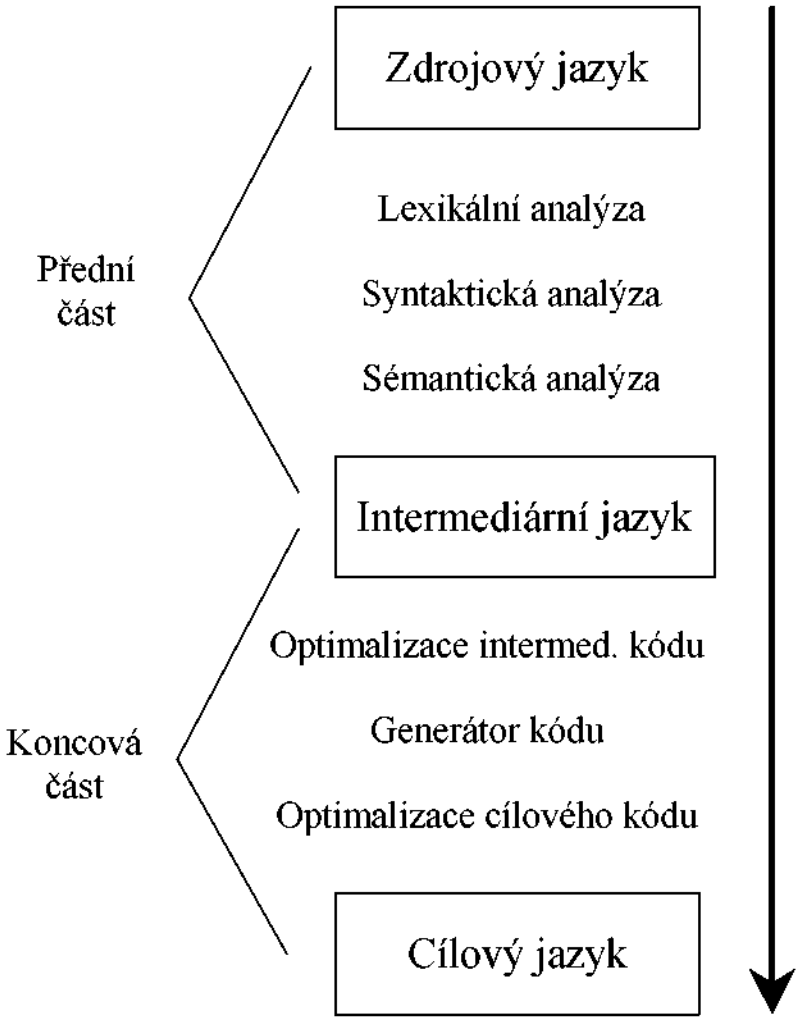


Obr 2-2: Interpretační překladač

Výběr vhodného přístupu, zda kompilovat nebo interpretovat, závisí obvykle na povaze jazyka a prostředí, ve kterém se používá. Pro časově náročné matematické výpočty se používají kompilační překladače, naopak pro účely výuky jazyků nebo na malých mikropočítačích se dává přednost interpretaci

Obecné schéma překladače z hlediska jeho členění na fáze je uvedeno na obr. 2.3. Toto členění odpovídá logické struktuře překladače, která však nemusí přímo odpovídat skutečné implementaci.

Jednotlivé fáze se často rozdělují na přední část (front end) a koncovou část (back end). Přední část se skládá z těch fází nebo jejich částí, které závisejí převážně na zdrojovém jazyku a jsou dosti nezávislé na cílovém počítači. Obvykle zahrnuje lexikální a syntaktickou analýzu, vytváření tabulky symbolů, sémantickou analýzu a generování intermediárního kódu. V přední části překladače lze provést rovněž jistou část optimalizace kódu. Obsahuje také obsluhu chyb, které vznikají během analýzy.



Obr 2-3: Fáze překladače

2.1 Kompilátor

Funkce kompilátoru spočívá v přečtení zdrojového programu zapsaném ve zdrojovém jazyce a překládá (transformuje) jej na ekvivalentní cílový program zapsaný v cílovém jazyce.

Zdrojovým jazykem kompilátoru nemusí být vždy nějaký programovací jazyk. Může se jednat také o některý přirozený jazyk (např. angličtinu), speciální jazyk popisující strukturu křemíkového integrovaného obvodu nebo strukturu grafických informací, které se mají zobrazit na tiskárně. Cílovým kódem takového kompilátoru pak může být třeba jiný přirozený jazyk, maska integrovaného obvodu nebo posloupnost příkazů pro ovladač laserové tiskárny. Programovacím jazykem tohoto typu je například PostScript, který se používá pro vytváření grafiky, nebo Metafont, kterým se definují tvary znaků používaných při sazbě textů připravených programem TEX. Tyto jazyky mají i prostředky pro vytváření cyklů, podmíněných příkazů nebo pro definování vlastních procedur nebo funkcí.

Kompilátor musí provádět dvě základní činnosti: analyzovat zdrojový program a vytvářet k němu odpovídající cílový program. Analýza spočívá v rozkladu zdrojového programu na jeho základní součásti, na základě kterých se během syntézy vybudují moduly cílového programu. Obě části překladače, analytická i syntetická, využívají ke své činnosti společné tabulky.

Analýza zdrojového programu při překladu probíhá na následujících úrovních:

- **Lexikální (lineární) analýza**. Zdrojový program vstupuje do procesu překladu jako posloupnost znaků. Tato posloupnost se čte lineárně zleva doprava a sestavují se z ní lexikální symboly (tokeny) jako konstanty, identifikátory, klíčová slova nebo operátory.
- **Syntaktická (hierarchická) analýza**. Z posloupnosti lexikálních symbolů se vytvářejí hierarchicky zanořené struktury, které mají jako celek svůj vlastní význam, např. výrazy, příkazy nebo deklarace.
- **Sémantická analýza**. Během sémantické analýzy se provádějí některé kontroly, zajišťující správnost programu z hlediska vazeb, které nelze provádět v rámci syntaktické analýzy (např. kontrola deklarací, typová kontrola apod.).

Uvedené členění na úrovně analýzy vychází z toho, že běžné programovací jazyky jsou z hlediska Chomského klasifikace typu 1, tj. kontextové.

2.1.1 Lexikální analýza

Fáze lexikální analýzy čte znaky zdrojového programu a sestavuje je do posloupnosti lexikálních symbolů, v níž každý symbol představuje logicky související posloupnost znaků jako identifikátor nebo operátor obdobný „:=". Posloupnost znaků tvořících symbol se nazývá lexém.

Po lexikální analýze znaků např. v tomto přiřazovacím příkazu

pozice := počátek + rychlost * 60 (1.1)

by se vytvořily následující lexikální jednotky:

1. **identifikátor** pozice
2. **symbol přiřazení** :=
3. **identifikátor** počátek
4. **operátor** +
5. **identifikátor** rychlost
6. **operátor** *
7. **číslo** 60

Symboly, které zahrnují celou třídu lexikálních jednotek (identifikátor, číslo, řetězec), jsou reprezentovány obvykle jako dvojice <druh symbolu, hodnota>, přičemž druhá část dvojice může být pro některé symboly prázdná. Výstupem lexikálního analyzátoru pro příkaz (1.1) by tedy mohla být posloupnost

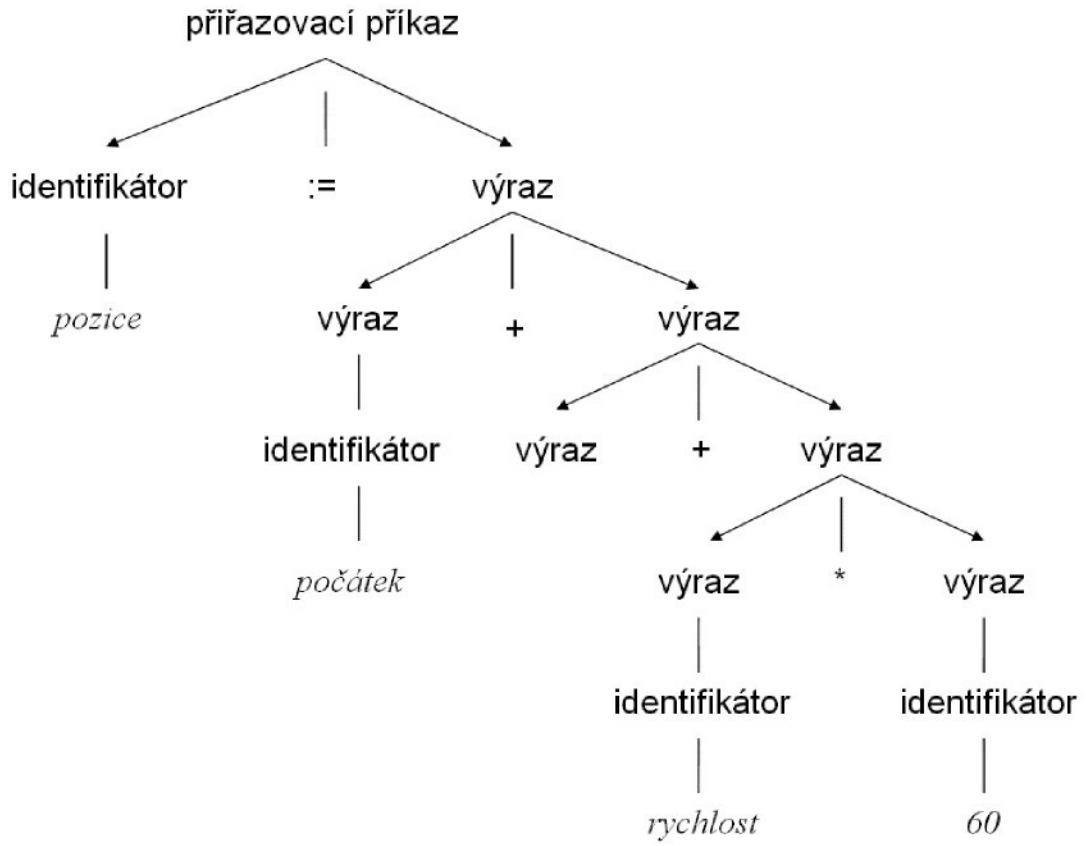
<id,pozice> <:=> <id,počátek> <+> <id,rychlost> <*> <num,60>

Mezery, konce řádků a poznámky oddělující lexikální symboly se obvykle během lexikální analýzy vypouštějí.

2.1.2 Syntaktická analýza

Syntaktická analýza spočívá v sestavování lexikálních jednotek ze zdrojového programu do gramatických frází, které překladač používá pro syntézu výstupu.

Gramatické fráze zdrojového programu se obvykle reprezentují derivačním stromem obdobným stromu na obr. 2.1.2.



Obr. 2.1.2: Derivační strom pro výraz pozice := počátek + rychlost * 60

Ve výrazu „počátek+rychlost*60“ je fráze „rychlost*60“ logickou jednotkou, neboť podle běžných matematických konvencí pro aritmetické výrazy se násobení provádí před sčítáním. Vzhledem k tomu, že za výrazem „počátek+rychlost“ následuje „*“, nevytváří tento výraz v situaci na obr. 2.1.2 frázi.

