

Wrocław, 2006

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Łukasz Szkup

Implementacja abstrakcyjnego modelu obliczeń - - Maszyny RAM

(Implementation of Abstract Computation Model – RAM Machine)

Praca magisterska
napisana pod kierunkiem
prof. Krzysztofa Lorysia

Podziękowania za pomoc w testowaniu programu (kolejność alfabetyczna) dla:
Agnieszki Kawałkowskiej
Natalii Madej
Rafała Szkup

Pracę dedykuję Rodzinie oraz żonie Aleksandrze.

SPIS TREŚCI

Wstęp	5
Specyfikacja kodu RAM	7
Złożoność obliczeniowa programów RAM	10
Obsługa programu „Maszyna RAM”	13
Informacje wstępne	13
Instalacja programu	13
Składowe edytora „Maszyny RAM”	16
Kontrolka procesora	17
Kontrolka pamięci	18
Kontrolka taśmy wejściowej i wyjściowej	20
Edytor programu	20
Kontrolka statystyk złożoności programu	22
Weryfikacja, uruchamianie i śledzenie programu	23
Import i Eksport	25
Drukowanie	27
Opcje – konfiguracja programu	29
Przykładowe programy w kodzie RAM	32
Dodawanie	32
Silnia	33
Największy wspólny dzielnik	34
Obsługa programu „Maszyna RAM” w wersji tekstowej	35
Implementacja i informacje dla kontynuatorów programu	37
Podsumowanie	42
Dodatek - Maszyny Turinga	43
Deterministyczna maszyna Turinga (DTM)	43
Niedeterministyczna maszyna Turinga (NDTM)	43
Klasy P i NP	43
Bibliografia	45

Wstęp

Realizacja niniejszej pracy jest odpowiedzią na lukę w dziedzinie oprogramowania edukacyjnego, którego brak powoduje trudności w przekazywaniu wiedzy młodym słuchaczom na lekcjach informatyki wprowadzających do świata programowania. Maszyna RAM jest modelem obliczeń, który jest łatwy do opanowania i jednocześnie wyrabia umiejętności logicznego myślenia, które mogą być przydatne w rozwijaniu zdolności programowania.

W przeszłości powstało wiele implementacji maszyny RAM (między innymi „Maszyna RAM” autorstwa prof. Krzysztofa Lorysia i dr Przemysławy Kanarek z 1990 roku), lecz pomysł napisania nowej wersji niesie ze sobą co najmniej kilka zalet jak: odświeżony wygląd programu czy atrakcyjna wizualizacja wraz z możliwością konfiguracji wielu elementów na potrzeby użytkownika. Także sama technologia (C# i .NET) użyta do napisania programu czyni go przenaszalnym na większość popularnych systemów operacyjnych, takich jak MS Windows® czy Linux. Ponieważ język C# jest aktualnie w czołówce najczęściej używanych języków programowania na świecie, istnieje spora szansa, że zachęci to do rozwinięcia aktualnej implementacji maszyny RAM (do pracy jest dołączony pełny kod źródłowy programu) o nowe elementy. Do napisania nowej wersji Maszyny RAM skłonił autora również fakt, iż uruchomienie starszych implementacji na nowej klasy komputerach sprawiało czasami spore problemy, co skutecznie zniechęcało zarówno uczniów jak i nauczycieli do używania takiego oprogramowania podczas lekcji, przez co zajęcia musiały się odbywać jedynie w teorii. Odbijało się to na spadku atrakcyjności i szansy zainteresowania młodych słuchaczy dziedziną informatyki, jaką jest programowanie. Zbierając te niezbyt optymistyczne fakty autor postanowił temu zaradzić i tak oto powstała nowa wersja Maszyny RAM.

Zanim rozpoczniemy przygodę z programowaniem w kodzie RAM (taką nazwę nosi język programowania dla maszyny RAM), autor przedstawi pokrótce, jakie zagadnienia zostały omówione w niniejszej pracy.

W pierwszych trzech rozdziałach zajmiemy się pełną specyfikacją Maszyny RAM, omawiając jej wszystkie składowe, opisując kod RAM oraz sposób, w jaki będziemy liczyć złożoność czasową i pamięciową dla wykonywanych programów.

Następnie zajmiemy się obsługą zaimplementowanej aplikacji, symulującej działanie Maszyny RAM. Omówimy szczegółowo każdy komponent, by korzystanie z programu było przyjemne i bezproblemowe, dzięki czemu użytkownik będzie mógł się skupić wyłącznie na pisaniu programów w kodzie RAM i ich uruchamianiu oraz śledzeniu krok po kroku.

Aby czytelnik mógł z większą łatwością utrwalić sobie specyfikację kodu RAM, kolejny rozdział będzie poświęcony na omówienie trzech krótkich programów. Szczegółowo przeanalizujemy każdy z programów, by dokładnie zrozumieć ich działanie.

Dla zwolenników programów uruchamianych w trybie poleceń została przygotowana także specjalna wersja maszyny. W kolejnym rozdziale opiszemy jak się taką wersją posługiwać.

„Implementacja i informacje dla kontynuatorów programu” to cenne informacje dla tych czytelników, którzy zapragną rozwijać implementowaną dla celów tej pracy aplikację. Zostanie omówiona struktura projektu, podział na moduły oraz wskazówki, pozwalające dodać do programu język programowania (oraz odpowiedni edytor), który może być tłumaczony na kod RAM, a następnie uruchamiany. Będzie to świetna okazja by zapoznać także się z technikami kompilacji i translacji kodu.

Ostatecznie podsumujemy wszystkie zebrane tu wiadomości oraz podamy bibliografię, którą autor się posługiwał podczas pisania niniejszej pracy.

Na końcu pracy został umieszczony dodatek omawiający deterministyczną i niedeterministyczną maszynę Turinga oraz klasy P i NP , który może być ciekawym uzupełnieniem do wiedzy zawartej w niniejszej pracy.

Maszyna RAM

Maszyna RAM bazuje na modelu jednej z abstrakcyjnych maszyn Turinga, a mianowicie modelu deterministycznej maszyny (DTM). Informacja ta jest ważna chociażby z szacunku do historii informatyki, ale nie na tyle istotna, by ją w tym miejscu szczegółowo omawiać. Teoria związana z maszyną Turinga nie jest wymagana, by zrozumieć ideę maszyny RAM, jednak dla zainteresowanych czytelników powstał odpowiedni dodatek znajdujący się w końcowej części pracy.

Maszyna RAM składa się z następujących elementów: pamięci, taśmy wejściowej, taśmy wyjściowej, programu oraz procesora.

Pamięć jest serią komórek, a każda z nich mieści dowolną liczbę całkowitą. Komórki są ponumerowane liczbami całkowitymi, począwszy od zera. Pierwsza, zerowa komórka, jest zwana także akumulatorem. Jest ona pomocniczą komórką, używaną między innymi przy operacjach arytmetycznych. Liczba komórek pamięci jest nieskończona. Również liczba, która może być przechowywana w każdej z komórek, nie posiada limitu na jej wielkość (wielkość liczby rozumiemy jako liczbę cyfr, z których się składa, przyjmując jakikolwiek system liczbowy, np.: binarny). Początkowo, przed uruchomieniem programu, wartość każdej komórki pamięci jest nieokreślona i próba odczytania tej wartości jest błędną operacją.

Maszyna RAM posiada dwie taśmy: jedną do odczytu i jedną do zapisu. Każda taśma, podobnie jak pamięć maszyny, jest podzielona na komórki, gdzie każda komórka mieści jedną, dowolną liczbę całkowitą. Obie taśmy mają nieskończoną długość. Do odczytu i zapisu na odpowiednich taśmach służą głowice, początkowo ustawione nad pierwszą komórką taśmy. Gdy następuje próba zapisania liczby na taśmę, głowica taśmy wyjściowej dokonuje odpowiedniego zapisu do komórki, nad którą się aktualnie znajduje i przesuwa się nad kolejną komórkę taśmy. Podobnie wygląda sytuacja dla taśmy wejściowej, z tym, że gdy maszyna spróbuje odczytać liczbę z komórki, która jest pusta, operacja zakończy się niepowodzeniem (i maszyna natychmiast się zatrzymuje).

Program dla Maszyny RAM składa się z jednej lub wielu linii programu, a każda z nich może być pusta lub zawierać jedną z kilkunastu dostępnych instrukcji, którą rozpoznaje maszyna. Program jest osobną częścią maszyny i nie jest przechowywany w pamięci (co zapobiega napisaniu samo-modyfikującego się programu).

Procesor jest najważniejszą jednostką Maszyny RAM. W każdym kroku maszyny, jego funkcja sprowadza się do wczytania kolejnej linii programu i wykonania instrukcji w niej zawartej. Procesor dodatkowo pamięta numer aktualnie wykonywanej linii programu oraz numer kolejnej, która zostanie wykonana jako następna. Zazwyczaj numer kolejnej linii programu przygotowanej do wykonania jest o jeden większy, niż numer linii aktualnie wykonywanej. Regułą tą łamią jednak instrukcje skoku warunkowego i bezwarunkowego, które pozwalają skoczyć do dowolnej linii programu (oznaczonej odpowiednią etykietą). Program jest wykonywany do momentu napotkania instrukcji stopu, lub gdy numer wskazujący na kolejną linię programu, która ma zostać wykonana, wykracza poza zakres łącznej liczby linii programu (czyli w sytuacji, gdy nie ma już następnych linii programu do wykonania).

Specyfikacja kodu RAM

Każda instrukcja kodu RAM składa się z dwóch części - kodu operacji i adresu. Oczywiście, zbiór instrukcji możemy wzbogacić o dowolne inne instrukcje, jak operacje logiczne, funkcje matematyczne czy operacje na znakach, nie wpływając przy tym na

zmianę rzędu złożoności programów. Adres instrukcji dzielimy na operandum i etykietę. Operandum może mieć następujący zapis (tabela poniżej).

Zapis	Znaczenie
$=x$	Liczba całkowita x
x	Liczba znajdująca się w komórce pamięci o indeksie x (wymóg: $x \geq 0$)
x	Liczba znajdująca się w komórce o indeksie y , gdzie y jest wartością komórki o indeksie x (wymóg: $x \geq 0, y \geq 0$)

Z powyższego zapisu wynika, że obok kodu operacji, możemy podać liczbę, zastosować adresowanie bezpośrednie, lub adresowanie pośrednie. Są to trzy techniki stosowane przez praktycznie wszystkie języki programowania, zarówno te niższego jak i wyższego rzędu, które są wystarczające do napisania dowolnego programu.

Instrukcje kodu RAM

Kod operacji	Adres
LOAD	Operandum
STORE	Operandum
ADD	Operandum
SUB	Operandum
MULT	Operandum
DIV	Operandum
READ	Operandum
WRITE	Operandum
JUMP	Etykieta
JGTZ	Etykieta
JZERO	Etykieta
HALT	Etykieta

Osoba, która miała wcześniej styczność z językiem assembler, analizując powyższą listę zauważy tu pewnie podobieństwo do kodu RAM. Jednak dla utrzymania prostoty (ale bez szkody dla funkcjonalności języka) lista rozkazów kodu RAM jest szczuplejsza, w stosunku do np.: assemblera dla procesora x86.

Możemy teraz zdefiniować sens programu P przy pomocy dwóch wielkości: przekształcenia m określonego na zbiorze nieujemnych liczb całkowitych o wartościach w zbiorze liczb całkowitych i indeksu lokalizacji, który ustala indeks następnej instrukcji do wykonania. Funkcja m jest mapą pamięci - $m(x)$ jest liczbą całkowitą umieszczoną w komórce pamięci o indeksie x . Początkowo, przed uruchomieniem programu P mamy:

- $m(x) = ?$ (wartość nieokreślona) dla każdego $x \geq 0$,
- indeks lokalizacji jest nastawiony na pierwszą instrukcję P ,
- taśma wyjściowa jest pusta.

