SLOVENSKÁ POĽNOHOSPODÁRSKA UNIVERZITA

V NITRE

TECHNICKÁ FAKULTA

2133239

DIPLOMOVÁ PRÁCA

2010 **Ján Horváth, Bc.**

SLOVENSKÁ POĽNOHOSPODÁRSKA UNIVERZITA
V NITRE
TECHNICKÁ FAKULTA

INTERPRETER PROGRAMOV RAM

Diplomová práca

Študijný program:	Informačná a automatizačná technika v kvalite produkcie
Študijný odbor:	5. 2. 57 Kvalita produkcie
Školiace pracovisko:	Katedra elektrotechniky, automatizácie a informatiky
Školiteľ:	doc. Ing. Zuzana Palková, PhD.

Nitra 2010 **Ján Horváth, Bc.**

Čestné vyhlásenie

Podpísaný Ján Horváth vyhlasujem, že som záverečnú prácu na tému „Interpreter programov RAM“ vypracoval samostatne s použitím uvedenej literatúry.

Som si vedomý zákonných dôsledkov v prípade, ak uvedené údaje nie sú pravdivé.

V Nitre 15. marca 2010

Ján Horváth

Poďakovanie

Touto cestou ďakujem predovšetkým doc. Ing. Zuzane Palkovej, PhD. za cenné rady, usmernenie a pripomienky počas tvorby tejto bakalárskej práce a v neposlednom rade za čas strávený výmenou drahocenných informácií, ktoré ma motivovali pri ďalšom písaní. Moje ďakujem patrí aj mojej najbližšej rodine za ich všestrannú podporu, pomoc a trpezlivosť.

Abstrakt

V diplomovej práci sa zaoberáme vytvorením teoretického modelu počítača s ľubovolným prístupom k pamäti (RAM). Vytvorením aplikácie simulujúcej všetky vlastnosti daného virtuálneho zariadenia sme používateľovi priblížili, akým spôsobom toto zariadenie pracuje. Aplikácia navyše poskytuje údaje o hodnotách výpočtových časových a pamäťových zložitostí. Práca obsahuje analýzu teoretického modelu počítača, vizuálnu predstavu aplikácie a návrh komponentov interpretovaného modelu. Ďalej implementáciu aplikácie a následné úpravy spojené s overím funkčnosti zariadenia. Dôležitý bod v práci zastáva testovanie a vyhodnocovanie výsledkov testov. Vytvorením aplikácie sme dokázali, že dané virtuálne zariadenie je možné zostrojiť s jeho presnými špecifikáciami. Vizualizácia približuje používateľom správne chápanie základných princípov práce tohto zariadenia.

Abstract

This research deals with creating abstract model of a computer a random access to the memory (RAM). Creating of the application simulating all facilities of existing virtual device shows the user all the functions. Application also provides a review about calculating times and the complexity of memory. The research includes further an analysis of the theoretical computer-model, a graphical vision of the application and a proposal for components of interpreted model. It contains the implementation of the application and consecutive adjustment related with functionality tests. Big attention is devoted to tests and the evaluation of test results. The graphical vision helps the user to understand the facility's basic principles.

Obsah

Úvod

Keď by niekto v druhej polovici 20. storočia vyslovil slovo počítač, či výpočtová technika, vybavili by sa mu v myšlienkach veci ako obrovské miestnosti, s veľkými skriňami, ktoré obsahujú veľké množstvá súčiastok, páni s okuliarmi, v bielych plášťoch skúmajúci čosi na veľkých monitoroch, alebo konzultujúci s kolegami rôzne papierové tlačivá, dierne pásky a v pozadí hlučný chod všetkých tých veľkých zariadení. V tých časoch si väčšina ľudí (nezainteresovaných v problematike) myslela: „Veď na čo to je dobré?“ Počítať vieme aj bez takých veľkých strojov, ba dokonca aj presnejšie. V prvopočiatku šlo skutočne najmä o rôzne zložité matematické úkony, ktoré by sa dali vykonať bez pera a papiera a rýchlejšie, ba dokonca presnejšie. No s postupom času výpočtová technika zasahovala stále výraznejšie do rôznych sfér ľudských činností, v ktorých má dnes nezastupiteľné miesto. Bez výpočtovej techniky by sme nedokázali poslať človeka na Mesiac, nezaznamenali by sme žiadne fotografie z Marsu, nedokázali by sme bezpečne ovládať jadrové elektrárne, nevideli by sme televízny program v digitálnej kvalite, nenapísali by sme v úhľadnej forme žiaden text, nemali by sme mobilné telefóny ako dnes, ktoré mimo telefonovania navyše dokážu zaznamenávať obraz so zvukom, následne ho aj prehrať.

Preto dnes zažíva výpočtová technika obrovský rozmach čo sa týka zväčšovania objemu dát, miniaturizácie, rýchlosti a efektívnosti prenosu dát a práce s nimi. Obzvlášť čo sa týka efektívnosti a rýchlosti pri práci s údajmi majú vo výpočtovej technike nezastupiteľné miesto pojmy ako napríklad algoritmus, pamäťové nároky, časová náročnosť, výpočet. Tieto pojmy patria do časti teoretickej informatiky, ktorá je neoddeliteľnou súčasťou celej výpočtovej techniky. Preto by sme mali poznať modely výpočtov, ich presnú definíciu a rozumieť ich princípom pri návrhu a zostavovaní rôznych aplikácií.

Pojem algoritmus je vlastne návod na riešenie nejakého problému a na jeho vyjadrenie môžeme použiť niektorý z programovacích jazykov ako napríklad: Pascal, Algol, Fortran, Cobol, alebo ho môžeme realizovať pomocou teoretických výpočtových modelov Turingovho stroja, či Počítača s ľubovoľným prístupom (RAM - Random Access Machine). Napriek odlišnému písaniu programov v týchto jazykoch, je možné ukázať, že každý z nich dokáže rovnakú úlohu vyriešiť veľmi podobným spôsobom, to znamená, že tieto postupy sa vo svojej podstate neodlišujú, no nemôžeme hovoriť o

rovnakých algoritmoch, ale skôr o ich ekvivalentnosti. Ak by sme chceli vytvoriť program napísaný v Pascale, dokázali by sme daný program vytvoriť aj v programovacom jazyku Algol. No zavedením jednotlivých algoritmov sa prejavia ich rozdiely, t.j. ako rýchlo sa nájde riešenie daného problému. Ak by sme však chceli použiť popis vo vyššom programovacom jazyku, bol by tento proces veľmi obtiažni. Posledný uvedený výpočtový model (RAM) je pre jeho jednoduchosť a podobnosť so skutočným počítačom na úrovni assembleru tou najvhodnejšou voľbou. Interpretovaním príkazov v teoretickom modely počítača s ľubovoľným prístupom sa priblížime k programovaniu skutočných jednočipových počítačov, ktoré zodpovedajú aj ich výsledkom.

Preto cieľom tejto práce je vytvoriť funkčnú aplikáciu na interpretáciu (simuláciu) jednotlivých príkazov takéhoto stroja, ktorý by dostatočným a tiež vhodným spôsobom, mohol byť reprezentovaný a použitý ako pomôcka pri výučbe v oblasti teoretickej informatiky a bol nápomocný pri pochopení základnej problematiky práce počítačov pri zostavovaní aplikácií v programátorskej práci. No nielen to. Pri stavbe daného reálneho výpočtového modelu sme chceli dodržať aj kritériá, ktoré musela daná aplikácia spĺňať aj z hľadiska definovania samotného teoretického modelu, kde sme museli brať dôraz na jeho špecifické vlastnosti. Nakoľko sa jednalo o teoretický model, podmienkou bolo dosiahnuť zhodu s praktickým modelom, alebo sa danému teoretickému výpočtového modelu čo najviac priblížiť. No nejednalo sa len o samotnú funkčnosť zostaveného zariadenia, ale chceli sme zahrnúť do tohto modelu možnosť spočítavania časových a pamäťových zložitostí, ktoré sú neodmysliteľnou súčasťou procesu teoretického modelu stroja RAM.

Interpreter programov RAM ako abstraktný stroj napomôže predstave ako môžu pracovať výpočtové aplikácie napr. na jednočipových procesoroch s jednoduchými inštrukciami. Daný stroj je popri Turingovom stroji z učebného hľadiska významnou časťou výučby teoretickej informatiky. Tu si používateľ overí svoje poznatky z teórie algoritmov najmä však pri stanovovaní časových aj pamäťových nárokov na ten ktorý algoritmus, alebo program.

1 Súčasný stav riešenia problematiky

1.1 Charakteristika RAM stroja

Pri riešení danej problematiky sme museli najprv zozbierať čo najviac informácií z daného okruhu informatiky. Hlavným zdrojom teoretických skutočností, poznatkov boli informácie získané prostredníctvom kníh a internetu. Teoreticky bol RAM stroj charakterizovaný vo všetkých z týchto zdrojov, dokonca výsledky príkladov, postupy riešení výpočtov a teda celej práce tohto virtuálneho zariadenia zostavovali samotní autori publikovaných zdrojov informácií. No naproti tomu mala praktická realizácia interpretovania tohto stroja len niekoľko realizátorov, ktorý práve prostredníctvom internetu tieto zariadenia zverejnili. Poznatky z danej oblasti informatiky siahajú do obdobia spred 40-tych rokov, no skutočnosti, charakteristiky RAM stroja sa nezmenili a sú stále aktuálne. Bibliografické aj elektronické zdroje informácií, sa v týchto charakteristikách a skutočnostiach zhodujú. Iba v niekoľkých prípadoch sme narazili na mierne odlišnosti. V prvom prípade šlo o charakteristiku inštrukcie, no na celkovú funkciu nemala výrazný vplyv a skôr by sa dalo povedať, že svojou existenciou obohatila dané virtuálne zariadenie (Jančar-Kot-Sawa, 2007). V druhom prípade sa jednalo o príkazy, ktoré majú rovnakú funkciu, ale mali iné pomenovanie. Na tieto aj ostatné odlišnosti poukážeme v charakteristike príkazov RAM stroja. Tieto skutočnosti sme zohľadnili aj v praktickom riešení.

RAM stroj čiže Random Access Machine (stroj s ľubovoľným prístupom k pamäti) je abstraktný stroj patriaci do triedy registrových strojov, ktorý pracuje na základe vkladaní a prepisu údajov v registroch, no na rozdiel od Turingovho stroja, viac pripomína štruktúru bežných počítačov (Kučera, 1983). Ľubovoľný prístup k pamäti neznamená žiadnu náhodnosť pri vstupe do pamäte, ale že v každom okamihu výpočtu môže stroj pristupovať k ľubovoľnej pamäťovej bunke. Jedná sa o univerzálny výpočtový model stroja.

Pracuje s jazykom veľmi nízkej úrovne. Najlepšie by sme ho vedeli porovnať s jazykom symbolických adries. Obsahuje komponenty ako bežný počítač alebo procesor. RAM má vlastnosti počítadlového stroja, ale s jednou výhodou navyše a to nepriamym adresovaním registrov s ľubovoľným prístupom k nim. To znamená, že v jednom kroku výpočtu môže tento stroj pristúpiť k pamäťovej bunke (registru) s ľubovoľnou adresou.

Do buniek pamäte tohto stroja môžeme vkladať celočíselné hodnoty a to znamená, že aj vstupné a výstupné údaje tohto stroja budú pozostávať z celočíselných hodnôt.

1.2 RAM stroj a jeho jednotlivé časti

Zo zdrojov, ktoré definovali časti daného zariadenia, sme museli zvoliť jeden zdroj informácií ako základ pre náš ďalší postup, nakoľko sme narazili na isté odlišnosti, ktorým sme sa v riešení danej problematiky uberali. Musíme podotknúť, že sa nejednalo o zásadné rozdiely, ktoré by menili celú filozofiu a princípy funkcie daného modelu zariadenia. Všetky nasledujúce definície aj opis jednotlivých komponentov stroja RAM pochádzajú zo zdrojov akademického prostredia (Lovászová-Vozár, 2007). V prípade odlišnosti sme tieto zdroje informácií citovali v priebehu charakteristík a popisu.

Možnosťami RAM stroj je zložený podobne ako bežný počítač z častí (hardware), s tým rozdielom, že jeho časti sú abstraktné, no nápadne sa podobajú, až na niektoré rozdiely, častiam bežných počítačov. Tento stroj pozostáva z:

- programovej jednotky,
- programového registra,
- pamäte,
- vstupnej jednotky,
- výstupnej jednotky.

1.2.1 Programová jednotka

Programová jednotka je tvorená konečnou postupnosťou príkazov $p_1, p_2, ..., p_n$, ktoré sú tvorené inštrukciami zapísanými v programe RAM. Programová jednotka je sprostredkovacím článkom medzi vykonávateľom inštrukcií (Programový register) a napísanými inštrukciami v programe RAM, za pomoci nej prebieha celý proces výpočtu stroja RAM.

1.2.2 Programový register

Programový register, alebo tiež čítač inštrukcií má za úlohu vybrať inštrukciu, ktorá sa má v danom okamihu vykonať, a to podľa toho pod akým poradovým číslom (alebo v akom riadku) je zapísaná v programe RAM. Počiatočný stav čítača inštrukcií stroja RAM musí obsahovať číslo „*1*“, to znamená že stroj je pripravený vykonať prvý príkaz z programovej jednotky. Po vykonaní danej inštrukcie sa hodnota programového

registra zvýši o „*1*“. Tento cyklus pripočítavania sa neustále opakuje až do načítania poslednej inštrukcie zapísanej v programe RAM, alebo tiež v priebehu výpočtu, ak dôjde k vykonaniu chybného príkazu, chybnej hodnote. Výnimku tvoria príkazy skoku, ktoré môžu túto hodnotu zmeniť.

Úlohou tejto jednotky je najmä vykonávanie:

- aritmetických operácií (sčitovanie, odčítavanie, násobenie a delenie),
- logických operácií (načítanie a uloženie údajov do pamäte),
- vstupno/výstupných operácií (zápis do výstupnej pásky a čítanie zo vstupnej pásky),
- príkazov skoku,
- zastavenia.

Samotný programový register, za použitia analógie, by sme mohli prirovnať k aritmeticko-logickej jednotke mikroprocesora bežného počítača. Programový register, ako aj programová jednotka, tvoria srdce celého RAM stroja. Ako celok by sme ich analogicky mohli prirovnať k procesoru bežného počítača.

1.2.3 Pamäť

Táto lineárne usporiadaná pamäť je zložená z registrov (pamäťových buniek), do ktorých sú vkladané, alebo sú v nich prepisované celočíselné hodnoty. Z popisu tejto jednotky RAM stroja tiež vyplýva, že tento stroj ich obsahuje nekonečné množstvo. Sú teda očíslované od hodnoty „*0*“ až po *n*, kde *n* môže nadobúdať hodnotu ľubovoľného prirodzeného čísla. Tieto čísla nazývame adresami pamäťových buniek. Analogicky by sme mohli nazvať túto jednotku ako pri bežných počítačoch operačnou pamäťou stroja RAM. Počiatočné hodnoty všetkých registrov sú rovné „*0*“. Do registrov môžeme zapísať neobmedzene veľkú hodnotu celého čísla, čo znamená, že nemusíme vkladať len jednociferné hodnoty (číslice).

Všetky registre majú rovnakú funkciu čítania a zápisu respektíve prepisu hodnoty. Špeciálnu funkciu v pamäti RAM stroja zastávajú:

- pracovný register,
- indexový register.

Pracovný register

V niektorých publikáciách je tiež nazývaný ako akumulátor „one-accumulator“ (Aho-Hopcroft-Ullman, 1974). Prostredníctvom programového registra v ňom prebiehajú takmer všetky operácie. Ako napríklad operácie s pamäťou, vstupno/výstupné a aritmeticko-logické operácie. Jeho adresa má hodnotu „*0*“.

Indexový register

Má tak ako pracovný register špeciálny význam, ale na rozdiel od pracovného registra slúži na nepriame adresovanie. Môže uchovávať hodnotu adresy ďalšieho registra, alebo môže použiť jeho hodnotu. Hodnota v tomto registri pri vykonaní príkazu s nepriamym adresovaním musí nadobúdať hodnotu prirodzeného čísla. Ak by teda hodnota v tomto registri bola záporná, výpočet by skončil chybou (záporná adresa cieľového registra neexistuje). Nie vo všetkých programoch využívame indexový register. Preto môžeme vytvárať programy, kde indexový register stráca svoju funkciu a zastáva úlohu bežného registra na uchovávanie údajov. V danom prípade nejde o funkčnú chybu, iba o zmenu náhľadu na typ, alebo zaradenie daného registra.

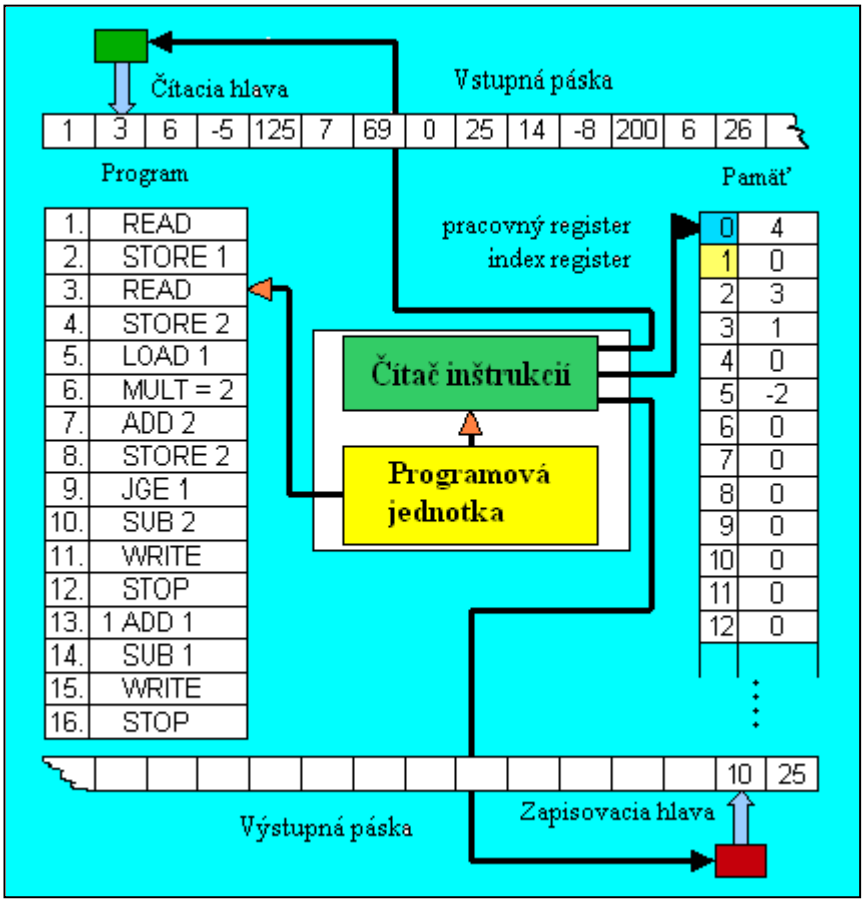
1.2.4 Vstupná jednotka

Je zložená zo vstupnej pásky a čítacej hlavy. Vstupná páska je vytvorená z buniek, kde každá bunka obsahuje jeden vstupný údaj, čiže celé číslo. Čítacia hlava má za úlohu danom okamihu zaznamenávať zo vstupnej pásky údaj, ktorý je uložený práve v jednej bunke. K údaju pristupuje sekvenčne. Čiže vždy sníma len jeden údaj. Následne sa posunie o jednu bunku doprava. Zo vstupnej pásky môže RAM stroj údaje len čítať.

1.2.5 Výstupná jednotka

Je zložená z výstupnej pásky a zapisovacej hlavy. Tak ako vstupná páska je aj výstupná páska vytvorená z buniek, ale s tým rozdielom, že do buniek bude zapisovaný údaj, čiže celé číslo. Zapisovacia hlava má za úlohu v danom okamihu vložiť do bunky výstupnej pásky práve jeden údaj, ktorý je uložený v pracovnom registri. Rovnako ako vo vstupnej jednotke je zachovaný sekvenčný prístup (zapisovacia hlava vloží vždy jeden údaj). Následne sa posunie o jednu bunku doľava. Na výstupnú pásku môže RAM stroj údaje len zapisovať.

Z nasledujúceho obrázku vyplýva ako jednotlivé časti na seba nadväzujú a môžeme tiež zistiť akým spôsobom pracuje stroj RAM.



Obr. 1 Zloženie stroja RAM

1.3 Príkazy RAM stroja

Nachádzajú sa v samotnom RAM programe programovej jednotky no vykonávané sú prostredníctvom programového registra. Rozdeľujeme ich do niekoľkých skupín a to podľa ich funkcie, ktorú vykonávajú.

1.3.1 Vstupno/výstupné príkazy

Ako nám názov napovedá jedná sa o základné príkazy, bez ktorých by sme vlastne nemali s čím pracovať a tiež by nebolo možné zistiť aké sú hodnoty výstupných dát. Dané príkazy pracujú bez operandov, keďže vždy čítajú/zapisujú jedinú hodnotu a to sekvenčne.

`READ` – prečíta údaj zo vstupnej pásky a uloží ho do pracovného registra 0, následne sa čítacia hlava posunutie o jednu bunku doprava.

`WRITE` – prečíta údaj z pracovného registra a zapíše do bunky výstupnej pásky a posunie zapisovaciu hlavu o jednu bunku doľava.

No stretávame sa aj s opisom vstupnej a výstupnej jednotky, kde je nutnosťou použitia takýchto príkazov aj operand, ktorý vyjadruje, ktoré políčko na pásky sa použiť (Aho-Hopcroft-Ullman, 1974). V tomto prípade však nejde o zautomatizovanie procesov čítania, alebo zápisu údajov, preto sme tento spôsob zapisovania programu vylúčili.

1.3.2 Príkazy pre prácu s pamäťou

Tieto príkazy sú určené na výber a uloženie dát v registroch. Tu sa jedná o manipuláciu, presun hodnôt údajov v registroch.

`LOAD` operand – týmto príkazom uloží do pracovného registra číslo dané operandom, obsah pracovného registra sa mení pričom ostatné hodnoty obsahu pamäťových buniek zostávajú nezmenené.

`STORE` operand – do pamäťovej bunky, ktorej adresa je daná operandom sa vloží obsah pracovného registra, pri tomto príkaze je neprípustný operand „`=i`“. Hodnoty ostatných pamäťových buniek zostávajú nezmenené.

1.3.3 Aritmetické operácie

Jedná sa o základné aritmetické operácie s dátami ako sčítanie, odčítanie, násobenie a delenie. RAM programy pracujú s celočíselnými hodnotami, preto výsledky dostávame v tvare celočíselnej hodnoty. A to aj pri vykonaní operácie delenia, kde RAM stroj neberie do úvahy zvyšok po delení.

`ADD` operand – hodnota daná operandom sa sčíta s hodnotou uloženou v pracovnom registri a uloží sa do pracovného registra.

`SUB` operand – hodnota daná operandom sa odčíta od hodnoty uloženej v pracovnom registri a uloží sa do pracovného registra.

`MULT` operand – hodnota daná operandom sa vynásobí s hodnotou uloženou v pracovnom registri a uloží sa do pracovného registra.

`DIV` operand – hodnota v pracovnom registri sa vydelí hodnotou danou operandom a uloží sa do pracovného registra.

1.3.4 Príkazy skoku

Príkazy skoku hrajú dôležitú úlohu vo všetkých programoch. V programoch RAM ich vieme použiť ako cyklus, tiež zastupujú funkciu vetvenia programu pri zadaní podmienok.

`JUMP` návestie – ide o nepodmienený skok, čiže výpočet bude pokračovať inštrukciou na danom návestí.

`JGE` návestie – ak hodnota pracovného registra je väčšia alebo rovná 0, potom program pokračuje inštrukciou na danom návestí. Ak táto podmienka nie je splnená, potom program pokračuje nasledujúcou inštrukciou.

`JZERO` návestie – ak hodnota pracovného registra je rovná 0, potom program pokračuje inštrukciou na danom návestí. Ak táto podmienka nie je splnená, potom program pokračuje nasledujúcou inštrukciou.

Nasledujúci príkaz skoku je v uvedený ako štandardný v učebnom texte (Jančar-Kot-Sawa, 2007). Tento príkaz zodpovedá príkazu JGE až na podmienkovú časť.

`JGTZ` návestie – ak hodnota pracovného registra je väčšia ako 0, potom program pokračuje inštrukciou na danom návestí. Ak táto podmienka nie je splnená, potom program pokračuje nasledujúcou inštrukciou.

1.3.5 Príkaz zastavenia výpočtu

Najjednoduchší spomedzi príkazov. Neobsahuje operand ani návestie. Po vykonaní tohto príkazu sa zastavuje celý výpočet. Výpočet sa znovu môže spustiť až po inicializácii daného programu.

STOP – výpočet sa zastaví, hodnota programového registra sa nezvýši o „*1*“.

HALT – výpočet sa zastaví, hodnota programového registra sa nezvýši o „*1*“. Podľa (Jančar-Kot-Sawa, 2007). Tento príkaz je ekvivalenciou príkazu „STOP“. Obyčajne neberieme do úvahy túto možnosť zápisu.

1.3.6 Význam operandov

Operandy môžu nadobúdať hodnoty rovnajúce sa konkrétnym číslam, alebo tiež adresám jednotlivých pamäťových buniek. Pri zápise operandov sme sa nestretli s odlišnosťami, ktoré by sme mohli ukázať. Každá publikácia a zdroj informácií uvádzali rovnaké charakteristiky daného objektu. Je ich možné zadať niekoľkými spôsobmi. Písmeno **i** v operande predstavuje číslo. Poznáme 3 možnosti zápisu:

i V tomto prípade môže nadobúdať hodnotu prirodzeného čísla alebo „*0*“. Jedná sa o obsah i-teho registra, alebo návestia pri skoku, preto je záporná hodnota neprípustná.