Hierarchická struktura programu se obvykle vyjadřuje pomocí rekurzivních pravidel, zapsaných ve formě bezkontextové gramatiky. Například pro definici části výrazu můžeme mít následující pravidla:

výraz -> identifikátor **(1)**

výraz -> číslo **(2)**

výraz -> výraz + výraz **(3)**

výraz -> výraz * výraz **(4)**

výraz -> (výraz) **(5)**

Pravidla (1) a (2) jsou (nerekurzivní) základní pravidla, zatímco (3){(5) definují výraz pomocí operátorů aplikovaných na jiné výrazy. Podle pravidla (1) jsou tedy

počátek a rychlost výrazy. Podle pravidla (2) je 60 výraz, zatímco z pravidla (4) můžeme nejprve odvodit, že rychlost*60 je výraz a konečně z pravidla (5) také počátek+rychlost*60 je výraz. Podobným způsobem jsou definovány příkazy jazyka, jako např.:

příkaz -> identifikátor := výraz
příkaz -> while (výraz) do příkaz **(1.2)**
příkaz -> if (výraz) then příkaz

Dělení na lexikální a syntaktickou analýzu je dosti volné. Obvykle vybíráme takové rozdělení, které zjednodušuje činnost analýzy. Jedním z faktorů, které přitom uvažujeme, je to, zda jsou konstrukce zdrojového jazyka regulární nebo ne. Lexikální jednotky lze obvykle popsat jako regulární množiny, zatímco konstrukce vytvořené z lexikálních jednotek již vyžadují obecnější přístupy.

Běžně rozpoznáváme identifikátory jednoduchým prohlížením vstupního textu, v němž očekáváme znak, který není písmeno ani číslice, a potom seskupíme všechna písmena a číslice nalezené až do tohoto místa do lexikální jednotky pro identifikátor. Znaky takto shromážděné zaznamenáme do tabulky (tabulky symbolů) a odstraníme je ze vstupu tak, aby mohlo pokračovat zpracování dalšího symbolu.

2.1.3 Sémantická analýza

Fáze sémantické analýzy zpracovává především informace, které jsou uvedeny v deklaracích, ukládá je do vnitřních datových struktur a na jejich základě provádí sémantickou kontrolu příkazů a výrazů v programu. K identifikaci operátorů a operandů těchto výrazů a příkazů využívá hierarchickou strukturu, určenou ve fázi syntaktické analýzy. Důležitou složkou sémantické analýzy je typová kontrola. Kompilátor zde kontroluje, zda všechny operátory mají operandy povolené specifikací zdrojového jazyka. Mnoho definic programovacích jazyků například vyžaduje, aby kompilátor hlásil chybu, kdykoliv je reálné číslo použito jako index pole. Specifikace jazyka však může dovolit některé implicitní transformace operandů, například při aplikaci binárního aritmetického operátoru na celočíselný a reálný operand. V tomto případě může kompilátor požadovat konverzi celého čísla na reálné.

2.2 Interpret

Interpretace je mnohem pomalejší než kompilace, neboť je třeba analyzovat zdrojový příkaz pokaždé, když na něj program narazí. Pro poměr mezi rychlostí interpretovaného a kompilovaného programu se uvádějí hodnoty mezi 10:1 až 100:1, v závislosti na konkrétním jazyce. Interprety bývají také náročné na paměťový prostor, neboť i při běhu programu musí být stále k dispozici celý překladač. Interprety však mají i své výhody oproti kompilačním překladačům. Při výskytu chyby máme vždy přesné informace o jejím výskytu a můžeme poměrně rychle odhalit její příčinu. Tento přístup je tedy vhodný zvláště při ladění programů. Interprety umožňují modifikaci textu programu i během jeho činnosti, což se využívá často u jazyků jako je Prolog nebo LISP. U jazyků, které nemají blokovou strukturu (např. BASIC, APL), se může změnit některý příkaz, aniž by se musel znovu překládat zbytek programu. Interprety se dále používají tam, kde se mohou typy objektů dynamicky měnit v průběhu provádění programu. Typickým příkladem je jazyk Smalltalk-80. Jejich zpracování je pro kompilační překladače značně obtížné. Interpretační překladače bývají značně strojově nezávislé, neboť negenerují strojový kód. Pro přenos na jiný počítač obvykle postačí interpret znovu zkompilovat.

3. Popis funkce vytvořeného simulátoru

Jak bylo již v úvodu řečeno, tento simulátor vznikl jako výuková pomůcka pro studenty Technické univerzity v Liberci při výuce předmětu Číslicové počítače. Jeho funkce byly navrženy tak, aby byl dostatečně jednoduchý a přitom studenty dostatečně naučil vytvářet programy v jazyku assembler. Tím se dobře seznámí s vlastní funkcí mikroprocesoru, neboť assembler pracuje přímo s jeho registry, vstupy, výstupy a pamětí.

Jako programovací jazyk pro vytvoření simulátoru byl zvolen jazyk Microsoft Visual C++ 6.0 s podporou funkcí MFC.

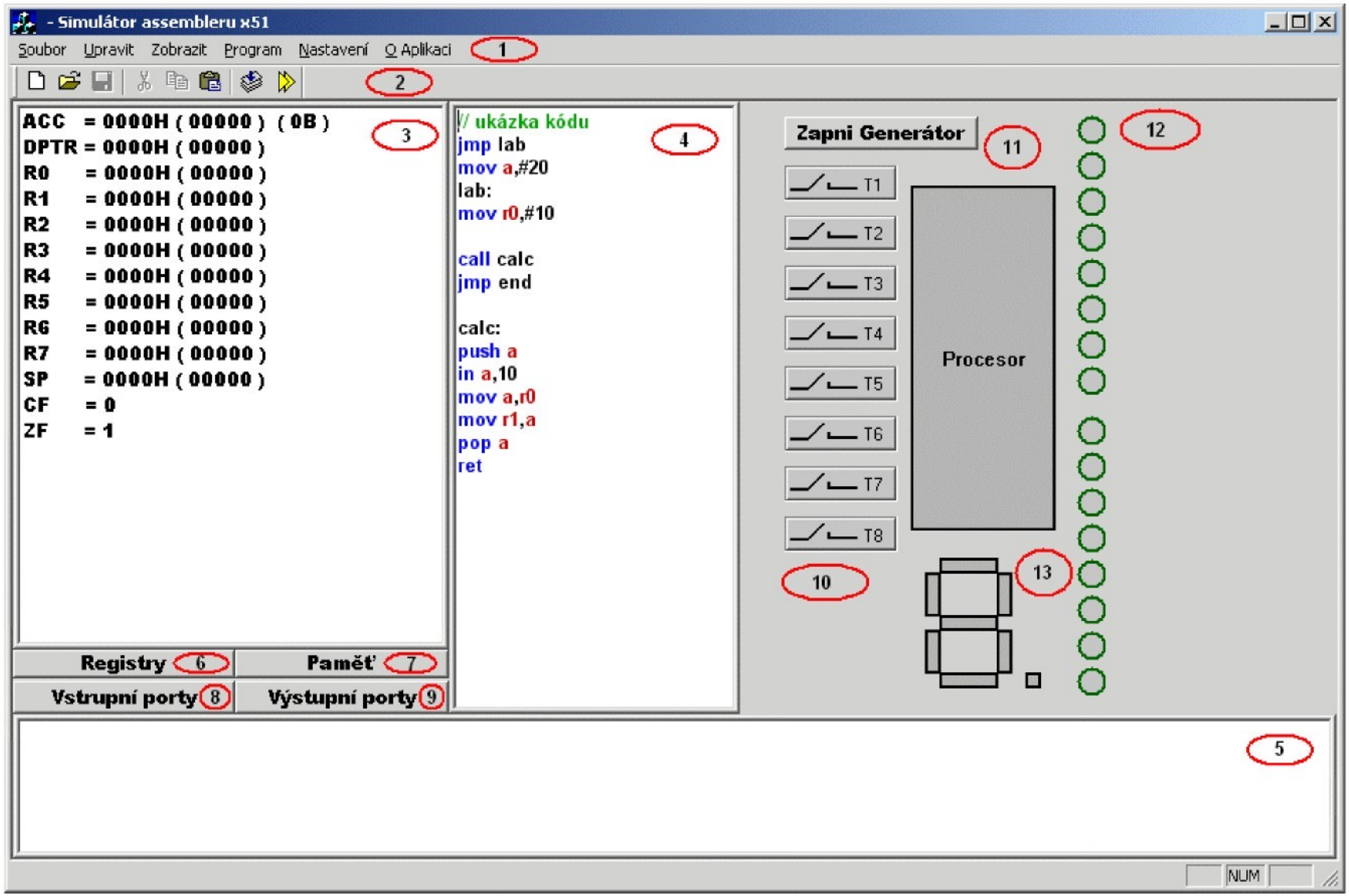
3.1 Popis vizuálního prostředí simulátoru

Základ vizuální části aplikace byl vytvořen průvodcem MFC AppWizard. V němž byl zvolen model rozhraní jednoho dokumentu (SDI – Single Dokument Interface). Tento model podporuje otevření pouze jednoho dokumentu. V našem případě jednu pracovní plochu simulátoru.

Dále byla vybrána volba podpory architektury dokument-pohled. V aplikacích tohoto typu jsou data reprezentována objektem dokumentu a pohled těchto dat je reprezentován objektem pohled. Tyto objekty spolupracují jak při zpracování uživatelského vstupu, tak při vytváření textové nebo grafické reprezentace výsledných dat. Třída CDocument, implementovaná v knihovně MFC, je výchozí třídou pro všechny objekty dokumentu, zatímco třída CView je výchozí třídou pro všechny objekty pohledu. Hlavní okno aplikace, jejíž model chování se zakládá na třídě CFrameWnd, není ústředním koordinačním bodem, kde jsou zpracovávány všechny příchozí zprávy, ale slouží v první řadě jako kontejner pro objekty pohled, panel nástrojů, stavový řádek a další objekty vytvářející uživatelské rozhraní aplikace. V dalším kroku byla ponechána volba podpory panelu nástrojů a stavové lišty.

V posledním kroku průvodce byla změněna volba základní třídy z CView na CFormView. Tím aplikace dostala vzhled formuláře na který bylo možné umístit další ovládací prvky.

Podobu a umístění jednotlivých ovládacích prvků je vidět na obrázku 3.1.