Po wykonaniu k -tej instrukcji P , indeks lokalizacji jest automatycznie ustawiany na $k + 1$ (tj. na następną instrukcję), chyba że k -tą instrukcją jest JUMP, JGTZ lub JZERO, które są instrukcjami skoku bezwarunkowego (JUMP) i warunkowego (JGTZ, JZERO). Aby określić sens instrukcji, najpierw zdefiniujemy $v(a)$, czyli *wartość operandum a* :

Użycie	Wartość	Opis
$v(=x)$	x ,	Liczba całkowita x

$v(x)$	$m(x)$,	Adresowanie bezpośrednie
$v(^x)$	$m(m(x))$	Adresowanie pośrednie

Zdefiniowanie $v(a)$, pozwala nam na krótszy i prostszy zapis dla tych instrukcji, które mogą posiadać każdą z powyższych trzech postaci operandum. Dla pozostałych instrukcji, które nie spełniają tego warunku, każdy z dopuszczalnych przypadków zostanie opisane w osobnym wierszu. W każdym opisie kolejnej instrukcji (kolumna: *Znaczenie*) przyjmujemy, że po jej wykonaniu indeks lokalizacji jest ustawiany na kolejną instrukcję programu, chyba, że w opisie jest zaznaczone inaczej.

Oto opis poszczególnych instrukcji kodu RAM (za operandum a możemy podstawić: $=x$, x lub x):

Instrukcja	Znaczenie	Opis słowny
LOAD a	$m(0) \leftarrow v(a)$	Kopiowanie liczby z $v(a)$ do akumulatora, czyli $m(0)$
STORE x	$m(x) \leftarrow m(0)$	Kopiowanie liczby z akumulatora do $m(x)$
STORE x	$m(m(x)) \leftarrow m(0)$	Kopiowanie liczby z akumulatora do $m(m(x))$
ADD a	$m(0) \leftarrow m(0) + v(a)$	Sumowanie wartości akumulatora i $v(a)$, a wynik jest składowany w akumulatorze
SUB a	$m(0) \leftarrow m(0) - v(a)$	Odejmowanie $v(a)$ od wartości akumulatora, a wynik jest składowany w akumulatorze
MULT a	$m(0) \leftarrow m(0) \times v(a)$	Mnożenie wartości akumulatora i $v(a)$, a wynik jest składowany w akumulatorze
DIV a	$m(0) \leftarrow \lfloor m(0) / v(a) \rfloor$	Dzielenie wartości akumulatora przez $v(a)$, a wynik, zaokrąglony w dół do najbliższej liczby całkowitej, jest składowany w akumulatorze
READ x	$m(x) \leftarrow$ wartość komórki taśmy wejściowej, nad którą jest aktualnie głowica	Odczytanie wartości z taśmy wejściowej i umieszczenie go w $m(x)$. Po tej operacji głowica taśmy przesuwa się nad kolejną komórkę taśmy.
READ x	$m(m(x)) \leftarrow$ wartość komórki taśmy wejściowej, nad którą jest aktualnie głowica	Odczytanie wartości z taśmy wejściowej i umieszczenie go w $m(m(x))$. Po tej operacji głowica taśmy przesuwa się nad kolejną komórkę taśmy
WRITE a	$v(a) \rightarrow$ aktualna komórka taśmy wyjściowej, nad którą znajduje się aktualnie głowica	Zapis liczby $v(a)$ na aktualną komórkę taśmy wyjściowej. Głowica taśmy przesuwa się na kolejną pozycję
JUMP e	Licznik wskazujący na numer kolejnej instrukcji do wykonania jest nastawiany na instrukcję z etykietą e .	Skok do linii programu zawierającej etykietę e
JGTZ e	Jeżeli $m(0) > 0$, indeks lokalizacji jest nastawiany na instrukcję z etykietą e . W przeciwnym razie indeks ten jest ustawiany na kolejną instrukcję	Skok do linii programu zawierającej etykietę e pod warunkiem, że wartość akumulatora jest większa od zera

	programu.	
JZERO e	Jeżeli $m(0) = 0$, indeks lokalizacji jest nastawiany na instrukcję z etykietą e . W przeciwnym razie indeks ten jest ustawiany na kolejną instrukcję programu.	Skok do linii programu zawierającej etykietę e pod warunkiem, że wartość akumulatora jest równa zero
HALT		Wykonywanie programu kończy się.

Ogólnie mówiąc, program napisany w kodzie RAM definiuje przekształcenie taśm wejściowych w taśmy wyjściowe. Skoro nie dla wszystkich taśm wejściowych program może się zatrzymać (możemy napisać program, który się zapętli), przekształcenie jest częściowe (czyli może być nieokreślone dla pewnych danych wejściowych).

Złożoność obliczeniowa programów RAM

Gdy chcemy, za pomocą programu komputerowego obliczyć jakiś problem, należy najpierw skonstruować algorytm. Jeśli jednak posiadamy do dyspozycji więcej niż jeden algorytm, zapewne chcielibyśmy wybrać ten lepszy. Tutaj zadajemy sobie pytanie: jak określić, który z nich jest bardziej optymalny oraz jakie kryterium mamy wziąć pod uwagę analizując kolejne algorytmy? Ten rozdział jest odpowiedzią na te pytania.

Miarami, którymi będziemy mierzyć jakość algorytmu to jego złożoność czasowa i pamięciowa w stosunku do rozmiaru danych. Jeżeli za złożoność, dla pewnego rozmiaru danych, weźmiemy złożoność maksymalną dla wszystkich danych tego rozmiaru, to złożoność tę nazywa się *złożonością najgorszego przypadku*. Jeżeli za złożoność weźmiemy średnią złożoność dla wszystkich danych pewnego rozmiaru, nazywamy ją wówczas *złożonością oczekiwaną*. Oszacowanie złożoności oczekiwanej algorytmu jest przeważnie trudniejsze od oszacowania złożoności najgorszego przypadku. Warunkiem do takiego szacowania jest pewne założenie o rozkładzie danych, co nie zawsze jest trywialnym zadaniem. Aktualnie skupimy się na złożoności najgorszego przypadku, która przy okazji ma bardziej uniwersalne zastosowanie, jednakże pamiętajmy, że algorytm o najlepszej złożoności najgorszego przypadku niekoniecznie musi mieć najlepszą złożoność oczekiwaną.

Złożoność czasowa najgorszego przypadku programu napisanego w kodzie RAM jest funkcją $f(n)$, która dla wszystkich danych rozmiaru n jest maksymalnym czasem użytym na wykonywanie programu, dla tych danych. Aby móc policzyć czas zużywany na wykonanie programu, musimy zdefiniować go dla każdej instrukcji z osobna. Głównym czynnikiem, który wpływa na zużywany czas dla danej instrukcji, jest długość liczby będącej jej operandem (zależnie od wybranego kryterium, o którym poniżej).

Dla programów RAM definiujemy dwa takie kryteria kosztu: zuniformizowany oraz logarytmiczny.

Kryterium kosztu zuniformizowanego mówi, że każda instrukcja kodu RAM wymaga jednej jednostki czasu, natomiast każda komórka pamięci, jednej jednostki pamięci.

Kryterium kosztu logarytmicznego uwzględnia długość operandu (co jest bardziej miarodajne dla programów operujących na liczbach o dowolnej wielkości).

Zdefiniujmy funkcję $lcost(x)$:

$$lcost(x) = \lfloor \log |x| \rfloor + 1 \text{ dla } x \neq 0, \text{ oraz} \\ lcost(x) = 1 \text{ dla } x = 0.$$

$lcost(x) = 0$ dla x o nieokreślonej wartości

Dla krótszego zapisu zdefiniujemy sobie $t(a)$ dla trzech możliwych postaci operandum a .

Operandum a	Koszt $t(a)$
$=x$	$lcost(x)$
x	$lcost(x) + lcost(m(x))$
x	$lcost(x) + lcost(m(x)) + lcost(m(m(x)))$

Posiadając odpowiednie narzędzia, możemy teraz opisać koszt każdej z instrukcji kodu RAM, dla każdego z wariantów operandum:

Instrukcja	Koszt
LOAD a	$t(a)$
STORE x	$lcost(m(0)) + lcost(x)$
STORE x	$lcost(m(0)) + lcost(x) + lcost(m(x))$
ADD a	$lcost(m(0)) + t(a)$
SUB a	$lcost(m(0)) + t(a)$
MULT a	$lcost(m(0)) + t(a)$
DIV a	$lcost(m(0)) + t(a)$
READ x	$lcost(input) + lcost(x)$
READ x	$lcost(input) + lcost(x) + lcost(m(x))$
WRITE a	$t(a)$
JUMP e	1
JGTZ e	$lcost(m(0))$
JZERO e	$lcost(m(0))$
HALT	1

W przedstawionej powyżej tabelce uwzględnione jest, że reprezentacja liczby całkowitej i w komórce pamięci wymaga $\lfloor \log i \rfloor + 1$ bitów. Przypomnijmy, że komórki pamięci, mogą zawierać dowolnie duże liczby całkowite.

Kryterium kosztu logarytmicznego opiera się na założeniu, że koszt wykonania instrukcji jest proporcjonalny do długości operandów tych instrukcji. Aby lepiej zobrazować obliczanie kosztu, posłużmy się przykładem i policzmy koszt dla instrukcji MULT x . Najpierw musimy ustalić koszt, gdzie $t(a)$ jest kosztem operandum a . Aby rozpoznać liczbę całkowitą x przez maszynę, potrzeba czasu $lcost(x)$. Następnie, aby odczytać $m(x)$ (zawartość komórki o indeksie x) oraz odszukać rejestr $m(x)$ potrzeba czasu $lcost(m(x))$. Z kolei czytanie zawartości rejestru $m(x)$ kosztuje $lcost(m(m(x)))$. Skoro instrukcja MULT x mnoży liczbę całkowitą $m(m(x))$ przez $m(0)$, widzimy, że ostatecznym kosztem, jaki należy przypisać instrukcji MULT x , jest $lcost(m(0)) + lcost(x) + lcost(m(x)) + lcost(m(m(x)))$.

Logarytmiczną złożoność pamięciową programu RAM definiujemy jako sumę $lcost(i_x)$ po wszystkich komórkach pamięci, gdzie i_x jest największą liczbą całkowitą, jaka była umieszczona w komórce pamięci o indeksie x podczas obliczeń. Przy obliczaniu sumy nie będzie nam przeszkadzało, że większość komórek będzie zawierała wartość nieokreśloną (czyli taką, jaką posiadają przed uruchomieniem programu wszystkie komórki pamięci), ponieważ dla takiego przypadku koszt wynosi 0 (patrz: definicja funkcji $lcost$).

Z powyższego jasno wynika, że dany program może mieć całkowicie różne złożoności czasowe i pamięciowe zależnie od tego, czy użyje się do szacowania kosztu

zuniformizowanego, czy logarytmicznego. Jeżeli zakładamy, że każda liczba zajmuje jedną jednostkę pamięci lub po prostu: stałą liczbę jednostek, wówczas stosujemy koszt zuniformizowany (np.: analizując program napisany w języku C, używający zmiennych typu `int`). W przeciwnym razie, dla realistycznej analizy złożoności, bardziej właściwy może być koszt logarytmiczny (w szczególności – analizując programy w kodzie RAM).

Obsługa programu „Maszyna RAM”

Informacje wstępne

„Maszyna RAM” jest programem komputerowym, napisanym w języku C# w technologii Microsoft® .NET. Możliwość uruchamiania napisanych w tej technologii programów istnieje na systemach operacyjnych z zainstalowanym Framework’iem .NET w wersji 1.1. Pakiet ten jest dostępny nieodpłatnie na stronie www.microsoft.com i udostępniany w wielu wersjach językowych. W/w framework jest zbiorem bibliotek niezbędnych do uruchamiania programów napisanych w technologii .NET. Oprócz systemów operacyjnych z rodziny MS Windows®, które jako pierwsze udostępniły obsługę programów opartych na .NET, obecnie trwają prace nad umożliwieniem uruchamiania takich programów na innych systemach operacyjnych, między innymi systemie Linux (patrz: Project MONO - www.mono-project.com). Wedle zapewnień twórców projektu MONO, programy napisane w powyższej technologii możemy uruchamiać na systemie Linux wprost, bez jakiegokolwiek uprzedniej rekompilacji kodu.

Program „Maszyna RAM” jest implementacją abstrakcyjnego modelu obliczeń o tej samej nazwie. Program, w ogólnym zarysie, służy do pisania, testowania i uruchamiania programów napisanych w kodzie RAM. Jest napisany z myślą o wygodzie użytkownika, co pozwoli na szybkie nauczanie się jego obsługi. Możliwa największa ilość akcji, którą można wykonać w programie, jest zaprojektowana w oparciu o wiedzę na temat HCI (Human Computer Interaction) oraz o obowiązujące światowe standardy pisania oprogramowania, by program był maksymalnie intuicyjny w obsłudze.