=i môže nadobúdať hodnotu celého čísla. V tomto prípade je hodnotou operandu samotné číslo **i**. Pre príkaz STORE je tento operand neprípustný.

***i** Jedná sa o nepriame adresovanie. Význam tohto operandu je nasledovný:

Určuje obsah registra, ktorého adresa je uložená v i-tom registri. To znamená, že číslo **i** nám určuje, v ktorom registri je uložený odkaz čiže adresa cieľového registra, kde je uložená hodnota. Hodnota **i** môže preto nadobúdať hodnoty prirodzených čísel alebo „*0*“. Zároveň ak hodnota odkazu je záporné číslo, celý program skončí (skolabuje), pretože adresa cieľového registra musí mať hodnotu prirodzeného čísla.

1.3.7 Význam návestí

Návestia v programoch RAM sa nachádzajú výlučne za príkazmi skokov. Vyjadrujú miesto, v ktorom má daný program pokračovať, preto majú v inštrukciách obsahujúcich skoky dôležitú funkciu. Čo znamená skok na ľubovoľné miesto v programe ním určené. Môžu nadobúdať hodnoty prirodzených čísel a musia byť totožné s pozíciou (hodnota prirodzeného čísla), kam bude daný program „odklonený“. Pozície skokov sú v programoch RAM vložené pred inštrukciu, kde je udané miesto skoku programu.

No exitujú aj ďalšie možnosti určenia presnej identifikácie, čiže akou inštrukciou má daný program pokračovať vo svojom priebehu. Je to napríklad presné určenie riadku, v ktorom má program pokračovať (Jančar-Kot-Sawa, 2007), kde navyše nemusíme zadávať pozíciu skoku. Alebo úplne iným spôsobom, kde je návestie a pozícia kam bude smerovať skok programu sú vyjadrené slovom, vyjadrujúcim charakteristický znak, funkcie cyklu alebo podmienky (Aho-Hopcroft-Ullman, 1974).

2 Cieľ práce

Podrobným štúdiom uvedených literárnych zdrojov sme zistili, že dané virtuálne zariadenie má dosť špecifických vlastností, ktoré svojou štruktúrou a charakteristikou pripomínajú dnešné počítače. Jednotlivé informácie okolo RAM strojov a ich vlastností sú interpretované len v tlačenej, či elektronickej forme ako sú výklady či texty, ktorých predstava je podporovaná vo forme obrázkov. Preto pri výklade tohto stroja by nám mohla napomôcť vizualizácia daného virtuálneho zariadenia, ktoré by navyše vhodným spôsobom interpretovalo funkciu modelu takéhoto zariadenia. Cieľom diplomovej práce bolo vytvoriť virtuálny ale funkčný model s nasledovnými vlastnosťami:

a) aplikácia s pomenovaním Interpreter stroja RAM,

b) jeho popis a charakteristiky sa musia zhodovať so skutočnosťami uvedenými v publikáciach,

c) musí byť schopný vykonávať a presne prezentovať dané inštrukcie v čase,

d) svojím pôsobením musí dospieť k pravdivým (žiadaným) výsledkom,

e) ďalšie jeho funkcie majú pomôcť zjednodušiť teoretický výklad pri použití v oblasti teoretickej informatiky (výpočet časovej a pamäťovej zložitosti),

f) Interpreter stroja RAM nesmie svojím chodom negatívne ovplyvňovať chod iných aplikácií (nakoľko samotný interpreter je aplikácia),

g) prostredie, v akom bude realizované musí byť jednoduché a ovládanie intuitívne zvládnuteľné.

3 Metodika práce

Základom celého návrhu aplikácie interpretera bolo dôkladne sa oboznámiť s problematikou, kde samozrejmou nutnosťou je pochopenie danej problematiky a tiež získanie čo najväčšieho množstva informácií v danej oblasti. Voľba presných charakteristík z množstva zdrojov informácií, podľa ktorých sme riešili dané úlohy.

Dôležitou súčasťou návrhu a riešenia je tiež:

- určenie programátorského prostredia,
- návrh vizuálnej stránky - rozvrhnutie častí výslednej aplikácie ako aj rozloženie zdrojového kódu v aplikácii a v neposlednom rade návrh celkovej vizuálnej podoby výslednej aplikácie, aby bola zrozumiteľná pre každého používateľa,
- implementácia – vytvorenie funkčnej aplikácie, vyhnutie sa chybám z hľadiska špecifík RAM stroja ako aj z hľadiska výslednej aplikácie. Zápis a prípadné korekcie zdrojového kódu pre zabezpečenie správneho chodu našej aplikácie RAM stroja,
- úprava aplikácie s dôrazom na kritické chyby pri vkladaní hodnôt a ostatné úpravy. Pri behu výpočtu, obohatenie aplikácie o údaj časovej a pamäťovej zložitosti,
- testovanie a vyhodnocovanie - z hľadiska správnosti výsledkov a z hľadiska vyhodnocovania údajov časových a pamäťových nárokov na program RAM.

3.1 Určenie programátorského prostredia

Danú aplikáciu sme mali možnosť vytvoriť niekoľkými programátorskými, ktorých je nepreberné množstvo. Pre daný účel sme zvolili vizuálne vývojové prostredie BORLAND DELPHI vo verzii 2005, ktoré je založené na mutácii jazyka Object Pascal, ktorým sme naprogramovali jednotlivé časti interpretera.

Tento vývojový prostriedok, zahŕňa v sebe silnú sadu vizuálnych nástrojov pre tvorbu jednotlivých častí aplikácie s programovými nástrojmi a výkonným kompilátorom a toto vývojové prostredie je primárne určené pre tvorbu aplikácií pod operačným systémom Windows a pre potreby už spomínaného simulátora plne vyhovuje.

3.2 Návrh vizuálnej stránky aplikácie

Z používateľského hľadiska si musíme RAM stroj predstaviť ako akúsi „skrinku“ a v našom prípade aplikáciu, ktorá dokáže na základe postupu zadaného konkrétneho programu a po vložení vstupných údajov vypočítať a prezentovať výstupné údaje. Preto je dôležité predstavu takého stroja čo najlepším spôsobom uskutočniť na konkrétnej aplikácii.

Jednotlivé časti tohto zariadenia boli reprezentované ako vizuálne komponenty, ktoré máme možnosť sledovať na monitore počítača a vytvoriť tak komunikáciu s používateľom a na druhej strane sú to nevizuálne komponenty, ktoré sú určené na obsluhu a bezpečný chod samotnej aplikácie.

Z charakteristiky zariadenia teda vyplýva, že sa jedná o stroj na teoretickej úrovni a jeho realizáciu musíme uskutočniť za pomoci interpretovania jednotlivých komponentov RAM stroja.

3.3 Implementácia interpretera programov RAM

Čo sa týkalo hľadiska zdrojového kódu sme si zvolili stratégiu rozdelenia jednotlivých procedúr podľa dôležitosti a funkčnosti na:

- obslužné procedúry – nemajú vplyv na výpočet, ale sú dôležitou súčasťou behu programu, teda jeho správnej funkcie. Medzi ne môžeme zahrnúť napríklad analýzu otvorených súborov, kontrolu správnosti syntaxe, vytvorenie pamäte a programového registra, kontrolu vloženej číselnej hodnoty, atď.
- vykonávacie procedúry – sú dôležité pri výpočtoch a sú rozhodujúcim prvkom v aplikácii.
- pomocné procedúry – tu by sme mohli spomenúť zarovnanie programovej jednotky a pamäte,
- inicializačné procedúry – tie sú využívané najmä pri akciách tlačidiel, pri vytvorení kontextového menu alebo po vstupe pri editovaní v programovom registri alebo pri editovaní popisu programu, atď.
- informačné procedúry – na celkový beh programu nemajú vplyv, no ich informačná úloha je nenahraditeľná. Sem patrí procedúra *Chybové hlásenia*.

Nasledujúca fáza bola zameraná na funkčnosť celku, kde predpokladom dokončenia aplikácie bolo vyhnutie sa chybám, ktoré by mali za následok tzv. „skolabovanie“ celej aplikácie a jej následné ukončenie už operačným systémom, alebo v inom prípade keď z hľadiska prípadného zlého návrhu programu RAM stroja, by program vytvoril nekonečnú slučku (nekonečný cyklus) a používateľ by nemohol prerušiť chod programu. Tieto fatálne následky bolo preto nutné eliminovať do takej miery, aby sa nemohli vyskytnúť.

Posledným a nemenej dôležitým krokom bolo testovanie skúšobnými programami pre jednotlivé procedúry, ktoré si takúto činnosť vyžadovali. Ak sa znova objavili skutočnosti v podobe vyššie uvedených funkčných chýb, alebo chybných výsledkov, ktoré sme nepredpokladali, bolo nutné zabezpečiť korekciu zdrojového kódu.

3.4 Úprava aplikácie

Do tejto kategórie by sme mohli zahrnúť odstránenie kritických chýb, úpravu aplikácie pridaním štandardnej výbavy, redukcie zdrojového kódu, pomoc vizualizácie pridaním jednoduchých efektov a posledným a najdôležitejším prvkom bolo obohatenie aplikácie o určenie výpočtovej časovej a pamäťovej zložitosti.

3.5 Testovanie a vyhodnocovanie

Koncovým bodom celej práce bolo testovanie, čím sme overovali pravdivosť dosiahnutých výsledkov, časovej a pamäťovej zložitosti algoritmov.

V tejto časti práce sme sa držali nasledovných bodov:

1. **Analýza časovej zložitosti (Príklady programov č.1 – č.5):**
   - výber dvoch vzoriek RAM programov a podrobenie analýze (jednotlivé príkazy),
     - výpočet a získanie výsledkov,
     - overenie výpočtu pomocou interpretera RAM programov,
     - porovnanie výsledkov,
   - výber troch vzoriek RAM programov, bez podrobenia analýze,
     - postup rovnaký ako pri dvoch vzorkách, ale viac vstupných hodnôt.
2. **Analýza pamäťovej zložitosti (Príklady programov č.6 – č.10):**
   - postup rovnaký ako pri analýze časovej zložitosti,
3. **Zhrnutie.**

4 Výsledky práce

4.1 Návrh vizuálnej stránky aplikácie

4.1.1 Vstupná jednotka

Špecifikácia: vkladanie a zobrazovanie celočíselných hodnôt.

Obmedzenie: nemožnosť vloženia iných hodnôt alebo znakov.

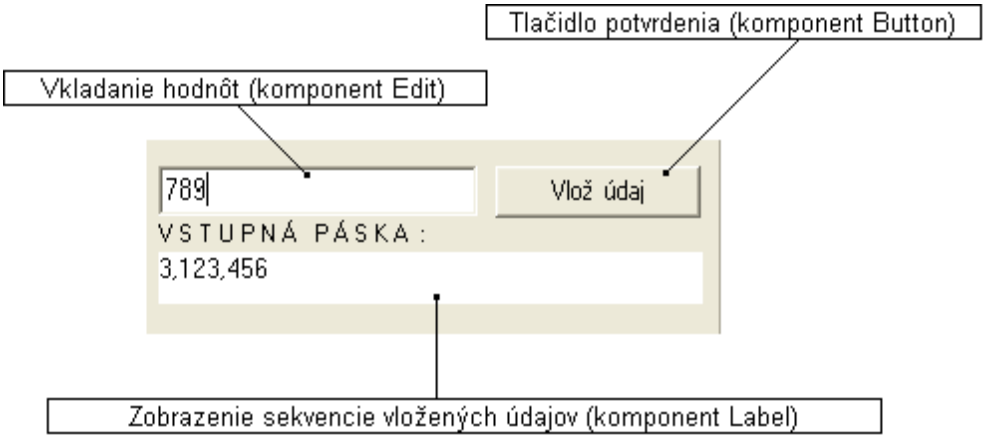
Rozširujúce možnosti: uloženie vstupných údajov do textového súboru.

Aby predstava tohto komponentu bola čo najdokonalejšia a aby najlepšie dokázala spĺňať samotnú predstavu, museli sme tento „hardware“ realizovať spôsobom priameho vkladania vstupných údajov prostredníctvom klávesnice počítača do aplikácie v našom prípade prostredníctvom komponentu *TEdit* na spracovanie. Namiesto čítacej hlavy sme použili tlačidlo potvrdenia pre vloženie údaja do aplikácie.

Sekvenciu vložených údajov sme mohli pozorovať na ďalšom vizuálnom komponente, kde sa zaznamenávané údaje počas vkladania zobrazovali. V našom prípade to bol komponent *TLabel*.

Všetky vložené údaje, ktoré boli zaznamenávané v tomto komponente, boli oddelené čiarkou. Čiarka predstavovala identifikátor konca bunky vstupnej pásky.

Vstupnú jednotku najlepšie vykresľuje nasledujúci obrázok so stručným popisom jeho jednotlivých častí nachádzajúci sa na nasledujúcej strane tejto práce.



Obr. 2 Vstupná jednotka stroja RAM

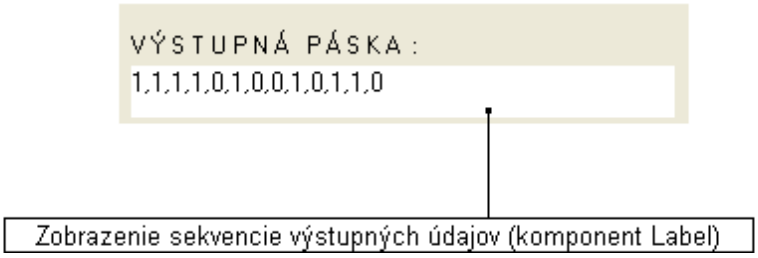
4.1.2 Výstupná jednotka

Špecifikácia: zobrazenie celočíselných hodnôt.

Obmedzenie: žiadne.

Rozširujúce možnosti: uloženie výstupných údajov (výsledkov) do textového súboru.

Výstupné údaje boli zobrazované podobným spôsobom ako vstupné údaje aplikácie. Počas výpočtu sa zobrazovali výsledky (výstupné údaje) iba prostredníctvom komponentu *Label*.



Obr. 3 Výstupná jednotka stroja RAM

4.1.3 Programová jednotka

Špecifikácia: tvorba, zobrazenie a editovanie postupnosti príkazov, ich presná identifikácia pri načítavaní zo súboru.

Obmedzenie: nemožnosť vloženia iných hodnôt - príkazov, len aké podporuje RAM stroj.

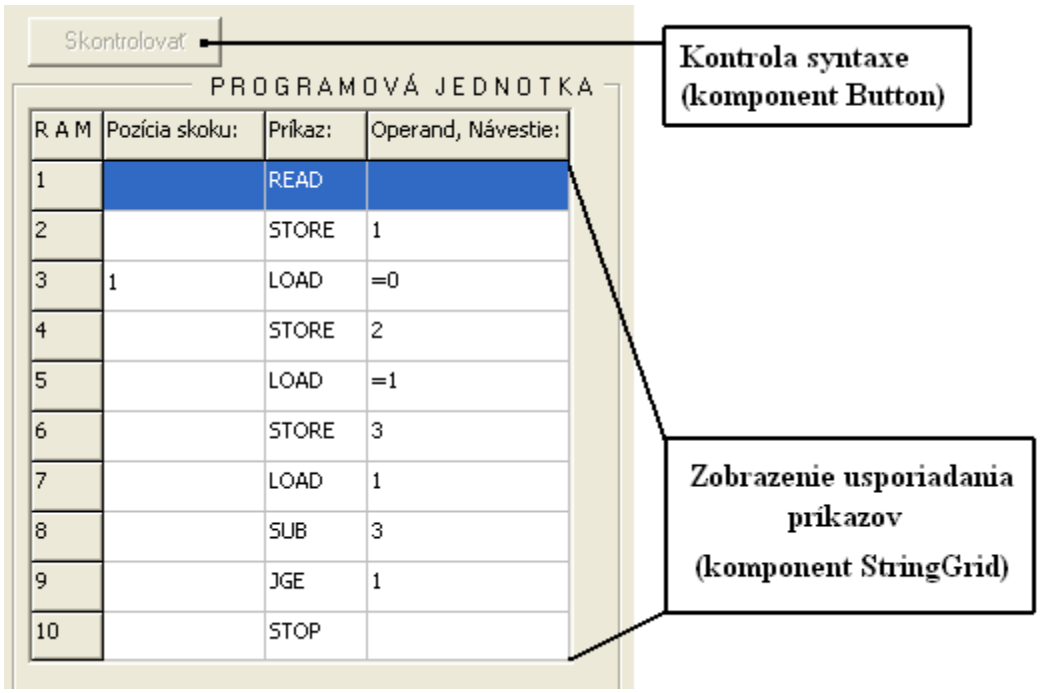
Rozširujúce možnosti: uloženie programu RAM stroja a stručného popisu tohto programu do textového súboru, zisťovanie syntaktických chýb pri otvorení alebo editovaní programu.

Úlohou programovej jednotky bolo teda zvládnuť jednotlivé príkazy nášho programu hlavne po syntaktickej stránke. K zobrazeniu programovej jednotky RAM stroja sme použili vizuálny komponent pre tabuľky *StingGrid*. Týmto elegantným spôsobom sa dá znázorniť práve programový register, jeho editovanie, prepis chybného záznamu, prípadné vymazanie niektorého príkazu, atď.

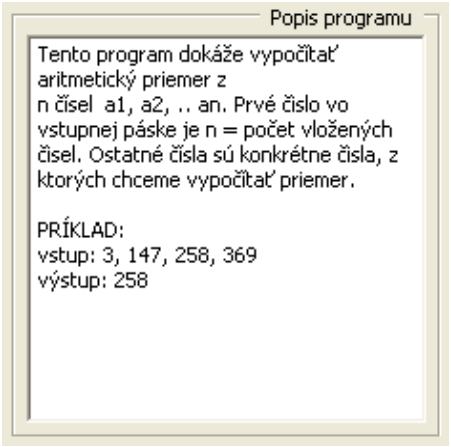
Aby program, podľa ktorého má bežať RAM stroj, bol úplný, je užitočné ho doplniť možnosťou popisu vlastností s jednoznačným určením, to znamená za akým účelom bol vytvorený a teda aký výpočet bude ním realizovaný. Pre tvorcu programu, teda pre samotného používateľa sa predkladá možnosť bližšieho popisu tohto programu, ku ktorému sa po uložení do súboru môže neskôr vrátiť. Podľa samotného popisu programu teda vieme čo program robí, poprípade aké vstupné hodnoty môžeme zadávať a aké výstupné údaje môžeme očakávať, ak chceme vyskúšať daný program. Na rozlíšenie tohto textu od samotného programu sme použili jednoduchý identifikátor, ktorý v uloženom textovom súbore začínal znakom „//“. Na interpretáciu popisu programu sme použili komponent textového editoru – *Memo*.

Ďalším doplňujúcim, a nemenej dôležitým komponentom súvisiacim s programovou jednotkou je tabuľka zistených syntaktických chýb. Po zostavení nového programu alebo otvorení či editovaní už existujúceho programu nám aplikácia zistí prípadné chybné zápisy v tabuľke čítača inštrukcií, na ktoré nás upozorní výpisom. Pre ich zápis sme použili komponent, ktorý je určený na výpis takýchto chybových hlásení *ListBox*. Každá zistená syntaktická chyba bola v nej zobrazená.

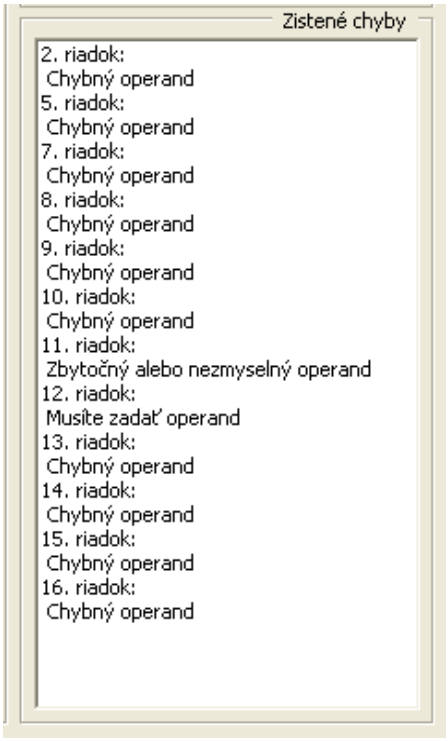
Po otvorení RAM programu zo súboru sa automaticky kontroluje jeho syntax, ale pri editovaní alebo pri novovytvorenom programe nie je možné takúto kontrolu previesť automaticky, a preto bol použitý ďalší vizuálny komponent, teda tlačidlo – *Button*, ktorým inicializujeme kontrolu príkazov zapísaných v čítači inštrukcií. Nasledujúce strany s obrázkami zobrazujú uvedené skutočnosti a popis jednotlivých komponentov.



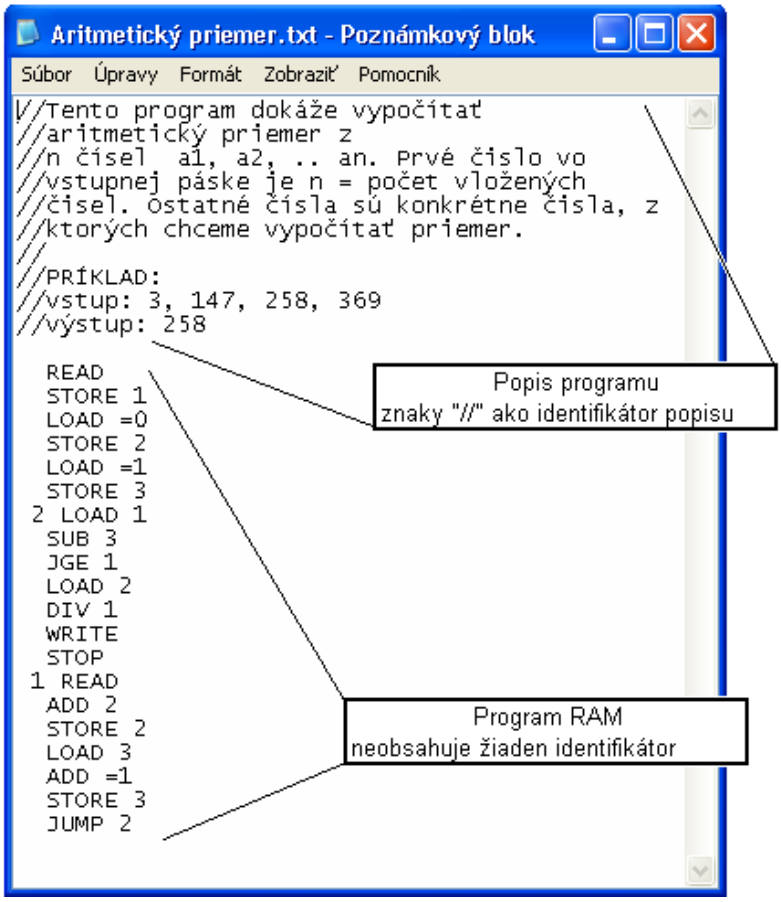
Obr. 4 Programová jednotka stroja RAM



Obr. 5 Komponent popisu programu



Obr. 6 Komponent výpisu zistených chýb



Obr. 7 Realizácia textového súboru s programom RAM

4.1.4 Programový register

Špecifikácia: spustenie a ukončenie behu programu, v prípade skokov pokračovaním behu programu návestím.

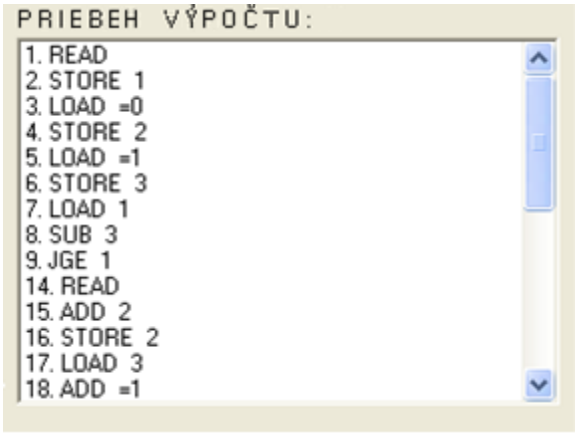
Obmedzenie: v prípade chybnej sémantiky programu možnosť prerušenia jeho chodu.

Rozširujúce možnosti: zaznamenávanie priebehu vykonávaných inštrukcií, výpis konečného výsledku, možnosť spustenia RAM stroja jednoduchým krokovaním alebo tiež spustením naraz.

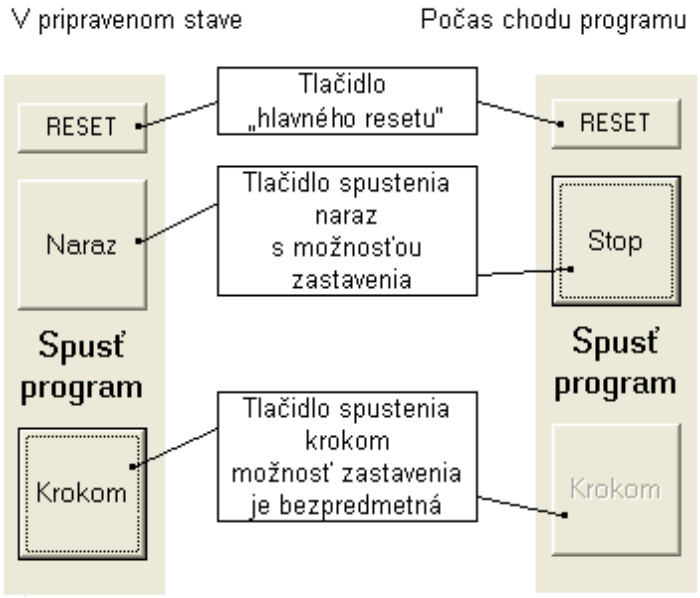
Samotný programový register alebo čítač inštrukcií sme interpretovali ako jednu z množstva procedúr výslednej aplikácie, jedná sa o nevizuálny komponent, preto sme pre danú aplikáciu vytvorili zvláštnu tabuľku, ktorá „odsleduje“ celý proces, to znamená ako postupuje beh programu, čiže aké inštrukcie boli práve vykonané. Tieto skutočnosti sme preniesli do komponentu *ListBox*.