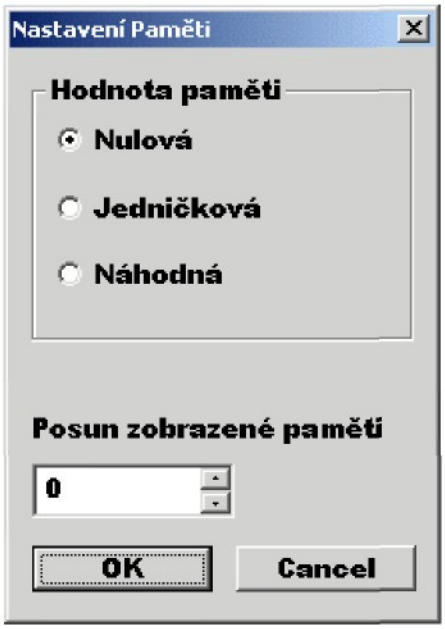


Obr. 3.1: Vizuální podoba aplikace simulátoru

Popis jednotlivých částí simulátoru:

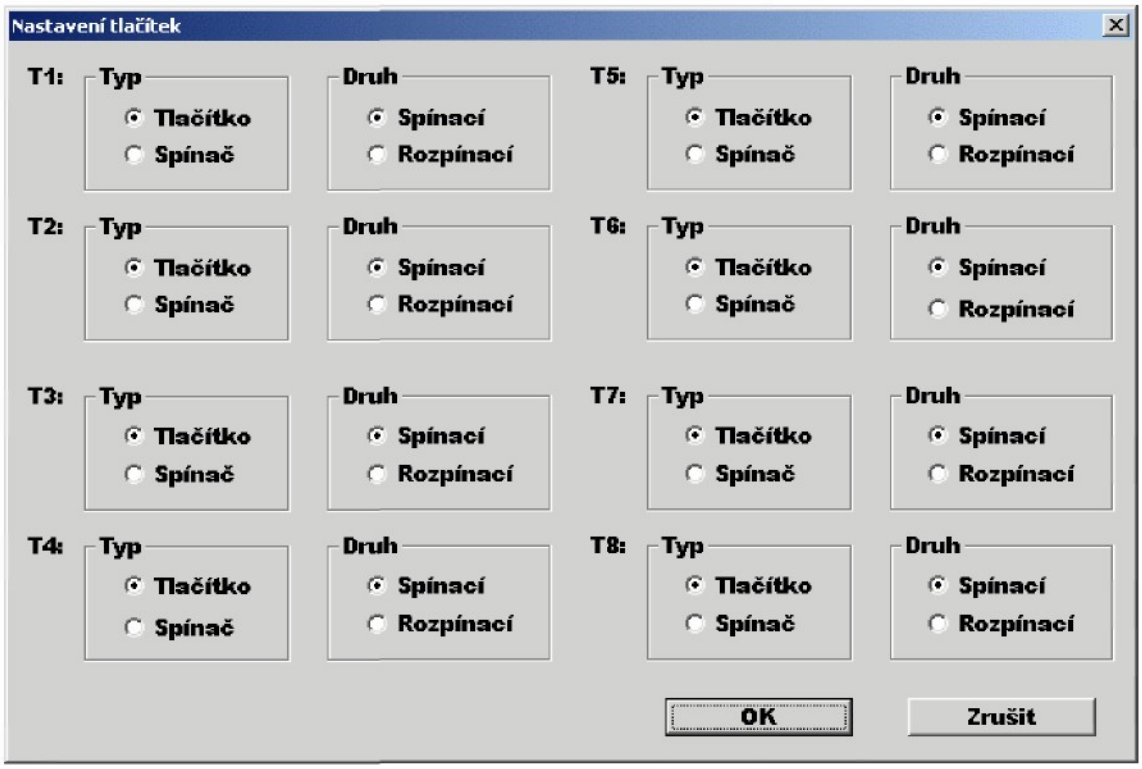
1. **Menu simulátoru**. Menu je rozděleno na části Soubor, Upravit, Zobrazit, Program, Nastavení a O Aplikaci.
   - položka **Soubor** (klávesová zkratka ALT+S) obsahuje možnosti Nový, Otevřít, Uložit, Uložit jako a Konec
     - položka **Nový** slouží k vytvoření nového programu. Vymaže se obsah textového pole realizovaného komponentou *RitchEdit* [část 4]. Vymazání provede funkce *RitchEdit.Clear()*. Hodnoty registrů, výstupní a vstupní porty a datová paměť se vynuluje funkcí objektu modelu mikroprocesoru *uP.Init()*. Nastavení tlačítek led-diod a generátoru se vrátí do výchozího nastavení zavoláním funkce *UpdateControlsFromDoc()*.
     - položka **Otevřít** umožní načtení programu ze souboru do ovládacího prvku *RitchEdit* [část 4]. Výběr souboru je realizován přes *CFileDialog*. Pomocí něho je získána cesta a jméno souboru pro otevření. Tem je poté načten callback funkcí *readFile(CString sFileName)* do *RitchEditu* funkcí *StreamIn*.
     - položka **Uložit a Uložit jako** slouží k uložení programu z ovládacího prvku *RitchEdit* [část 4] do souboru. Výběr cesty a jména souboru pro uložení je realizován třídou *CFileDialog*. Vlastní uložení realizuje objekt *EDITSTREAM* a jeho volání callback funkce *RitchEdit_WriteFile*.
     - položka Konec ukončí simulátor.
   - položka **Upravit** (klávesová zkratka ALT+U) obsahuje možnosti Vyjmout, Kopírovat a Vložit. Tyto položky slouží pro vkládání a vyjímání textu z a do ovládacího prvku RitchEdit [část 4] přes schránku Windows.
     - položka **Vyjmout** je aktivní pouze pokud je v ovládacím prvku RitchEdit [část 4] označen nějaký text. A slouží k vyjmutí označeného textu z prvku RitchEdit funkci *Cut()* a jeho vložení do schránky Windows.
     - položka **Kopírovat** je aktivní pouze pokud je v ovládacím prvku RitchEdit označen nějaký text a zkopíruje označený text funkcí *Copy()* do schránky Windows.
     - položka **Vložit** je aktivní pouze pokud je ve schránce Windows nějaký text ke vložení a tento text vloží do ovládacího prvku RitchEdit funkcí *Paste()* na pozici kursoru.

- položka **Zobrazit** (klávesová zkratka ALT+Z) obsahuje možnosti Registry (F5), Paměť (F6), Vstupní porty (F7) a Výstupní porty (F8). Vždy může být označena pouze jedna položka výběru. Tyto položky slouží jako přepínač, který rozhoduje o tom co bude zobrazeno v panelu výpisu stavů modelu mikroprocesoru. Tento panel reprezentuje komponenta ListBox [část 3]. Vkládání jednotlivých položek je naprogramováno pomocí funkce *InsertString*.
- položka **Program** (klávesová zkratka ALT+P) obsahuje možnosti Přeložit, Spustit a Ukončit běh.
  - položka **Přeložit** zkompiluje program napsaný v ovládacím prvku RitchEdit [část 4] pomocí funkce *compile*. Pokud jsou v programu chyby zobrazí se v ovládacím prvku ListBox [část 5] příslušná zpráva informující o druhu chyby. Přidání chyby realizuje funkce *Show_errHint*. Dále se barva písma řádku na kterém byla chyba nalezena změní na světle zelenou a podtrhne. To zajišťuje funkce *Set_ErrLine*.
  - položka **Spustit** provede simulaci programu na virtuálním modelu [kapitola 1.3]. Před vlastní simulací provede ještě kompilaci a pokud nebyla nalezena chyba simuluje program. Simulaci vykonává funkce *Execute*.
  - Položka **Ukončit běh** ukončí běh simulace programu.
- položka **Nastavení** (klávesová zkratka ALT+N) obsahuje možnosti Paměť, Tlačítka, Generátor, Led diody a Sedmisegmentovka. Tato nabídka slouží pro individuální nastavení.
  - Položka **Paměť** vyvolá dialog nastavení paměti. V něm je možné nastavit hodnoty paměti na nula,jedna nebo náhodné. Dále je možné nastavit od které paměťové buňky se bude zobrazovat obsah paměti v panelu výpisu [část 4]. Podoba dialogu je na obr. 3.1-1.
  - položka **Tlačítka** vyvolá dialog nastavení osmi tlačítek [část 10] obr 3.1-2. Volba typu tlačítka určuje zda tlačítko zůstane po stisknutí dole (volba Spínač) nebo ne (volba Tlačítko). Volba druh určuje zda bude tlačítko spínací nebo rozpínací.
  - položka **Generátor** vyvolá dialog nastavení generátoru signálu [část 11]. Podoba dialogu je na obr 3.1-3. V tomto dialogu je možné nastavit frekvenci generování signálu, číslo portu a bit příslušného portu na který je připojen. Generovaný signál má obdélníkový tvar s danou frekvencí.



Obr 3.1-1: Dialog nastavení paměti

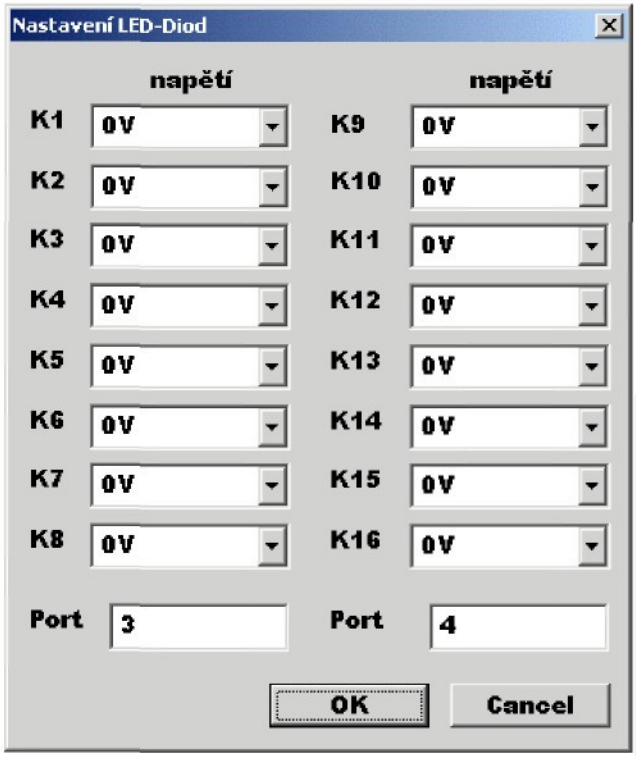
Obr 3.1-3: Dialog nastavení generátoru



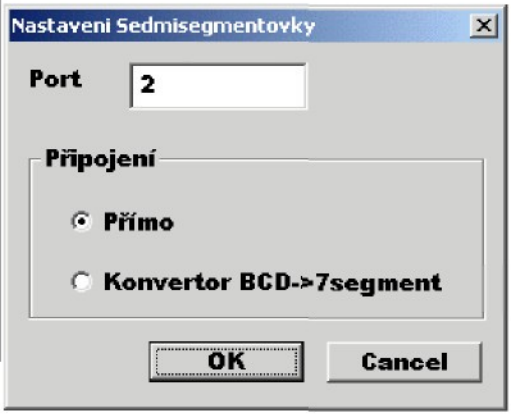
Obr 3.1-2: Dialog nastavení tlačítek

- položka **Led diody** vyvolá dialog nastavení 16ti diod [část 12]. Podoba dialogu je na obr 3.1-4. V tomto dialogu je možné nastavit 2 porty, ke kterým budou diody připojeny. Dále je možno vybrat zda budou diody připojeny na 5V či na 0V.
- položka **Sedmisegmentovka** obr. 3.1-5 vyvolá dialog s nastavením sedmisegmentového displeje. Dialog obsahuje nastavení čísla portu připojení a způsob připojení. Způsobem připojení je myšleno zda jednotlivé segmenty displeje jsou připojeny přímo k danému portu nebo zda-li je mezi porty a displejem zapojen konvertor BCD->7segment. Tento konvertor převádí dekadickou číslici na signály odpovídající příslušnému číslu pro zobrazeni na sedmisegmentovce.