Abstrakcyjny model Maszyny RAM nie posiada ograniczenia na liczbę komórek pamięci, liczbę komórek taśmy wejściowej i wyjściowej oraz wielkości samych liczb, co powoduje, że żaden realny komputer nie okiełzna takich wymagań. Tak też jest w przypadku omawianego tutaj programu. Autor programu nie nałożył żadnych ograniczeń co do podanych powyżej wielkości, jednak takie ograniczenia nakłada sprzęt i system operacyjny, na którym program będzie uruchamiany. Ale są to jedyne ograniczenia. Wymagania co do prędkości komputera są tutaj minimalne, wystarczy bowiem dowolny sprzęt z procesorem co najmniej 500Mhz, choć i wolniejszy także nie byłby specjalnym problemem.

Program „Maszyna RAM” był intensywnie testowany przez trzy osoby, w tym dwóch zawodowych testerów, co pozwala mieć nadzieję, że jest dostatecznie stabilny. Autor jednak bierze pod uwagę fakt, iż w programie mogą się znajdować jeszcze jakieś ukryte błędy, dlatego też w przypadku jego odnalezienia przez potencjalnych użytkowników, autor jest otwarty na wszelkie zgłoszenia w tej sprawie. W miarę możliwości zgłaszane błędy będą poprawiane, a poprawiona, najnowsza wersja programu będzie zawsze dostępna pod adresem:

<http://www.szkup.com>

Instalacja programu

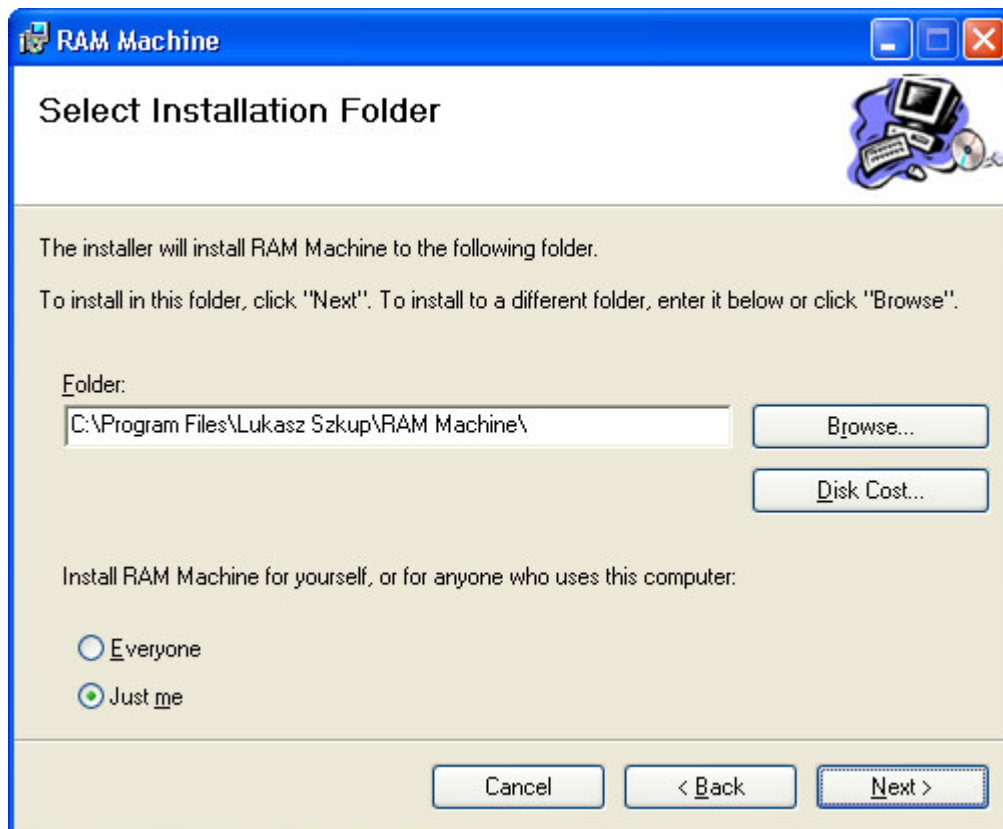
Na płycie CD dostarczonej wraz z niniejszą pracą znajduje się plik instalacyjny RAMMachineSetup.msi, który należy uruchomić, by rozpocząć instalację. Plik ten jest także dostępny na stronie autora (pod wskazanym adresem powyżej). Jeśli nie posiadamy w systemie zainstalowanego Framework’a .NET w wersji 1.1, należy go wcześniej zainstalować. Wersja polska i angielska została dostarczona na płycie CD z pracą magisterską (odpowiednio pliki „dotnetfx_pl.exe” oraz „dotnetfx_en.exe”), a pozostałe

wersje językowe można bezpłatnie pobrać ze strony www.microsoft.com (słowa kluczowe: NET Framework 1.1).

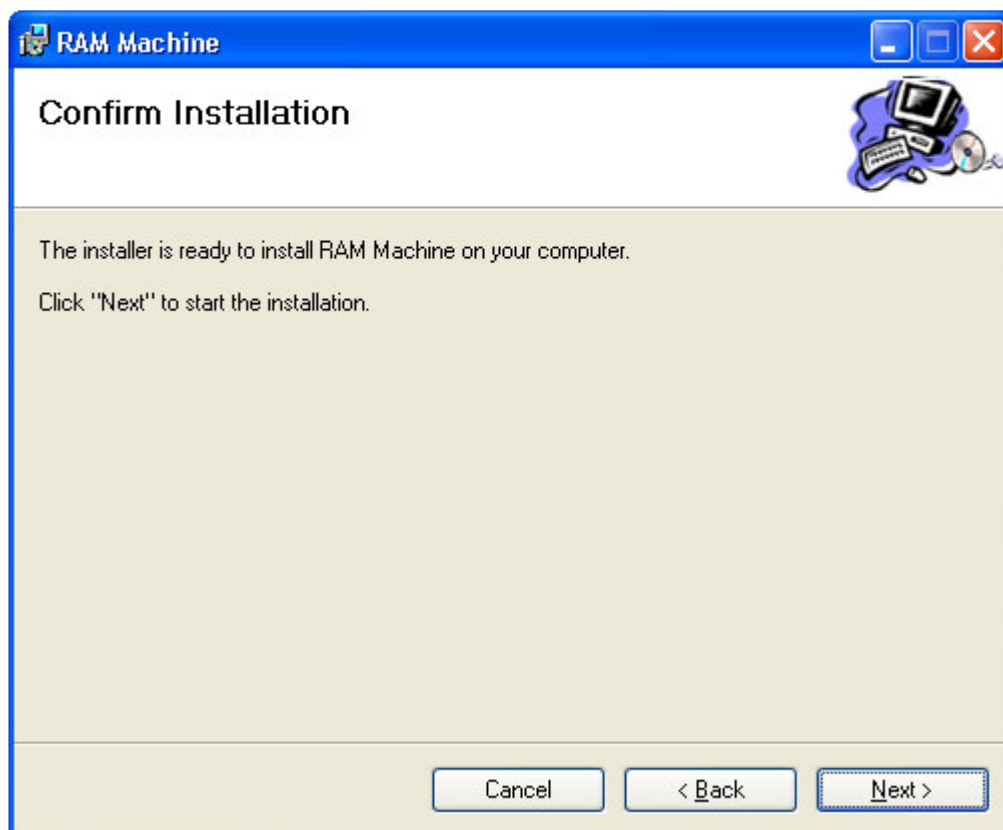
Po uruchomieniu pliku instalacyjnego otrzymamy następujące okno na ekranie (Rysunek 1). Naciskamy przycisk „Next” i przechodzimy do następnego okna (Rysunek 2). Możemy teraz wybrać ścieżkę do katalogu, pod którą zostanie zainstalowany program. Początkowa propozycja będzie wyświetlona przez instalator, jednak czasami należy zmienić tą ścieżkę, gdy na danym komputerze nie ma praw do instalowania oprogramowania w katalogu „Program Files”. Wybrać możemy także, czy program ma być zainstalowany tylko dla aktualnie zalogowanego użytkownika, czy dla wszystkich, którzy posiadają konta na danym komputerze. Następnie naciskamy przycisk „Next” i przechodzimy do kolejnego okna (Rysunek 3). Ponownie naciskamy „Next”, po czym pojawi się ostatnie okno instalacji (Rysunek 4). Po wciśnięciu przycisku „Close” kończymy instalację i program jest gotowy do uruchomienia. Jeśli instalacja przebiegła prawidłowo, na pulpicie zostaną dodane dwie ikonki (Rysunek 5). Pierwsza z nich służy do uruchomienia programu „Maszyna RAM”, natomiast druga jest skrótem do folderu, w którym są przechowane przykładowe programy napisane w kodzie RAM.