Spustenie a ukončenie programu zabezpečovali ďalšie procedúry, ktoré boli inicializované dvoma navzájom nezávislými tlačidlami, z ktorých jedno bolo určené na spustenie behu programu krokom, aby bolo možné sledovať priebeh výpočtu a zmeny zaznamenávané v pamäti. Druhé tlačidlo je určené na spustenie programu naraz. V jeho prípade sme museli toto tlačidlo využiť aj takým spôsobom, aby v prípade nekonečného cyklu programu alebo zastavenia bolo možné zastaviť chod programu.

Na prerušenie chodu programu v prípade chybnej sémantiky, alebo úplné zrušenie činnosti RAM stroja sme pridali do aplikácie ďalší vizuálny komponent *Button*. Tento komponent zabezpečí hlavný „reset“ výpočtu.



Obr. 8 Realizácia programového registra



Obr. 9 Realizácia spustenia a zastavenia programu

4.1.5 Pamäť

Špecifikácia: dočasné ukladanie (prepis) a výber hodnôt jednotlivých registroch.

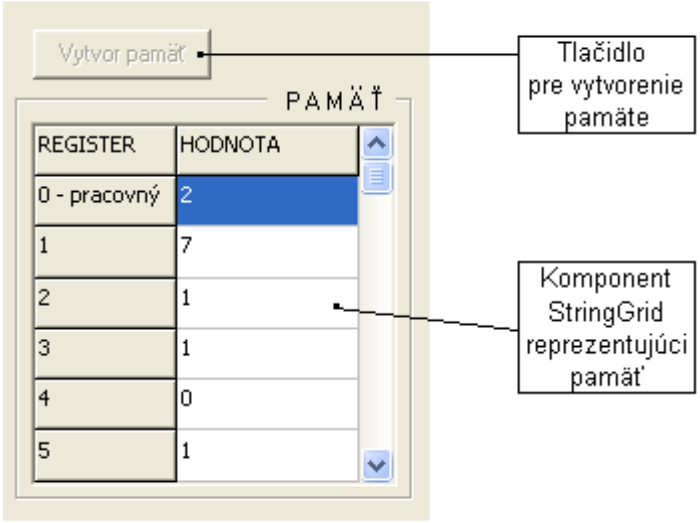
Obmedzenie: veľkosť pamäte musí byť obmedzená (reálne nemôže byť nekonečne veľká), nemožnosť vkladania iných hodnôt ako celočíselných, nemožnosť adresovania inak ako celými číslami.

Rozširujúce možnosti: žiadne.

Pamäť RAM stroja sme vytvorili pridaním ďalšieho tabuľkového komponentu *StringGrid*. Ako už vieme z definície RAM stroja je zrejmé, že pri inicializácií tohto zariadenia budú všetky bunky pamäte obsahovať hodnotu číslo nula „0“.

Ako by sme sa mohli nesprávne domnievať tento komponent by nemal byť problematický z hľadiska jeho vizuálnej prezentácie. Práve vizuálne prezentovať tento komponent nie je jednoduché, pretože neobmedzene veľkú pamäť nemožno realizovať v skutočných aplikáciách. Pri počiatočnom určení počtu registrov sme za kľúč k zisteniu tejto skutočnosti zvolili príkaz „`STORE`“ zapísaného v programovej jednotke. Dôvodom k tomuto rozhodnutiu nás viedlo zistenie, vyplývajúce z definície RAM stroja, kde vložením údaja s najväčšou adresou je limitovaná len operandom pozostávajúceho z čísla, nachádzajúcom sa v programe RAM stroja práve na mieste za príkazom „`STORE`“. To znamená, že zo všetkých riadkov RAM programu, v ktorých sa nachádza už spomínaný príkaz, vyberie aplikácia takú hodnotu operandu, ktorá je

najväčšia a táto hodnota je považovaná za kľúčovú a podľa tohto čísla bude stanovená počiatočná veľkosť pamäte (počet registrov). Veľkosť pamäte tiež ovplyvňuje skutočnosť práce s operandom nepriameho adresovania, ktorá môže ďalej zväčšovať jej veľkosť. Túto skutočnosť sme zabezpečili v zdrojovom kóde našej aplikácie. Je dôležité uviesť, že pri načítaní RAM programu zo súboru alebo po jeho vytvorení a následnej úspešnej kontrole musíme zakaždým vytvoriť aj túto pamäť, práve podľa horeuvedených skutočností. Pre tento prípad sme použili ďalší vizuálny komponent, čiže tlačidlo, ktoré túto funkciu zabezpečuje.



Obr. 10 Realizácia pamäte a tlačidla jej vytvorenia

4.1.6 Celková podoba návrhu

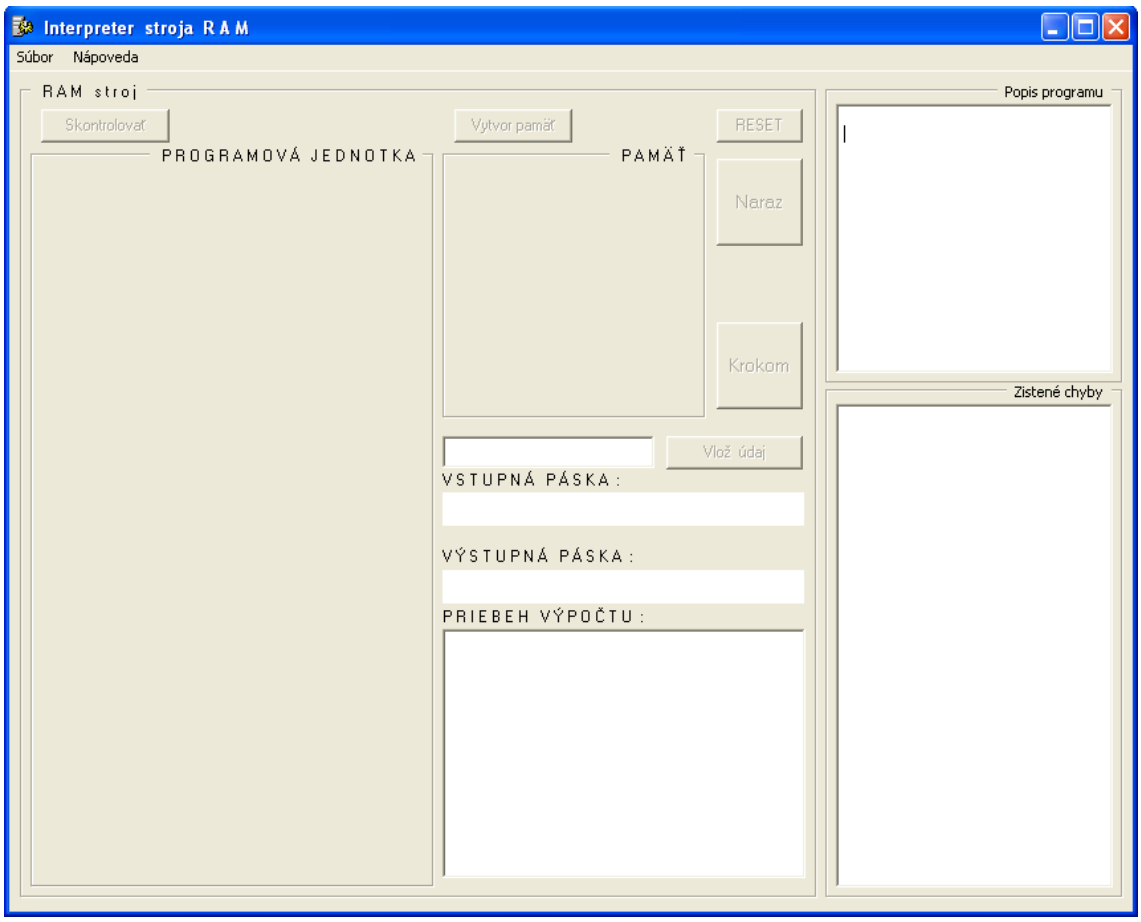
Výsledný RAM stroj pozostával už z horeuvedených a popísaných komponentov, ale pre výslednú aplikáciu, by to bolo nedostačujúce.

Preto aplikáciu interpretera stroja RAM sme doplnili ďalšími komponentmi aby mal podobu bežnej aplikácie a aby bol pre používateľa zrozumiteľný a kompaktný. V prípade tohto rozhrania má intuitívne ovládanie svoj zmysel, to znamená aby samotný používateľ hneď pri spustení programu s ním dokázal „manipulovať“, realizovať konkrétny výpočet.

Náš stroj RAM sme nakoniec doplnili o komponenty známe z klasických aplikácií ako napríklad *Menu*, prípadne upresňujúce vizuálnu podobu jednotlivých zoskupení daných komponentov, ku ktorým prislúchajú. Tu bol použitý komponent *GroupBox*. Ďalšími vizuálnymi komponentmi pre názvy sme znova použili už skôr spomínaný *Label*. Pri chode programu a to hlavne v prípade spustenia „naraz“ sa môže stať, že

program prejde veľmi rýchlo ku svojmu záveru a pre mierne spomalenie sme našu aplikáciu ošetrili nevizuálnym komponentom *Timer*. No a na koniec tým najdôležitejším je vlastne „podklad“ celej aplikácie na ktorom sú dané komponenty „položené“ je formulár v programátorskej aplikácií Delphi označovaný *Form*.

Celkový pohľad na rozloženie komponentov a podobu stroja RAM nám ponúka nasledujúci obrázok pri spustení aplikácie.



Obr. 11 Celkový pohľad na Interpreter stroja RAM

Predstava a návrh boli v tejto fáze dokončené a to znamenalo, že nasledujúce kroky smerovali k implementácií celého stroja RAM.

4.2 Implementácia interpretera programov RAM

4.2.1 Obslužné procedúry

Ako bolo spomenuté tieto procedúry nemajú vplyv na výpočet, ale sú dôležité pre správnu funkciu výslednej aplikácie. Všetky ostatné procedúry tohto typu je možné nájsť v prílohe tejto práce kde možno nájsť kompletný výpis zdrojového kódu. Pre priblíženie môžeme uviesť niekoľko príkladov týchto procedúr, ktoré vidno na nasledujúcich troch obrázkoch.

```
procedure TForm1.vymaz_pamat;
var                                    // Vymazanie
  i: integer;                          // StringGrid2 = pamäťe
begin
  for i:=1 to StringGrid2.RowCount do
  StringGrid2.Cells[1,i]:= '0';
end;
```

Obr. 12 Procedúra pre vymazanie pamäte

```
procedure TForm1.reset_p;
begin
  ides:= false;      // Vykonanie resetu počítania
  pamat;
  vymaz_pamat;
  Timer1.Enabled:= false;
  Button1.Enabled:= true;
  Button4.Enabled:= true;
  Button4.Caption:= 'Naraz';
  Button2.Enabled:= false;
  Edit1.Text:= '';
  Listbox3.Items.Clear;
  Listbox2.Items.Clear;
  Label1.Caption:= '';
  Label2.Caption:= '';
  pom:='';
  pom_vst:= '';
  riadok:= 1;
  Form1.ActiveControl:= Button1;
end;
```

Obr. 13 Procedúra vykonania reset

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  StringGrid1.ColWidths[0]:= 35;     // Vytvorenie prvého riadku
  StringGrid1.ColWidths[1]:= 80;     // pri menu "Nový súbor"
  StringGrid1.ColWidths[2]:= 50;
  StringGrid1.ColWidths[3]:= 100;
  StringGrid1.Cells[0,0]:= ('R A M');
  StringGrid1.Cells[1,0]:= ('Pozícia skoku:');
  StringGrid1.Cells[2,0]:= ('Príkaz:');
  StringGrid1.Cells[3,0]:= ('Operand, Návestie:');
  StringGrid1.Cells[0,1]:= ('1');
end;
```

Obr. 14 Procedúra pri vytvorení nového súboru

Ale pre úplnosť môžeme aspoň vymenovať a stručne popísať ďalšie procedúry našej aplikácie:

- ***vytvorenie nového súboru*** – vytvorí prvý riadok aplikácie v programovom registri,
- ***otvoriť program RAM*** – načítanie programu zo súboru a rozdelenie na samotný program a popis programu,
- ***povolenie uložiť*** – zistí zo správy, či chce používateľ uložiť súbor alebo nie,
- ***uložiť program RAM*** – uloženie programu RAM do súboru spolu s jeho popisom,
- ***uložiť výsledok*** – do textového súboru uloží načítané vstupné a výstupné hodnoty,
- ***odpoveď správy*** – zistí zo správy, či chce používateľ ukončiť aplikáciu s uložením súboru alebo nie,
- ***uzatvorenie aplikácie*** – uzatvorenie,
- ***správa*** – zistí od používateľa, či môže uložiť súbor,
- ***prevod súboru*** – po načítaní súboru, vykoná analýzu inštrukcií, upraví ich a vloží do programového registra,
- ***kontrola syntaxe*** – skontroluje správnosť už vpísaných inštrukcií do programového registra,
- ***pamäť*** – jedná sa o konštrukciu teda zostavenie pamäte, bez inicializácie vykonania tejto procedúry,
- ***vykonanie spustenia naraz*** – spustenie tejto procedúry až po inicializácii,
- ***vymazanie pamäte*** – postará sa o vymazanie dát, predošlého výpočtu,

- ***vymazanie programového registra*** – postará sa o vymazanie inštrukcií z programového registra,
- ***vykonanie reset*** – postará sa o celkové vymazanie pamäte, výsledkov aj vstupných hodnôt, vymazanie priebehu postupu výpočtu až po inicializácii,
- ***vypnúť tlačidlá*** – z hľadiska funkčnosti veľmi dôležitá procedúra, zabraňuje nechceným spusteniam niektorých ďalších procedúr,
- ***kontrola čísla*** – z hľadiska funkčnosti dôležitá procedúra, zabraňuje vloženiu iných hodnôt ako celočíselných,
- ***zmazanie, pridanie riadku na koniec*** – stlačením klávesu „-“ vymaže posledný riadok v programovom registri, pri zatlačení „+“ pridá riadok na koniec tohto registra,
- ***úprava, vloženie znaku*** – pri vložení alebo úprave znakov kontroluje zmeny, teda ak sa vyskytnú, môže byť súbor uložený,
- ***dvojklik na programový register*** – aj týmto spôsobom je zabezpečená možnosť editovania riadku v programovom registri,
- ***vloženie riadku*** – možnosť výberu tohto komponentu kontextového menu zabezpečí vloženie kopírovaného riadku na pozícií kurzora myši v programovom registri,
- ***zmazanie riadku*** – podobne ako v predošlej situácií avšak s rozdielom vymazania riadku programového registra,
- ***kopírovanie riadku*** – podobne ako v predošlej situácii avšak s rozdielom kopírovania riadku programového registra,
- ***vloženie prázdneho riadku*** – podobne ako v predošlej situácii avšak s rozdielom vloženia prázdneho riadku na pozícií kurzora myši v programovom registri,
- ***kontrola záznamu*** – táto procedúra určí, či sa zmenil obsah. Ak áno, potom vykoná procedúru zmena,
- ***zmena*** – súvisí s procedúrou kontroly záznamu a táto procedúra povolí použitie tlačidla na kontrolu syntaxe, po zmene obsahu programového registra alebo popisu programu,
- ***podmienka pri zmazaní riadku v kontext. menu*** – procedúra umožní mazanie cez už spomínané kontextové menu, ak minimálny počet riadkov nedosiahne 2,
- ***úprava v popise programu*** – pri tejto úprave popisu programu nám procedúra umožní uložiť program RAM do súboru.

4.2.2 Vykonávacie procedúry

Sú najdôležitejšími procedúrami pri realizácií výpočtov. Ukázať príklady týchto procedúr by bolo možné, ale vzhľadom na ich rozsiahlosť sme uviedli iba názvy s ich stručným popisom. Všetky procedúry sú zapísané v zdrojovom kóde aplikácie, ktorý sa nachádza v prílohe tejto práce. Medzi vykonávacie procedúry sme zahrnuli len tieto:

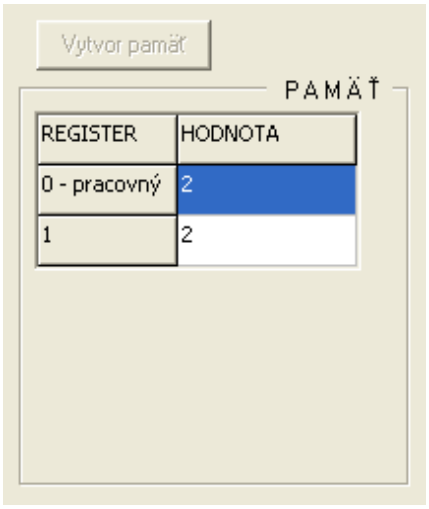
- **vykonanie výpočtu** – inicializujú ju procedúry spustenia krokovaním alebo naraz. Určením riadku v programovom registri a následným prečítaním celej inštrukcie z programového registra, vykonáva táto vykonávacia procedúra skok na ďalšiu procedúru vykonania operácie alebo vykonania skoku v závislosti od prečítania danej inštrukcie. Niektoré inštrukcie nevyužívajú tieto procedúry, lebo sú súčasťou práve tejto procedúry. Tu by bolo vhodné spomenúť príkazy `READ`, `WRITE`, `STOP`,
- **vykonanie operácií** – táto procedúra obsahuje nástroje na analýzu operandu, identifikáciu príkazu a v prípade zistenia chyby sémantiky s tým spojené odkazy na procedúru chybových hlásení. Ale hlavnou úlohou je realizovať výpočty aritmetických operácií a to podľa správneho identifikovania inštrukcie, a tiež realizáciu presunu údajov z pracovného registra do ostatných častí pamäte a naopak,
- **vykonanie skokov** – procedúra obsahuje identifikáciu príkazu aj návestia a nakoniec aj identifikáciu pozícií skokov. Ako v predchádzajúcej procedúre aj túto sme ošetrili v prípade chýb aktivovaním procedúry chybových hlásení. Jednotlivé skoky sme realizovali len jednoduchým prepísaním hodnoty riadku. A to s podmienku, v prípade inštrukcií `JGE` a `JZERO` ktoré boli limitované práve svojimi názvami a hodnotou v pracovnom registri. V prípade skoku `JUMP` neexistovali žiadne obmedzenia.

4.2.3 Pomocné procedúry

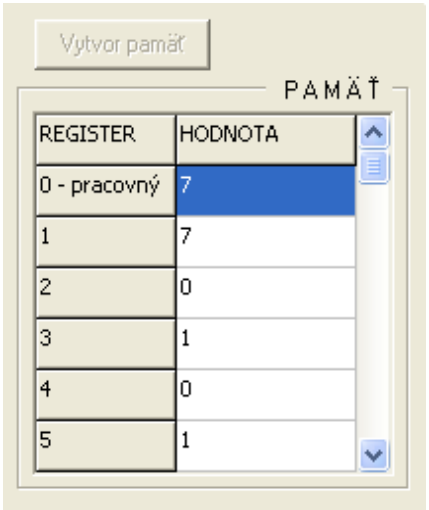
Bez pomocných procedúr by táto aplikácia bola plne funkčná, avšak niektoré vizuálne komponenty by stratili svoj pekný vzhľad. Týmito procedúrami sme chceli:

- zarovnať programový register – vizuálna procedúra jednoduchého zarovnania okrajov programového registra,
- zarovnať pamäť – vizuálna procedúra jednoduchého zarovnania okrajov pamäte.

Ukážky procedúry zarovnania okrajov pamäte a jej význam je zrejmý z nasledujúcich dvoch obrázkov.



Obr. 15 Zarovnanie pri 2 hodnotách v pamäti



Obr. 16 Zarovnanie pri 7 hodnotách v pamäti

4.2.4 Inicializačné procedúry

- Kontextové menu – kliknutím pravého tlačidla myši, inicializujeme v komponente programového registra 4 možnosti ovládnutia editačných pomôcok pre daný riadok programového registra. Túto funkciu je možné využiť len na tomto komponente.
- Nový súbor – inicializuje všetky prvky potrebné k editovaniu a úvodnému zobrazeniu.
- Spustenie krokovaním – spúšťa výpočet vždy po jednom kroku, to znamená, že za každým krokom výpočtu pripočíta ku svojej hodnote číslo 1.
- Spustenie naraz – má za úlohu aktivovať nevizuálny komponent *Timer*, ktorý zabezpečí funkciu mierneho spomalenia výpočtu (inak by proces prebehol veľmi rýchlo) a tiež sám aktivuje procedúru spustenia krokom. Takýmto spôsobom sme zabezpečili plynulý beh programu.
- Vytvorenie pamäte – obsahuje viacero odkazov na ostatné obslužné procedúry, ale aj vykonanie procedúry reset a priraďuje riadku číslo 1, čiže celý program RAM uvádza do počiatočnej polohy.
- Reset počítania – v danom prípade to je len inicializácia tejto funkcie tlačidlom.

Môžeme uviesť takisto dva príklady na jednom obrázku, kde vidíme, že takéto druhy procedúr nemusia mať vždy rozsiahly zdrojový kód a na ďalšom obrázku vidíme viacero odkazov na iné procedúry.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  rataj;                // spustenie krokovaním
  inc(riadok);
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  if ides = true then  // spustenie naraz
  ides:= false         // s podmienkou nutnosti
  else               // prerušenia výpočtu pri nekonečnom cykle
  ides:= true;
  naraz;
end;
```

Obr. 17 Dve jednoduché inicializačné procedúry

```
procedure TForm1.Button5Click(Sender: TObject);
begin
  Label3.Caption:= ('Spusť'+char(13)+'program');
  Label3.Visible:= true;
  Button5.Enabled:= false;        // vytvorenie pamäte
  Button6.Enabled:= true;         // načítanie do tabulky /SringGrid2
  ListBox2.Items.Clear;
  riadok:= 1;
  ListBox3.Items.clear;
  ListBox3.Enabled:= true;
  ides:= true;
  reset_p;
  pamat;
end;
```

Obr. 18 Jedna zložitejšia inicializačná procedúra

4.2.5 Informačné procedúry

V tejto skupine sa nachádza len jedna procedúra a tou je procedúra chybových hlásení. Jej úlohou je informovať používateľa o syntaktických chybách zistených počas otvorenia alebo editovaní či zostavovaní programu RAM, a tiež pri zistení chýb sémantiky. Ak počas výpočtu aplikácia nájde chybu oznámi túto skutočnosť aj oznamom v komponente *ShowMessage*. Uvedie aj príčinu, z akého dôvodu došlo k chybe. Výpis jednotlivých zistených chýb sa zobrazí v tabuľke zistených chýb. Ukážka časti zdrojového kódu chybového hlásenia je na nasledujúcom obrázku.

```
if ident = 'w' then begin
ListBox2.Items.Add(' Chybné návestie alebo');
ListBox2.Items.Add(' neexistujúca pozícia skoku');
ShowMessage('Nemôžem skočiť na návestie č.'+cislo+','+#13+
'    neviem nájsť pozíciu skoku.');
end;
```

Obr. 19 Časť zdrojového kódu chybového hlásenia

4.3 Úprava aplikácie

4.3.1 Odstránenie kritických chýb

S odstupom času, teda do okamihu finalizácie aplikácie, sme náhodne dospeli k niektorým nepríjemným vlastnostiam vyskytujúcich sa v našej aplikácii, ktoré môžu nepriaznivo vplývať na priebeh výpočtu ako aj na hodnotu, presnosť výsledku, ba dokonca aj samotného základu (definície) RAM stroja.

Dialo sa to najmä v súvislosti so zadávaním vstupných hodnôt, od ktorých sa v podstate odvíja celý výpočet, čím sme narazili na problém č.1. Ten sme podrobili analýze, pokúsili sme sa nájsť riešenie.

Ďalším problém, problém č.2, ktorý sme objavili mal za následok zastavenie priebehu výpočtu, kde podľa definície RAM stroja sa nemal zastaviť. Takisto sme celý problém rozanalyzovali, kde sme dospeli k riešeniu a k niektorým zaujímavým skutočnostiam.