- položka **O Aplikaci** (klávesová zkratka ALT+O) vyvolá dialog s detaily o verzi aplikace, jménu autora atd..

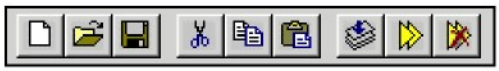


Obr 3.1-4: Dialog nastavení LED Diod



Obr 3.1-5: Dialog nastavení Sedmisegmentovky

2. **Panel nástrojů** obr 3.1-6**.** Umožňuje rychle použít vybraná volby menu. Obsahuje tlačítka Nový, Otevřít, Uložit, Kopírovat, Vyjmout, Vložit, Přeložit, Spustit a Ukončit běh.



Obr 3.1-6: Panel nástrojů

3. **Ovládací prvek ListBox**. Slouží pro zobrazení hodnot modelu mikroprocesoru. Pro výběr zobrazení slouží položka menu→Zobrazení, ikony na panelu nástrojů nebo tlačítka označená na obr 3.1 čísly 6,7,8 a 9.

4. **Ovládací prvek RitchEdit**. Tento ovládací prvek slouží jako textové pole pro psaní a úpravu programu. Obsahuje zvýraznění syntaxe příkazů (modrá barva), jmen registrů (červená barva) a komentářů (zelená barva). Pro označení komentáře v programu slouží vložení znaků „//“ před komentář. Pokud je během překladu programu nalezena chyba je příslušný řádek s chybou označen světle zeleně a podtržen. Tím je usnadněno nalezení chyby.

5. **Ovládací prvek ListBox**. Slouží pro zobrazení chybových hlášek vzniklých při kompilaci programu. Tyto chybové hlášky byly navrženy tak, aby co možná nejvíce vystihovaly druh chyby a způsob její nápravy. Dvoj klikem na příslušnou chybu se nastaví kursor textového pole [část 4] na řádek který chybu obsahuje a uživatel jí může opravit.

6. Ovládací prvek **tlačítko Registry**. Po stisku způsobí vypisování hodnot registrů do ovládacího prvku ListBox [část 3].

7. Ovládací prvek **tlačítko Paměť**. Po stisku způsobí vypisování hodnot paměti do ovládacího prvku ListBox [část 3].

8. Ovládací prvek **tlačítko Vstupní porty**. Po stisku způsobí vypisování hodnot vstupních portů do ovládacího prvku ListBox [část 3].

9. Ovládací prvek **tlačítko Výstupní porty**. Po stisku způsobí vypisování hodnot výstupních portů do ovládacího prvku ListBox [část 3].

10. **Ovládací prvky tlačítka**. Jde o 8 nastavitelných tlačítek (viz. Obr 3.1.1 nastavení tlačítek), které po stisku způsobí přivedení 5V (logická 1) na příslušný bit (0-7) vstupního portu ke kterému jsou tlačítka připojeny. Pokud je tlačítko nastaveno jako spínací spínač zůstane po kliknutí v sepnutém stavu a na portu bude příslušný bit neustále v logické 1. Při nastavení rozpínací spínač je logická 1 nastavena v rozepnutém stavu tlačítka. Pokud je tlačítko v režimu tlačítko je logická hodnota 1 respektive 0 na příslušném bitu portu pouze při stisknutí respektive rozepnutí. Podle nastavení spínací/rozpínací.

11. Ovládací prvek tlačítko **Zapni** respektive **Vypni Generátor.** Po stisknutí se na příslušném portu podle nastavení začne generovat obdélníkový periodický signál. A popisek tlačítka se změní na Vypni Generátor. Po opakovaném stisku se tlačítko vymáčkne a popisek se změní na Zapni Generátor. Na port přestane přicházet signál.

12. **Ovládací prvek diody**. Jde o 16 nastavitelných led diod sloužících jako virtuální signalizace. V nastavení lze nastavit způsob zapojení diod. Konec diody je připojen na 0V nebo 5V a začátek na příslušný bit nastaveného portu. Podoba těchto prvků je realizována ve funkci *OnPaint()* třídy pohledu *CSim_x51View*.

13. **Ovládací prvek Sedmisegmentovka**. Slouží pro zobrazení číslice složené ze sedmi segmentů viz obr 3.1. V nastavení tohoto prvku lze nastavit port ke kterému je připojen. Dále volba způsobu připojení umožňuje připojit jednotlivé segmenty přímo k výstupním portům (volba přímo), nebo před segmentovku umístit konvertor BCD na 7segment (volba BCD→7segment). Tento konvertor převádí dekadickou číslici na příslušném portu na signály odpovídající příslušnému číslu pro zobrazení na sedmisegmentovce. Vizuální podoba je realizována ve funkci *OnPaint()* třídy pohledu *CSim_x51View*.

3.2 Realizace modelu mikroprocesoru

Model mikroprocesoru je realizován objektem *CuProcesor*, který je deklarován v souborech uProcesor.h a procesor.cpp. Jde o následující strukturu :

```
class CuProcesor
{
public:
        unsigned char A;
        unsigned char R0;
        unsigned char R1;
        unsigned char R2;
        unsigned char R3;
        unsigned char R4;
        unsigned char R5;
        unsigned char R6;
        unsigned char R7;

        TYPE_PSW_REGISTER       PSW;
        TYPE_DPTR_REGISTER      DPTR;
        TYPE_SP_REGISTER        SP;
        unsigned short PC;

        unsigned int T1;
        unsigned int T2;

        unsigned char in[POCET_VSTUPU];
        unsigned char out[POCET_VYSTUPU];

        unsigned char data[DAT_PAMET];

        CuProcesor();
        Init();
        ~CuProcesor();
        int GetRegisterByID(int id);
        void SetRegisterBy_2ID(int id1,int id2);
        void SetRegisterBy_ID(int id,unsigned int data);
};
```

Registr **PSW** je realizován unií *TYPE_PSW_REGISTER*. Tuto unii tvoří struktura PSWBITS (ta umožňuje snadno nastavit bit *Carry* a *Zero*) a unsigned char. Unie byla volena z důvodů zjednodušení práce s registrem PSW, neboť k němu můžeme přistupovat jako k 8-bitovému registru nebo pracovat jen s jeho bity C a Z.

Registry **DPTR** a **SP** (viz. deklarace) jdou realizovány taktéž unií, neboť jde o 16-bitové registry složené ze dvou 8-bitových. Pracuje se s nimi jak 16-bitově tak pomocí 8-bitových registrů DPL, DPH, SPL a SPH.

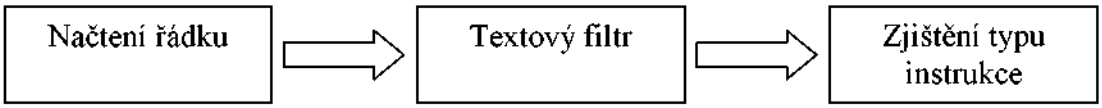
Deklarace registrů PSW, DPTR a SP:

```
struct PSWBITS
{
        unsigned C: 1;
        unsigned notUse: 6;
        unsigned Z: 1;
};
union TYPE_PSW_REGISTER
{
        unsigned char   PSW;
        PSWBITS         PSWbits;
}

struct REG16
{
        unsigned char L;
        unsigned char H;
};

union TYPE_DPTR_REGISTER
{
        unsigned short   fullDPTR;
        REG16            DP;
};

union TYPE_SP_REGISTER
{
        unsigned short   fullSP;
        REG16            SP;
};
```

- Funkce *GetRegisterByID(int id)* vrací podle identifikátoru id hodnotu v příslušného registru. Definice identifikátorů obsahuje soubor *instrukcni_kody.h*.
- Funkce *SetRegisterBy_2ID(int id1,int id2)* uloží do registru daného id1 hodnotu v registru určeném identifikátorem id2.
- Funkce *SetRegisterBy_ID(int id,unsigned int data)* uloží do registru daného identifikátorem id 8-bitovou hodnotu.

3.3 Analýza instrukce

Jelikož instrukce assembleru jsou jednořádkové probíhá analýza programového kódu po jednotlivých řádcích. Struktura algoritmu je zobrazena na obr. 3.2-1.



Obr. 3.2-1 princip analýzy instrukce

Daný řádek programu je načten z ovládacího prvku RitchEdit funkcí *GetLine*. Takto získaný řetězec znaků je předán jako parametr do funkce *prepareCheckSyntax*. Ta filtruje vstupní řetězec znaků a takto upravený řetězec vrací jako výstup. Její činnost spočívá v následujících krocích:

- Všechny znaky tabulátor nahradí mezerami.

  replaceChar(řetězec,znak tabulátor, znak mezera)
- Znak konec řádků (\n nebo 13) nahradí znakem konec řetězce (tj. 0) .

  replaceChar(řetězec, 13, 0)
- Pokud je nalezeno více mezer za sebou jsou nahrazeny jednou.

  skip(řetězec, znak mezera)
- Odstranění poznámek ze zpracovávaného řádku programu. Poznámka je uvozena znaky „//“ a pokračuje až do konce řetězce.

Takto upravená instrukce vstupuje jako parametr do funkce *checkSyntax*, která zajišťuje kontrolu syntaxe daného řádku programu. Tato funkce rozpozná jednotlivé části instrukcí. Rozpoznání je založeno na znalosti možností zápisu instrukcí. Aby byla dosažena nezávislost rozpoznání programu na velikosti písmen převádí se identifikovaný řetězec znaků na velká písmena. Činnost této funkce názorně vysvětluje obr. 3.2-2.