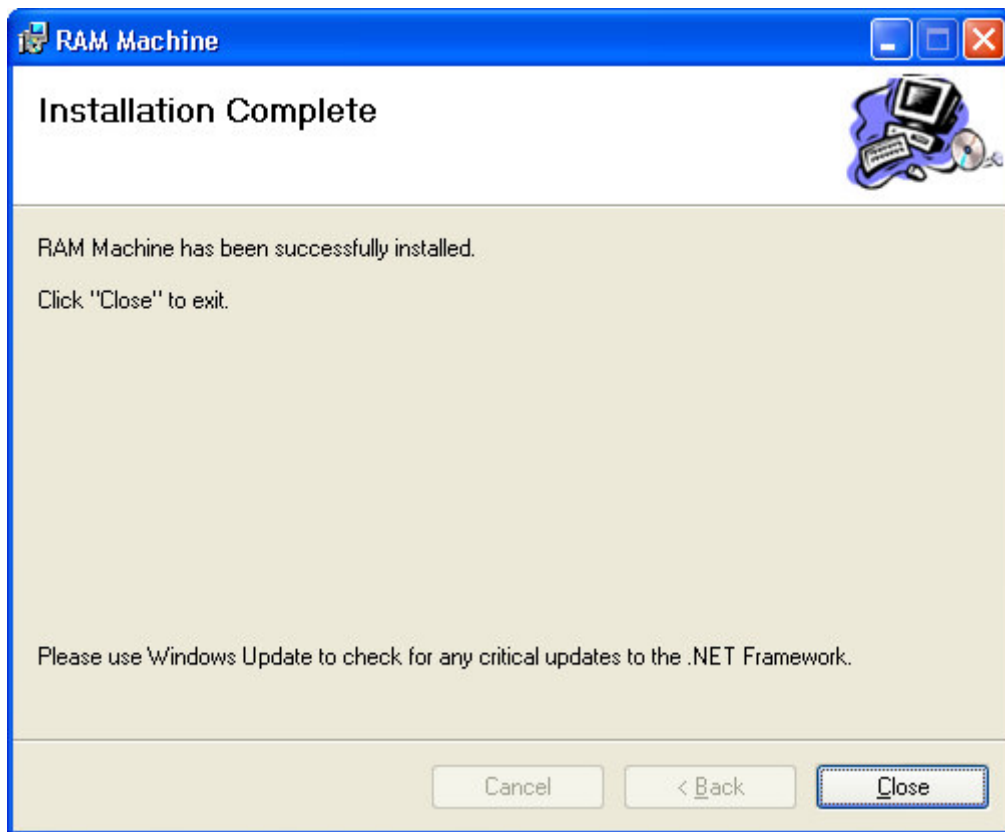
Rysunek 1



Rysunek 2



Rysunek 3



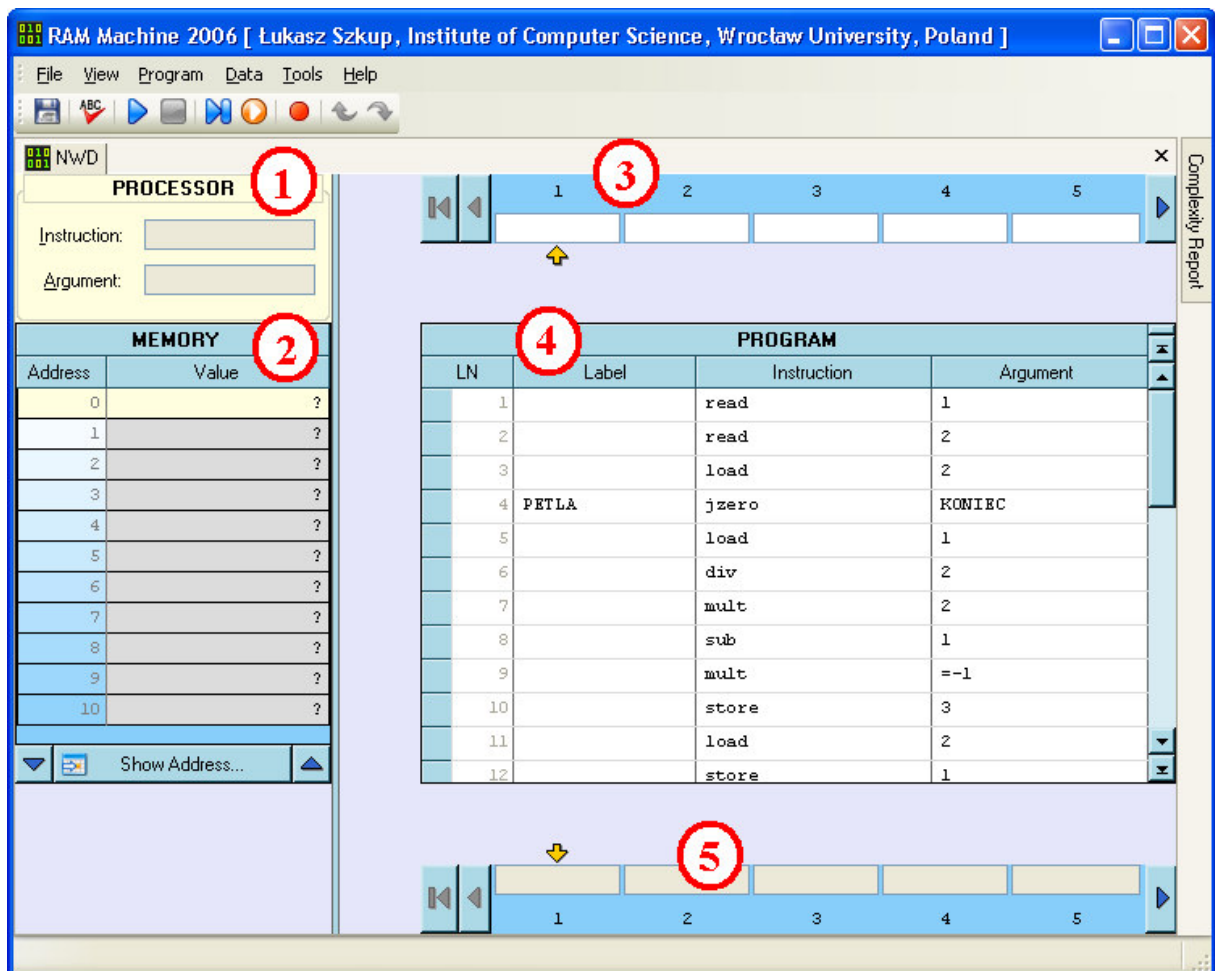
Rysunek 4



Rysunek 5

Składowe edytora „Maszyny RAM”

Program „Maszyna RAM” składa się z pięciu głównych składników (Rysunek 6): procesora (1), pamięci (2), taśmy wejściowej (3), edytora programu (4) i taśmy wyjściowej (5). Wszystkie składniki zostały w taki sposób umieszczone na ekranie, by maksymalnie ułatwić pracę z programem. Każdy ze składników zostanie opisany szczegółowo w poniższych rozdziałach. Składniki te będziemy dalej nazywali kontrolkami (jest to powszechnie używana nazwa w stosunku do graficznych komponentów w aplikacjach okienkowych).



Rysunek 6

Kontrolka procesora

Kontrolka procesora jest najprostszą kontrolką w programie (Rysunek 7). Służy do wyświetlania aktualnie wykonywanego rozkazu oraz jego adresu.

PROCESSOR

Instruction:

Argument:

Rysunek 7


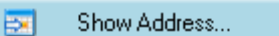

Początkowo, przed uruchomieniem programu w kodzie RAM, kontrolka ta nie zawiera żadnych danych. Gdy uruchomimy program, tuż przed wykonaniem danej linii kodu, maszyna ładuje rozkaz oraz jego adres (argument) do procesora. Jeśli w opcjach programu jest to włączone, operacja ta będzie poprzedzona odpowiednią animacją. Rozkaz będzie widoczny w polu na lewo od napisu „Instruction”, a argument w polu na lewo od napisu „Argument”. Jeśli nazwa rozkazu lub argumentu będzie zbyt długa, by się zmieścić w w/w polach, pojawi się jedynie jej początek zakończony znakiem trzykropka. Aby ujrzyć pełny napis można regulować wielkość okna lub zmieniać położenie pionowej podziałki przylegającej do prawej strony kontrolki procesora i pamięci. Można także skopiować wartość dowolnego z pól do schowka (np.: za pomocą myszki i menu kontekstowego –

opcja „Copy”) i wkleić jego zawartość do dowolnego innego edytora tekstowego, np.: Notepad czy MS Word®. Gdy program ładuje kolejną instrukcję do wykonania, poprzednia zawartość pól rozkazu i argumentu jest zastępowana nowymi wartościami. Po wykonaniu programu, w/w pola są czyszczone.

Kontrolka pamięci

Kontrolka pamięci (Rysunek 8) pozwala nam śledzić zawartość pamięci maszyny RAM. Posiada dwie kolumny: „Address” i „Value”, które odpowiednio wyświetlają adres (indeks) komórki pamięci oraz jej zawartość. Komórka o adresie zero jest zaznaczona na inny kolor, niż pozostałe komórki. Komórka ta – zwana akumulatorem – jest szczególna, ponieważ jest używana przez większość operacji do obliczeń/porównań i dlatego też zasłużyła na to, by lepiej ją uwidocznić.

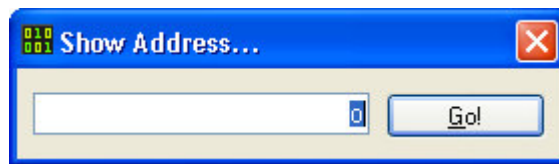
MEMORY	
Address	Value
0	69
1	?
2	?
3	?
4	334
5	?
6	?
7	7755
8	?
9	?
10	?

Rysunek 8

Pola ukazujące wartości komórek pamięci, które posiadają wartość nieokreśloną, wyświetlają symbol „?” oraz są zaznaczone kolorem szarym. Jeśli w danej komórce pamięci znajduje się liczba, pole to zmieni kolor na biały. Dzięki takiej manipulacji użytkownik łatwiej będzie mógł się zorientować w zawartości pamięci.

Kontrolka pamięci pokazuje jedynie pewien wycinek pamięci maszyny RAM, jednak mamy dostęp do dowolnego adresu. Do tego celu służą trzy przyciski znajdujące się na dole kontrolki. Przyciskami po lewej i prawej stronie (z ikonkami strzałek) przewijamy widok o jeden adres w górę/dół. Dłuższe przytrzymanie wciśniętego przycisku spowoduje automatyczne przewijanie o kolejne komórki, aż do momentu zwolnienia przycisku. Środkowym przyciskiem („Show Address...”) możemy szybko przejść do dowolnej komórki pamięci. Gdy go naciśniemy, pojawi się nam okno „Show Address...” (Rysunek 9). Możemy podać dowolny adres komórki pamięci i nacisnąć przycisk „Go!”, po czym kontrolka pamięci przewinie nam widok w taki sposób, że pierwszą widoczną komórką (u góry kontrolki) będzie ta, przed chwilą podana przez użytkownika.



Rysunek 9

Teraz się zajmiemy kolumną z adresami komórek pamięci. Jak widzimy na rysunku Rysunek 8, pola z adresami są odpowiednio cieniowane, począwszy od koloru białego do niebieskiego. W przypadku pokazywania przez kontrolkę ciągłego bloku pamięci, taki sposób cieniowania może okazać się zbyt cenny, ale spójrzmy jak jest to istotne, gdy stosujemy adresowanie bezpośrednie (Rysunek 10) lub pośrednie (Rysunek 11).

Kontrolka pamięci podczas wykonywania programu w kodzie RAM, automatycznie dopasowuje swój widok w taki sposób, by były widoczne wszystkie komórki pamięci biorące udział w danej operacji. Dlatego też, jeśli wykonywana jest instrukcja odwołująca się do argumentu poprzez adresowanie bezpośrednie, np.: ADD 20 (czyli: dodaj do zawartości akumulatora wartość komórki o adresie 20), wtedy kontrolka pamięci pogrupuje nam widok w dwa ciągłe bloki pamięci, aby na raz widoczne były komórki o numerze 0 (akumulator) i 20 (Rysunek 10).

W przypadku adresowania pośredniego sytuacja wygląda analogicznie, lecz tutaj należy pokazać trzy komórki pamięci. Na kolejnym obrazku (Rysunek 11) jest pokazany widok pamięci po wykonaniu instrukcji: LOAD ^300 (czyli: zajrzyj do komórki o adresie 300, następnie znaleziona tam wartość niech będzie adresem do komórki, której wartość załaduj do akumulatora).

Teraz, analizując rysunki Rysunek 10 i Rysunek 11, widać jak bardzo istotne jest cieniowanie komórek z adresami pamięci. Dzięki temu użytkownik nie będzie miał problemu z odróżnieniem osobnych, ciągłych bloków pamięci maszyny RAM, co powoduje, że kontrolka pamięci jest przejrzysta i czytelna.

MEMORY	
Address	Value
0	123
1	?
2	?
3	?
4	?
5	?
20	6
21	?
22	?
23	?
24	?

Rysunek 10

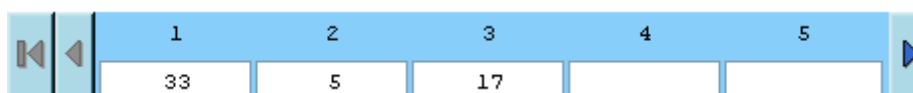
MEMORY	
Address	Value
0	44
1	?
2	?
3	?
300	570
301	?
302	?
570	44
571	?
572	?
573	?

Rysunek 11

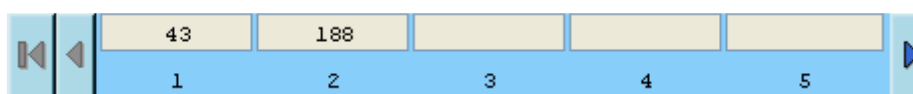
Kontrolka taśmy wejściowej i wyjściowej

Odzwierciedleniem taśmy wejściowej i wyjściowej maszyny RAM w programie są odpowiednie kontrolki, jak na rysunku Rysunek 12 oraz Rysunek 13. Użytkownik ma możliwość wprowadzać dane jedynie do komórek taśmy wejściowej, ponieważ taśma wyjściowa służy do zapisywania wyników jedynie przez program w kodzie RAM. Jednakże każda z taśm udostępnia użytkownikowi odczyt wartości z dowolnej taśmy i jej komórki. Nad każdą komórką taśmy wejściowej znajduje się jej numer (do numeracji są użyte kolejne liczby całkowite począwszy od 1). Taśma wyjściowa także posiada taką numerację, lecz pod spodem komórek taśmy. Numeracja została wprowadzona, by ułatwić użytkownikowi orientację w pozycji taśmy podczas nawigowania. Do nawigacji po taśmie służą trzy przyciski: dwa po lewej stronie i jeden po prawej. Pierwszym po lewej przewijamy widok taśmy do pierwszej komórki taśmy, a pozostałymi przyciskami przewijamy taśmę o jedną komórkę w lewo/prawo. Tak jak w przypadku przycisków na kontrolce pamięci, dłuższe przyciśnięcie przycisku spowoduje przewijanie o większą liczbę komórek, aż do momentu zwolnienia przycisku.

Przy każdej z taśm widoczna jest strzałka (Rysunek 6), pokazująca aktualną pozycję głowicy taśmy. Gdy program w kodzie RAM wykona instrukcję czytania (READ) lub zapisywania (WRITE) wartości na taśmę, strzałka ta przesunie się na kolejną komórkę taśmy, symbolizując nowe położenie głowicy. Gdy pozycja głowicy przesuwa się nad komórkę taśmy, która nie jest widoczna na ekranie, kontrolka taśmy automatycznie przewinie widok taśmy w taki sposób, by komórka ta była widoczna.



Rysunek 12



Rysunek 13

Edytor programu

Za pomocą edytora możemy napisać dowolny program w kodzie RAM. Edytor składa się łącznie z 7 kolumn o nazwach: LN, Label, Instruction, Argument, Comment, EC, EP.