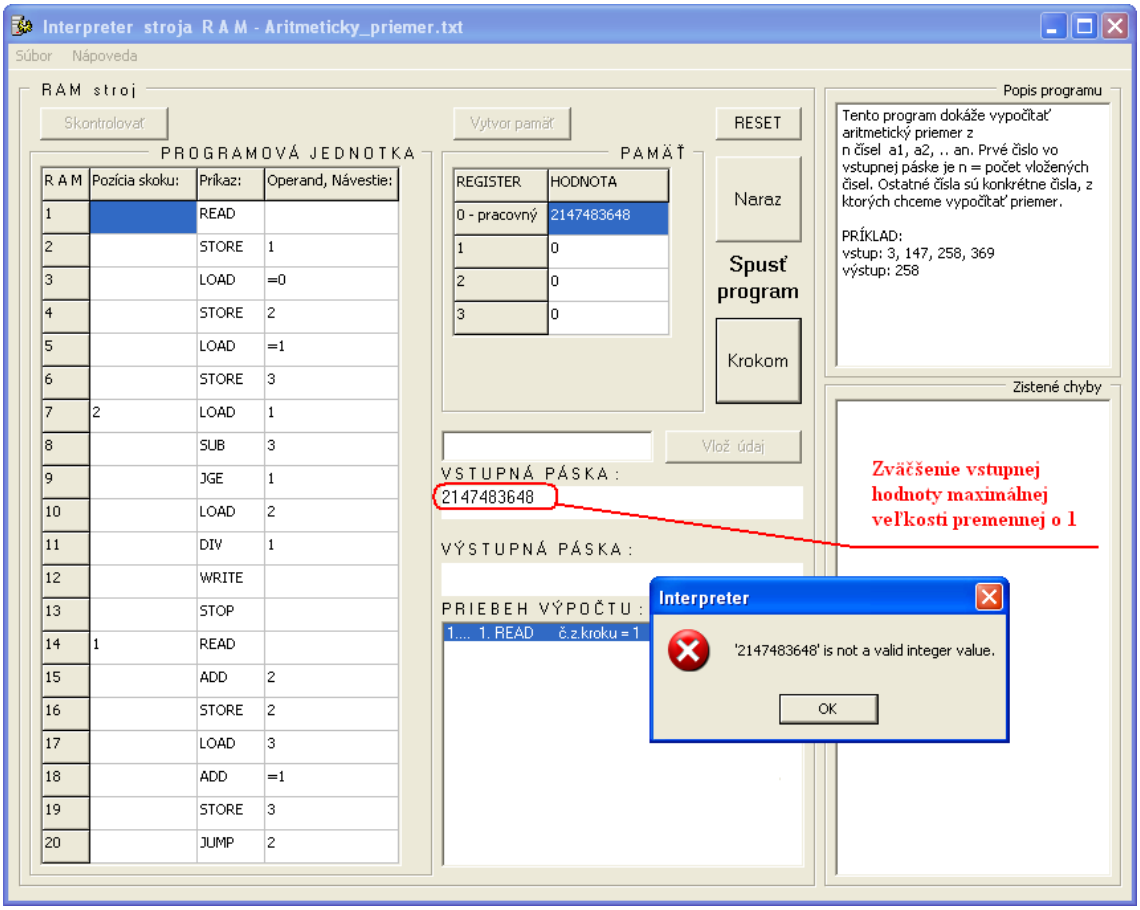
4.3.1.1 Problém č. 1

V charakteristike počítača s ľubovoľným prístupom sme nedali žiadne obmedzenia na veľkosť čísel, ktoré môžu byť uchované v jednej pamäťovej bunke. Nevýhodou uniformného hodnotenia je možnosť využitia tejto nedôslednosti k vytváraniu zdanlivo rýchlych algoritmov spôsobom, ktorý ukážeme na príklade: čísla *254*, *168*, *224*, *789* a *471* zlúčime do jediného čísla *254168224789471* a presuny uvedených čísel v pamäti a v obmedzenej miere a tiež aritmetické operácie s nimi je možné vykonávať jedným príkazom miesto piatych. Pri výpočtoch s maticami je takto možné zlúčiť celé riadky alebo stĺpce, takto vzniknutá „úspora“ času je ešte väčšia. Úvodzovky sme použili z toho dôvodu, že pri vykonávaní výpočtu na reálnom počítači je nutné uchovávať veľmi veľké čísla vo viacpamäťových bunkách, takže popísané zrýchlenie výpočtu nie je použiteľné.

Lenže definícia pamäte RAM stroja má práve toto obmedzenie - neobmedzenie. Tento problém má skutočne veľkú váhu, pretože ak v definovaní premenných pamäťových buniek v skutočnom interpreteri RAM budeme uvažovať „len“ o premenných typu integer, teda celočíselných premenných, a už len pri vkladaní obrovských čísel (hodnoty nad 2^{31} alebo pod -2^{31}), alebo násobení veľkými číslami

nehovoriac o výpočte faktoriálu, polynómu sú tieto podmienky mierne povedané veľmi obmedzené.

Ako prvý uvádzam príklad, toho keď zadaný vstupný údaj mal veľkú hodnotu (za hranicou veľkosti premennej integer, teda celého čísla v našom prípade o *1*). To platí aj v prípade zadania vstupnej hodnoty čísla menšej ako je hranica rozsahu najnižšej z možných hodnôt so znamienkom „-“, teda pod hodnotou menšou ako *-2147483648*.

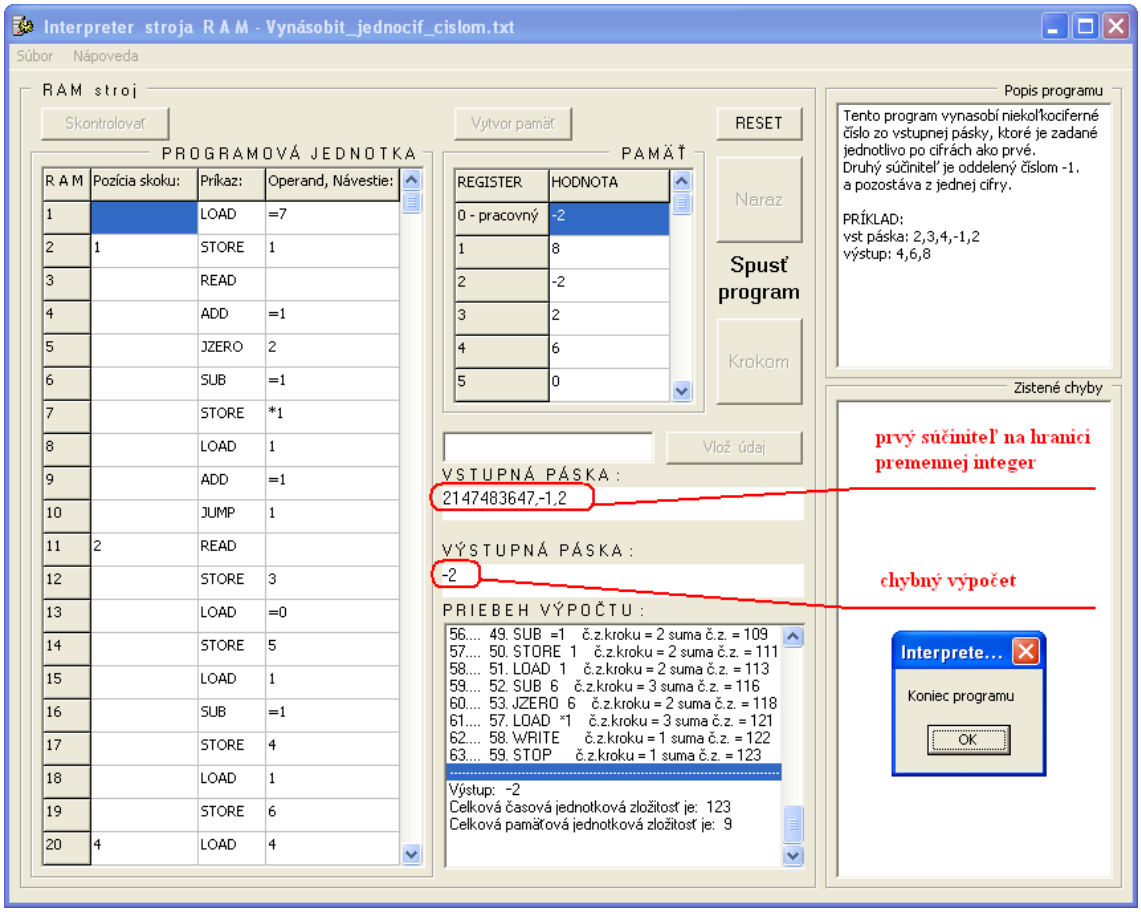


Obr. 20 Prekročenie veľkosti vstupnej hodnoty

V druhom prípade, ako to demonštruje nasledujúci obrázok je vstupná hodnota na hranici rozsahu (ešte aplikácia akceptuje hodnotu), no pri ďalších krokoch sa dopúšťa chýb, pretože medzivýsledok a tým aj celkový výsledok je nesprávny.

Čiže: *2147483647 x 2* sa nikdy nemôže rovnať číslu *-2* !!!

Aplikácia teda „nezaregistrovala chybný výsledok“ ako vidíme, ani výsledok nebol presný.



Obr. 21 Vstupná hodnota na hranici rozsahu a chybný výpočet

Možnosti riešenia:

V takýchto prípadoch máme viac možností nápravy, alebo skôr korekcie:

1) zmeníme typ premennej (napríklad z `integer` na `int64` v DELPHI),

2) informujeme prostredníctvom samotnej aplikácie, používateľa o prekročení veľkosti hodnoty premennej, alebo medzivýsledku a teda možnej chyby konečného výpočtu,

3) zmeníme samotný program RAM v tom zmysle, že sám používateľ (taký, ktorý chce vytvárať programy RAM) si navrhne a vytvorí taký postup pri písaní programu, ktorý zabezpečí presnosť výsledku v každom okamihu výpočtu. Musí si vytvoriť vlastnú bezpečnostnú filozofiu pri písaní tohto programu, ktorý musí spĺňať kritéria a špecifiká, ktoré vyplývajú zo stavby a definície stroja RAM.

V prvej možnosti, teda pri zmene typu (rozsahu) premennej nemusí vždy výpočet vykazovať správny výsledok. Zvlášť vtedy, keď medzivýsledok niektorej z operácii bol čo i len raz mimo rozsahu hodnôt tejto premennej. No pri zadávaní vstupných hodnôt má istú váhu. Nasledujúce vety nám ponúknu vysvetlenie. V oboch prípadoch je dôležitým predpokladom nutnosť zmeny zdrojového kódu aplikácie.

Ak si uvedomíme, že premenná typu integer má rozsah hodnôt celých čísel = 2^{32}, čo predstavuje hodnoty od *-2147483648* do *+2147483647*, tak pri vkladní hodnoty napríklad o *1* väčšej ako je maximálna prípustná hodnota, nám aplikácia vyhodnotí tento postup ako chybovým hlásením, ktorá oznamuje, že zadané číslo nie je v rozsahu typu celého čísla, a preto musíme vykonať nápravu, čiže vykonať „reset“ počítania inicializovaný tlačidlom „RESET“. A celý výpočet môžeme začať odznova. Táto situácia môže nastať len ak spúšťame výpočet pomocou inicializačného tlačidla „KROKOM“.

Celkom iný účinok má spôsob inicializácie výpočtu kliknutím na tlačidlo „NARAZ“. Pri tomto spôsobe, spustenia RAM programu nie je možné zastaviť aplikáciu, ktorá cyklicky oznamuje chybové hlásenie o prekročení hodnoty vstupnej premennej. Je nutné zastaviť chod aplikácie (napríklad vo WINDOWS XP) spoločne známym spôsobom, a teda súčasným zatlačením tlačidiel „Alt + Ctrl + Del“, ktorým spustíme „Správcu úloh systému Windows“, kde v záložke vyhľadáme názov našej aplikácie a našu aplikáciu ukončíme tlačidlom „Ukončiť úlohu“.

Pre obmedzenie účinku takejto fatálnej chyby môžeme vykonať úpravu zdrojového kódu našej aplikácie Interpreter stroja RAM zmenou v deklarácii premennej, s ktorou pracujeme. Jej rozsah zmeníme na 2^{64}, čo predstavuje hodnoty v rozmedzí od *-9223372036854775808* do *+9223372036854775807*.

Teda málo účinný spôsob eliminácie výskytu chýb čo sa týka výpočtu, ale lepší spôsob pri fatálnom zlyhaní pri zadávaní vstupného čísla do vstupnej pásky RAM stroja. V tomto prípade je tu menší predpoklad, že by sme do vstupnej pásky zadávali väčšiu hodnotu premennej, ktorá by mala veľkosť väčšiu ako 2^{64}. Túto činnosť môže vykonať programátor.

Riešenie č. 1 (Zmena typu premennej)

V zdrojovom kóde aplikácie, v procedúre vykonania operácií `Tform1.p_operacia` sme zmenili v deklaračnej časti tejto procedúry premenné s názvami `cislo`, `st_cislo` na typ `int64`, čo znamená zväčšenie rozsahu danej premennej. Tým istým spôsobom sme v procedúre `TForm1.p_skok` zmenili deklarovanie premennej s názvom `reg_0` na typ `int64`, čím sme zmenili aj na tomto mieste v zdrojovom kóde rozsah danej premennej. V tejto procedúre sa síce nevykonávajú žiadne aritmetické operácie, no pre podmienené skoky ako `JGE`, `JGTZ` a `JZERO` majú význam, pretože porovnávajú číslo „*0*“ s hodnotou v pracovnom registry. Preto, ak by sme nevykonali túto zmenu a po spustení aplikácie a po vložení programu RAM, v ktorom nie je zadaný žiadny príkaz skoku okrem príkazu `JUMP`, aplikácia by pracovala korektne, teda bez chybových hlásení.

Ak by sme zmenili len deklaráciu premenných, tak by počas výpočtu nad kritickými rozsahmi (nad hodnotami premenných = 2^{64}) aplikácia oznamovala chybovým hlásením skutočnosť, že sme stále mimo pôvodného rozsahu premennej, hoci sme už zmenili typ premennej celého čísla na danú hodnotu. Musíme aj v samotných procedúrach zmeniť všetky konverzie reťazcov na celé 64 bitové číslo, čo zabezpečíme zmenou `StrToInt` na `StrToInt64`. Opačná konverzia `IntToStr` zostáva zachovaná.

Riešenie č. 2 (Informovanie prostredníctvom aplikácie)

Druhou možnosťou v súčinnosti s prvou docielime tiež obmedzenie prípustnosti chyby a to v tom zmysle, že ak by používateľ, teda ten, ktorý nevytvoril program RAM, ale ho iba používa, by program spustil a neprečítal by v „popise programu“ dané obmedzenia, odporúčania, by bol znova vystavený nefunkčnosti daného RAM programu. Teda tento popis programu môžeme brať ako návod na použitie, či manuál akéhokoľvek prístroja, ktorý si kupujeme v obchode.

Takýmto spôsobom by bolo docielené iba použitie prvej možnosti korekcie možných chýb a používateľ, ktorý by tento program obsluhoval (nie tvorca programu RAM) by bol sklamaný. Preto je veľmi dôležité čítať „návod na použitie“, ktorý musí napísať tvorca RAM programu. Pričom nutným predpokladom je, aby najprv otestoval daný program RAM toľkými hodnotami, pokiaľ nedôjde ku chybnému výsledku. Tým by bolo možné stanoviť aj rozsah vstupných hodnôt, a tým predísť aj výsledným chybám.

Ku príkladu v popise programu vložíme informácie typu: „Môžete zadávať vstupné hodnoty len od – do“ a podobne. Túto činnosť môže vykonať používateľ, teda tvorca RAM programu. Tento druh korekcie má lepší účinok ako predchádzajúci spôsob.

Riešenie č. 3 (Zmena programu RAM)

Tretia možnosť je na používateľovi programu, akým spôsobom vytvorí samotný program RAM, ktorý môže byť ešte bezpečnejším voči chybným výsledkom. Bohužiaľ proti zadaniu vstupnej hodnoty mimo spomenutý rozsah hodnôt, je neúčinný. Tu sa jedná o ponímanie vstupnej hodnoty nie ako celku, ale vstupnú hodnotu sme rozdelili na viacero častí poprípade na jednotlivé číslice.

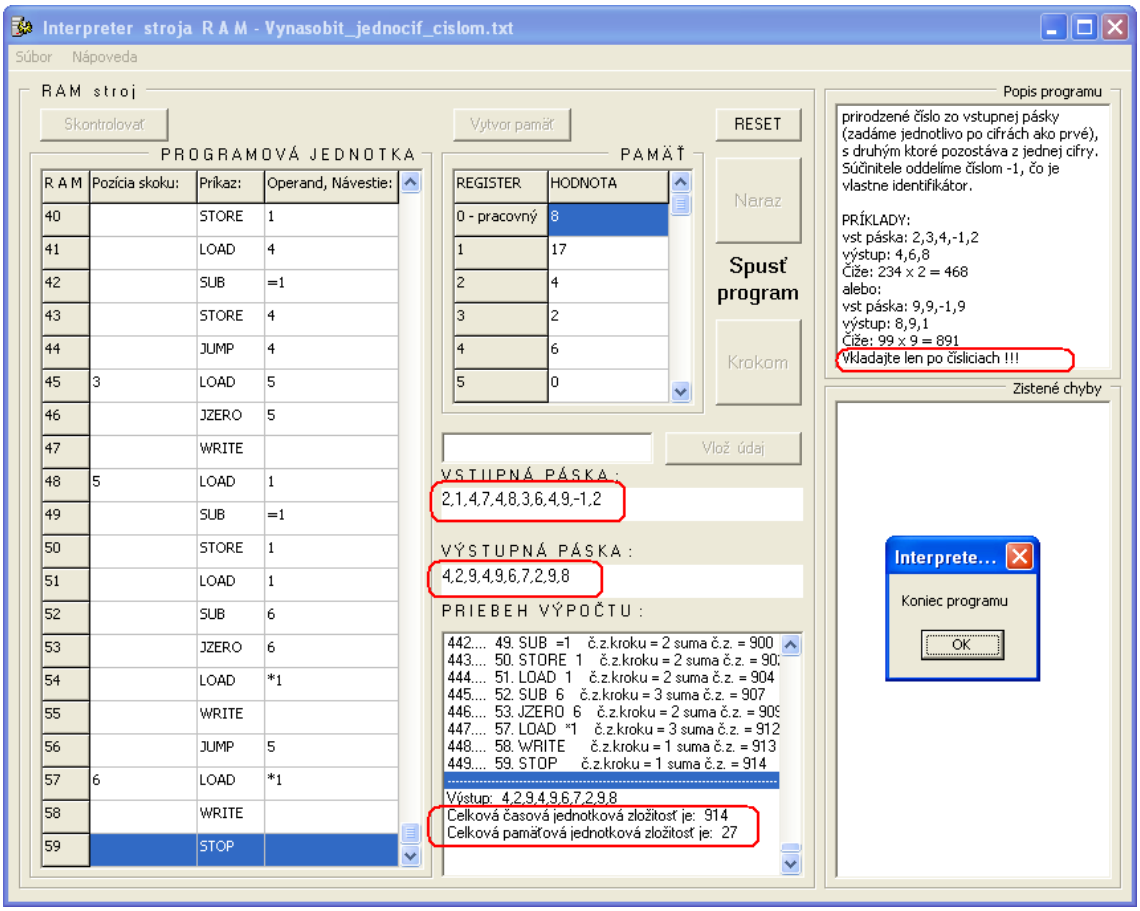
Napríklad: vstupnú hodnotu *2174783649* vkladáme zo vstupnej pásky po jednotlivých čísliciach *2*, *1*, *7*, *4*, *7*, *8*, *3*, *6*, *4*, *9* a aplikácia dokáže tieto údaje spracovať bez obmedzenia tým dospeje k správnosti výsledku. Zámerne sme na ukážku vybrali číslo väčšie ako je rozsah vstupných hodnôt, aby sme mohli ukázať, že s takýmto spôsobom sa dajú riešiť príklady programov RAM.

No musíme si tiež uvedomiť skutočnosť, že týmto spôsobom tvorby programu zvyšujeme pamäťovú náročnosť RAM stroja, no výsledok dostávame presný. Tvorca programu teda môže postupnosťou krokov v programovej jednotke zabezpečiť skutočnú

funkčnosť, čiže dopracovanie sa ku správnosti konečného výsledku. Túto činnosť môže vykonať používateľ (tvorca programu RAM). Tento spôsob riešenia problému č.1 má ešte lepší účinok ako predchádzajúci spôsob.

Jedným z pekných príkladov kombinácie informovania používateľa o obmedzeniach a vytvorením tzv. „bezpečného“ RAM programu, čiže kombinácie 2. a 3. možnosti je nasledujúci program RAM, ktorý dokáže vynásobiť akokoľvek veľké číslo s iným jednociferným no za cenu zvyšovania počtu pamäťových buniek (registrov), s ktorými sa bude pracovať a teda pamäťová zložitosť sa zväčšuje úmerne počtu cifier v čísle prvého súčiniteľa.

Z nasledujúceho obrázku môžeme zistiť, že skutočne daný program presne vypočítal hodnotu násobenia čísel *2174783649* a *2*, kde sme dostali výsledok rovnajúci sa *4349567298* a tým potvrdil náš predpoklad. V časti „Popis programu“ je veta upozorňujúca na spôsob vkladania vstupných hodnôt. V časti priebehu výpočtu sme tiež zvýraznili skutočnosť, že daný program mal zvýšené nároky na pamäť a tým sa zvýšila aj jeho časová náročnosť.



Obr. 22 Vstupná hodnota za hranicou rozsahu a presný výsledok

Zhrnutie

Ak by sme chceli zhrnúť riešenie problému č.1, dané možnosti riešenia, určite zlepšili chod niektorých náročnejších výpočtov. Samozrejme zmeniť celkovo zdrojový kód a to natoľko, aby sme vytvorili aplikáciu, ktorá bezpečne dokáže dôjsť až do konečnej fázy, bez toho aby nedochádzalo ku spomenutým chybám, pri zadaní veľkých vstupných hodnôt, sa samozrejme dá, ale myslím, že dané tri riešenia predchádzajúceho problému sú postačujúce na vykonanie, alebo overenie všetkých programov RAM a potierajú zistené riziká vzniku problému. Ako sme spomenuli, všetky doteraz uvedené riešenia spoločne napomôžu tomu, aby sme sa pri výpočtoch nedopúšťali chýb.

Pre daný účel učebnej pomôcky, alebo priblíženia predstavy funkcie počítačov nemá význam veľkého zviditeľňovania tohto problému.

4.3.1.2 Problém č. 2

Daný problém súvisí s otázkou definovania RAM stroja. Spočiatku sme uvažovali nad veľkosťou použitej pamäte, pričom sme vychádzali so skutočnosti, že v reálnych podmienkach nie je možné vytvoriť neobmedzený počet pamäťových buniek, teda registrov. Uspokojili sme sa s tým, že vloženie údaja do pamäťového registra, vlastne limitoval veľkosť celej pamäte, teda keď sme chceli pracovať s nejakým údajom v registri, museli sme ho pred tým uložiť do určitého pamäťového miesta. Týmto sme získali aj informáciu o adrese registra. To znamená, že ak sme vložili ľubovoľný údaj do pamäťovej bunky, s najväčšou hodnotou adresy, táto bola určujúcou hodnotou veľkosti použitej pamäte. Čo malo za následok, že ak sme chceli vykonať inštrukciu niektorej z aritmetických operácií (príkaz `ADD i`, `ADD *i` a ostatné aritmetické operácie), ktorá požadovala výber z registra s väčšou adresou ako bola alokovaná pamäť, aplikácia nám oznámila, že daná pamäťová bunka ešte nebola vytvorená, a preto nemôže pokračovať výpočet. Čo nie je v súlade s definíciou samotného stroja RAM, pretože pamäť musí obsahovať nekonečný počet registrov.

Navyše, ak sme chceli vybrať údaj z pamäťovej bunky mimo rozsah pamäti interpretera (príkaz `LOAD i`, `LOAD *i`), aplikácia oznámila chybovým hlásením túto skutočnosť, že do daného registra ešte nebol vložený žiaden údaj. Čo tiež nebolo v súlade s definíciou počítača s ľubovoľným prístupom, pretože pred spustením výpočtu na teoretickej úrovni, je v každom registri programu vložená hodnota „*0*“.

Ak sme chceli z časti dodržať definíciu, že pamäť má vždy aspoň použiteľný počet registrov (nie nekonečný ako to vyplýva z definície) a hodnota údaja vložená v tomto registri má mať hodnotu „*0*“ (keď tam ešte nebola zadaná žiadna hodnota), museli sme vykonať zásah do zdrojového kódu aplikácie. No pokiaľ by sme vytvorili program RAM tým spôsobom, že by sme vôbec neuvažovali o nepriamom adresovaní a dbali by sme na to, aby sme sa vyhli horeuvedeným prípadom, nemuseli by sme meniť vôbec nič. Preto sme vykonali niektoré úpravy v zdrojovom kóde aplikácie. Síce sme vizuálne neinterpretovali nekonečný počet registrov, no vždy, keď si to vyžadoval priebeh výpočtu boli dané registre vždy k dispozícii. Navyše táto skutočnosť značne zjednodušovala stanovenie pamäťovej zložitosti RAM stroja, čo sme vysvetlili už v predošlej kapitole.

Riešenie

Riešením bola zmena v procedúre s príznačným názvom `Tform1.pridaj_riadky`, kde sme vložili testovací cyklus, ktorý zistil koľko riadkov z komponentu `StringGrid2` musíme vložiť k doterajšej pamäti a následne na to vytvoril hľadanú pamäťovú bunku, do ktorej vložil hodnotu „0“. Čím sme dodržali pravidlá definovania samotného chodu stroja RAM. Hneď za tým sme vstúpili do procedúry `zarovnat_2`, čím sme vizuálne upravili veľkosť pamäte. S takto vytvoreným registerom sa už mohlo pokračovať vo výpočte, to znamená, že za behu programu sme vytvorili ďalšiu pamäťovú bunku (register).

No nestačilo nám iba vytvoriť danú procedúru, bolo nutné aby sme do nej vstupovali z iného miesta v programe. Konkrétne z procedúry vykonania operácií, teda `TForm1.p_operacia`, a to na miestach súvisiacich s vykonaním aritmetických operácií a tiež so všetkými operáciami, ktoré obsahovali vo svojom operande „*“, čiže pri operáciách s nepriamym adresovaním.