Přehled tvaru zápisu programového kódu:

návěští: jméno_instrukce první_operand, druhý_operand, třetí operand

návěští:

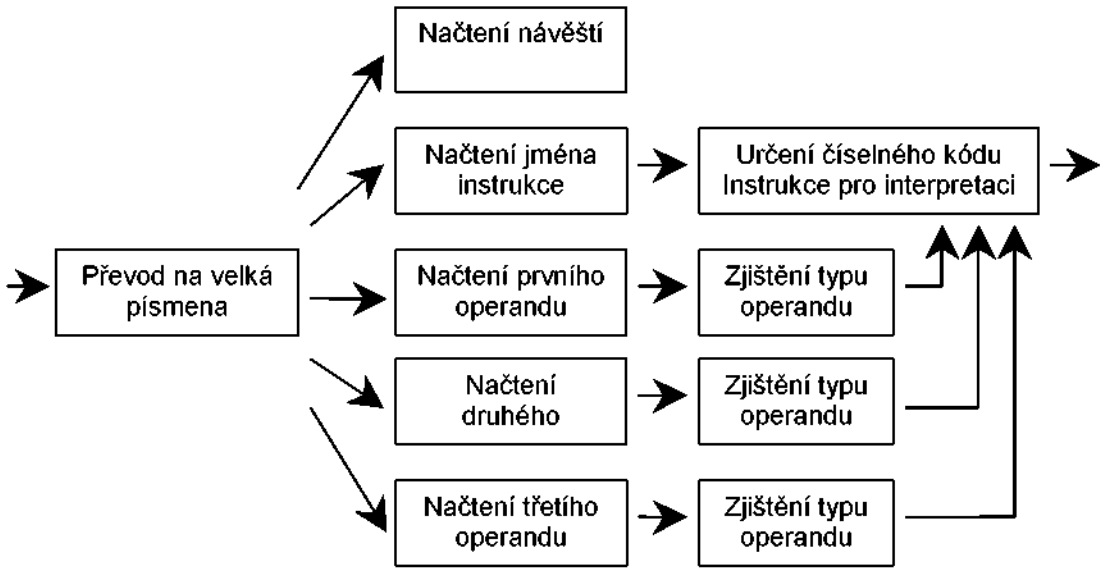
- může obsahovat libovolné znaky zakončené dvojtečkou.

jméno instrukce:

- instrukce je označena dvou až čtyř písmenným identifikátorem. A jsou to MOV, PUSH, POP, IN, OUT, CPL, ANL, ORL, XRL, RR, RRC, RL, RLC, INC, DEC, ADD, ADDC, SUBB, JMP, JZ, JNZ, JC, JNC, CJNE, DJNZ, CALL, RET, NOP

operand:

- počet operandů má model 18 a jsou to A, R0, R1, R2, R3, R4, R5, R6, R7, DPTR, DPL, DPH, SP, SPL, SPH, PSW, T1 a T2. Pro svojí činnost překladač rozpoznává ještě druhy Adresa8, Adresa16, Data8, Data16, Data32
- počet operandů a jejich možnosti pro dané instrukce jsou rozepsány v příloze: Soubor instrukcí pro osmibitový mikroprocesor řady x51.



Obr 3.2-2: Činnost funkce checkSyntax

Funkce zjištění typu operandu provádějí převod znakového vyjádření operandu na číselnou konstantu, která ho zastupuje. Z vrácené hodnoty můžeme poznat o jaký operand jde, nebo zda je špatně zapsán. Tyto zjištěné hodnoty vstupují do vůbec nejdůležitější části programu a tou je funkce *checkInstruction*. Ta zjišťuje jestli zpracovávaný řádek programu je nějaká povolená kombinace instrukce a jejich

operandů. Pokud je nějaká taková možnost nalezena, vrací funkce číselný kód specifický pro danou kombinaci uvedených parametrů. Není-li shoda nalezena vrací funkce záporný chybový kód. Chybové kódy a jejich hodnoty jsou definovány v hlavičkovém souboru *instrukcni_kody.h*. Názvy těchto kódů byly voleny tak aby z nich bylo na první pohled jasné o jakou chybu v zápisu instrukce se jedná a simulátor pak mohl zobrazit co nejpřesnější nápovědu.

3.4 Vytvoření tabulky skoků

Vytvoření tabulky skoků se provádí před vlastní kontrolou syntaxe. Je to nutné z toho důvodu, aby simulátor mohl kontrolovat chyby vznikající opakovanou deklarací stejného návěští nebo chybou v překlepu jeho názvu. Jelikož jde o simulátor určený pro tvorbu krátkých programů je vytvoření tabulky uděláno tak, že cyklem projde všechny řádky programu a hledá v nich deklaraci návěští. Je-li nalezeno porovnává se s již nalezenými návěští pro kontrolu opakované deklarace. Není-li shoda nalezena je nové jméno návěští uloženo na konec tabulky. Velikost tabulky je určena konstantou *POCET_SKOKU* a nastavena na hodnotu 1024. Takto velká hodnota by měla zaručit dostatečně velkou tabulku pro jakýkoliv program který bude v tomto simulátoru napsán.

3.5 Interpretace instrukce

Interpretační část byla udělána tak, že po stisku tlačítka Spustit, se objekt modelu mikroprocesoru nastaví do výchozího stavu. To provede funkce *Init()*. Dále se spustí časovač ID_TIMER_EXECUTE, který po uplynutí nastavené doby aktivuje funkci *OnTimer*, která zpracuje jeden řádek kódu napsaného programu. Funkce *OnTimer* pracuje tak, že zavolá funkci *Execute* ze souboru Interpret.cpp. Funkce *Execute* používá objekt *COMMANDLINE*, který obsahuje informace o zpracovávané instrukci zjištěné v analýze. Tento objekt slouží jako parametr pro funkci *DoCommand*. Ta se stará o vykonání zpracovávané instrukce podle hodnot v objektu *COMMANDLINE*. Po provedení instrukce se zvětší počítadlo zpracovávaného řádku *lineCounter* o jedna a pomocí funkce *UpdateControlsFromDoc* se aktualizují zobrazené hodnoty modelu mikroprocesoru.

4. Popis jednotlivých souborů programu

Sim_x51.cpp a Sim_x51.h :

Obsahují objekt třídy *CSim_x51App*, který reprezentuje aplikaci. V souboru Sim_x51.Cpp je definovaný jediný globální objekt *theApp* třídy *CSim_x51App*. Chování objektu *theApp* určuje převážně základní třída *CWinApp*, od které dědí většinu vlastností. Tento objekt obsahuje jedinou členskou funkci a to *InitInstance*, která provede volání potřebná k vytvoření a zobrazení hlavního rámcového okna aplikace.

MainFrm.cpp a MainFrm.h :

Obsahují objekt třídy Mainframe. Ten reprezentuje hlavní rámcové okno aplikace. Obsahuje konstruktor, který zavolá funkci *Create* základní třídy *CFrameWnd*. Windows poté vytvoří vlastní strukturu okna a aplikační systém ji spojí s objektem C++. Funkce *ShowWindow* a *UpdateWindow*, které jsou taká členskými funkcemi základní třídy, musí být zavolány pro zobrazení okna.

Sim_x51Doc.cpp a Sim_x51Doc.h :

Tyto soubory obsahuji objekt *CSim_x51Doc*, který vychází z objektu *CDocument*. Tento objekt obsahuje data aplikace, což v našem případě je model mikroprocesoru, který reprezentuje objekt *CuProcesor*.

Sim_x51View.cpp a Sim_x51View.h :

Obsahují objekt CSim_x51View, který tvoří vizuální interpretaci a reprezentaci dokumentu na obrazovce. Slouží také k překládání uživatelského vstupu (konkrétně zpráv myši a klávesnice) do příkazů, které operují nad daty dokumentu. Tento objekt je odvozen od základní třídy *CFormView*. Obsahuje vizuální prvky CButton, CListbox, CComboBox, CRichEditCtrl a dále funkce obsluhující uživatelské akce (podrobnosti viz. uvedené soubory na CD přiloženém k diplomové práci).

uProcesor.cpp a uProcesor.h :

V těchto souborech je definována třída modelu mikroprocesoru *CuProcesor*. Obsahuje funkce:

- konstruktor *CuProcesor()* nastavuje PSW.PSWbits.Z na 1 a hodnoty ostatních proměnných modelu na 0.
- void Init() vykonává podobnou činnost jako konstruktor až na to, že nevynuluje hodnoty vstupních portů.
- *int GetRegisterByID(int id)* která vrací podle identifikátoru id hodnotu v příslušného registru. Definice identifikátorů obsahuje soubor *instrukcni_kody.h*.
- *void SetRegisterBy_2ID(int id1,int id2)* která uloží do registru daného id1 hodnotu v registru určeném identifikátorem id2.
- *void SetRegisterBy_ID(int id,unsigned int data)* která uloží do registru daného identifikátorem id 8-bitovou hodnotu.

Kompilator.cpp a Kompilator.h :

V těchto souborech, jsou umístěny funkce, které mají nějakou souvislost s realizací kompilátoru. Mezi hlavní patří tyto:

- *char *prepareCheckSyntax(char *sz)* filtruje vstupní řetězec znaků a takto upravený řetězec vrací jako výstup. Její činnost spočívá v následujících krocích:
    - Všechny znaky tabulátor nahradí mezerami.
    - Znak konec řádků (\n nebo 13) nahradí znakem konec řetězce (tj. 0) .
    - Pokud je nalezeno více mezer za sebou jsou nahrazeny jednou.
    - Odstranění poznámek ze zpracovávaného řádku programu. Poznámka je uvozena znaky „//“ a pokračuje až do konce řetězce.
- *int isAdresa(char *sz,int base,int *nOut)* zjišťuje jestli řetězec sz obsahuje operand op_Adresa8 nebo op_Adresa16 podle parametru base (8,16). Podrobnější detaily viz. poznámky v souboru kompilator.cpp před funkcí *isAdresa*.
- *int isData(char *sz,int base,unsigned int *nOut)* zjišťuje jestli řetězec sz obsahuje operand op_Data8, op_data16 nebo op_Data32 podle parametru base (8,16,32). Podrobnější detaily viz. poznámky v souboru kompilator.cpp před funkcí *isData*.

- *int checkOp1(char *szOp)* zkontroluje první operand instrukce. Vrácená hodnota je buť konstanta operandu (kladná hodnota) nebo kód chyby (záporná hodnota). Přesný popis vracených hodnot viz. poznámky v souboru kompilátor.cpp před funkcí *checkOp1*.
- *int checkOp2(char *szOp)* zkontroluje druhý operand instrukce. Detaily viz. funkce *checkOp1* a poznámky v souboru kompilátor.cpp před funkcí *checkOp2*.
- int checkOp3(char *szOp) zkontroluje třetí operand instrukce. Detaily viz. funkce *checkOp1* a poznámky v souboru kompilátor.cpp před funkcí *checkOp3*.
- *int checkInstruction(char *szInst,int op1,int op2, int op3)* z parametru *szInst* a kódů operandů *op1*, *op2* a *op3* určí číselnou konstantu, která udává druh instrukce zpracovávaného řádku kódu (parametr szInst). Definice hodnot konstant viz. soubor instrukcni_kody.h.
- *int checkSyntax(char *szPrikaz)* analyzuje řetězec *szPrikaz* a vrací pomocí funkcí *checkOp1*, *checkOp2*, *checkOp3* a *checkInstruction* konstantu která udává výsledek analýzy. Definice hodnot konstant viz. soubor instrukcni_kody.h.
- *int readJumps(char *szPrikaz,int *adrCount,int *lineCount)* vytvoří tabulku skoků pro simulaci programu. Tabulka ja obsažena v globálním poli *skoky[POČET_SKOKU]* definovaném v souboru kompilator.cpp.
- void *Show_errHint(int err,CListBox *pLB,char *sz,int line)* podle hodnoty err vypíše do ListBoxu (*pLB*) příslušnou chybovou hlášku. Proměnná *sz* obsahuje řetězec instrukce, kde se chyba vyskytla. V parametru *line* je číslo řádky na které byla chyba nalezena.
- int compile(CRichEditCtrl *pRitchEdit,CListBox *pLB) zkontroluje pomocí výše uvedených funkcí syntaxi celého programu. Kladná vrácená hodnota udává typ správně zapsané instrukce. Záporná označuje druh nalezené chyby.