PROGRAM						
LN	Label	Instruction	Argument	Comment	EC	EP
1		read	1		0	0.000 %
2		read	2		0	0.000 %
3		load	2		0	0.000 %
4	PETLA	jzero	KONIEC		0	0.000 %
5		load	1		0	0.000 %
6		div	2		0	0.000 %
7		mult	2		0	0.000 %
8		sub	1		0	0.000 %
9		mult	==1		0	0.000 %
10		store	3		0	0.000 %
11		load	2		0	0.000 %
12		store	1		0	0.000 %
13		load	3		0	0.000 %
14		store	2		0	0.000 %
15		jump	PETLA		0	0.000 %
16	KONIEC	write	1		0	0.000 %

Rysunek 14

Pierwsza kolumna pokazuje numer linii programu. Druga kolumna („Label”) służy do umieszczenia unikalnej etykiety dla danej linii, do której może się odwoływać dowolna z instrukcji skoku. Kolejna kolumna („Instruction”) służy do wpisania jednej z dostępnych instrukcji maszyny RAM. Podczas wpisywania instrukcji pojawi się automatyczna podpowiedź, jeśli wpisany ciąg znaków jest prefiksem jednej z dostępnych instrukcji. Możemy taką instrukcję wybrać również z rozwijanej listy. By ją aktywować, należy kliknąć na przycisk, który się pojawia z prawej strony pola, gdzie wpisujemy instrukcję. Następną kolumną („Argument”) służy do podania argumentu instrukcji (jedynie instrukcja HALT tego nie wymaga). Podczas wpisywania argumentu, jeśli jest on etykietą skoku, pojawia się podpowiedź, tak samo jak w przypadku wpisywania instrukcji, z tym, że podpowiadane są etykiety, które zostały wcześniej wpisane w kolumnie „Label”. Klikając na przycisk po prawej stronie pola można także rozwinąć pełną listę propozycji i wybrać jedną z nich. Natomiast kolumna „Comment” jest zwykłą kolumną tekstową, gdzie można wpisać dowolny tekst (przeważnie będzie to komentarz do programu).

Kolejne dwie kolumny to „EC” (execution counter) i „EP” (execution percent). Początkowo nie są widoczne, ale w każdym momencie można je uwidocznić (lub z powrotem schować) za pomocą odpowiedniej opcji w menu lub za pomocą skrótów klawiszowych (odpowiednio) Ctrl+6 i Ctrl+7. Kolumny te pokazują odpowiednio licznik wykonania danej linii kodu oraz procent wykonania (w stosunku do łącznej liczby wykonanych linii). Kolumny te są szczególnie przydatne, gdy analizujemy tzw. „wąskie gardła” programu.

Do przemieszczania kursora w edytorze możemy używać klawiszy: Tab, Enter oraz strzałek. Klawiszem Tab przechodzimy do następnej komórki na prawo. Jeśli aktualnie jesteśmy w ostatniej edytowalnej komórce danego wiersza, kursor przechodzi do pierwszej komórki po lewej w wierszu o jeden niżej. W razie potrzeby, edytor doda nowy wiersz programu, jeśli przejście do poniższego wiersza nie jest możliwe (z powodu jego braku).

Klawisz Enter działa podobnie jak w przypadku typowych edytorów – tworzony jest nowy wiersz programu, tuż pod tym, w którym znajduje się aktualnie kursor, następnie kursor przechodzi do nowej linii edytora, do pierwszej komórki po lewej stronie wiersza.

Klawiszami strzałek możemy z kolei dowolnie przemieszczać kursor po obszarze edytora. Kierunek przemieszczania się kursora jest zgodny ze strzałkami, które są naciskane. Oczywiście po edytorze możemy także nawigować za pomocą myszki, klikając w interesującą nas komórkę, by przenieść do niej kursor. Pomocny jest także suwak znajdujący się z prawej strony edytora przeznaczony do przewijania programu, gdy nie mieści się w całości na ekranie.

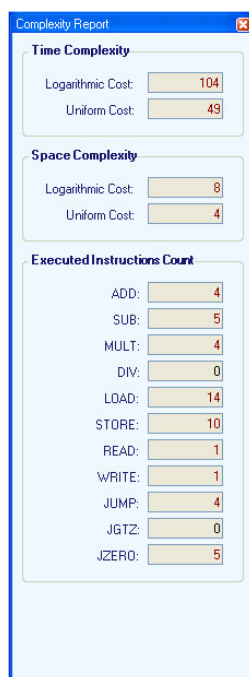
Podczas pisania programu, każda zmiana jest rejestrowana, dzięki czemu, możemy wrócić do dowolnego wcześniejszego stanu. Za pomocą skrótu klawiszowego Ctrl+Z cofamy się o jedną zmianę. Jeśli użyjemy tego skrótu, by cofnąć się do jednego z poprzednich stanów i nie zmodyfikujemy nic w programie, możemy z powrotem wrócić do pierwotnych zmian za pomocą skrótu Ctrl+Y.

Pomocny jest także klawisz Insert, za pomocą którego dodajemy jeden pusty wiersz w miejscu, gdzie jest aktualnie kursor. Wiersze znajdujące się od pozycji kursora w dół, zostaną przesunięte o jeden wiersz do dołu, a tuż nad nimi zostanie dodany nowy wiersz. Po tej operacji kursor pozostaje w nowo dodanym wierszu.

Zaimplementowana jest także obsługa skrótów klawiszowych Ctrl+A, Ctrl+C, Ctrl+V i Ctrl+X, które działają podobnie jak w typowych edytorach tekstu, a mianowicie (odpowiednio): „zaznacz wszystkie wiersze programu”, „skopiuj do schowka zaznaczone wiersze programu”, „wklej wiersze programu znajdujące się w schowku” (w miejscu zaznaczonego wiersza) oraz „skopiuj do schowka a następnie wytnij zaznaczone wiersze programu”.

Kontrolka statystyk złożoności programu

Kontrolka statystyk (okno Complexity Report - Rysunek 15) pokazuje nam, w trakcie wykonywania programu, aktualne statystyki złożoności oraz ilości wykonanych instrukcji.



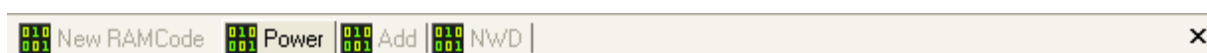
Rysunek 15 – kontrolka statystyk

Na samej górze okienka mamy sekcję złożoności czasowej (Time Complexity), która pokazuje koszt logarytmiczny oraz zuniformizowany (jednostkowy), który jest obliczany zgodnie z założeniami w rozdziale „Złożoność obliczeniowa programów RAM”. Następna sekcja przedstawia koszt pamięciowy (Space Complexity), w której również jest przedstawiony koszt zarówno logarytmiczny jak i jednostkowy, zgodnie z ustaleniami znajdującymi się we wspomnianym powyżej rozdziale. Ostatnia sekcja (Executed Instruction Count) to lista instrukcji, jakie potrafi wykonywać maszyna RAM oraz licznik, który mówi, ile razy dana instrukcja została wykonana przez maszynę od czasu uruchomienia programu.

Kiedy program „Maszyna RAM” obsługuje na raz więcej niż jeden uruchomiony program, napisany w kodzie RAM, kontrolka pokazuje statystyki jedynie dla tego, który jest aktualnie aktywny (wyświetlony na ekranie). Jeśli chcemy przesunąć inny program na pierwszy plan, by oglądać jego statystyki, należy go wskazać, klikając na odpowiednią zakładkę na górnej części okna (Rysunek 16).

Istnieje także możliwość wyzerowania aktualnie pokazywanych statystyk. Może się to okazać pomocne, gdy chcemy zbadać złożoność programu jedynie od pewnego momentu, np.: po wczytaniu danych z taśmy wejściowej, by sprawdzić jedynie złożoność samego algorytmu programu. Czyszczenie listy wywołujemy z poziomu menu programu. Gdy nie chcemy oglądać statystyk programu, możliwe jest schowanie okna poprzez kliknięcie w ikonkę krzyżyka lub użycie klawisza F3. By ponownie pokazać okno statystyk naciskamy F3 lub odpowiednią opcję w menu programu.

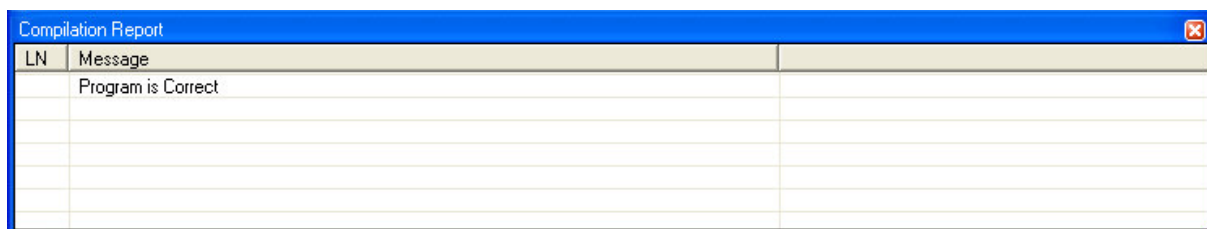
Początkowo, po uruchomieniu „Maszyny RAM”, okno statystyk jest „zaczepione” po prawej stronie głównego okna aplikacji, w taki sposób, że widać jedynie pionową belkę z nazwą okienka. Po najechaniu myszką na belkę, okno statystyk się rozwinie. Po przesunięciu myszki z powrotem na inne części aplikacji, okno statystyk się zwinie do postaci belki. Jeśli chcemy zablokować efekt chowania, należy kliknąć ikonkę pinezki w prawej górnej części okna statystyk. Tą samą ikonkę klikamy, by przywrócić efekt chowania.



Rysunek 16 - belka z załadowanymi programami

Weryfikacja, uruchamianie i śledzenie programu

Gdy już napiszemy program lub jego fragment możemy sprawdzić jego poprawność. Można to zrobić za pomocą odpowiedniej pozycji w menu lub za pomocą klawisza F7. Gdy program zostanie pozytywnie zweryfikowany, pojawi się na dole okna programu okienko informacyjne z napisem *Program is correct*.



Rysunek 17- okno raportu kompilacji

Jeśli jednak znajdą się jakiegokolwiek błędy, to okienko to zostanie wypełnione listą błędów (ich opisów). Kliknięcie myszką na dowolnym wierszu z opisem błędu na w/w liście spowoduje zaznaczenie błędnej linii programu w edytorze. Dzięki temu można łatwo się przemieszczać po błędnych liniach programu i sprawnie je poprawiać.

Po zweryfikowaniu poprawności napisanego (lub wczytanego z pliku) programu zazwyczaj chcemy go uruchomić, by sprawdzić jego działanie. Do tego celu służą klawisze F5, Shift+F5 oraz Shift+Ctrl+F5. Klawisz F5 służy do uruchomienia programu w trybie pracy krokowej - zatrzymanie wykonywania następuje automatycznie, gdy indeks aktualnie wykonywanego wiersza programu napotka wiersz z ustawionym punktem zatrzymania – tzw. breakpoint'em.

PROGRAM			
LN	Label	Instruction	Argument
1		read	1
2		read	2
3		load	2
4	PETLA	jzero	KONIEC
5		load	1
6		div	2
7		mult	2
8		sub	1
9		mult	=-1

Rysunek 18 – punkty zatrzymania programu (czerwone ikonki)

W każdym momencie pracy, czy to podczas pisania programu, czy też podczas jego uruchamiania, możemy w dowolnym wierszu ustawić punkt zatrzymania. Dokonuje się to poprzez zaznaczenie co najmniej jednego wiersza i naciśnięcie odpowiedniej opcji w menu lub klawisza F9. Punkty zatrzymania będą miejscami, w którym program zatrzyma swoje działanie (podczas pracy krokowej), a wznowienie działania programu jest możliwe po ponownym naciśnięciu F5.

Skrót klawiszowy Shift+F5 służy do tego, aby maszyna wykonała tylko jedną instrukcję programu i zatrzymała się. Ponowne wykonanie kolejnej instrukcji jest możliwe poprzez skrót Shift+F5. Jest też możliwe użyciu klawisza F5, by program zatrzymał się dopiero po napotkaniu punktu kontrolnego, a nie po wykonaniu jednej instrukcji.

Tryb uruchamiany za pomocą skrótu Shift+Ctrl+F5 („Run Program at Full Speed”) służy do wykonania programu w całości, bez możliwości śledzenia jego działania krok po kroku, tak jak w w/w opisanych trybach. W tym trybie, jak też w każdym innym, możliwe jest natychmiastowe zatrzymanie pracy maszyny za pomocą skrótu Shift+F10 lub odpowiedniej opcji w menu.