Priblíženie korekcie v zdrojovom kóde (vytvorenie novej procedúry zväčšujúcu pamäť, čiže zvyšuje počet registrov) je na nasledujúcom obrázku.

```
//======  P R I D A N I E   R E G I S T R O V   P R I   V Ý P O Č T E
procedure TForm1.pridaj_riadky(var operand: string);
var
  i: integer;
begin
  for i:= StringGrid2.RowCount to StrToInt(operand)+1 do begin
    StringGrid2.RowCount:= StringGrid2.RowCount + 1;
    StringGrid2.Cells[1,StringGrid2.RowCount]:= '0';
    StringGrid2.Cells[0,StringGrid2.RowCount]:= IntToStr(i);
  end;
  zarovnat_2;
end;
```

Obr. 23 Úprava zdrojového kódu „Pridanie registrov“

Na ďalšom obrázku sú označené časti zdrojového kódu, kde bol vložený odkaz na vstúpenie do podprogramu `TForm1.pridaj_riadky`, ktorý sme pridávali do procedúry `Tform1.p_operacia`. Na obrázku sú tieto miesta označené ako doteraz (červený ovál).

```
    delete(operand,1,1);                              // odstránime "*" a zostane
    if StringGrid2.RowCount < StrToInt64(operand)+2    // musí byť nezáporné
      then pridaj_riadky(operand);
    if StrToInt64(StringGrid2.Cells[1,StrToInt64(operand)+1]) >= 0 then begin
      odkaz:= StringGrid2.Cells[1,StrToInt64(operand)+1];
      operand:= odkaz;
      cas_narocnost:= 1;
      zarovnat_2;                                     //  poradové číslo registr
    end else begin                                    //  nemôže byť záporné
      Timer1.Enabled:= false;                         //             |
      i:= riadok;                                     //             |
      chyby('q',operand, i);  //  <---------------------------------?
      stop:= true;
      exit;
    end;
  end;

  // Skontrolujeme či máme dosť registrov,
  // do ktorých chceme niečo uložiť, alebo z nich vybrať
  if StringGrid2.RowCount < StrToInt64(operand)+2 then pridaj_riadky(operand);


  // LOAD
  if prikaz = 'L' then begin
    if StringGrid2.RowCount < StrToInt64(operand)+2 then
      pridaj_riadky(operand);
    cislo:= StrToInt64(StringGrid2.Cells[1,StrToInt64(operand)+1]);
    StringGrid2.Cells[1,1]:= IntToStr(cislo);
    cas_narocnost:= cas_narocnost +2;
  end;
```

Obr. 24 Príkaz vstupu do podprogramu „Pridanie registrov“

Zhrnutie

Daným krokom v korekcii zdrojového kódu, sme podstatným spôsobom zmenili chod aplikácie. Podľa definície RAM stroja naša aplikácia spoľahlivo pracovala, čo sme si samozrejme overili aj praktickou skúškou, ale paradoxne týmto spôsobom sme do určitej miery obmedzili overovaciu schopnosť našej aplikácie. Čo sme zistili až dodatočne. Týka sa to najmä tvorby programov RAM.

Ukážme si situáciu pred zmenou aplikácie v zdrojovom kóde. Kde pri nepriamom adresovaní napríklad v inštrukcii `ADD *i`, keď sme nemali vložený žiaden údaj v registri, ktorý ukazuje na register, v ktorom by mal byť uložený, nám aplikácia vypísala hlásenie práve o tejto udalosti a priebeh výpočtu bol ukončený. Mohli sme daný program RAM zmeniť.

Pri situácii po zmene v zdrojovom kóde a pri rovnakom prípade nepriameho adresovania, keď sme štandardne nemali vytvorený register, na ktorý ukazoval register `i` svojou hodnotou, sme nedostali žiadne chybové hlásenie od aplikácie, pretože sme pracovali podľa definície stroja RAM, to znamená pred tým sme vytvorili pamäťovú bunku s vloženou hodnotou „*0*“.

V podstate sme simulovali inštrukciu vcelku nezmyselnú, teda `ADD =0`, čím sme k hodnote pracovného registra pripočítali hodnotu „0“, čo nijak nezmení medzivýsledok. A čo sa týka časovej zložitosti, tak sme použili inštrukciu s najväčšou čiastkovou pamäťovou zložitosťou, no a nakoniec vytvorením nového registra sme zväčšili aj pamäťovú zložitosť celého programu RAM.

Je možné, že v niektorých prípadoch ako napríklad pri inštrukcii `MULT *i` kde sa zmení výsledná hodnota v pracovnom registri na hodnotu „0“, môže mať takýto spôsob pri vytváraní programu svoje opodstatnenie. Jednako však najdôležitejšie hľadisko pri vytváraní programov RAM je obozretnosť pri ich tvorbe no a nakoniec otestovanie a následné porovnávanie už dosiahnutých výsledkov. Vyriešením tohto problému sme sa priblížili o ďalší kus k modelu teoretického stroja RAM.

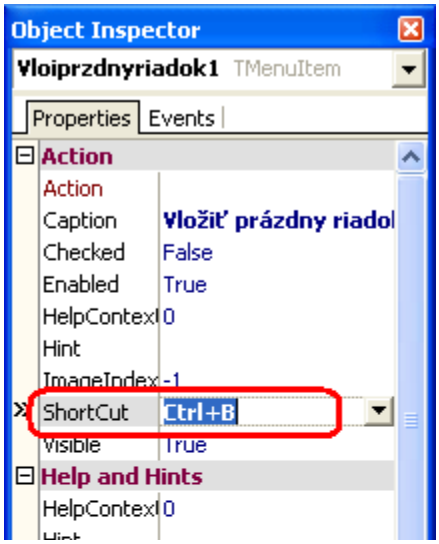
4.3.2 Úprava aplikácie pridaním „štandardnej“ výbavy

4.3.2.1 Pomoc

Každá aplikácia by mala mať štandardné vybavenie, ako napríklad pomoc, kde si používateľ vie poradiť s riešením problematických otázok spojených so spustením, či písaním alebo otváraním, ukladaním vytvorených programov RAM. Všetky spomenuté problematické otázky sme zhrnuli do jedného dokumentu, s názvom „`Pomoc.pdf`“, ktorý je súčasťou aplikácie. Tieto potrebné informácie si vieme hneď vyhľadať prostredníctvom vytvoreného obsahu a hypertextových odkazov, ktorý má pôsobnosť v rámci celého dokumentu. Pomoc k ovládaniu danej aplikácie nájdeme v hlavnom menu, teda na hlavnej lište pod názvom „Nápoveda“, alebo zatlačením klávesy (klávesovou skratkou) „F1“ máme túto pomoc hneď k dispozícii.

4.3.2.2 Skrátená voľba

K štandardnej výbave mnohých aplikácií patrí aj Skrátenú voľba (tzv. klávesové skratky), ktoré pomáhajú pri manipulácii. Úpravu zdrojového kódu sme vykonali tým spôsobom, že pre komponent, v ktorom sme sa rozhodli použiť skrátenú voľbu, teda v časti vlastností nastavení „Object Inspector“ v príslušnej kolónke „ShortCut“ sme umiestnili názov klávesy (tiež skrátene), poprípade ich kombinácií. Nasledujúci obrázok vykresľuje túto skutočnosť.



Obr. 25 Vytvorenie skrátenej voľby „Klávesovej skratky“

4.3.2.3 „Bublinová nápoveda“

Ďalším príspevkom radu štandardných vlastností môže byť vybavenie „bublinová nápoveda“ čo nie je nič iné ako opis či popis komponentu, významu, alebo vlastnosti, nad ktorou je vykonaná nejaká udalosť (zatlačenie tlačidla, alebo tlačidlo myši, alebo len prechod kurzoru myši nad daným komponentom). Úpravu sme vykonali v rovnakej časti nastavení ako pri definovaní klávesových skratiek v kolónke „Hint“, alebo vložením do tela niektorej z procedúr ako to znázorňuje nasledujúci obrázok.

```
procedure TForm1.Button5Click(Sender: TObject);
begin
  Label3.Caption:= ('Spusť'+char(13)+'program');
  Label3.Visible:= true;
  Button5.Enabled:= false;        // vytvorenie pamäte
  Button6.Enabled:= true;         // načítanie do tabuľky /SringGrid2
  ListBox2.Items.Clear;
  riadok:= 1;
  ListBox3.Items.clear;
  ListBox3.Enabled:= true;
  ides:= true;
  reset_p;
  pamat;
  StringGrid2.Hint:= 'Modrá = aktívna bunka';
  StringGrid1.row:= riadok;
end;
```

Obr. 26 Vytvorenie „bublinová nápoveda“ nad pamäťou

4.3.3 Redukcia zdrojového kódu

Medzi niektoré korekcie by sme mohli zahrnúť aj redukovanie zdrojového kódu, čo v konečnom dôsledku zmenší veľkosť aplikácie a ovplyvní celkovú rýchlosť aplikácie. Musí sa to ale diať spôsobom, ktorý negatívne neovplyvní funkčnosť výslednej aplikácie. Nakoľko pri behu aplikácie sme využívali tie isté úkony, ktoré boli zapísané v tele viacerých procedúr, zväčšovali sme zdrojový kód a tým aj veľkosť celej aplikácie. Zmeny sme vykonali v procedúrach vykonania operácií „`TForm1.p_operacia`“ a vykonania skokov `Tform1.p_skok`. Použitím tejto metódy sme zredukovali veľkosť zdrojového kódu a tým ho aj sprehľadnili. Pre veľkosť týchto zmien v zdrojovom kóde neuvádzame na ukážku obrázky.

4.3.4 Pridanie jednoduchých efektov

Ďalšie úpravy aplikácie boli zamerané na vizuálny efekt pri priebehu výpočtu. Mali sme týmto spôsobom možnosť sledovať postupnosť krokov v programovej jednotke, pri sledovaní pamäte sme mohli vidieť, ktorá z pamäťových buniek sa práve používa a pri sledovaní priebehu programu, presnejšie po jeho ukončení, bol časť priebehu a výsledku výrazne oddelená. Všetky tieto efekty boli zvýraznené štandardnou farbou (modrá). Úpravy sme vykonali na komponente „Pamäť“ vložením príkazu `StringGrid2.Row`, ktorý zvýraznil políčko tohto komponentu práve v čase aktivity prebiehajúcej procedúry vykonania operácií `Tform1.p_operacia` a v procedúre výpočtu `TForm1.rataj` a inicializácie výpočtu `Tform1.reset_p`. V komponente „Programová jednotka“ vložením príkazu `StringGrid1.Row`, do procedúr otvorenia súboru `Tform1.N1Click`, výpisu priebehu výpočtu `Tform1.vypis`, do procedúry výpočtu `Tform1.rataj`, do procedúry vytvorenia pamäte `Tform1.Button5Click`, a inicializácie výpočtu `Tform1.reset_p`. Všetky tieto zmeny interpretovali v aplikácii vykonávanie príkazov v čase.

4.3.5 Obohatenie aplikácie o nové inštrukcie

Hlavným dôvodom tejto úpravy bolo to, aby RAM stroj dokázal pracovať aj s ostatnými príkazmi, ktoré sa odlišovali názvom alebo funkciou od inštrukcií, s ktorými pracuje naša aplikácia. Pre skupinu používateľov využívajúcich dané inštrukcie `JGTZ` a `HALT` sme zmenili zdrojový kód. Bez tejto úpravy by naša aplikácia nevedela identifikovať dané príkazy. Bolo by nutné prepísať program RAM. Najprv sme prikročili ku zmene v zdrojovom kóde v procedúre prevodu programu `Tform1.prevod_suboru`, pri kontrole syntaxe `TForm1.kontrola_instrukcii` a nakoniec v procedúre výpočet `Tform1.rataj`. Pre rozsiahle zmeny v zdrojovom kóde neuvádzame na ukážku obrázky.

4.3.6 Obohatenie aplikácie o výpočtovú zložitosť

Zostavenie interpretera RAM, čiže návrh vizuálnej stránky aplikácie, definovanie jednotlivých funkcií, vytvorenie jednotlivých komponent, samotná funkčnosť ako celku bolo dôležitou súčasťou práce. No pri ďalšom analyzovaní algoritmov, nás zajíma nielen to, či dostaneme riešenie požadovanej úlohy, ale ako dlho potrvá výpočet a koľko pamäte je treba vymedziť pri práci na vykonanie tejto úlohy, a preto bolo nutné zaviesť pojem výpočtovej zložitosti, ktorá vlastne zahŕňa dva už spomenuté aspekty:

- **časová výpočtová zložitosť** (ako dlho trvá vykonanie výpočtu),
- **pamäťová výpočtová zložitosť** (koľko pamäte potrebujeme na výpočet).

V oboch prípadoch nám nejde o určenie, nejakého presného času vypočítaného od začiatku spustenia až po koniec algoritmu (aplikácie), alebo presné určenie počtu pamäťových buniek, ale o určitú charakteristickú črtu, ktorá nám jednotlivé algoritmy pomôže porovnať a ukáže nám určitý trend rastu alebo poklesu výpočtovej zložitosti v závislosti na veľkosti vstupných údajov.

Samozrejme danú problematiku bolo nutné riešiť s ohľadom na reálne možnosti vytvorenej aplikácie interpretera RAM. Na základe podmienok, definícii, obmedzení, vychádzajúcich z teórie, ktoré sme museli buď akceptovať, alebo v najhoršom prípade k nim čo najviac priblížiť, sme upravovali jednotlivé časti zdrojového kódu. To znamená vytvorenie nových procedúr a funkcií, pridanie nových premenných pri dodržaní funkčnosti celého projektu ako aj samotnej funkčnosti týchto vylepšení vytvorenej aplikácie. Ďalším krokom bolo porovnávanie teoretických výsledkov s praktickými. Ak sme pri porovnaní dospeli k nesúladu bolo nutné nájsť a zmeniť, teda upraviť aplikáciu.

4.3.6.1 Časová zložitosť

Existuje mnoho rôznych spôsobov, ako zmerať rýchlosť výpočtu. Najjednoduchším spôsobom je zapísať algoritmus v niektorom z programovacích jazykov, vykonať výpočet na počítači a určiť, ako dlho trval. Tento pokus možno pochopiteľne vykonať len pre malé množstvo skúšobných príkladov, z ktorých nemôžeme stanoviť obecné závery. Navyše podľa tohto hľadiska sú dosiahnuté výsledky závislé aj na technických parametroch a programovom vybavení použitého počítača, na voľbe programovacieho jazyka a tiež na schopnostiach, zručnosti a skúsenostiach programátora. Celý proces by bol zdĺhavý a značne neefektívny.

Ak teda hovoríme o tom, že určitý algoritmus je „rýchly“ alebo tiež „pomalý“, musíme mať možnosť sledovať určitú charakteristiku, ktorou by sme určité algoritmy mohli porovnať. Pri tejto časti diplomovej práce, je preto tiež nutné aspoň stručne sa oboznámiť s definíciou časovej zložitosti.

Teoretické úvahy

Jedna z metód odhadu doby, potrebnej k vykonaniu príkazu počítača s ľubovoľným prístupom, vychádza z toho, že pre binárny /dvojkovo kódovaný/ zápis čísla n potrebujeme $[\log (|n| + 1)]$, čo zaokrúhlene zodpovedá $[\log |n|$ bitov (+ znamienko)$]$ a väčšinu príkazov (s výnimkou násobenia a delenia) možno vykonať v čase priamoúmernému súčtu dĺžok binárnych zápisov čísel, s ktorými sa pracuje, teda súčtu dĺžok ich logaritmov. Celková doba výpočtu sa potom určí ako súčet trvania všetkých vykonaných príkazov.

Logaritmické hodnotenie nezodpovedá celkom presne reálnej situácii, pretože spracovávané čísla sa na konkrétnych počítačoch obvykle zmestia do jednej až dvoch pamäťových buniek a doba potrebná k vykonaniu operácie s pamäťovou bunkou závisí len veľmi málo od jej obsahu.

Lepšiu zhodu s empirickými skúsenosťami preto dosahuje tzv. uniformné hodnotenie, ktoré predpokladá, že každá operácia počítača s náhodným prístupom trvá jednotku času, to znamená že dĺžka výpočtu je určená počtom vykonaných príkazov. Dopustíme sa síce zjednodušenia, pretože napr. príkaz `MULT *i` trvá v praxi dlhšie ako napríklad `READ` alebo `STOP`, no pritom pomer času trvania najrýchlejšej a najpomalšej inštrukcie je konštantný. A preto môžeme hovoriť o uniformnom hodnotení

umožňujúcom stanoviť rozmedzie, v ktorom leží skutočná doba výpočtu, s presnosťou až na multiplikatívnu (násobnú) konštantu, čo je vlastne pre naše účely dostatočné.

Znamenalo to, že sme časovú zložitosť určili spočítaním všetkých hodnôt čiastkových časových zložitostí (či už sú uložené v registroch, alebo vo vstupnej páske), s ktorými jednotlivé príkazy RAM programu pracujú. Ide vlastne o uniformné hodnotenie.

Majme teda $k : Z \rightarrow N$, kde k je pamäťová funkcia, kde zložitosti všetkých príkazov môžeme vyjadriť, prirodzeným číslom N (hodnotou), kde jednotlivý príkazom s ohľadom na operand či návesť sme priradili čiastkové hodnoty časových zložitostí:

Tab. 1 Priradenie hodnôt čiastkových časových zložitostí inštrukciám

Príkaz	Zložitosť inštrukcie	Hodnota
`READ`	k (vstup)	1
`WRITE`	k (výstup)	1
`LOAD =i`	k (i)	1
`LOAD i`	k (i) + k (c(0))	2
`LOAD *i`	k (i) + k (c(i)) + k (c(c(0)))	3
`STORE i`	k (i) + k (c(i))	2
`STORE *i`	k (i) + k (c(i)) + k (c(c(i)))	3
`ADD =i`	k (i) + k (c(0))	2
`ADD i`	k (i) + k (c(i)) + k (c(0))	3
`ADD *i`	k (i) + k (c(i)) + k (c(c(i))) + k (c(0))	4
`SUB =i`	k (i) + k (c(0))	2
`SUB i`	k (i) + k (c(i)) + k (c(0))	3
`SUB *i`	k (i) + k (c(i)) + k (c(c(i))) + k (c(0))	4
`MULT =i`	k (i) + k (c(0))	2
`MULT i`	k (i) + k (c(i)) + k (c(0))	3
`MULT *i`	k (i) + k (c(i)) + k (c(c(i))) + k (c(0))	4
`DIV =i`	k (i) + k (c(0))	2
`DIV i`	k (i) + k (c(i)) + k (c(0))	3
`DIV *i`	k (i) + k (c(i)) + k (c(c(i))) + k (c(0))	4
`JZERO`	k (i) + k (návesť)	2
`JUMP i`	k (návesť)	1
`JGE i`	k (i) + k (návesť)	2
`JGTZ i`	k (i) + k (návesť)	2
`STOP` alebo `HALT`		1

Pri letmom pohľade je zrejmé, ktoré z daných príkazov, trvá najdlhšiu dobu, ktoré kratšiu čo v teoretickom modeli musíme akceptovať. Teda aj v našom interpreteri RAM. I keď sa dá povedať, že v skutočnosti, dané kroky aplikácie ležia možno v inom časovom rozpätí (reálna časová zložitosť) ako v uniformnom hodnotení. No uniformné hodnotenie je pre nás smerodajné, pretože interpretujeme aj tento údaj, ktorý môžeme tiež použiť ako vyučovaciu pomôcku, ktorá nám umožní pochopiť (aj keď len v hrubých rysoch) fungovanie počítača, alebo ako môže daný algoritmus zmeniť dĺžku času výpočtu, respektíve ako limitovaná veľkosť pamäte, alebo ako všetky tieto aspekty navzájom súvisia.

Definícia 1. Časová výpočtová zložitosť algoritmu *T(n)*, ktorá udáva, koľko elementárnych operácií sa vykoná pri riešení problému rozsahu *n* (Lovászová-Vozár, 2007).

Pri tejto príležitosti zavedieme aj pojem asymptotickej výpočtovej zložitosti, ktorú definujeme ako rád stúpania funkcie zložitosti. Práve tento ukazovateľ je dôležitým, pri určení v akom rozmedzí zložitostí sa daný algoritmus nachádza. Rozdeľujeme dva druhy asymptotickej výpočtovej zložitosti.

Definícia 2. Asymptotická horná hranica funkcie *g(n)* je funkcia *f(n)* taká, že $\exists c,m>0 \forall n \in N$, $n \geq m$ platí $g(n) \leq c.\ f(n)$ a zapisujeme $g(n) = O(f(n))$. (Hynek, 2008)

Táto nám udáva hornú hranicu časovej zložitosti, najhoršiu možnosť, alebo ľudovo povedané „v najhoršom prípade“.

Definícia 3. Asymptotická dolná hranica funkcie *g(n)* je funkcia *f(n)* taká, že $\exists c,m>0 \forall n \in N$, $n \geq m$ platí $g(n) \geq c.\ f(n)$ a zapisujeme $g(n) = \Omega(f(n))$. (Hynek, 2008)

Táto nám udáva dolnú hranicu časovej zložitosti, najlepšiu možnosť, alebo ľudovo povedané „v najlepšom prípade“. Obe funkcie asymptotickej zložitosti vieme využiť pri určení časovej ako aj pamäťovej zložitosti algoritmu.

Napríklad máme za úlohu pomocou algoritmu tzv. bublinkového triedenia zoradiť čísla od najmenšieho po najväčšie. Kde *n* je počet vložených čísel. Dospeli by sme k výsledkom:

Ak do vstupu vkladáme hodnoty od najnižšej po najvyššiu, daným algoritmom, môžeme dosiahnuť:

Asymptotickú dolnú hranicu časovej zložitosti $\Omega(f(n)) = n$.

Pri vložení 100 čísel by program v najlepšom prípade musel vykonať $\Omega(f(n)) = 100$ operácií, čiže *100*.

Ale naopak, ak najskôr vkladáme hodnoty od najvyššej po najnižšiu, môžeme dosiahnuť: Asymptotickú hornú hranicu časovej zložitosti $O(f(n)) = n^2$.

Kde vložením 100 čísel by program v najhoršom prípade musel vykonať $O(f(n)) = 100^2$ operácií, čiže *10 000*.

Asymptotická zložitosť algoritmu predstavuje trend, ktorým sa či už časová, alebo pamäťová zložitosť uberá.

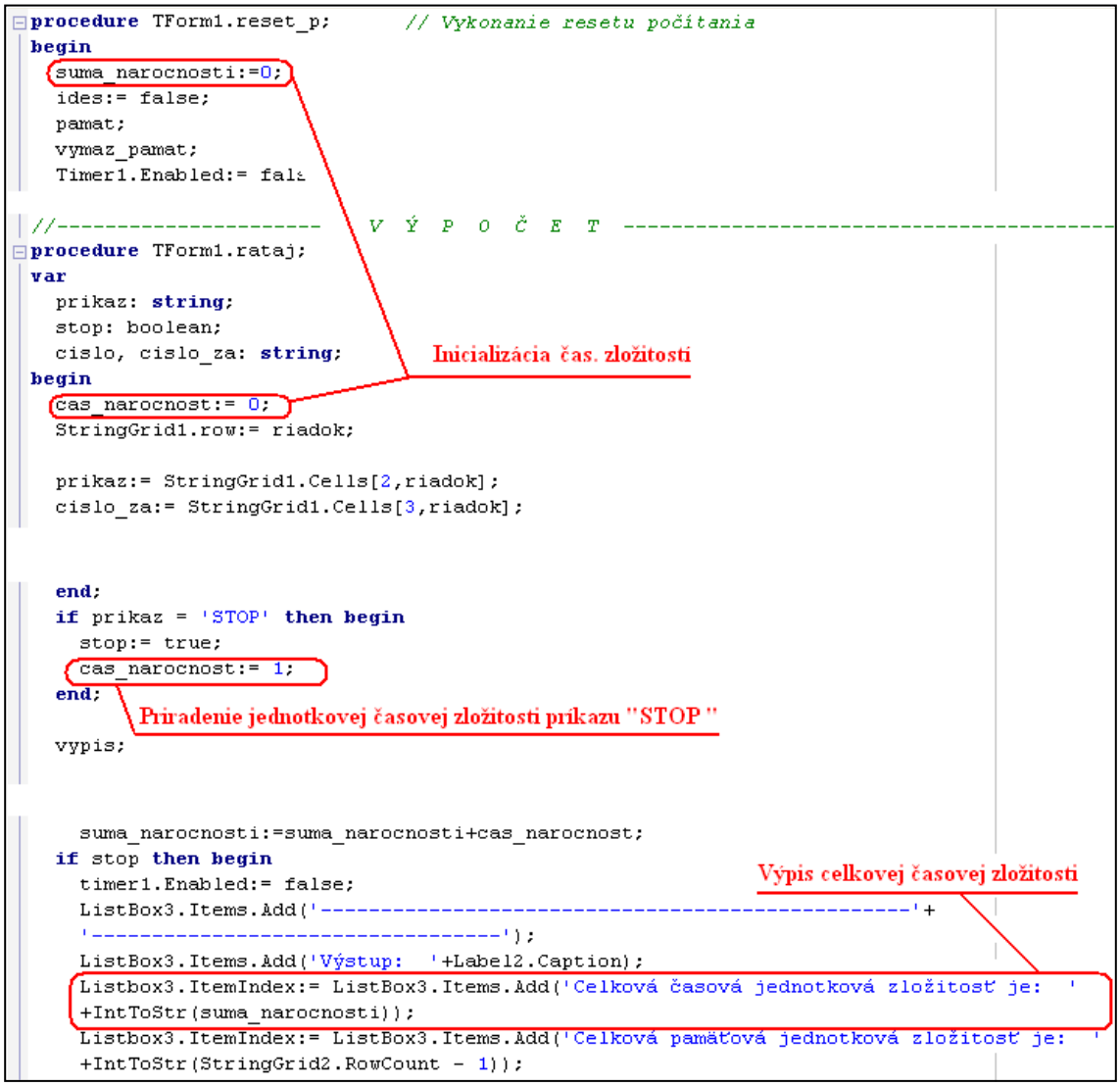
Praktické riešenie

Riešenie daného problému spočíva v priradení jednotkových hodnôt časových zložitostí ku každej už vykonanej inštrukcii. V priebehu výpočtu aplikácia tieto hodnoty spočítava. Po ukončení, alebo zastavení (pri chybe) výpočtu dostávame vlastne konečný údaj celkovej časovej zložitosti konkrétneho programu RAM.

Realizácia úpravy zdrojového kódu.