Intepret.cpp a Interpret.h :

V těchto souborech, jsou umístěny dvě funkce, které realizují interpretaci instrukcí. Jsou to funkce :

- *void Execute(CSim_x51Doc* pDoc,CRichEditCtrl *pRE,int *line)* načte řádek udaný parametrem *line* z RitchEditu, na který ukazuje *pRE*. Tento řádek analyzuje a zjištěnými údaji naplní objekt *cmd* typu *COMMANDLINE*. Poté zavolá funkci *DoCommand* s parametry *pDoc*, *&cmd* a *line*.
- *void DoCommand(CSim_x51Doc* pDoc,COMMANDLINE * cm,int *line)* upraví model mikroprocesoru podle instrukce dané objektem cm.

Je tu také definován objekt COMMANDLINE, který je použit pro uložení jednotlivých částí programového řádku.

MyButton.cpp a MyButton.h :

Tyto soubory obsahují objekt CMyButton, odvozený od objektu CButton, který je použit pro realizaci tlačítek s obrázkem spínače [viz. kapitola 3.1 bod 10]. Objekt CMyButton přepisuje tři virtuální funkce objektu CButton a jsou to *OnLButtonDown*, *OnLButtonUp* a *OnClicked*. Tyto funkce realizují změnu zobrazené bitmapy podle stavu sepnutí tlačítka a jeho nastavení. Nahraní bitmapy na tlačítko realizuje funkce *SetBitmap(LoadBitmap(GetModuleHandle(NULL),MAKEINTRESOURCE(id)))*.
Parametr *id* udává označení příslušné bitmapy vložené do souboru zdrojů (sim_x51.rc).

Instrukcní_kody.h :

Tento soubor obsahuje deklarace konstant použitých v programu. Jsou to kódy instrukcí, kódy operandů, návratové hodnoty některých funkcí a další.

NastaveníTlacitek.cpp a NastaveniTlacitek.h :

Tyto soubory realizují výměnu dat mezi pohledem CSim_x51View a dialogem pro nastavení tlačítek (*IDD_NASTAVENI_TLACITEK*).

NastaveníGenerator.cpp a NastaveniGenerator.h :

Tyto soubory realizují výměnu dat mezi pohledem CSim_x51View a dialogem pro nastavení tlačítek (*IDD_ GENERATOR*).

NastaveníLED.cpp a NastaveniLED.h :

Tyto soubory realizují výměnu dat mezi pohledem CSim_x51View a dialogem pro nastavení led diod (*IDD_NASTAVENI_LED*).

Nastavení7Segment.cpp a Nastavení7Segment.h :

Tyto soubory realizují výměnu dat mezi pohledem CSim_x51View a dialogem pro nastavení sedmisegmentovky (*IDD_SEMENTOVKA*).

NastavPamet.cpp a NastavPamet.h :

Tyto soubory realizují výměnu dat mezi pohledem CSim_x51View a dialogem pro nastavení paměti (*IDD_NASTAV_PAMET*).

5. Závěr

Cílem bodu 1 zadání bylo vytvořit překladač zjednodušeného assembleru, který se vyučuje v rámci předmětu Číslicové počítače, jehož výstupem bude binární soubor nebo přehled syntaktických chyb. Část překladače byla plně začleněna do programu simulátor. Z tohoto důvodu nebylo nutné vytvářet binární soubor. Příslušné zjištěné informace o kontrole syntaxe či analýze jednotlivých instrukcí se ihned vyhodnocují. Překladač nebyl studenty při výuce testován. Z tohoto důvodu nelze s jistotou říci zda-li neobsahuje nějaké neodhalené logické chyby při vyhodnocení syntaxe.

Cílem bodu 2 zadání bylo naprogramovat interpret, jehož vstupem bude binární soubor z překladače assembleru. Jelikož překladač nevytváří binární soubor překladu, byl stejně jako kompilátor začleněn do programu simulátor. Údaje potřebné pro interpretaci jsou získávány přímo z kompilátoru. Vykonávání jednotlivých instrukcí obstarává model (kapitola 1.3). Tento model byl navržen podle zadání a plně dostačuje pro vykonání všech instrukcí.

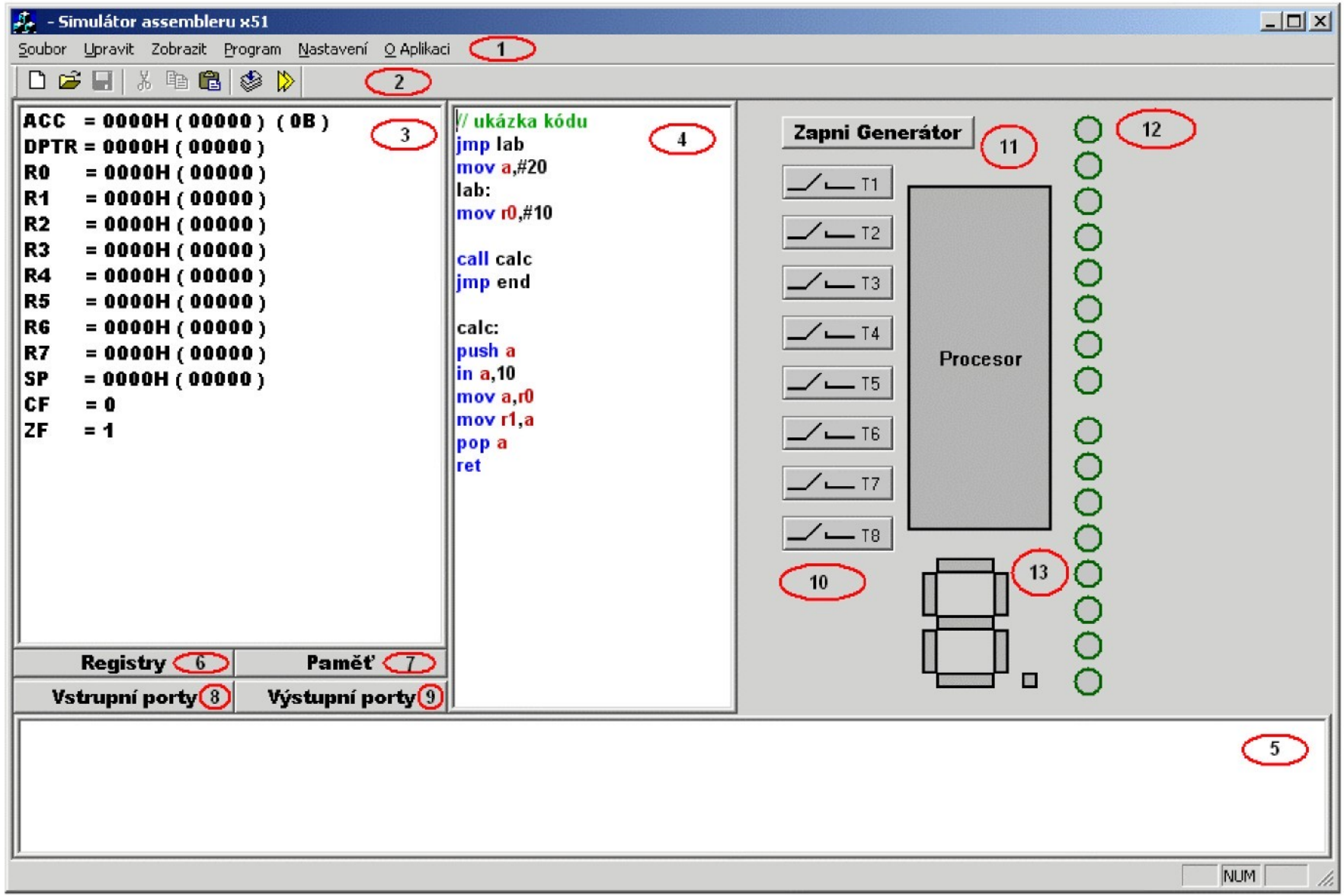
Cílem bodu 3 bylo z částí 1 a 2 vytvořit program simulátoru, který bude sloužit jako pomůcka pro výuku assembleru. Simulátor obsahuje pro editaci a úpravu zdrojového textu se zvýrazněnou syntaxí komponentu textové pole. Pro překlad a interpretaci má simulátor tlačítka Přeložit, Spustit a Ukončit běh.

Cílem bodu 4 bylo v simulátoru vytvořit přehledné zobrazení obsahu paměti a jednoduchých periferií. To bylo splněno a jako periferie bylo vytvořeno 16 led diod, 8 tlačítek a jeden sedmisegmentový displej.

Seznam literatury:

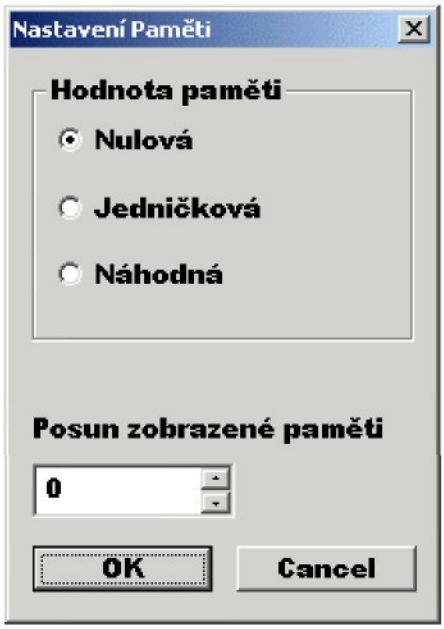
1. Radek Chalupa: 1001 Tipů a triků pro Visual C++. Computer Press, Brno 2003, ISBN 80-7226-842-2
2. Jeff Prosise: Programování ve Windows pomoci MFC. Computer Press, Praha 2000, ISBN 80-7226-309-9
3. David J. Kruglinski, Georgie Shepherd, Scot Wingo: Programujeme v microsoft Visual C++. Computer Press, Praha 2000, ISBN 80-7226-362-5
4. Herbert Schildt: Nauč se sám C. SoftPress, Praha 2001, ISBN 80-86497-16-X
5. Herbert Schildt: Nauč se sám C++. SoftPress, Praha 2001, ISBN 80-86497-13-5
6. S. Ferguson: The history of Computeer Programing Languages. [online]. 2004. Dostupné na www: http://www.princeton.edu/~ferguson/adw/programing_languages.shtml
7. Historie a vznik počítačů. www: http://utf.mff.cuni.cz/win/vyuka/OFY016/F2001/Hola/referat.html