Podczas śledzenia programu podczas jego działania obserwować możemy wszystkie kontrolki znajdujące się na ekranie. Każda z nich informuje nas graficznie o jakiegokolwiek zmianie, która jej dotyczy. Począwszy od taśmy wejściowej i wyjściowej: kiedy maszyna uruchomi instrukcję READ lub WRITE odpowiednia strzałka (informująca o położeniu głowicy taśmy) zostanie właściwie przesunięta na następną komórkę taśmy. Dzięki temu użytkownik może obserwować ruchy głowicy na każdej z taśm.

Kolejno, kontrolka pamięci także poinformuje nas o jakiegokolwiek zmianie w dowolnej z komórek. Także przy adresowaniu bezpośrednim lub pośrednim kontrolka pokaże nam wszystkie komórki pamięci, które biorą w tym udział. W razie potrzeby, gdy komórki te są bardzo odległe od siebie, by pokazać je w jednym ciągłym bloku pamięci,

kontrolka podzieli widok na odpowiednią ilość ciągłych podbloków, tak by cała operacja była widoczna dla użytkownika. By nie powtarzać informacji o pozostałym zachowaniu i obsłudze kontrolki pamięci, odsyłam czytelnika do jednego z powyższych rozdziałów poświęconych właśnie tej kontrolce (to samo się tyczy kontrolki procesora).

Teraz możemy się zająć kontrolką edytora programu, która podczas wykonywania programu posiada liczne cechy wspomagające jego śledzenie. Po pierwsze, pomocna będzie kolumna EC (execution counter), która pokazuje, ile razy dana linia kodu została wykonana przez maszynę. Druga kolumna, EP (execution percent), pokazuje procentowo ile razy dana linia kodu została wykonana na przestrzeni łącznej liczby wykonanych linii w programie do danego momentu działania maszyny RAM. Dzięki tym kolumnom możemy analizować i znajdować tzw. wąskie gardła programu, a więc miejsc, gdzie maszyna spędza najwięcej czasu na uruchamianie.

Dla lepszego ukazania linii programu, które są najczęściej uruchamiane, została wprowadzona odpowiednia kolorystyka. Po wykonaniu każdej linii programu, linia ta zostaje delikatnie zabarwiona na kolor czerwony. Dzieje się tak dla każdej wykonanej linii. Gdy dana linia zostanie wykonana po raz kolejny, odpowiednia linia ponownie jest zabarwiana, tym razem intensywniej, na kolor czerwony. Oczywiście jest tutaj nadane pewne ograniczenie, które zapobiega zabarwieniu wierszy zupełnie na czerwono, a więc tak intensywnie, że uniemożliwiłoby to odczytanie jego zawartości. Dodatkowo, gdy choć jeden z wierszy osiągnie maksymalny próg zabarwienia, zabarwienie pozostałych wierszy jest odpowiednio przeskalowane, by była dalej zachowana zasada proporcjonalnego dostosowania zabarwienia zależnie od częstości wykonania danego wiersza.

PROGRAM				
LN	Label	Instruction	Argument	
1		read	1	
2		read	2	
3		load	2	
4	PETLA	jzero	KONIEC	
5		load	1	
6		div	2	
7		mult	2	
8		sub	1	
9		mult	=-1	
10		store	3	
11		load	2	
12		store	1	
13		load	3	
14		store	2	
15		jump	PETLA	
16	KONIEC	write	1	

Rysunek 19 – zabarwianie linii kodu podczas uruchamiania

O tym, czy podczas uruchamiania programu linie mają się odpowiednio zabarwiać jak i kolorze zabarwienia można samemu zdecydować za pomocą okna z opcjami programu (szczegółowo będzie o tym poniżej).

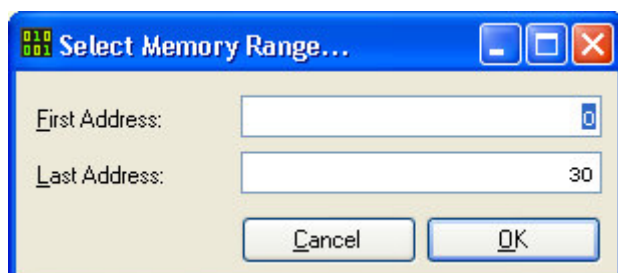
Import i Eksport

W programie „Maszyna RAM” jest możliwość importowania i eksportowania pewnych plików (np.: do dalszej obróbki w innym programie). Wyeksportować do pliku możemy:

- aktualny stan pamięci maszyny,
- stan taśmy wejściowej,
- stan taśmy wyjściowej,
- raport złożoności programu.

Z kolei import jest możliwy tylko dla taśmy wejściowej.

Podczas eksportu stanu pamięci maszyny zostaniemy dodatkowo spytani o zakres komórek pamięci, jaki nas interesuje. Możemy podać dowolny zakres, przy czym należy pamiętać, że przy dużym zakresie wygenerowany plik będzie zajmował dużo przestrzeni dyskowej lub proces eksportu zakończy się błędem przepełnienia dysku.



Rysunek 20 - okno wyświetlane przy eksporcie stanu pamięci

Wyeksportowany plik stanu pamięci zawiera serię linii tekstu, każda zakończona znakiem końca wiersza. Na początku każdej linii znajduje się numer komórki pamięci objętej w kwadratowe nawiasy. Następnie, zapisana jest wartość komórki lub znak „?”, jeśli wartość komórki w momencie eksportu nie była określona. Numer komórki pamięci jest odseparowany od wartości komórki za pomocą jednego znaku spacji. Oto przykładowa zawartość wyeksportowanego stanu pamięci dla przedziału 0 – 11:

```
[ 0] -4
[ 1] ?
[ 2] ?
[ 3] 99
[ 4] 17
[ 5] ?
[ 6] ?
[ 7] 625
[ 8] ?
[ 9] ?
[10] -117797
[11] ?
```

Format wyeksportowanego stanu taśmy wejściowej i wyjściowej to seria linii, zakończonych znakiem końca wiersza każda, gdzie w każdej linii znajduje się zawartość kolejnej komórki taśmy lub znak „?”, gdy dana komórka była pusta w momencie eksportowania pliku. Liczba eksportowanych komórek taśmy jest ograniczona przez największy numer komórki, w którym znajdowała się liczba. Oto przykład zawartości wyeksportowanego stanu taśmy wejściowej zawierającej liczby jednie w pierwszej i czwartej komórce:

33
?
?
4

Oczekiwany format przy imporcie taśmy wejściowej jest identyczny jak format eksportowanego pliku taśmy wejściowej i wyjściowej.

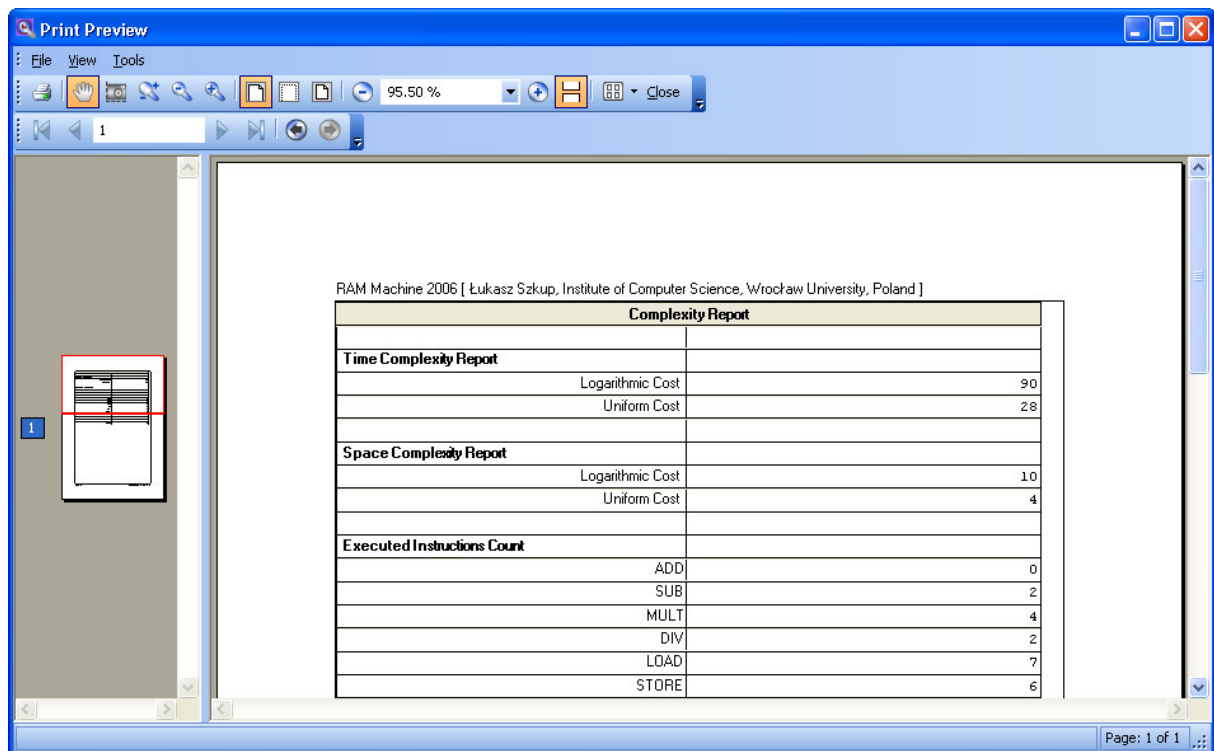
Format złożoności programu przypomina układ okna „Complexity Report” w programie. W tym przypadku od razu przejdę do przykładu, który najlepiej zobrazuje strukturę wyeksportowanego pliku:

```
Complexity Report - RAM Machine 2006  
Łukasz Szkup, Institute of Computer Science, Wrocław University, Poland  
-----
```

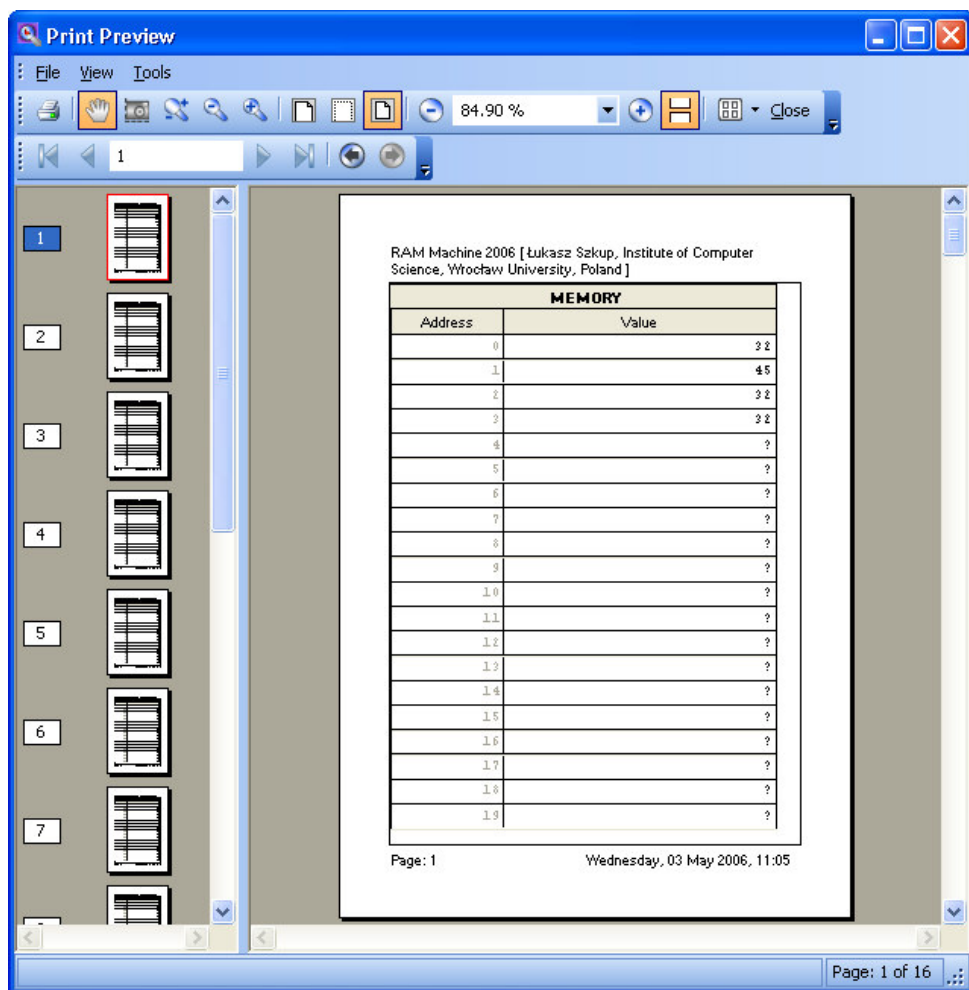
```
- Time Complexity Report  
    Logarithmic Cost: 104  
    Uniform Cost: 49  
  
- Space Complexity Report  
    Logarithmic Cost: 8  
    Uniform Cost: 4  
  
- Executed Instructions Count  
    ADD: 4  
    SUB: 5  
    MULT: 4  
    DIV: 0  
    LOAD: 14  
    STORE: 10  
    READ: 1  
    WRITE: 1  
    JUMP: 4  
    JGTZ: 0  
    JZERO: 5
```