Na začiatku sme zvolili (deklarovali) premenné `cas_narocnost` - jednotková časová zložitosť a `suma_narocnosti` - súčet všetkých jednotkových časových zložitostí. Nakoľko sme s nimi pracovali vo viacerých procedúrach, zaradili sme ich medzi globálne premenné. V procedúre `TForm1.rataj` sme najprv inicializovali jednotkovú časovú zložitosť a tak isto aj celkovú časovú zložitosť v procedúre `TForm1.reset_p` (vždy na začiatku má tento údaj hodnotu = 0). Na konci procedúry `TForm1.rataj` v časti kde sa program zastaví sa nachádza vizuálny komponent (`Listbox3`), v ktorom aplikácia vypíše tieto údaje časových zložitostí.

Počas chodu výpočtu, každý prechod danou inštrukciou priradí hodnotu jednotkovej časovej zložitosti v príslušnej veľkosti podľa tabuľky „*Obr. 27“*. Po každej vykonanej inštrukcii nám aplikácia čiastkovú časovú zložitosť inicializovala do východiskového stavu `cas_narocnost:= 0`, zatiaľ čo premenná `suma_narocnosti` zväčšovala svoju hodnotu každou ďalšou hodnotou celkovej jednotkovej časovej zložitosti do zastavenia výpočtu poprípade vykonaním v procedúry `TForm1.reset_p`.



Obr. 27 Zmeny v súvislosti s určením čiastkovej časovej výpočtovej zložitosti

So zameraním sa na teoretický rozbor algoritmov, s programami pre počítač s ľubovoľným prístupom RAM, nám postačí určenie časovej výpočtová zložitosť algoritmu *T(n)* a asymptotickej hornej hranice časovej výpočtovej zložitosti algoritmu *O(f(n))*.

4.3.6.2 Časová zložitosť

Podobným spôsobom ako v prípade časovej zložitosti algoritmu aj pamäťovú zložitosť vieme definovať. Pri určovaní pamäťovej zložitosti musíme zasa vychádzať z predpokladu, že RAM stroj pracuje s pamäťou skladajúcou sa z nekonečného množstva registrov, čo v reálnych podmienkach výpočtovej techniky nie je realizovateľné. Navyše každá z týchto buniek môže uchovávať neobmedzene veľkú hodnotu, čo v reálnych podmienkach tiež nie je možné. Preto budeme naše nasledujúce, rozbory usmerňovať v zmysle definícií a to tým spôsobom aby sme sa čo najviac priblížili k skutočnej realizácii v interpreteri RAM stroja.

Teoretické úvahy

Definícia 4. Pamäťová výpočtová zložitosť algoritmu *S(n)* je funkcia, ktorá udáva, koľko pamäte sa spotrebuje pri riešení problému rozsahu *n.* (Lovászová-Vozár, 2007).

Ďalej vychádzame z toho, že veľkosť pamäte používajúca daný register chápeme ako jednotku, čiže znova môžeme uvažovať o jednotkovej (uniformnej) miere. Do jedného registra môžeme zapísať obrovské číslo, čo v reálnych podmienkach nevieme vytvoriť, no vzhľadom na pamäťovú zložitosť stroja RAM musíme uvažovať len o jednotkovej miere.

Podľa toho ako ja definovaný RAM stroj by sme mali na zápis vstupného slova veľkosti *n,* použiť $n + 2 + k$ registrov. Z dôvodu, že prvý register, podľa číslovania je pracovný „nultý“ a v ňom sa vykonávajú všetky aritmetické operácie preto do neho neukladáme, žiadne vstupné údaje na uchovávanie. Druhý register je rezervovaný pre nepriame adresovanie (inštrukcie, kde sa v operande nachádza „*“). To znamená, že by sme mali registre v pamäti obsadzovať od registra č.2. No do úvahy môžeme brať aj ďalšie registre, ktoré v niektorých prípadoch môžu zohrávať tiež dôležitú funkciu pomocných registrov, teda určených na istú „riadiacu funkciu“ vo výpočtoch, pre túto prípad sme použili symbol *k*.

Ale vzhľadom na skutočnosť, že index register sa používa aj ako ukladač vstupných údajov a navyše aj ostatné registre vedia vykonávať inštrukcie nepriameho adresovania, môžeme pokojne na zápis vstupného slova použiť $n + 1$ registrov.

Definícia 5. Povedzme, že výpočet vykonaný počítačom s ľubovoľným prístupom trval dobu *t,* ak v *t*-tom kroku výpočtu došlo k vykonaniu príkazu zastavenia, alebo k adresovej chybe, alebo k deleniu nulou, a zatiaľčo pre *i = 0, 1, ..., (t - 1)* v *i*-tom kroku k chybe ani k vykonaniu príkazu zastaveniu nedošlo. (Kučera, 1983)

Definícia 6. Povedzme, že výpočet počítača s ľubovoľným prístupom pracoval s pamäťou veľkosti *m*, a ak nebol vykonaný žiadny z príkazov, ktorý by mal okamžitú adresu operandu väčšiu ako *m*, ale bol vykonaný príkaz s okamžitou adresou operandu *m*. (Kučera, 1983)

Praktické riešenie

V prípade pamäťovej zložitosti nie sú obmedzenia prakticky žiadne a ak existujú nemajú takmer žiadny vplyv na určenie pamäťovej zložitosti alebo správnosť výsledku. Pamäťovú zložitosť výpočtu sme realizovali jednoduchým súčtom registrov (pamäťových buniek), s ktorými pracoval RAM stroj. Inicializáciu pamäťovej zložitosti vykonáva procedúra zostavenia pamäte a tiež všetky procedúry výpočtu, skokov, a preto inicializáciu pamäťovej zložitosti, nie je potrebné realizovať úpravou zdrojového kódu, nakoľko táto bola realizovaná už pri vytváraní Interpretera stroja RAM, čo bolo predmetom bakalárskej práce.

Realizácia úpravy zdrojového kódu.

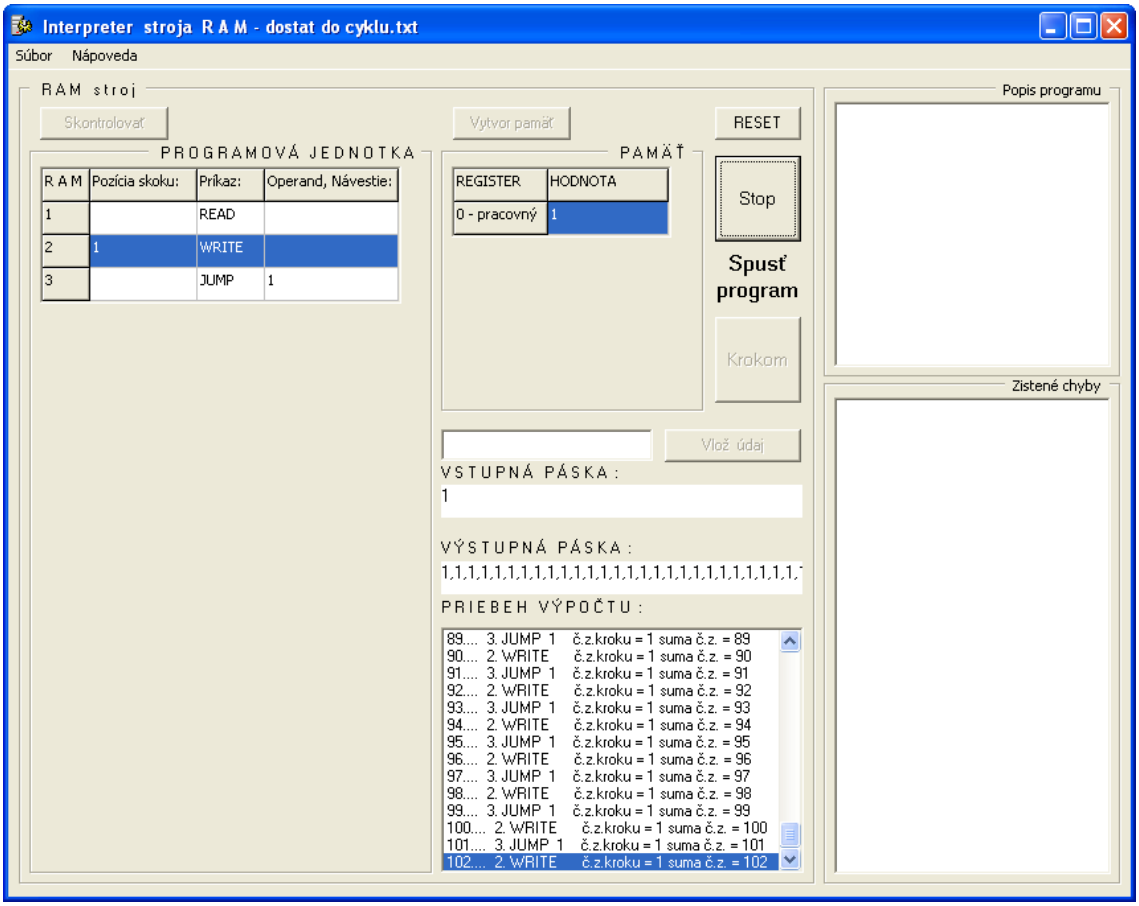
Táto zmena v zdrojovom kóde aplikácie je jednou z najjednoduchších. V procedúre samotného výpočtu `Tform1.rataj` sme vo výpise priebehu programu, ktorý sa nachádza v časti kde sa program zastaví, teda v komponente (`Listbox3`), údaj o celkovej pamäťovej zložitostí, čiže súčtu všetkých riadkov komponentu `StringGrid2`.

```
else
  Uloivysledok1.Enabled:= false;
  suma_narocnosti:=suma_narocnosti+cas_narocnost;
if stop then begin
  timer1.Enabled:= false;
  ListBox3.Items.Add('-------------------------------------------------'+
  '-----------------------------------');
  ListBox3.Items.Add('Výstup:  '+Label2.Caption);
  Listbox3.ItemIndex:= ListBox3.Items.Add('Celková časová jednotková zložitosť je:  '
  +IntToStr(suma_narocnosti));
  Listbox3.ItemIndex:= ListBox3.Items.Add('Celková pamäťová jednotková zložitosť je:  '
  +IntToStr(StringGrid2.RowCount - 1));
  Listbox3.ItemIndex:= ListBox3.Items.Add('');
  Listbox3.ItemIndex:= ListBox3.Items.Add('');
  Listbox3.ItemIndex:= Listbox3.count-6;
  Button4.Enabled:= false;
```

Obr. 28 Zmena v súvislosti s určením celkovej časovej výpočtovej zložitosti

Podľa uvedených definícií 5. a 6., ak nedôjde k zastaveniu výpočtu, nie je určená doba trvania výpočtu a nemusí byť definovaná ani veľkosť použitej pamäte.

Hodnoty pamäťovej zložitosti sú zobrazené iba v prípadoch zastavenia výpočtu, alebo pri dokončení posledného kroku výpočtu (príkaz „STOP“, chybové hlásenie = chybná sémantika, delenie nulou a pod.). V prípade, kedy sa RAM stroj nezastaví, ako napríklad pri nekonečnom cykle, nám interpreter neposkytne žiadne údaje o veľkosti pamäte. Je možné iba vizuálne zistiť (aj to len v niektorých prípadoch) daný stav pamäte a vtedy máme možnosť zistiť tento údaj len ako priebežný a nie konečný. V prípade, keď zastavíme priebeh výpočtu kliknutím na tlačidlo „STOP“ opäť nedostávame od Interpretera RAM údaj o konečnej veľkosti pamäte, čiže pamäťovej zložitosti. Máme teda k dispozícii iba momentálny stav pamäte, z ktorej nemusíme vedieť konečnú pamäťovú náročnosť.

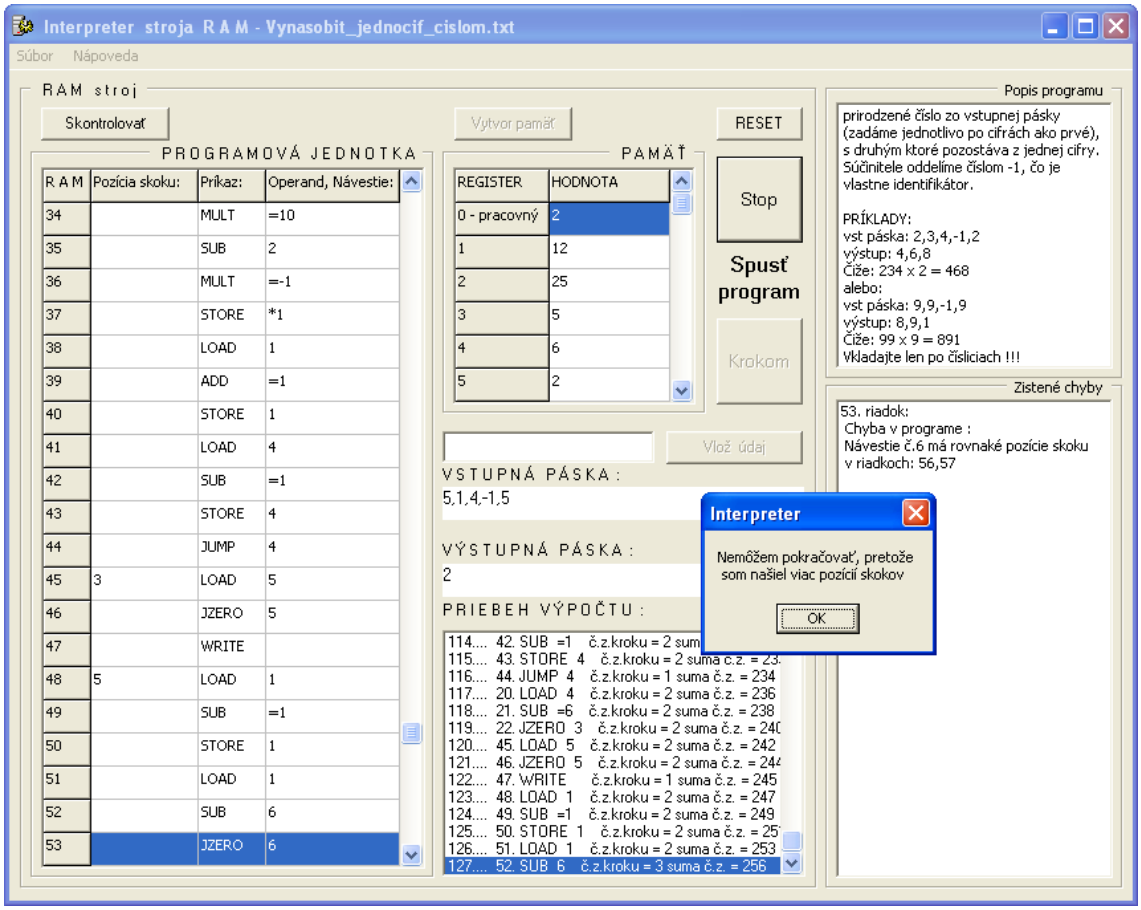


Obr. 29 Aplikácia neposkytla údaj o pamäťovej výpočtovej zložitosti

Evidentne šlo o nekonečný cyklus a zistili sme, že informácia o pamäťovej zložitosti nám nebola dostupná ani prostredníctvom Interpretera RAM stroja.

Samozrejme túto skutočnosť sme mohli si overiť aj v praxi, na to stačilo spustiť aplikáciu a vložiť do nej program RAM, ktorý je obsiahnutý v prílohe na CD médiu, alebo si vytvoríme vlastný program, ktorý dokáže simulovať práve takúto situáciu. Vidíme, že hodnota výsledku sa nemení, tým aj pamäťová zložitosť, ale môžeme si byť istý, že vždy pamäť v tomto programe RAM bude mať veľkosť jediného registra, ibaže samotný Interpreter nesmie túto informáciu poskytnúť.

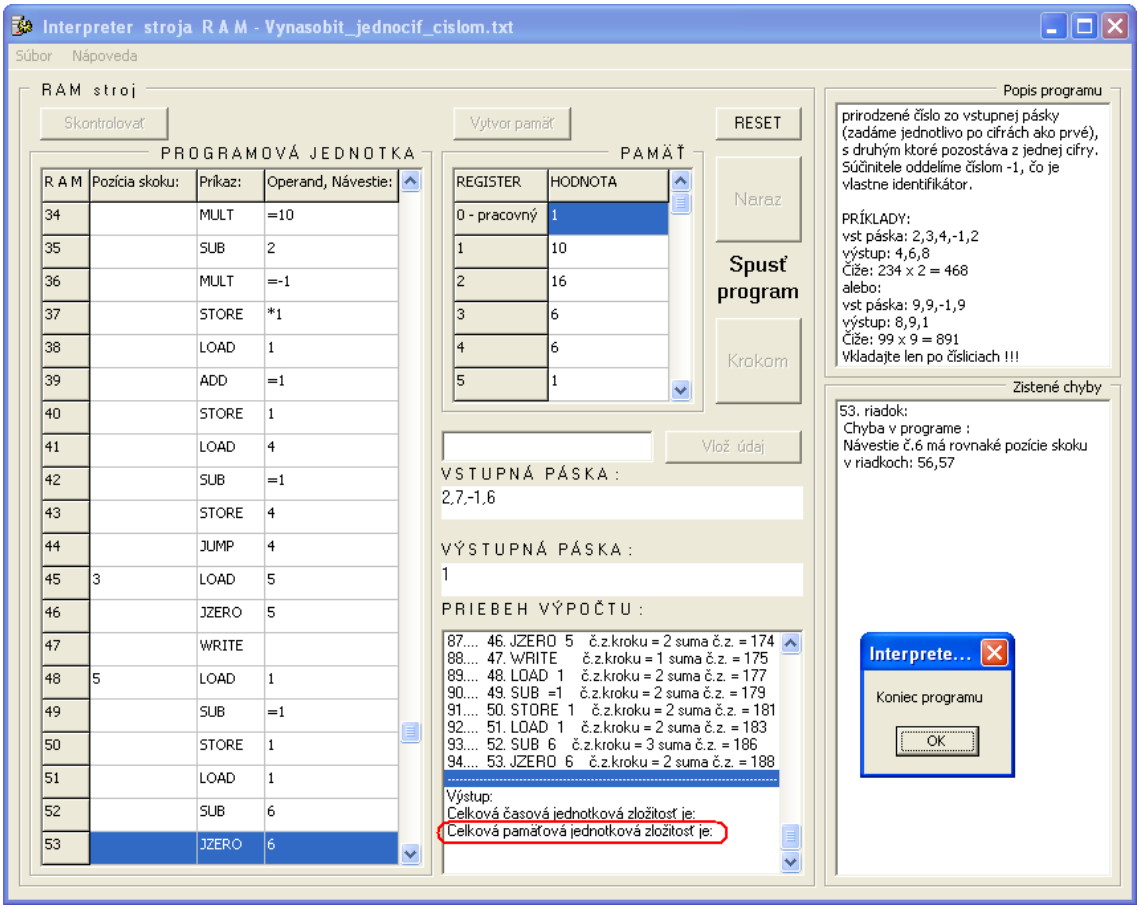
Nasledujúci obrázok znázorňuje tiež reálnu skutočnosť, kedy sa počas výpočtu vyskytla chyba sémantiky a aj aplikácia vyhodnotila túto situáciu ako chybu, následne zastavila celý priebeh výpočtu a upozornila touto informáciou o chybe chybovým hlásením.



Obr. 30 Aplikácia neposkytla údaj ani v prípade chyby

Nasledujúci obrázok znázorňuje situáciu po akceptovaní predošlej správy (zatlačiť na tlačidlo „OK“ správy) o vykonanej chybe, kde vidíme oznam o ukončení výpočtu a viditeľný dôkaz o poskytnutí informácie celkovej pamäťovej zložitosti daného programu RAM.

Táto situácia sa podobá tej, ktorá prebehne korektne, keď program vykoná všetky príkazy, bez oznamu o vykonanej chybe, alebo delení nulou a podobne.



Obr. 31 Aplikácia neposkytla údaj v prípade chyby a následnom ukončení

4.4 Testovanie a vyhodnocovanie

4.4.1 Testovanie časovej výpočtovej zložitosti

Príklady č.1 a č.2 sme podrobili hlbšej analýze (výpočet a overenie na intrepreteri). Príklady č.3 – č.5 boli zamerané na porovnanie výpočtov s výsledkami našej aplikácie pri určovaní výpočtovej zložitosti. Testovali sme viacerými hodnotami v logaritmickom meradle.

4.4.1.1 Príklad č. 1

Majme program RAM, ktorý zistí najväčšiu hodnotu troch zadaných celých čísel, ktoré prečíta RAM stroj zo vstupnej pásky. Vypočítajme časovú zložitosť daného programu. Celý program sme zapísali do tabuľky a tým sprehľadnili daný príklad.

Tab. 2 Rozbor programu s popisom inštrukcií podľa príkladu č. 1

Inštrukcia	Hodnota	Popis priebehu výpočtu
READ	1	Načítanie prvého čísla zo vstupnej pásky
STORE 1	2	Vloženie tohto čísla do registra č. 1
READ	1	Načítanie prvého čísla zo vstupnej pásky
STORE 2	2	Vloženie tohto čísla do registra č. 2
READ	1	Načítanie prvého čísla zo vstupnej pásky
STORE 3	2	Vloženie tohto čísla do registra č. 3
LOAD 2	2	Vložíme do pracovného registra údaj z registra č. 2
SUB 1	3	Odpočítame túto hodnotu od hodnoty v registri č.1
JGE 1	2	Ak je tento rozdiel ≥ 0, potom číslo v reg. č. 2 ≥ v reg. č. 1, skok
LOAD 3	2	Ak nie je tak vložíme do pracovného registra údaj z reg. č.3
SUB 1	3	Odpočítame túto hodnotu od hodnoty v registri č.1
JGE 2	2	Ak je tento rozdiel ≥ 0, potom číslo v reg. č. 3 ≥ v reg. č. 1, skok
LOAD 1	2	Ak nie je tak, znamená že číslo v registri č.1 je najväčšie
WRITE	1	Zápis čísla
STOP	1	Zastavenie, koniec výpočtu
2 LOAD 3	2	Znamená, že číslo v registri č.3 je najväčšie
WRITE	1	Zápis čísla
STOP	1	Zastavenie, koniec výpočtu
1 LOAD 3	2	Vložíme do pracovného registra údaj z registra č. 3
SUB 2	3	Odpočítame túto hodnotu od hodnoty v registri č. 1
JGE 3	2	Ak je tento rozdiel ≥ 0, potom číslo v reg. č. 3 ≥ v reg. č. 2, skok
LOAD 2	2	Inak je číslo v registri č. 2 najväčšie
WRITE	1	Zápis čísla
STOP	1	Zastavenie, koniec výpočtu
3 LOAD 3	2	Znamená, že číslo v registri č. 3 je najväčšie
WRITE	1	Zápis čísla
STOP	1	Zastavenie, koniec výpočtu

Rozbor a výpočet príkladu:

Rozložili sme si ho niekoľko menších. Podľa postupnosti inštrukcií sme priradili hodnoty časových zložitostí.

$T(n) = 1+2+1+2+1+2+2+3+2+2+3+2+2+1+1+2+1+1+2+3+2+2+1+1+2+1+1.$

Po 8. inštrukciu sme všetky hodnoty jednoducho sčítali (podčiarknuté hodnoty).

$T(n) = \underline{1+2+1+2+1+2+2+3}+2+2+3+2+2+1+1+2+1+1+2+3+2+2+1+1+2+1+1,$

$T(n) = \underline{14} + 2+2+3+2+2+1+1+2+1+1+2+3+2+2+1+1+2+1+1.$

Po ďalšej inštrukcii nastalo vetvenie programu (nenastal cyklus), to znamená, že vznikla stromová štruktúra programu:

1. Ak hodnota v registri č.2 je väčšia ako v registri č.1, ako aj v registri č.3 potom najväčšia hodnota je v registri č.2 (podčiarknuté hodnoty zložitostí). Ostatné hodnoty neuvažujeme, pretože program končí.

   $T(n) = \underline{14} + \underline{2}+2+3+2+2+1+1+2+1+1+\underline{2+3+2+2+1+1}+2+1+1,$

   $T(n) = \underline{14} + \underline{2} + \underline{11},$

   $T(n) = 14 + 2 + 11,$

   $\underline{\underline{T(n) = 27}}.$

2. Ak hodnota v registri č.2 je väčšia ako v registri č.1, ale menšia ako v registri č.3 potom najväčšia hodnota je v registri č.3 (podčiarknuté hodnoty zložitostí). Ostatné hodnoty neuvažujeme, pretože program končí.

   $T(n) = \underline{14} + \underline{2}+2+3+2+2+1+1+2+1+1+\underline{2+3+2}+2+1+1+\underline{2+1+1},$

   $T(n) = \underline{14} + \underline{2} + \underline{7} + \underline{4},$

   $T(n) = 14 + 2 + 7 + 4,$

   $\underline{\underline{T(n) = 27}}.$

3. Ak hodnota v registri č.2 je menšia ako v registri č.1, a menšia ako v registri č.3 potom najväčšia hodnota je v registri č.1 (podčiarknuté hodnoty zložitostí). Ostatné hodnoty neuvažujeme, pretože program končí.

   $T(n) = \underline{14} + \underline{2+2+3+2+2+1+1}+2+1+1+2+3+2+2+1+1+2+1+1,$

   $T(n) = \underline{14} + \underline{13},$

   $T(n) = 14 + 13,$

   $\underline{\underline{T(n) = 27}}.$

4. Ak hodnota v registri č.2 je menšia ako v registri č.1, a ak hodnota v registri č.1 je menšia ako v registri č.3 potom najväčšia hodnota je v registri č.3 (podčiarknuté hodnoty zložitostí). Ostatné hodnoty neuvažujeme, pretože program končí.