Příloha A - Popis vizuálního prostředí simulátoru



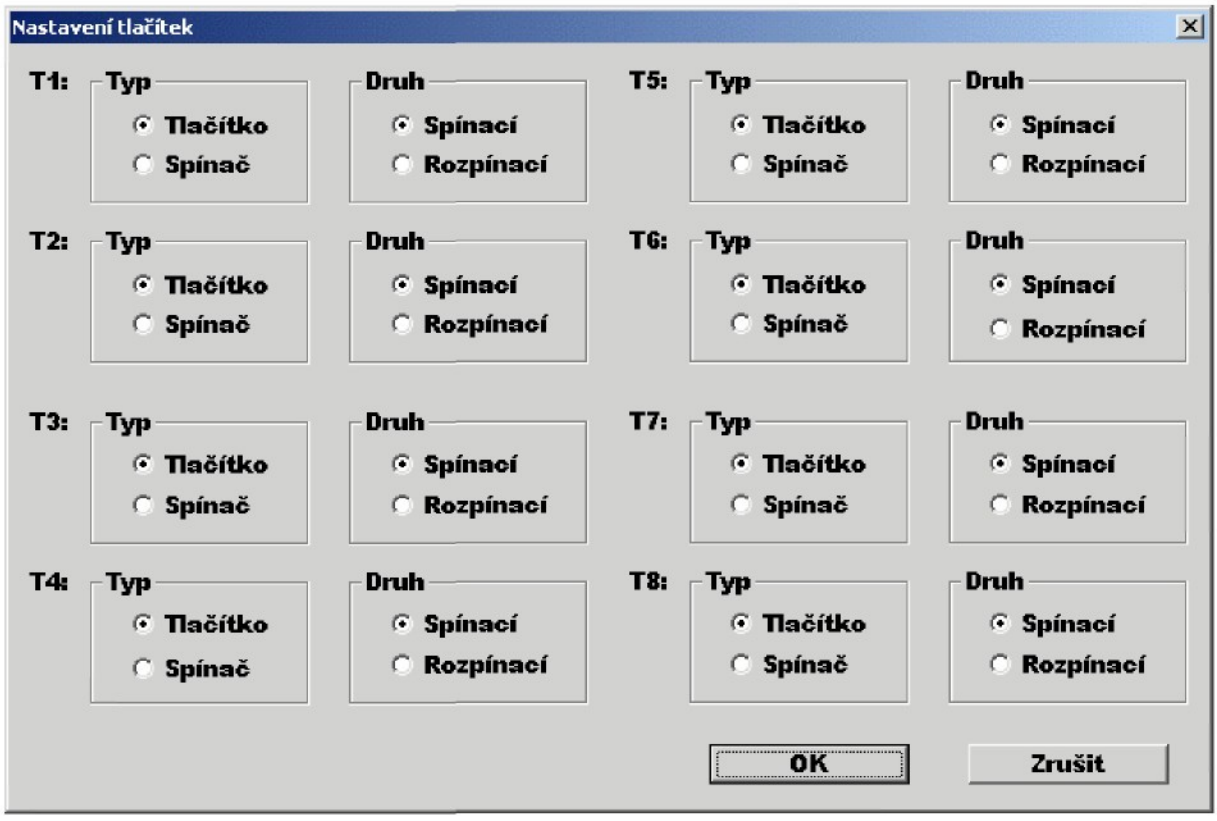
1. **Menu simulátoru**. Menu je rozděleno na části Soubor, Upravit, Zobrazit, Program, Nastavení a O Aplikaci.
   - položka **Soubor** (klávesová zkratka ALT+S) obsahuje možnosti Nový, Otevřít, Uložit, Uložit jako a Konec
     - položka **Nový** slouží k vytvoření nového programu. Vymaže se obsah textového pole [část 4]. Hodnoty registrů, výstupní a vstupní porty a datová paměť se vynuluje. Nastavení tlačítek led-diod a generátoru se vrátí do výchozího nastavení.
     - položka **Otevřít** umožní načtení programu ze souboru do textového pole [část 4].
     - položka **Uložit** a **Uložit jako** slouží k uložení programu z textového pole [část 4] do souboru. Přípona souboru je defautně nastavena na .asm.
     - položka Konec ukončí simulátor.
   - položka **Upravit** (klávesová zkratka ALT+U) obsahuje možnosti Vyjmout, Kopírovat a Vložit. Tyto položky slouží pro vkládání a vyjímání textu z a do textového pole [část 4] přes schránku Windows.
     - položka **Vyjmout** je aktivní pouze pokud je v textovém poli [část 4] označen nějaký text. A slouží k vyjmutí označeného textu a jeho vložení do schránky Windows.
     - položka **Kopírovat** je aktivní pouze pokud je v ovládacím prvku textové pole [část 4] označen nějaký text a zkopíruje ho do schránky Windows.
     - položka **Vložit** je aktivní pouze pokud je ve schránce Windows nějaký text ke vložení a tento text vloží do textové pole [část 4] na pozici kursoru.
   - položka **Zobrazit** (klávesová zkratka ALT+Z) obsahuje možnosti Registry (F5), Paměť (F6), Vstupní porty (F7) a Výstupní porty (F8). Tyto položky slouží jako přepínač který rozhoduje o tom co bude zobrazeno v panelu výpisu [část 3].
   - položka **Program** (klávesová zkratka ALT+P) obsahuje možnosti Přeložit, Spustit a ukončit beh..
     - položka **Přeložit** zkompiluje program napsaný v textovém poli [část 4]. Pokud jsou v programu chyby zobrazí se v panelu chyb [část 5] příslušná zpráva informující o druhu chyby.
     - položka **Spustit** provede simulaci programu na virtuálním modelu [kapitola 1.3]. Před vlastní simulací provede ještě kompilaci a pokud nebyla nalezena chyba simuluje program.
     - Položka **Ukončit běh** ukončí běh simulace programu.

- položka **Nastavení** (klávesová zkratka ALT+N) obsahuje možnosti Paměť, Tlačítka, Generátor, Led diody a Sedmisegmentovka. Tato nabídka slouží pro individuální nastavení simulátoru.
  - Položka **Paměť** vyvolá dialog nastavení paměti. V něm je možné nastavit hodnoty paměti na nula,jedna nebo náhodné. Dále je možné nastavit od které paměťové buňky se bude zobrazovat obsah paměti v panelu výpisu [část 4]. Podoba dialogu je na obr. 3.1-1.
  - položka **Tlačítka** vyvolá dialog nastavení osmi tlačítek [část 10] obr 3.1-2. Volba typu tlačítka určuje zda tlačítko zůstane po stisknutí dole (volba Spínač) nebo ne (volba Tlačítko). Volba druh určuje zda bude tlačítko spínací nebo rozpínací.
  - položka **Generátor** vyvolá dialog nastavení generátoru signálu [část 11]. Podoba dialogu je na obr 3.1-3. V tomto dialogu je možné nastavit frekvenci generování signálu a číslo portu na který je připojen. Generovaný signál má obdélníkový tvar s danou frekvencí.
  - položka **Led diody** vyvolá dialog nastavení 16ti diod [část 12]. Podoba dialogu je na obr 3.1-4. V tomto dialogu je možné nastavit 2 porty, ke kterým budou diody připojeny. Dále je možno vybrat zda budou diody připojeny na 5V či na 0V.
  - položka **Sedmisegmentovka** obr. 3.1-5 vyvolá dialog s nastavením sedmisegmentového displeje. Dialog obsahuje nastavení čísla portu připojení a způsob připojení. Způsobem připojení je myšleno zda jednotlivé segmenty displeje jsou připojeny přímo k danému portu nebo zda-li je mezi porty a displejem zapojen konvertor BCD->7segment. Tento konvertor převádí dekadickou číslici na signály odpovídající příslušnému číslu pro zobrazeni na sedmisegmentovce.
  - položka **O Aplikaci** (klávesová zkratka ALT+O) vyvolá dialog s detaily o verzi aplikace, jménu autora atd..

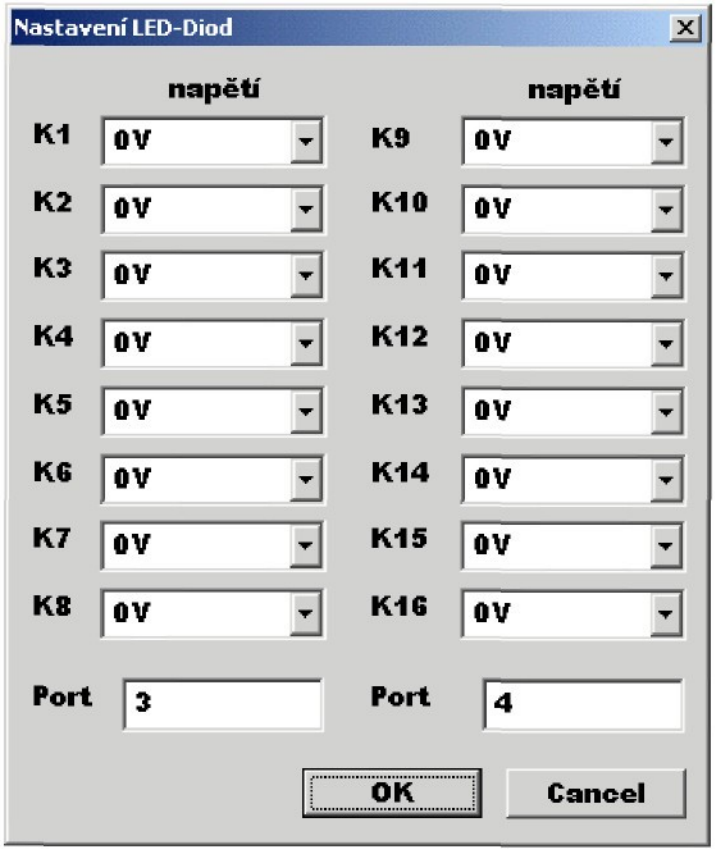


Obr 3.1-1: Dialog nastavení paměti

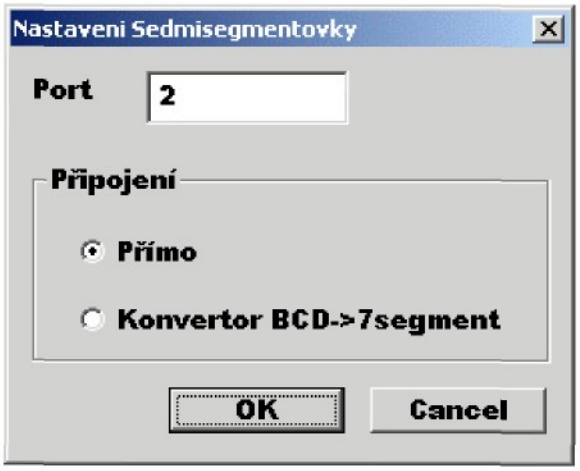
Obr 3.1-3: Dialog nastavení generátoru



Obr 3.1-2: Dialog nastavení tlačítek



Obr 3.1-4: Dialog nastavení LED Diod



Obr 3.1-5: Dialog nastavení Sedmisegmentovky

2. **Panel nástrojů.** Umožňuje rychle použít vybrané volby menu. Obsahuje tlačítka Nový, Otevřít, Uložit, Kopírovat, Vyjmout, Vložit, Přeložit, Spustit a přepínače zobrazeni Registry, Paměť, Vstupní a Výstupní porty.

3. **Panelu výpisu stavů modelu mikroprocesoru**. Slouží pro zobrazení hodnot modelu mikroprocesoru. Pro výběr zobrazení slouží položka menu→Zobrazení, ikony na panelu nástrojů nebo tlačítka označená na obr 3.1 čísly 6,7,8 a 9.