Drukowanie

Program „Maszyna RAM” umożliwia tworzenie wydruków (opcja „Print”) poszczególnych elementów jak: stan pamięci maszyny, zawartość taśm wejścia/wyjścia, raportu złożoności oraz programu napisanego w kodzie RAM. Dla każdego z elementów możemy także skorzystać z opcji „Print Preview”, gdzie możemy podglądać wydruk (przed lub zamiast drukowania). Tuż przed wydrukowaniem zostanie nam wyświetlone okienko, gdzie możemy wybrać docelową drukarkę oraz dokładnie skonfigurować parametry wydruku. Użyteczna może też być opcja „Page Settings”, gdzie możemy dokładnie ustalić jakiej wielkości papier posiadamy w drukarce oraz jakie życzymy sobie pozostawić marginesy z każdej ze stron kartki. Oto przykładowe podglądy wydruków dla wybranych elementów programu:



Rysunek 21 – podgląd wydruku raportu złożoności



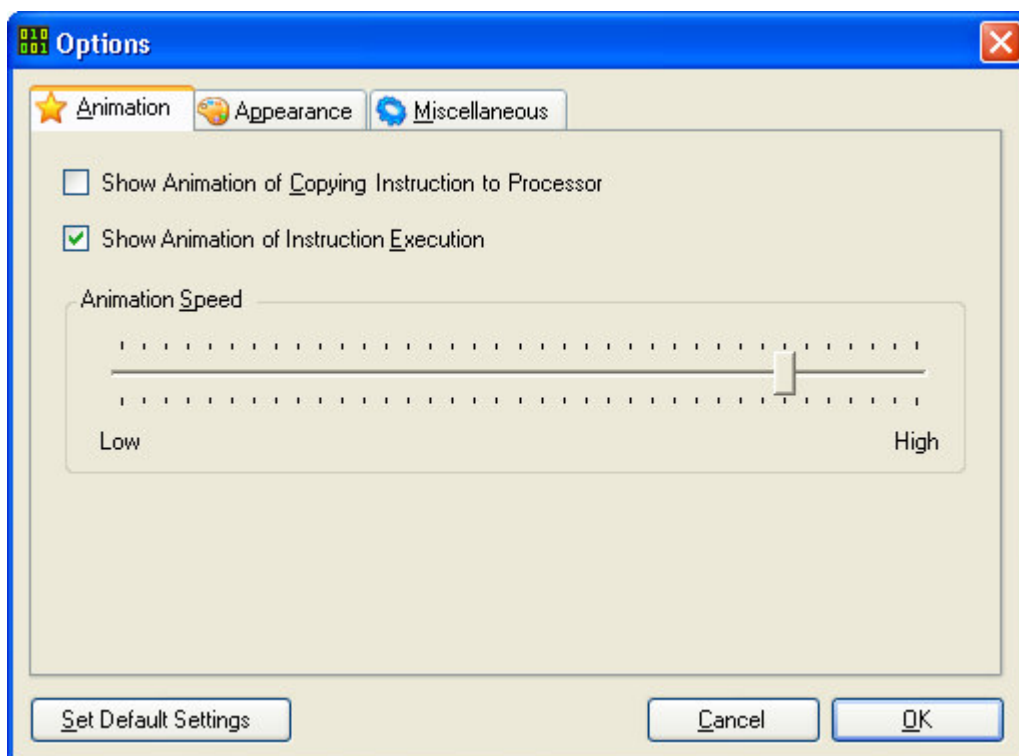
Rysunek 22 – podgląd wydruku kontrolki pamięci (przedział adresów: 0-300 oraz rozmiar kartki: A6)

Opcje – konfiguracja programu

Po wybraniu z menu programu opcji „Tools”, na następnie „Options” otrzymujemy dostęp do wszystkich opcji programu, gdzie możemy dowolnie skonfigurować wiele zachowań programu, jak też jego wygląd. Okno opcji jest podzielone tematycznie na zakładki: „Animation”, „Appearance” oraz „Miscellaneous”.

Zakładka „Animation” posiada następujące ustawienia:

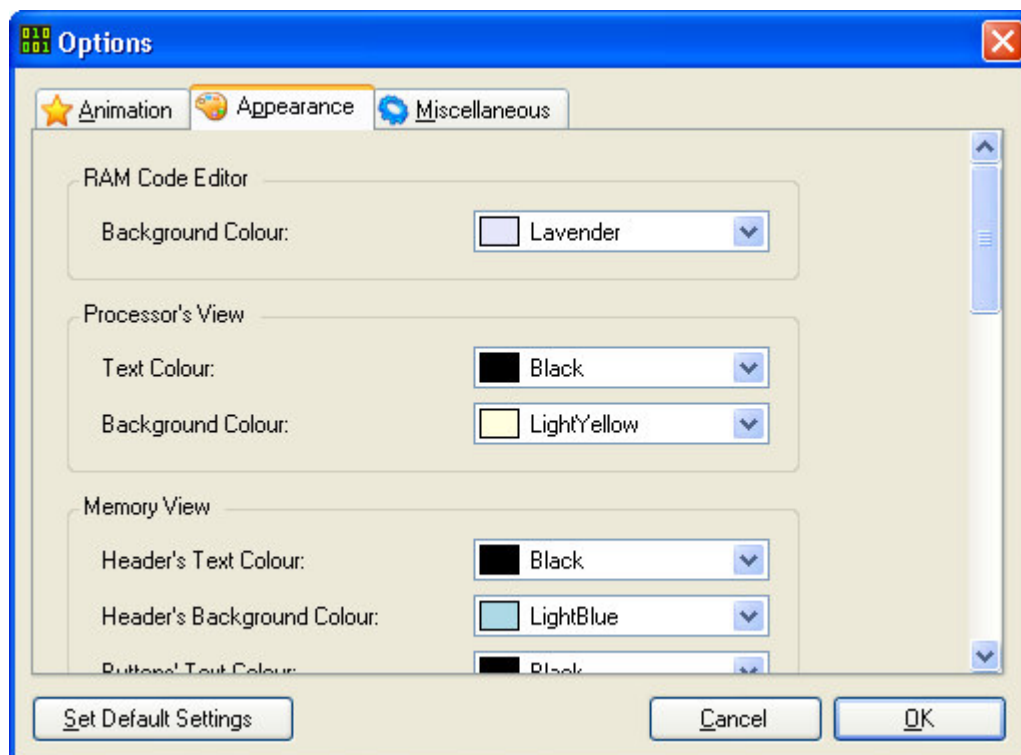
- „*Show Animation of Copying Instruction to Processor*” – ustala czy kopiowanie kolejnej instrukcji programu do procesora ma być poprzedzone odpowiednią animacją;
- „*Show Animation of Instruction Execution*” – ustala czy efekt działania załadowanej instrukcji do procesora ma być wizualizowane za pomocą odpowiedniej animacji;
- „*Animation Speed*” – suwak, za pomocą którego ustalamy szybkość animacji.



Na zakładce „Appearance” znajdują się poszczególne elementy programu, które podlegają zmianie koloru. Dzięki temu użytkownik samodzielnie dostosuje wygląd programu do własnych preferencji. Lista elementów posiadających możliwość zmiany koloru jest następująca:

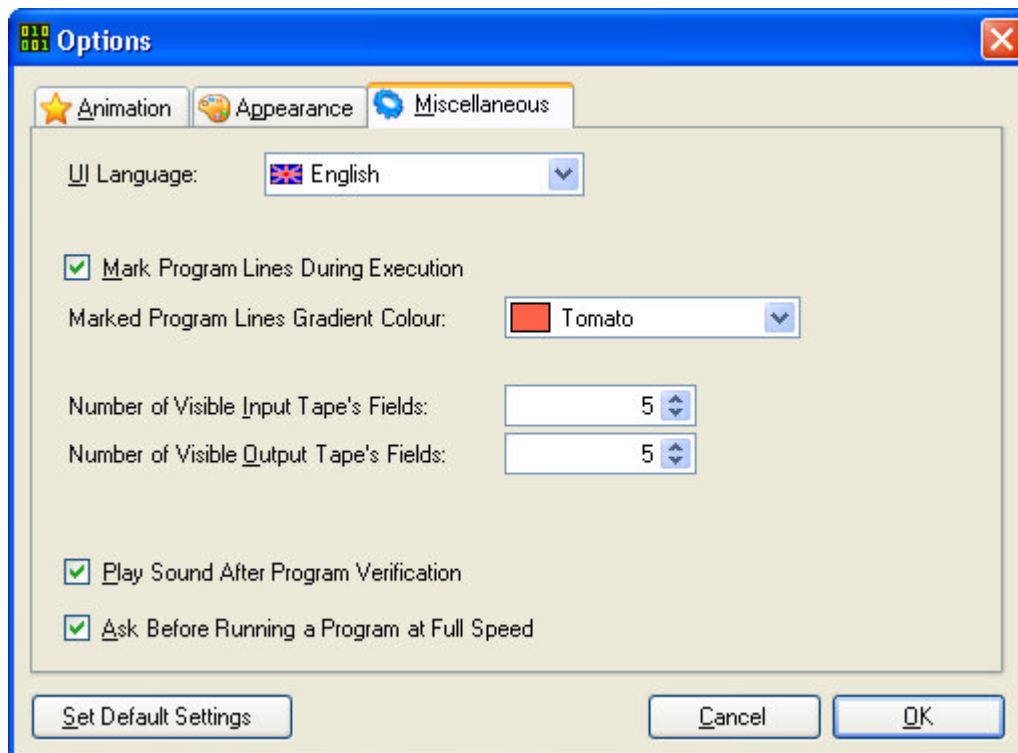
- „*RAM Code Editor*”
 - „*Background Colour*” – tło okna zawierającego wszystkie kontrolki programu jak procesor, pamięć, taśmy i edytor do pisania programów w kodzie RAM;
- „*Procesor's View*” – kontrolka procesora
 - „*Text Colour*” – kolor tekstu w kontrolce procesora
 - „*Background Colour*” - kolor tła w kontrolce procesora

- „*Memory View*” – kontrolka pamięci;
 - „*Header’s Text Colour*” – kolor tekstu w nagłówkach kolumn „Address” i “Value”;
 - „*Header’s Background Colour*” – kolor tła w nagłówkach kolumn „Address” i “Value”;
 - „*Button’s Text Colour*” – kolor tekstu przycisków służących do nawigowania po kontrolce pamięci;
 - „*Button’s Background Colour*” – kolor tła przycisków służących do nawigowania po kontrolce pamięci;
 - „*Accumulator’s Background Colour*” – kolor tła komórki nr 0
 - „*Address’ Text Colour*” – kolor tekstu w komórkach zawierających adres komórki pamięci;
 - „*Gradient Background Colour*” – kolor tła komórek z wartością pamięci używany do cieniowania, począwszy na kolorze białym i skończywszy na tym właśnie kolorze;
- „*Input Tape’s View*” – kontrolka taśmy wejściowej
 - „*Text Colour*” – kolor tekstu z numeracją komórek;
 - „*Background Colour*” – kolor tła z numeracją komórek;
 - „*Buttons’ Colour*” – kolor przycisków służących do nawigowania po taśmie;
- „*Output Tape’s View*” – kontrolka taśmy wyjściowej
 - „*Text Colour*” – kolor tekstu z numeracją komórek;
 - „*Background Colour*” – kolor tła z numeracją komórek;
 - „*Buttons’ Colour*” – kolor przycisków służących do nawigowania po taśmie;
- „*Program Editor*” – kontrolka służąca do pisania program w kodzie RAM
 - „*Header’s Text Colour*” – kolor tekstu w nagłówkach kolumn kontrolki;
 - „*Header’s Background Colour*” – kolor tła w nagłówkach kolumn kontrolki;
- “*Complexity Report*” – kontrolka złożoności programu
 - „*Text Colour*” – kolor tekstu kontrolki;
 - „*Background Colour*” – kolor tła kontrolki;