   $T(n) = \underline{14} + \underline{2+2+3+2}+2+1+1+\underline{2+1+1}+2+3+2+2+1+1+2+1+1,$

   $T(n) = \underline{14} + \underline{9} + \underline{4},$

   $T(n) = 14 + 9 + 4,$

   $\underline{\underline{T(n) = 27}}.$

Celková časová zložitosť daného programu RAM má funkciu rovnajúcu sa konštantnému číslu:

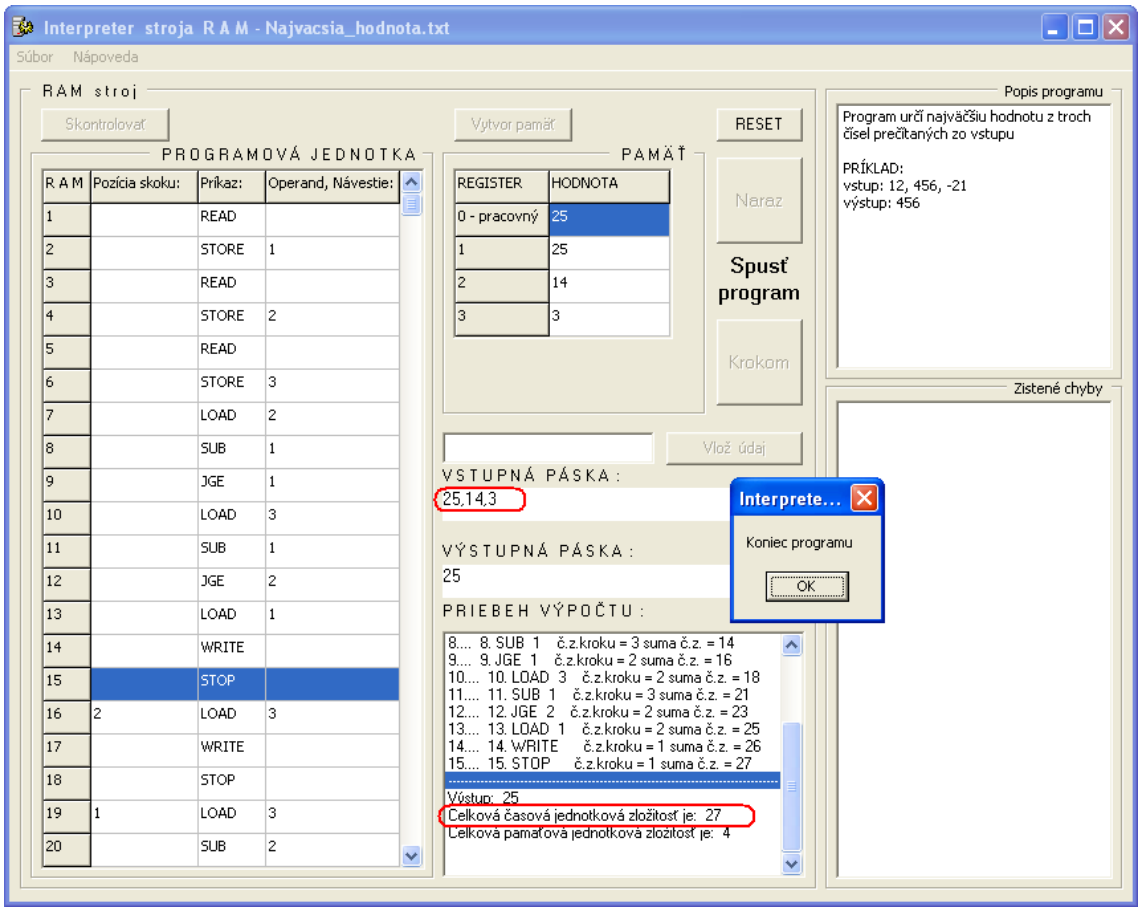
$\underline{\underline{T(n) = c}}.$

Asymptotická horná časová zložitosť algoritmu je:

$\underline{\underline{O(n) = c}}.$

Praktický dôkaz:

Na nasledujúcom obrázku máme možnosť vidieť, akým spôsobom náš interpreter dospel k výsledkom, čo sa týka konečného výsledku ako aj konečnej hodnoty časovej zložitosti. Vložením 3 náhodne zvolených celých čísel do vstupnej pásky, sme dostali výsledok.



Obr. 32 Údaje poskytnuté interpreterom podľa príkladu č. 1

Zhrnutie:

Časová zložitosť daného programu RAM nezávisí od hodnôt zadaných vo vstupnej páske, pretože je pevne stanovený počet vstupných údajov. Preto $\underline{T(n) = 27}$.

Samozrejme ak chceme získať ďalšie dôkazy môžeme tento program ďalej testovať rôznymi inými vstupnými hodnotami. Vždy dospejeme k súladu teoretického výpočtu a praktického overenia výpočtu na Interpreteri stroja RAM.

4.4.1.2 Príklad č. 2

Majme program RAM, ktorý vypočíta faktoriál prirodzeného čísla *n* zadaného zo vstupnej pásky, čiže počítame *n!*. Vypočítajme časovú zložitosť daného programu.

Daný program počíta hodnoty rekurzívnym spôsobom.

Tab. 3 Rozbor programu s popisom inštrukcií podľa príkladu č. 2

Inštrukcia	Hodnota	Popis priebehu výpočtu
READ	1	Načítanie *n* zo vstupnej pásky
STORE 1	2	Vloženie čísla *n* do index registra
LOAD =1	1	Do pracovného registra vložíme číslo 1, pretože začíname od 1
STORE 2	2	Vloženie tohto čísla do registra č. 2
STORE 3	2	Vloženie tohto istého čísla do registra č. 3
1 LOAD 1	2	Z index registra vyberieme hodnotu *n* vložíme do prac. registra
SUB 2	3	Odpočítame od nej hodnotu reg. č. 2 (znížime hodnotu na *n -1*)
JZERO 2	2	Ak *n = 0*, skočí na návesť 2 (ukončenie výpočtu)
LOAD 2	2	Ak nie, z registra č. 2 vložíme hodnotu do pracovného registra
ADD =1	2	Pričítame k nemu hodnotu *1*
STORE 2	2	Vložíme ho do registra č. 2
LOAD 3	2	Hodnotu z registra č. 3 vložíme do pracovného registra
MULT 2	3	Násobíme s hodnotou registra č. 2
STORE 3	2	Uchováme tento výsledok v registri č. 3
JUMP 1	1	Skok na ďalší výpočet
2 LOAD 3	2	Výber konečného výsledku z registra č. 3
WRITE	1	Zápis
STOP	1	Zastavenie, koniec výpočtu

Rozbor a výpočet príkladu:

Rozložili sme problém na niekoľko menších. Podľa postupnosti inštrukcii sme priradili hodnoty časových zložitostí.

$$T(n) = 1 + 2 + 1 + 2 + 2 + 2 + 3 + 2 + 2 + 2 + 2 + 2 + 3 + 2 + 1 + 2 + 1 + 1,$$

Po 5. inštrukciu sme všetky hodnoty jednoducho sčítali (podčiarknuté hodnoty).

$$T(n) = \underline{1 + 2 + 1 + 2 + 2} + 2 + 3 + 2 + 2 + 2 + 2 + 2 + 3 + 2 + 1 + 2 + 1 + 1,$$

$$T(n) = \underline{8} + 2 + 3 + 2 + 2 + 2 + 2 + 2 + 3 + 2 + 1 + 2 + 1 + 1.$$

Od 6. inštrukcie až po 15. inštrukciu máme cyklus (inštrukcia `JUMP`), ktorá sa opakuje práve $(n - 1)$ krát, pokiaľ $n = 0$, potom musíme pripočítať ešte zostávajúce 3 inštrukcie, pretože nasleduje podmienený skok.

$$T(n) = \underline{8} + \underline{2 + 3 + 2} + 2 + 2 + 2 + 2 + 2 + 3 + 2 + 1 + 2 + 1 + 1,$$

$$T(n) = \underline{8} + \underline{\underline{7}} + \underline{21(n - 1)} + 2 + 1 + 1.$$

Posledným zjednodušením je vlastne súčet hodnôt časových zložitostí, ktoré sme dostali po vykonaní podmieneného skoku a to znamená súčet 3 posledných inštrukcií.

$$T(n) = \underline{8} + \underline{\underline{7}} + \underline{21(n - 1)} + \underline{4}.$$

Nasledujúce kroky sú vlastne bežnými matematickými úkonmi.

$$T(n) = 8 + 7 + 21(n - 1) + 4,$$

$$T(n) = 19 + 21(n - 1),$$

$$T(n) = 19 + 21n - 21,$$

$$\underline{\underline{T(n) = 21n - 2}}.$$

Asymptotická horná časová zložitosť algoritmu je:

$$\underline{\underline{O(n) = n}}.$$

Celková časová zložitosť programu RAM, ktorý vypočíta faktoriál čísla n, teda $n!$, má tvar lineárnej funkcie:

$$\underline{\underline{T(n) = 21n - 2}}.$$

Ak napríklad do vstupnej pásky vložíme číslo $n = 8$, celková časová zložitosť nám musí vyjsť:

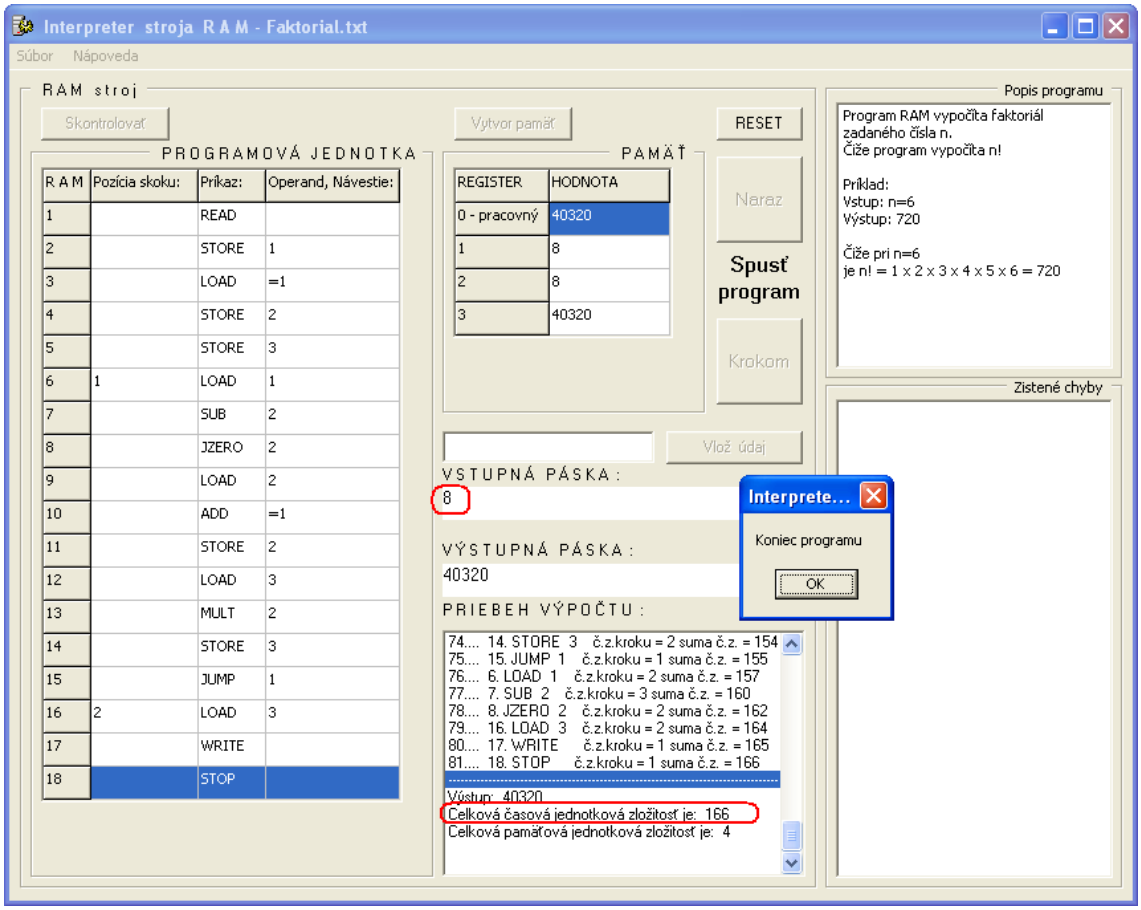
$$T(n) = 21n - 2,$$

$$T(n) = 21.8 - 2 = 168 - 2,$$

$$\underline{\underline{T(n) = 166}}.$$

Praktický dôkaz:

Na nasledujúcom obrázku máme možnosť vidieť, ako si náš interpreter poradil s daným programom. Vložením čísla *n* do vstupnej pásky a spustením samotného výpočtu sme dostali výsledok. Po vykonaní poslednej inštrukcie, kde po výstupnej hodnote, spočítal celkovú časovú zložitosť, programu *n!*, pri danej vstupnej hodnote *n*. Vstupná hodnota ako aj výsledná časová zložitosť sa na obrázku nachádzajú v červenom ovále.



Obr. 33 Údaje poskytnuté interpreterom podľa príkladu č. 2

Zhrnutie:

Z dôkazu je zrejmé, že pri danom programe výpočtu faktoriálu je časová zložitosť lineárne závislá od vstupnej hodnoty.

Ako v prvej úlohe tak aj v tejto môžeme získať ďalšie dôkazy a tento program ďalej testovať rôznymi inými vstupnými hodnotami. Môžeme zhodnotiť, že výpočet = praktické overenie programu RAM na Interpreteri stroja RAM.

4.4.1.3 Príklad č. 3

Vypočítajte aritmetický priemer z n čísel $a_1, a_2, ..., a_n$. vložených do vstupnej páske.

Prvé číslo vo vstupnej páske je n = počet vložených hodnôt.

Tab. 4 Rozbor programu s popisom inštrukcií podľa príkladu č. 3

READ	Načítanie počtu vložených hodnôt
STORE 1	Vložíme ho do 1.registra, na konci tento údaj delíme súčtom z reg. č. 1
LOAD =0	Vynulujeme pracovný register, vložíme hodnotu „*0*“
STORE 2	Túto hodnotu vložíme do 2.registra, tu bude súčet všetkých hodnôt
LOAD =1	Do pracovného registra vložíme hodnotu *1*
STORE 3	Vložíme ho do registra č. 3, sem vkladáme počty jednotlivých vstupov + *1*
2 LOAD 1	Načítame hodnotu n
SUB 3	Odčítaním od počtu vložených hodnôt testujeme koniec programu
JGE 1	Ak počet $n \leq$ počet zadaných vstupov +*1*, pokračujeme vo vkladaní
LOAD 2	Ak nie pokračuje načítaním súčtu všetkých hodnôt
DIV 1	Delíme počtom n
WRITE	Zapíšeme výsledný priemer do výstupnej pásky
STOP	Štandardne ukončujeme celý výpočet
1 READ	Opätovné načítanie zo vstupnej pásky
ADD 2	Pridáme do sumy všetkých hodnôt
STORE 2	Túto uložíme
LOAD 3	Vyberieme testovaciu hodnotu konca vkladania
ADD =1	Pridáme 1, lebo sme urobili ďalší krok
STORE 3	Tento uložíme znova do registra
JUMP 2	Vykonanie kolobehu načítavania vstupných hodnôt

Tab. 5 Výsledky testov programu podľa príkladu č. 3

Počet vložených hodnôt	Celková časová zložitosť
2	43
3	63
4	83
6	123
11	223
21	423
31	623
51	1023

Výpočtom sme tiež zistili celkovú časovú zložitosť programu:

$$T(n) = 20n + 3.$$

Asymptotická časová zložitosť (trend časovej výpočtovej zložitosti):

$$O(n) = n.$$

Zhrnutie:

Podľa výsledkov testovaní sme dospeli k záveru, že časovú zložitosť daného programu RAM môžeme zhodnotiť ako lineárne závislú od vstupnej hodnoty. Samozrejme musíme brať do úvahy aj konštantu, s ktorou pracuje program RAM.

4.4.1.4 Príklad č. 4

Vytvorte program RAM, ktorý zozrkadlí vstupné slovo $w \in \{0,1\}^*$ a je ukončené číslom *2*, čo je identifikátor konca načítavania hodnôt zo vstupnej pásky.

Tab. 6 Rozbor programu s popisom inštrukcií podľa príkladu č. 4

LOAD =2	Načítame číslo konca zoznamu
1 STORE	Vložíme ho do registra č.1
READ	Načítame hodnotu zo vstupnej pásky
SUB =2	Odčítame hodnotu *2* (testujeme či nie sme na konci zoznamu)
JZERO 2	Ak áno program skočí na vypisovanie
ADD =2	Ak nie pridáme číslo *2*, čím rekonštruujeme vstupnú hodnotu
STORE	Vložíme, ju do registra, kam ukazuje hodnota (naša adresa)
LOAD 1	Vložíme hodnotu registra *1* čiže krok v načítavaní
ADD =1	Zvýšime hodnotu registra 1, čiže o *1* krok v načítavaní
JUMP 1	Vykonanie tohto cyklu
2 LOAD 1	Ak sme dokončili načítavanie, začneme odčítavať vykonané kroky
SUB =1	Odčítavame *1*, znížime počet krokov o *1*
STORE 1	Zapamätáme si tento údaj
3 LOAD 1	Znova ho načítame do pracovného registra
SUB =1	Otestujeme, či nie sme na konci počtu vložených hodnôt
JZERO 4	Ak áno skočíme na koniec programu
LOAD *1	Ak nie, spätne načítavame údaje z registrov, hodnota je zozrkadlená
WRITE	Pritom zapisujeme tieto do výstupnej pásky
LOAD 1	Znova načítame údaj o kroku
SUB =1	Zmenšíme ho o *1* krok
STORE 1	A uchovávame k ďalšiemu načítavaniu - spätnému
JUMP 3	Skočíme späť na načítavanie údajov v registroch aj počtov krokov
4 STOP	Korektne ukončujeme program

Tab. 7 Výsledky testov programu podľa príkladu č. 4

Počet vložených hodnôt	Celková časová zložitosť
2	55
3	89
4	123
6	191
11	361
21	701
31	1041
51	1721

Výpočtom sme tiež zistili celkovú časovú zložitosť programu:

$$\underline{\underline{T(n) = 34n - 13}}.$$

Asymptotická časová zložitosť (trend časovej výpočtovej zložitosti):

$$\underline{\underline{O(n) = n}}.$$

Zhrnutie:

Podľa výsledkov testovaní sme dospeli k záveru, že časovú zložitosť daného programu RAM môžeme zhodnotiť ako lineárne závislú od vstupnej hodnoty. Samozrejme musíme brať do úvahy aj konštantu, čo predstavuje určitú réžiu na obslužné údaje, s ktorými pracuje program RAM.

4.4.1.5 Príklad č. 5

Zostrojte RAM stroj, ktorý vstupné slovo *{0,1, ..., 9}* ∪{1}* vynásobí jednociferným číslom. Na vstupnej páske bude zadané vstupné slovo a jednociferné číslo oddelené číslom *-1*.

Výpis programu RAM:

Pre dĺžku programu neuvádzame výpis ani popis. Výpis je prístupný k nahliadnutiu v prílohe práce, nachádzajúcej sa na CD médiu s názvom: Vynasobit_jednocifernym_cislom.txt.

Tab. 8 Výsledky testov programu podľa príkladu č. 5

Počet vstupných hodnôt	Celková časová zložitosť	
	Ak násobenie neprechádza cez 10	Ak násobenie prechádza cez 10
3	122	123
4	210	211
5	298	299
6	386	387
12	914	915
22	1794	1795
32	2674	2675
52	4434	4435

Výpočtom sme tiež zistili celkovú časovú zložitosť programu:

Záleží od toho či cifra pri násobení prechádza číslom 10,

ak áno: $T(n) = 88n - 141$,

ak nie: $T(n) = 88n - 141 + 1$.

Asymptotická časová zložitosť (trend časovej výpočtovej zložitosti):

$O(n) = n$.

Zhrnutie:

Podľa výsledkov testovaní sme dospeli k záveru, že časovú zložitosť daného programu RAM môžeme zhodnotiť ako lineárne závislú od vstupnej hodnoty. Samozrejme musíme brať do úvahy aj konštantu, s ktorou pracuje program RAM.

4.4.2 Zhrnutie výsledkov časovej výpočtovej zložitosti

Samozrejme existujú úlohy, ktoré majú rôzne stupne zložitosti. Ako sme v úvode tejto kapitoly naznačili, zložitosť algoritmu bublinkového triedenia môže nadobúdať hodnoty v rozmedzí n až n^2, tak pri triedení môžeme inými algoritmami dosiahnuť aj lepších hodnôt priemernej zložitosti algoritmu (Quicksort, Mergesort, Heapsort ...) $T(n) = n \log n$. No existujú aj úlohy, ktoré sú riešiteľné v polynomiálnom čase, kde $T(n) = n^c$, kde c – konštanta, kde n predstavuje počet vstupných hodnôt. To znamená n^2, n^3, n^4 … a tak ďalej. Alebo v exponenciálnom čase, kde $T(n) = c^n$, čo predstavuje obrovské časové zložitosti.

Ku príkladu: ak funkciu $f(n) = 999999.n^2$ v porovnávaní s funkciou $g(n) = 2^n$ sa zdá byť dosť veľká, no stačí zvoliť napríklad počet vstupných hodnôt $n = 1000$ a zisťujeme, že $f(n) = 999999000000$, zatiaľ čo $g(n)=2^{1000}$ čo predstavuje zhruba 10^{300}, názorne by táto hodnota predstavovala číslo začínajúce číslicou *1* a pokračovala by *300* nulami! No a pri väčších hodnotách n je rozdiel ešte výraznejší!!!

V praxi to znamená, že je dôležité zhodnotiť či dané problémy vieme algoritmicky vyriešiť v časovom rozmedzí, ktoré sme si stanovili, alebo či existuje iný algoritmus, ktorý daný problém zvládne za kratší čas, ba dokonca či niektoré problémy sa vôbec dajú algorimicky riešiť a to aspoň v polynomiálne obmedzenom čase (chceme vylúčiť riešenie v exponenciálnom čase). Niektoré úlohy sú dokonca zvládnuteľné (niektoré sú dokázateľné takmer okamžite) v polynomiálne obmedzenom čase, ale za pomoci algoritmu majú časovú zložitosť exponenciálnu!

Na predošlých príkladoch sme dokázali, že výpočty, ktoré dokážeme realizovať vieme aj určitým spôsobom (naprogramovaním) interpretovať v našej aplikácii Interpreter stroja RAM. I keď by sa dalo namietať, že aplikácia nedokáže plne pracovať s vysokými hodnotami, predsa však z hľadiska časovej zložitosti podáva interpreter presné výsledky.

4.4.3 Testovanie pamäťovej výpočtovej zložitosti

Príklady č.6 a č.7 sme podrobili hlbšej analýze (výpočet a overenie na intrepreteri). Príklady č.8 – č.10 boli zamerané na porovnanie výpočtov s výsledkami našej aplikácie pri určovaní výpočtovej zložitosti. Testovali sme viacerými hodnotami v logaritmickom meradle.

4.4.3.1 Príklad č. 6

Majme program RAM, ktorý zistí najväčšiu hodnotu troch zadaných celých čísel, ktoré prečíta RAM stroj zo vstupnej pásky. Vypočítajme pamäťovú zložitosť daného programu. Použijeme rovnaký program ako pri zisťovaní časovej zložitosti.