4. Ovládací prvek **textové pole**. Tento ovládací prvek slouží jako textové pole pro psaní a úpravu programu. Obsahuje zvýraznění syntaxe příkazů (modrá barva), jmen registrů (červená barva) a komentářů (zelená barva). Pro označení komentáře v programu slouží vložení znaků „//“ před komentář. Pokud je během překladu programu nalezena chyba je příslušný řádek s chybou označen světle zeleně a podtržen. Tím je usnadněno nalezení chyby.

5. Ovládací prvek **panel chyb**. Slouží pro zobrazení chybových hlášek vzniklých při kompilaci programu. Tyto chybové hlášky byly navrženy tak, aby co možná nejvíce vystihovaly druh chyby a způsob její nápravy. Dvoj klikem na příslušnou chybu se nastaví kursor textového pole [část 4] na řádek který chybu obsahuje a uživatel jí může opravit.

6. Ovládací prvek **tlačítko Registry**. Po stisku způsobí vypisování hodnot registrů do ovládacího prvku panel výpisu [část 3].

7. Ovládací prvek **tlačítko Paměť**. Po stisku způsobí vypisování hodnot paměti do ovládacího prvku panel výpisu [část 3].

8. Ovládací prvek **tlačítko Vstupní porty**. Po stisku způsobí vypisování hodnot vstupních portů do ovládacího prvku panel výpisu [část 3].

9. Ovládací prvek **tlačítko Výstupní porty**. Po stisku způsobí vypisování hodnot výstupních portů do ovládacího prvku panel výpisu [část 3].

10. **Ovládací prvky tlačítka**. Jde o 8 nastavitelných tlačítek (viz. Obr 3.1.1 nastavení tlačítek), které po stisku způsobí přivedení 5V (logická 1) na příslušný bit (0-7) vstupního portu ke kterému jsou tlačítka připojeny. Pokud je tlačítko nastaveno jako spínací spínač zůstane po kliknutí v sepnutém stavu a na portu bude příslušný bit neustále v logické 1. Při nastavení rozpínací spínač je logická 1 nastavena v rozepnutém stavu tlačítka. Pokud

je tlačítko v režimu tlačítko je logická hodnota 1 respektive 0 na příslušném bitu portu pouze při stisknutí respektive rozepnutí. Podle nastavení spínací/rozpínací.

11. Ovládací prvek tlačítko **Zapni** respektive **Vypni Generátor.** Po stisknutí se na příslušném portu podle nastavení začne generovat obdélníkový periodický signál. A popisek tlačítka se změní na Vypni Generátor. Po opakovaném stisku se tlačítko vymáčkne a popisek se změní na Zapni Generátor. Na port přestane přicházet signál.

12. **Ovládací prvek diody**. Jde o 16 nastavitelných led diod sloužících jako virtuální signalizace. V nastavení lze nastavit způsob zapojení diod. Konec diody je připojen na 0V nebo 5V a začátek na příslušný bit nastaveného portu.

13. **Ovládací prvek Sedmisegmentovka**. Slouží pro zobrazení číslice složené ze sedmi segmentů. V nastavení tohoto prvku lze nastavit port ke kterému je připojen. Dále volba způsobu připojení umožňuje připojit jednotlivé segmenty přímo k výstupním portům (volba přímo), nebo před segmentovku umístit konvertor BCD na 7segment (volba BCD→7segment). Tento konvertor převádí dekadickou číslici na signály odpovídající příslušnému číslu pro zobrazení na sedmisegmentovce.

Příloha B - Soubor instrukcí pro 8-bitový mikroprocesor řady x51

Přesun dat

MOV

Obecný tvar MOV kam,odkud

Přesune byte (word) z místa „odkud“ na místo „kam“. Pro mikroprocesor jsou povoleny tyto kombinace:

1. mov A, R_n
2. mov A, DPH
3. mov A, DPL
4. mov A, SPH
5. mov A, SPL
6. mov R_n, A
7. mov DPH, A
8. mov DPL, A
9. mov SPH, A
10. mov SPL, A
11. mov A, #Data8
12. mov R_n, #Data8
13. mov DPTR, #Data16
14. mov SP, #Data16
15. mov A, Adresa8
16. mov A, @R_n
17. mov @R_n, A
18. mov A, @DPTR
19. mov @DPTR, A
20. mov Adresa8, A
21. mov T1, #Data32
22. mov T2, #Data32

Operand R_n znamená libovolný z registrů R0 až R7.

PUSH

Obecný tvar PUSH co

Uloží „co“ na zásobník. Nejprve zvětší obsah registru SP o 1 a poté uloží „co“ na adresu v SP.

1. push A
2. push R_n
3. push PSW

Nemění příznaky.

POP

Obecný tvar POP co

Načte do „co“ vrchol zásobníku, poté sníží obsah registru SP o 1.

1. pop A
2. pop R_n
3. pop PSW

Nemění příznaky.

IN

Obecný tvar IN kam, odkud

Přečte port určený „odkud“ a výsledek uloží do „kam“.

1. in A, Adresa8

Nemění příznaky.

OUT

Obecný tvar OUT kam, co

Na port „kam“ zapiše „co“.

1. out Adresa8, A

Nemění příznaky.

Logické instrukce

CPL

Obecný tvar CPL co

Provede negaci „co“ a výsledek uloží do „co“.

Operace logická negace (NOT)

1. bit	výsledek
0	1
1	0

1. cpl A

ANL

Obecný tvar ANL kam, co

Provede logický součin „kam“ a „co“. Výsledek uloží do „kam“.

Operace logický součin (AND)

1. bit	2. bit	výsledek
0	0	0
0	1	0
1	0	0
1	1	1

1. anl A, R_n
2. anl A, #Data8
3. anl A, @ R_n
4. anl A, Adresa8

ORL

Obecný tvar ORL kam, co

Provede logický součet „kam“ a „co“. Výsledek uloží do „kam“.

Operace logický součet (OR)

1. bit	2. bit	výsledek
0	0	0
0	1	1
1	0	1
1	1	1

1. anl A, R_n
2. anl A, #Data8
3. anl A, @R_n
4. anl A, Adresa8

XRL

Obecný tvar XRL kam, co

Provede logický exkluzivní součet „kam“ a „co“. Výsledek uloží do „kam“.

Operace logický exkluzivní součet (XOR)

1. bit	2. bit	výsledek
0	0	0
0	1	1
1	0	1
1	1	0

1. xrl A, R_n
2. xrl A, #Data8
3. xrl A, @R_n
4. xrl A, Adresa8

Instrukce rotací

RR

Obecný tvar RR co

Posune „co“ o 1 bit doprava. Vypadávající bit dá na místo nultého bitu.

1. RR A

RRC

Obecný tvar RRC co

Posune „co“ o 1 bit doprava. Vypadávající bit dá do C (carry). Původní C dá na místo nultého bitu.

1. RRC A

RL

Obecný tvar RL co

Posune „co“ o 1 bit doleva. Vypadávající bit dá na místo nejvyššího bitu.

1. RL A

RLC

Obecný tvar RLC A

Posune „co“ o 1 bit doleva. Vypadávající bit dá do C (carry). Původní C dá na místo nejvyššího bitu.

1. RLC A

Aritmetické instrukce

INC

Obecný tvar INC co

Zvětší „co“ o 1.

1. inc A
2. inc R_n
3. inc DPTR

Nemění příznaky.

DEC

Obecný tvar DEC co

Zmenší „co“ o 1.

1. dec A
2. dec R_n

Nemění příznaky.

ADD

Obecný tvar ADD kam, co

Přičte ke „kam“ „co“ a uloží do „kam“.

1. add A, R_n
2. add A, #Data8
3. add A, @R_n
4. add A, Adresa8 (16?)

ADDC Obecný tvar ADDC kam,co

Provede aritmetický součet obsahu „kam“, druhého operandu „co“ a jednobitového registru C. Výsledek uloží do „kam“. Devátý bit se ukládá do jednobitového registru C.

1. addc A, R_n
2. addc A, #Data8
3. addc A, @R_n
4. addc A, Adresa8 (16?)

SUBB Obecný tvar SUBB kam, co

Provede aritmetický rozdíl obsahu „kam“, druhého operandu „co“ a jednobitového registru C. Výsledek uloží do „kam“. Je-li výsledek nezáporný, je C = 0 jinak C = 1.

1. subb A, R_n
2. subb A, #Data8
3. subb A, @R_n
4. subb A, Adresa8 (16?)

Instrukce skoků a volání

JMP Obecný tvar JMP adresa

Instrukce předá řízení (nahrání nové adresy do registru PC a následně se vykonávají instrukce od této adresy) na místo určené jako „adresa“. Adresu lze zadat jako číselnou konstantu nebo jako symbolické návěští (např. návěští:). Pro skok na aktuální instrukci můžeme místo návěští použít znak $. Instrukce JMP $ provede skok na sebe a tím vytvoří nekonečnou smyčku.

1. jmp Adresa16

JZ Obecný tvar JZ adresa

(Jump if zero) Provede skok na adresu pokud je jednobitovy registr Z = 1 (tzn. Acc = 0). Jinak se chova jako instukce NOP.

1. jz Adresa16

JNZ Obecný tvar JNZ adresa

(Jump if not zero) Provede skok na adresu pokud je jednobitový registr Z = 0 (tzn. Acc ≠ 0). Jinak se chová jako instrukce NOP.

1. jnz Adresa16

JC Obecný tvar JC adresa

(Jump if carry) Provede skok pokud je jednobitový registr C = 1 (tzn. při poslední matematické operaci došlo k přetečení). Jinak se chová jako instrukce NOP.

1. jc Adresa16

JNC Obecný tvar JNC adresa

(Jump if not carry) Provede skok pokud je jednobitový registr C = 0 (tzn. při poslední matematické operaci nedošlo k přetečení). Jinak se chová jako instrukce NOP.

1. jnc Adresa16

CJNE Obecný tvar CJNE co,hodnota,adresa

(Compare and Jump if Not Equal) Provede porovnání obsahu „co“ s „hodnota“. Pokud „co“ ≠ „hodnota“ provede skok na „adresa“. Jinak se pokračuje další instrukcí. Navíc pokud „co“ < „hodnota“ nastaví jednobitový registr C na 1. Jinak C = 0.

1. cjne A, #Data8, Adresa16
2. cjne R_n, #data8, Adresa16
3. cjne T1, #Data32, Adresa16
4. cjne T2, #Data32, Adresa16

DJNZ Obecný tvar DJNZ co,adresa

(Decrement and Jump if Not Zero) Nejprve sníží obsah „co“ o 1 a pokud je výsledek různý od 0, pak provede skok na zadanou adresu. Jinak pokračuje následující instrukcí.

1. djnz R_n, Adresa16

CALL Obecný tvar CALL adresa

Uloží do zásobníku nejprve nižší a poté vyžší byte adresy instrukce následující za CALL. Potom provede skok na zadanou adresu.

1. call Adresa16

RET Obecný tvar RET

Odebere ze zásobníku nejprve vyšší a potá nižší byte adresy instrukce, které poté předá řízení.

1. ret

Řídící instrukce

NOP Obecný tvar NOP

Tato instrukce říká mikroprocesoru aby jeden strojový takt nic nedělal.

1. nop