Tab. 9 Výpis programu podľa príkladu č. 6

Inštrukcia	Poradové číslo registra, ktorý sme v programe použili
READ	0
STORE 1	1
READ	0
STORE 2	2
READ	0
STORE 3	3
LOAD 2	2
SUB 1	1
JGE 1	0
LOAD 3	3
SUB 1	1
JGE 2	0
LOAD 1	2
WRITE	0
STOP	
2 LOAD 3	3
WRITE	1
STOP	
1 LOAD 3	3
SUB 2	2
JGE 3	0
LOAD 2	2
WRITE	1
STOP	
3 LOAD 3	3
WRITE	1
STOP	

Rozbor a výpočet príkladu:

Jednoduchým spôsobom vieme zistiť s koľkými registrami pracoval RAM stroj. Zisťujeme najvyššiu hodnotu z tabuľky, ktorou je v tomto prípade počet vstupných hodnôt $n = 3$, a ktoré podľa toho zaberajú 3 registre v pamäti, z čoho vypočítame pamäťovú zložitosť podľa už spomenutej definície:

(počet vstupných hodnôt n) + ($1x$ pracovný register) + ($0x$ indexový register) + ($0x$ pomocný register)

$$S(n) = n + 1 + 0 + 0,$$

$$\underline{\underline{S(n) = n + 1}},$$

Asymptotická pamäťová zložitosť (trend pamäťovej výpočtovej zložitosti):

$$\underline{\underline{O(n) = n}}.$$

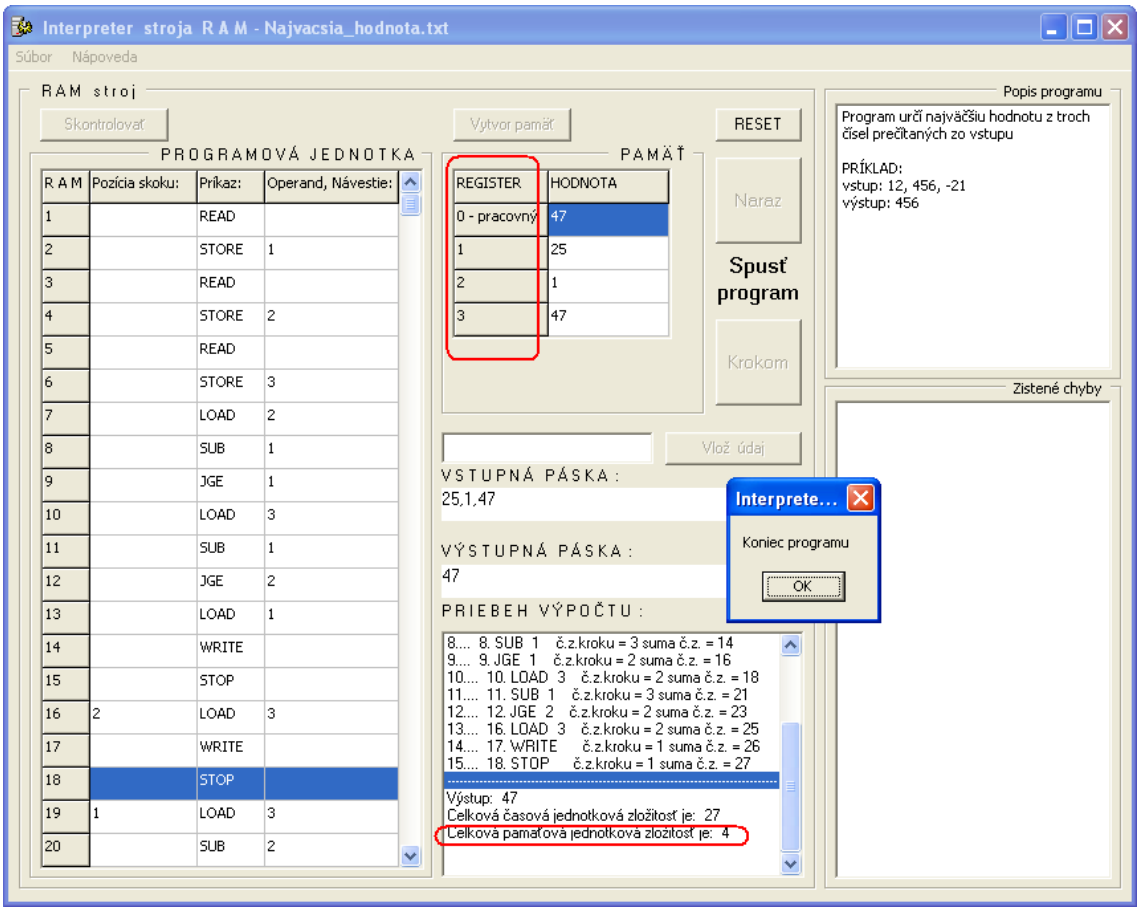
Pre zvolené hodnoty, ktorých počet musí byť rovný 3, tak je tento program vytvorený nám vždy vyjde hodnota pamäťovej zložitosti podľa definície č. 5:

$$S(n) = 3 + 1 + 0 + 0 = \underline{4},$$

$$\underline{\underline{S(n) = 4}}.$$

Praktický dôkaz:

Vložením 3 náhodne zvolených celých čísel do vstupnej pásky, sme dostali výsledok. Po vykonaní poslednej inštrukcie, kde po výstupnej hodnote, spočítal celkovú pamäťovú zložitosť programu. Výsledná pamäťová zložitosť ako aj zobrazenie pamäte, z ktorej vieme zistiť priamo pamäťovú zložitosť sú na obrázku označené v červenom ovále.



Obr. 34 Údaje poskytnuté interpreterom podľa príkladu č. 6

Zhrnutie:

Na obrázku vidíme, že pamäť má veľkosť štyroch registrov. Použitím ďalších hodnôt stále dostaneme tú istú veľkosť pamäťovej zložitosti konkrétneho RAM stroja. Tento dôkaz je vlastne triviálny, no patrí k určovaniu takéhoto detailu akým je pamäťová zložitosť.

4.4.3.2 Príklad č. 7

Majme program RAM, ktorý zistí najväčšiu hodnotu troch zadaných celých čísel, ktoré prečíta RAM stroj zo vstupnej pásky. Vypočítajme pamäťovú zložitosť daného programu.

V tomto príklade použijeme úplne iný program RAM pri pamäťovej zložitosti. V tabuľke sa nenachádza popis inštrukcii ako v prípade časovej zložitosti, pretože nás skôr zaujíma počet použitých registrov a v tomto príklade navyše používame indexový register, preto nás zaujíma, koľko pamäťových adries ukladá do svojho registra.

Tab. 10 Výpis programu podľa príkladu č. 7

Inštrukcia	Poradové číslo registra, ktorý sme v programe použili
LOAD =2	0
1 STORE 1	1
READ	0
SUB =2	0
JZERO 2	0
ADD =2	0
STORE *1	n - 1
LOAD 1	0, 1
ADD =1	1
JUMP 1	
2 LOAD 1	1
SUB =1	0
STORE 1	1
3 LOAD 1	0
SUB =1	0
JZERO 4	0
LOAD *1	n - 1
WRITE	0
LOAD 1	0, 1
SUB =1	0
STORE 1	1
JUMP 3	
4 STOP	

Rozbor a výpočet príkladu:

Z predošlej tabuľky vieme zistiť, že maximálnu hodnotu vstupu, teda množstvo pamäte použitej pri zadaní vstupných údajov v tomto RAM programe je *n - 1*. Prečo sme udali hodnotu *n - 1*? Je to prosté. Posledné číslo zadané vo vstupnej páske je len ako identifikátor konca zoznamu vstupných hodnôt a v inštrukciách pre prácu s indexovým registrom ako napríklad: `STORE *1`, `LOAD *1` nefiguruje. Pri výpočte sme používali indexový register a samozrejme zoznamu registrov pamäte musíme zahrnúť aj pracovný register.

$$S(n) = (n - 1) + 1 + 1 + 0,$$

$$\underline{\underline{S(n) = n + 1}}.$$

Asymptotická pamäťová zložitosť (trend pamäťovej výpočtovej zložitosti):

$$\underline{\underline{O(n) = n}}.$$

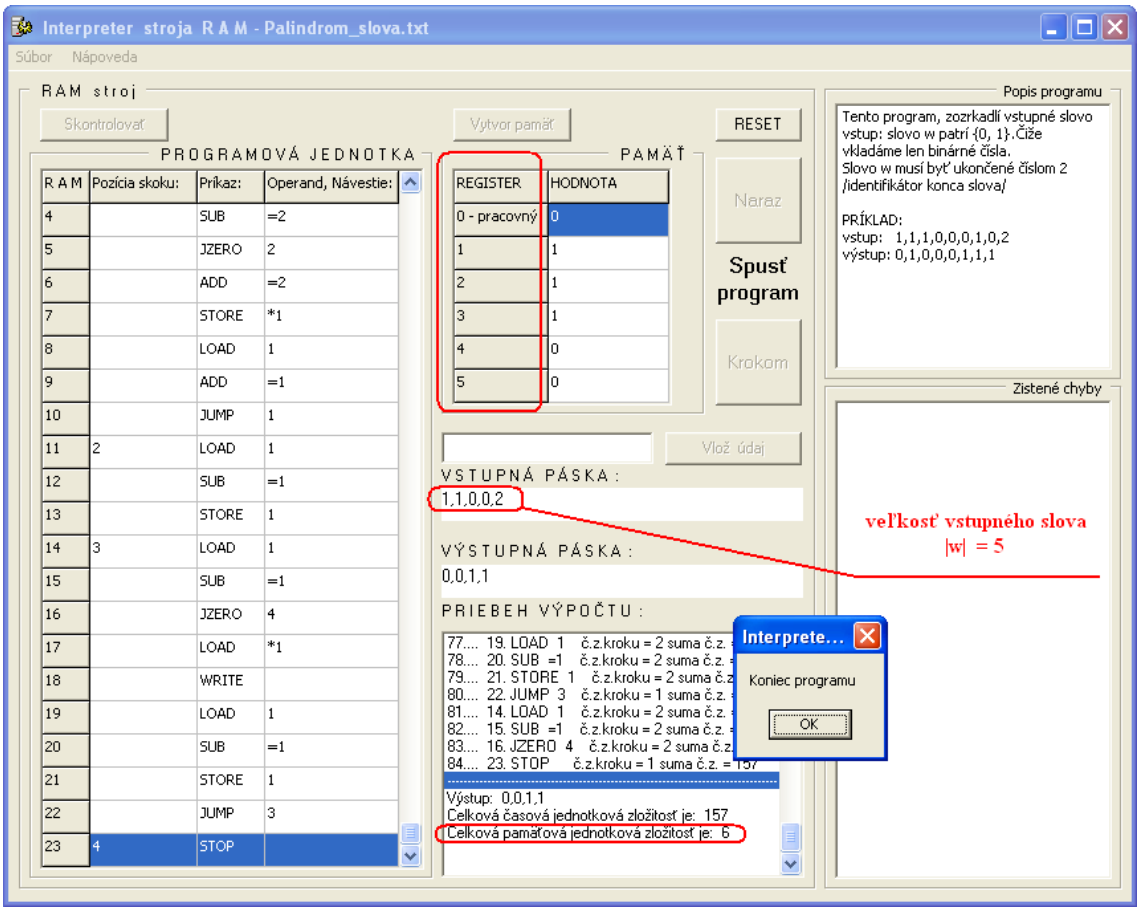
Pre zvolené vstupné hodnoty v počte $n = 5$ a po dosadení do vzťahu nám vychádza:

$$S(n) = (5 - 1) + 1 + 1 + 0,$$

$$\underline{\underline{S(n) = 6}}.$$

Praktický dôkaz:

Vložením *5* náhodne zvolených binárnych čísel do vstupnej pásky, sme dostali výsledok. Výsledná pamäťová zložitosť ako aj zobrazenie pamäte, z ktorej vieme zistiť priamo pamäťovú zložitosť sú na obrázku označené v červenom ovále.



Obr. 35 Údaje poskytnuté interpreterom podľa príkladu č. 7

Zhrnutie:

Z obrázku vidíme, že pamäť má veľkosť šiestych registrov. Vložením ďalších vstupných hodnôt narastá jej veľkosť podľa funkcie $n+1$, čo je typické pre konkrétny program RAM. Z daného vyplynulo, že trend pamäťovej zložitosti je závislí od počtu vstupných údajov.

4.4.3.3 Príklad č. 8

Vypočítajte aritmetický priemer z *n* čísel $a_1, a_2, ..., a_n$. Prvé číslo vo vstupnej páske je *n,* počet vložených hodnôt.

Výpis programu RAM:

V predchádzajúcej časti, teda pri stanovovaní časovej zložitosti sme tento istý program podrobne popísali, a preto zostavíme tabuľku s výsledkami testov, ukážeme si obecný vzťah, ktorý je platný pre daný program a nakoniec zhrnieme na všetky poznatky pamäťovej zložitosti tohto programu.

Tab. 11 Výsledky testov programu podľa príkladu č. 8

Počet vložených hodnôt	Celková pamäťová zložitosť
2	4
3	4
4	4
6	4
11	4
21	4
31	4
51	4

Výpočtom sme tiež zistili celkovú pamäťovú zložitosť programu:

$\underline{\underline{S(n) = 4}}$.

Asymptotická pamäťová zložitosť (trend pamäťovej výpočtovej zložitosti):

$\underline{\underline{O(n) = c}}$.

Zhrnutie:

Podľa výsledkov testovaní sme dospeli k záveru, že pamäťovú zložitosť daného programu RAM môžeme zhodnotiť ako konštantne závislú od vstupnej hodnoty. Samozrejme musíme mohli by sme vytvoriť aj program s iným algoritmom, ktorý by bol lineárne závislý.

4.4.3.4 Príklad č. 9

Zostrojte RAM stroj, ktorý vstupné slovo *{0,1, ..., 9}* ∪{1}* vynásobí jednociferným číslom. Na vstupnej páske bude zadané vstupné slovo a jednociferné číslo oddelené číslom *-1*.

Výpis programu RAM:

Je prístupný k nahliadnutiu v prílohe práce, nachádzajúcej sa na CD médiu s názvom: Vynasobit_jednocifernym_cislom.txt. Miesto tohto údaja ponúkame výsledky testovaní.

Tab. 12 Výsledky testov programu podľa príkladu č. 9

Počet vložených hodnôt	Celková časová zložitosť
3	9
4	11
5	13
6	15
12	27
22	47
32	67
52	107

V tomto prípade, ak sa jedná o výpočet pamäťovej zložitosti, ktorá nezáleží od toho či cifra pri násobení prechádza číslom 10, ako pri zisťovaní časovej zložitosti tohto programu.

$$\underline{\underline{S(n) = 2n + 3}}.$$

Asymptotická pamäťová zložitosť (trend pamäťovej výpočtovej zložitosti):

$$\underline{\underline{O(n) = n}}.$$

Zhrnutie:

Podľa výsledkov testovaní sme dospeli k záveru, že pamäťovú zložitosť daného programu RAM môžeme zhodnotiť ako lineárne závislú od vstupnej hodnoty. Samozrejme musíme brať do úvahy aj konštantu, s ktorou pracuje program RAM.

4.4.3.5 Príklad č. 10

Vytvorte program RAM, ktorý vytriedi všetky čísla zo vstupnej pásky od najmenšej hodnoty po najväčšiu. Ukončenie vkladania zo vstupnej pásky bude ukončené číslom *0*, čo je identifikátor konca načítavania hodnôt.

Výpis programu RAM:

Pre dĺžku programu neuvádzame výpis ani popis. Výpis je prístupný k nahliadnutiu v prílohe práce, nachádzajúcej sa na CD médiu s názvom: Bubble_Sort.txt. Miesto tohto údaja ponúkame výsledky testov.

Tab. 12 Výsledky testov programu podľa príkladu č. 10

Počet vložených hodnôt	Celková časová zložitosť
2	8
3	9
5	12
11	17
21	27
31	37
51	57

Výpočtom sme tiež zistili celkovú pamäťovú zložitosť programu:

$$\underline{\underline{S(n) = n + 6}}.$$

Asymptotická pamäťovej zložitosť (trend pamäťovej výpočtovej zložitosti):

$$\underline{\underline{O(n) = n}}.$$

Zhrnutie:

Podľa výsledkov testovaní sme dospeli k záveru, že pamäťovú zložitosť daného programu RAM môžeme zhodnotiť ako lineárne závislú od vstupnej hodnoty. Samozrejme musíme brať do úvahy aj konštantu, čo predstavuje určitú pamäťovú réžiu na obslužné údaje, ktorými pracuje program RAM.

4.4.4 Zhrnutie výsledkov pamäťovej výpočtovej zložitosti

Za pomoci výpočtov a testov sme dokázali funkčnosť našej aplikácie a tiež pravdivosť výsledkov hľadiska pamäťovej výpočtovej zložitosti programov RAM. Vo väčšine príkladov sa jednalo o pamäťové zložitosti veľkosti n až na dve výnimky kde veľkosť bola limitovaná hodnotou konštanty, alebo násobky veľkosti počtu vstupných hodnôt.

V príklade už spomenutého bublinkového triedenia, kde časová zložitosť (trend zložitosti) tohto algoritmu dosahovala hodnôt maximálne n^2, alebo minimálne n, no v prípade pamäťovej zložitosti sa jednalo „len“ o veľkosť n, čiže veľkosti počtu vložených hodnôt. Aj týmto môžeme potvrdiť fakt, že pamäťová výpočtová zložitosť sa môže v niektorých algoritmov rovnať časovej výpočtovej zložitosti, alebo môže byť menšia. $S(n) \leq T(n)$. Nikdy však veľkosť pamäťovej zložitosti neprevyšuje hodnoty časovej výpočtovej zložitosti. Môžeme teda povedať, že naša aplikácia aj v prípade určenia pamäťovej výpočtovej zložitosti pracovala úplne presne.

Záver

Ako bolo uvedené na začiatku, výpočtová technika v súčasnosti zažíva obrovský rozmach vo vývoji nových technológií. Na túto skutočnosť nadväzuje fakt, ktorý musíme rešpektovať a týmto faktom je zdokonaľovanie aplikácií pre používateľov ako aj aplikácií vývojárskych nástrojov. No vždy je základným prvkom spôsob, akým možno priebeh týchto aplikácií ovplyvňovať, to znamená aký postup na riešenie problému použijeme. V úvode spomenuté jazyky sú dôkazom rozmanitosti takýchto návodov, čiže algoritmov.

V našej aplikácii sme interpretovali jednu z pomôcok zavedenia algoritmu, čiže sme vytvorili aplikáciu na simuláciu výpočtu abstraktného výpočtového modelu – stroja RAM. Hlavnou úlohou bolo realizovať taký abstraktný stroj, ktorý by bol plne funkčný a demonštroval tak jeho teoretický model. Pri tejto práci sme dospeli k záveru, že možnosť vytvorenia takejto aplikácie je možná za predpokladu dodržania všetkých špecifík daného abstraktného stroja a tiež nutnosti zvládnutia takejto práce z hľadiska programovania.

Tento praktický model pri rôznych skúšobných RAM programoch adekvátne reagoval, čo znamená že pri spustení výpočtu daného programu a vložení vstupných údajov prezentoval očakávané výstupné údaje. Samozrejme, tieto sme si najprv overili, spôsobom najjednoduchším, čiže za pomoci pera, papiera. Z hľadiska realizácie takéhoto funkčného modelu táto práca splnila očakávania.

Danú aplikáciu by bolo možné využiť napríklad ako pomôcku pri vyučovaní teórie algoritmov. Mohli by sme porovnať skutočný počítač s počítačom s ľubovoľným prístupom (Interpreter stroja RAM), a to v tom zmysle, že oba počítače majú úložisko dát, pamäť, aritmeticko-logickú jednotku pod. Napríklad v RAM stroji máme vstupnú a výstupnú pásku, programovú jednotku, pamäť s registrami, zatiaľ čo bežné PC obsahuje CD mechaniku, pevný disk, procesorovú jednotku (mikroprocesor), matematický koprocesor a operačnú pamäť. Môžeme tiež povedať, že samotná pamäť RAM stroja je rozdelená na pracovný register, indexový register, pomocné registre, údajové registre čo môžeme prirovnať v bežných počítačoch k pamäti RAM (pozor podobnosť s Random Access Memory), ktorú by sme mohli rozdeliť na pracovnú a virtuálnu pamäť. Ak by sme chceli zájsť v úvahách o interpreteri stroja RAM ešte ďalej, mohli by sme interpreter (našu aplikáciu) prirovnať k operačnému systému bežných PC a program RAM, k bežnej aplikácii, pracujúcej v ňom. Takýmto spôsobom by sme napríklad mohli

aspoň približne prezentovať činnosť bežných počítačov a vytvoriť predstavu ľuďom, ktorí sa stretávajú po prvý raz s funkciou počítača a chceli by vedieť z akých častí sa skladá počítač, aké procesy a akým spôsobom prebiehajú v jednotlivých častiach počítača. Práve touto prácou sme chceli podať pomocnú ruku tým ľuďom, ktorý nemajú ucelenú predstavu o fungovaní počítačov. Pri výučbe v predmetu Algoritmy a údajové štruktúry, čo je vlastne študijný predmet z oblasti výpočtovej techniky, môže takáto aplikácia ako pomôcka poslúžiť účelu vizualizácie v danej problematike.

Ak by sme chceli, používať túto aplikáciu na zložité vedecké výpočty, je portebné podotknúť, že týmto smerom by sme celý skutočný výpočtový čas predlžovali, a to práve z dôvodu, že sa jedná o interpreter. Každý krok výpočtu (každý príkaz, inštrukcia, operand, číslo, ich identifikáciu, presný spôsob výpočtu) aplikáciou interpretujeme, čiže nevypočítavame „priamo“, ako napríklad pri použití aplikácie „Kalkulačka“, ktorá je súčasťou operačného systému Windows XP. Týmto spôsobom vykonávame dvojitú transformáciu.

Samozrejme danú aplikáciu možno stále vylepšovať, teda je možné zvyšovať rýchlosť chodu, vizuálne aj farebne doplniť rôznymi funkciami. Ale môžeme vyjadriť uspokojenie nad tým, že sme dokázali vytvoriť funkčnú aplikáciu, ktorá pomôže pri overovaní niektorých skutočností, alebo pri použití ako pomôcky z oblasti teoretickej informatiky. Každá, aj tá najlepšie prevedená aplikácia má svojho nasledovníka, teda ďalšiu verziu. Bolo by na škodu, ak by sa aj táto aplikácia prestala zdokonaľovať, ak má poslúžiť ako učebná pomôcka vizualizácie funkcií, prvkov teoretického stroja RAM. Navyše podáva informáciu o základných funkciách bežných počítačov.

Zoznam použitej literatúry

AHO, V. Alfred - HOPCROFT, E. John - ULLMAN, D. Jeffrey. 1974. *The Design and Analysis of Computer Algoritms.* Massachusetts : Addison Wesley Publishing Company. 470 s. ISBN 0-201-00029-6.

CANTÚ, Marco. 2003. *Myslíme v jazyku Delphi 7.* Praha : Vydavateľstvo Grada, 2003. 580 s. ISBN 80-247-0694-6.

DAVIS, D. Martin – SIGAL, Ron – WEYUKER, J. Elaine. 1994. *Computability, Complexity and Languages*. 2.vyd. Boston: Academic Press, 1994. 327 s. ISBN 0-12-206382-1.

HOPCROFT, E. John – MOTWANI, Rajeev. – ULLMAN, D. Jeffrey. 2006. *Introduction to Automata Theory, Languages and Computation.* 2.vyd. Boston : Addison Wesley Publishing Company, 2006. 521s. ISBN 0-321-45536-3.

HYNEK, Josef. 2008. *Teoretická informatika* [online]. b. m. : b. v., 2008 [cit. 2010-16-01] Dostupné na: <http://lide.uhk.cz/fim/ucitel/hynekjo1/TINF/TInf008.ppt>.

JANČAR, Petr - KOT, Martin - SAWA, Zdeněk. 2007. *Teoretická informatika* [online] Ostrava : Technická univerzita Ostrava, 2007 [cit. 2010-20-04]. 269 s. Dostupné na: <http://www.cs.vsb.cz/kot/down/ti2007/ti-text-new.pdf>.

JINOCH, Josef - MÜLLER, Karel - VOGEL, Josef. 1988. *Programování v jazyku Pascal.* 3. vyd. Praha : Vydavateľstvo SNTL, 1988. 328 s.

KOZEN, C. Dexter. 1997. *Automata and Computability*. New York : Springer-Verlag, 1997. 400 s. ISBN 0-387-94907-0.

KUČERA, Luděk. 1983. *Kombinatorické algoritmy*. Praha : Vydavateľstvo SNTL, 1983. 283 s.

LEWIS, R. Harry – PAPADIMITRIOU, H. Christos. 1997. *Elements of the Theory of Computation*. 2.vyd. Upper Saddle River : Prentice Hall, 1997. 361 s. ISBN 0-13-272741-2.

LOVÁSZOVÁ, Gabriela - VOZÁR, Martin. 2007. *Matematické modely počítača* [online]. b. m.: b. v., 2007 [cit. 2009-20-11]. Dostupné na: <http://edu.ukf.sk/course/view.php?id=43>.

PINEDO, Michael. 2008. *Scheduling: theory, algorithms, and systems* [online] New York : Prentice Hall. aktualizované 2008. [cit. 2010-01-20]. 671 s. Dostupné na: <http://www.google.com/books?hl=sk&lr=&id=EkpDak9kEs0C&oi=fnd&pg=PR7&dq=complexity+theory+algorithm&ots=GLfAJwLQqc&sig=JB6Foz2iEFWatdro61_g2Ci7WbM#v=onepage&q=complexity%20theory%20algorithm&f=false>. ISBN 978-0-387-78935-4.

SAVICKÝ, Petr. 2004. *Výpočtová složitost* [online]. b. m. : b. v., 2004 [cit. 2009-16-10] Dostupné na: <http://www2.cs.cas.cz/~savicky/vyuka/vypsl/vypsl2004zs1.pdf>.

SIPSER, Michael. 1997. *Introduction to the Theory of Computation*. Boston : Course Technology, 1997. 416 s. ISBN 0-534-94728-X.

SCHÖNHAGE, Arnold. 1979. *Lecture Notes in Computer Science : Automata, Languages and Programming* [online] Berlin : Springer. aktualizované 2006. [cit. 2010-01-20]. 529 s. Dostupné na: <http://www.springerlink.com/content/p7562n5438n24727/>. ISBN 978-3-540-09510-1.

SUDKAMP, A. Thomas. 1996. *Languages and Machines*. 2.vyd. Boston : Addison Wesley Publishing Company, 1996. 569 s. ISBN 0-201-82136-2.

WIRTH, Niklaus. 1989. *Algoritmy a štruktúry údajov*. 2. vyd. Bratislava : Vydavateľstvo Alfa, 1989. 488 s. ISBN 80-05-00153-3.

Prílohy

Príloha A

Elektronický nosič CD

Priložené CD obsahuje:

Súbor výslednej aplikácie:

- Interpreter_RAM.exe - výsledná aplikácia interpretera programov RAM.

Priečinok - RAM_programy obsahujúci skúšobné súbory pre interpreter stroja RAM:

- Aritmeticky_priemer.txt,
- Bubble_sort.txt,
- Faktorial.txt,
- Fibonacci.txt,
- Mocnina 2_na _n.txt,
- Najvacsia_hodnota.txt,
- Palindrom_slova.txt,
- Vynasobit_jednocif_cislom.txt.

Súbory určené pre úpravu, ladenie výslednej aplikácie:

- Interpreter_RAM.pas - obsahuje zdrojový kód výslednej aplikácie,
- Interpreter_RAM.dfm - obsahuje popis vlastností umiestnených na formulári,
- Interpreter.dpr - súbor pre riadenie systému formulárov programátorskom prostredí DELPHI, obsahuje zoznam formulárov a štartovací kód.

Súbory tejto diplomovej práce spracované v elektronickej forme:

- Dilpomova_praca_Jan_Horvath.doc - pre používateľov kancelárskych balíkov MS Office, Open Office, atď.
- Dilpomova_praca_Jan_Horvath.pdf - pre používateľov aplikácie Adobe Reader, Foxit Reader, atď.