Ostatnia zakładka, „Miscellaneous”, zawiera pozostałe opcje programu. Oto one:

- „*UI Language*” – określa język interfejsu użytkownika (angielski/polski)
- „*Mark Program Lines During Execution*” – określa czy chcemy, aby podczas wykonywania programu były zaznaczane (odpowiednim kolorem tła) wykonane linie programu, gdzie intensywność koloru będzie oznaczała proporcjonalnie częstość wykonania danej linii kodu w stosunku do ilości całkowitej wykonanych linii programu;
- „*Marked Program Lines Gradient Colour*” – określa maksymalną intensywność koloru, jaką będą oznaczane wykonane linie programu podczas jego wykonywania. Linie programu będą posiadały proporcjonalną do ilości wykonań intensywność pomiędzy kolorem białym, a tym właśnie kolorem;
- „*Number of Visible Input Tape's Fields*” – liczba komórek taśmy wejściowej, jaka ma być jednocześnie widoczna na ekranie;
- „*Number of Visible Output Tape's Fields*” – liczba komórek taśmy wyjściowej, jaka ma być jednocześnie widoczna na ekranie;
- „*Play Sound After Program Verification*” – określa, czy po weryfikacji poprawności programu będzie odegrany odpowiedni dźwięk zależnie od wyniku weryfikacji;
- „*Ask Before Running a Program at Full Speed*” – określa, czy użytkownik ma zostać spytany o potwierdzenie (za pomocą dodatkowego okna dialogowego) tuż przed próbą uruchomienia programu w kodzie RAM przy maksymalnej możliwej prędkości.



Przykładowe programy w kodzie RAM

Ten rozdział ma na celu przekazanie podstaw programowania w kodzie RAM na przykładzie trzech programów: „Dodawanie” (plik ‘Add’), „Silnia” (plik ‘Power’), „Największy wspólny dzielnik” (plik ‘NWD’). Omawiane tu programy są dostępne w katalogu „RAM Machine Examples” znajdującym się na pulpicie, utworzonym podczas instalacji „Maszyny RAM”. Po przeczytaniu tego rozdziału polecam załadowanie za pomocą aplikacji poniższych przykładów i uważnego prześledzenia ich działania, co pomoże w lepszym zrozumieniu ich konstrukcji. Przy omawianiu każdego z programów najpierw będzie przedstawione oczekiwane działanie programu, następnie kod gotowego programu, a na końcu w miarę szczegółowe wyjaśnienie jego działania.

Dodawanie

Zadaniem tego programu będzie wczytanie dwóch liczb z taśmy wejściowej, wyliczenie ich sumy i wypisaniu wyniku na taśmę wyjściową. A zatem pamiętajmy, żeby przed uruchomieniem poniższego programu przygotować dwie liczby w pierwszych dwóch komórkach taśmy wejściowej.

```
1. read 0
2. read 1
3. add 1
4. write 0
```

Jak czytelnik zauważy, program jest dość krótki i prosty. W linii nr 1 wczytujemy do akumulatora (czyli komórki pamięci nr 0) pierwszą liczbę z taśmy wejściowej. W drugiej linii kodu wczytujemy kolejną (drugą) liczbę z taśmy wejściowej i ładujemy ją do

komórki pamięci nr 1. W trzeciej linii następuje główna operacja, którą jest dodanie do zawartości akumulatora zawartość komórki nr 1 i wpisanie wyniku do akumulatora. W ostatniej linii (nr 4) następuje zapisanie wartości akumulatora do pierwszej komórki taśmy wyjściowej.

Silnia

Program „Silnia” będzie nam obliczał silnię dla dowolnie zadanej liczby całkowitej. Ze względu na chęć utrzymania prostoty programu, program nie będzie sprawdzał poprawności danych wejściowych, a mianowicie faktu, czy pierwsza liczba oczekiwana na taśmie wejściowej jest nieujemna. Dopisanie do prezentowanego programu odpowiednich instrukcji, które będą ten fakt sprawdzały, pozostawiam czytelnikowi jako ćwiczenie.

Proszę pamiętać, żeby przed uruchomieniem programu taśma wejściowa zawierała nieujemną liczbę w pierwszej komórce taśmy.

```

1.          READ  1
2.          LOAD  =1
3.          STORE 2
4.          STORE 3
5.      petla: LOAD  1
6.          SUB   2
7.          JZERO koniec
8.          LOAD  2
9.          ADD   =1
10.         STORE 2
11.         LOAD  3
12.         MULT  2
13.         STORE 3
14.         JUMP  petla
15.      koniec: WRITE 3
16.         HALT

```

W linii pierwszej następuje wczytanie do komórki pamięci nr 1 liczby znajdującej się w pierwszej komórce taśmy wejściowej. Następnie (linia nr 2) zostanie do akumulatora załadowana stała = 1. W linii nr 3 i 4 liczba ta zostaje skopiowana do komórki o adresie 2 i 3. Teraz, w linii piątej zaczynamy wykonywać pętlę, w której będzie obliczana silnia (linie 5-14). Pętla będzie wykonywana dokładnie tyle razy, na ile wskazuje wartość liczby wczytanej z taśmy wejściowej (co oznacza, że algorytm jest liniowy w stosunku do danych wejściowych).

Algorytm wykonywany w powyższej pętli jest następujący. Niech wczytana z taśmy wejściowej liczba nazywa się N . W każdym przebiegu pętli komórka nr 2 jest używana jako licznik, który początkowo posiada wartość 1 i w każdym przebiegu pętli zwiększa swoją wartość o 1, aż osiągnie wartość N . Przy okazji każdego przebiegu wartość komórki nr 3 (początkowo równa 1) jest mnożona przez wartość komórki nr 2. Oznacza to, że wartość komórki nr 3 jest zawsze równa (po ukończeniu danego przebiegu pętli) $m(2)!$, gdzie $m(x)$ nazywamy wartością komórki o numerze x . Na początku przebiegu pętli jest sprawdzane każdorazowo (linia 6 i 7), czy wartość komórki nr 2 osiągnęła N , co jest warunkiem stopu algorytmu. Jeśli warunek jest spełniony, przechodzimy do linii nr 15, której zadaniem jest wypisanie na taśmę wyjściową oczekiwanego wyniku. Linia nr 16

informuje maszynę, by zakończyła swoje działanie, choć nie jest ona konieczna (gdy maszyna nie znajduje kolejnej instrukcji do wykonania, automatycznie kończy działanie).

Największy wspólny dzielnik

Celem tego programu jest obliczenie największego wspólnego dzielnika dla dowolnych dwóch liczb naturalnych x i y , czyli takiej maksymalnej liczby naturalnej z , przez którą zarówno x i y dzieli się bez reszty. Najprostszym algorytmem jest przejście zbioru liczb naturalnych $\langle 1, \min(x,y) \rangle$ i wybrania maksymalnej liczby, która spełnia powyższe założenia. Nieco lepszym algorytmem jest algorytm Euklidesa (autorstwa greckiego matematyka - Eudoksosa z Knidos, żyjącego w drugiej połowie IV wieku p.n.e.), na którym bazuje poniższa implementacja. Algorytm bazuje na poniższej obserwacji:

$$NWD(a, b) = \begin{cases} a & \text{dla } b = 0 \\ NWD(b, a \bmod b) & \text{dla } b \geq 1 \end{cases}$$

Implementacja powyższego wzoru w języku C wygląda następująco:

```
int NWD( int x, int y )
{
    int temp;
    while ( y != 0 )
    {
        temp = x % y; // temp = x MODULO y;
        x = y;
        y = temp;
    }
    return x;
}
```

Teraz przedstawimy rozwiązanie w kodzie RAM.

```
1.          read  1
2.          read  2
3.          load  2
4.      PETLA: jzero KONIEC
5.          load  1
6.          div   2
7.          mult  2
8.          sub   1
9.          mult  ==-1
10.         store 3
11.         load  2
12.         store 1
13.         load  3
14.         store 2
15.         jump  PETLA
16.      KONIEC: write 1
```

Ponieważ w kodzie RAM nie dysponujemy operacją *modulo*, została ona zastąpiona w powyższym programie dzieleniem, mnożeniem i odejmowaniem. Wzór jest następujący:

$$a \bmod b = a - \lfloor a / b \rfloor * b.$$

Arytmetyka maszyny RAM zapewnia nam, że podczas dzielenia dwóch liczb całkowitych otrzymamy liczbę całkowitą, która jest wynikiem dzielenia zaokrąglonym „w dół”. Dzięki temu użycie powyższego wzoru w kodzie RAM nie będzie problemem.

Powyższy program prezentuje algorytm Euklidesa. Serce algorytmu mieści się w pętli, w liniach 4-15. Tak jak w przypadku programu w C, w każdym przebiegu pętli obliczane jest *modulo* dwóch liczb, następnie ich zamiana i nadpisanie drugiej z nich wynikiem działania operacji *modulo*. Pętla jest powtarzana do momentu, aż wynik operacji *modulo* osiągnie wynik 0. Wczytując plik NWD dostarczony wraz z instalacją „Maszyny RAM” możemy prześledzić dokładnie działanie tego programu, krok po kroku.

Obsługa programu „Maszyna RAM” w wersji tekstowej

Wraz z instalacją „Maszyny RAM” w wersji okienkowej, została także przygotowana odrębna wersja obsługiwana z linii poleceń. Po przejściu do katalogu, gdzie została zainstalowana aplikacja (standardowo katalog: `C:\Program Files\Lukasz Szkup\RAM Machine\`), odnajdziemy plik `RamMachine.cmd.exe`. Plik ten służy do uruchamiania „Maszyny RAM” w wersji konsolowej, z linii poleceń. Poniżej znajduje się linia poleceń, którą uruchamiamy program:

```
RamMachine.cmd.exe -p PROGRAM_FILE -it INPUT_TAPE_FILE -c SOME_COMMENT -et -ip
```

gdzie:

-p PROGRAM_FILE

ścieżka do pliku z programem w kodzie RAM;

-it INPUT_TAPE_FILE [opcjonalnie]

ścieżka do pliku zawierającego dane dla taśmy wejściowej;

-c SOME_COMMENT [opcjonalnie]

dowolny komentarz, który zostanie umieszczony na początku pliku z raportem złożoności;

-et [opcjonalnie]

podanie tego parametru spowoduje dodanie informacji o rzeczywistym czasie, jaki został zużyty na wykonanie programu; informacja zostanie dodana na końcu pliku z raportem złożoności;

-ip [opcjonalnie]

podanie tego parametru spowoduje umieszczenie (na końcu pliku z raportem złożoności) kodu źródłowego programu, który został uruchomiony.

Należy tutaj wyraźnie zaznaczyć, że podając parametr zawierający znaki spacji, należy taki parametr objąć w znaki cudzysłowia, niezależnie czy jest to ścieżka do pliku, czy komentarz. Przykładowo, zamiast niepoprawnego wywołania:

`-p Mój program.RAMCode -c Jakiś komentarz na temat programu`

powinno być:

`-p „Mój program.RAMCode” -c „Jakiś komentarz na temat programu”.`

Tuż po uruchomieniu programu maszyny następuje faza weryfikacji programu w kodzie RAM. W przypadku niepowodzenia weryfikacji, zostaną wypisane numery linii programu zawierające błędy oraz krótki opis błędu dla każdej z linii. W przeciwnym przypadku następuje faza uruchomienia programu. Jeśli podczas wykonywania programu w kodzie RAM nastąpi próba czytania z taśmy wejściowej, gdy aktualna komórka taśmy jest pusta lub zostanie napotkana instrukcja DIV i dzielną będzie 0, program zakończy wykonywanie oraz powiadomi użytkownika o przyczynie natychmiastowego przerwania pracy (zostanie także podana linia programu, która spowodowała krytyczne zatrzymanie).

Po wykonaniu programu w kodzie RAM zostanie wypisany na standardowe wyjście dokładny rzeczywisty czas jego wykonywania. Dodatkowo, w katalogu z programem w kodzie RAM, który był podany jako parametr w linii poleceń, zostaną utworzone dwa pliki:

PROGRAM_FILE_Output.txt

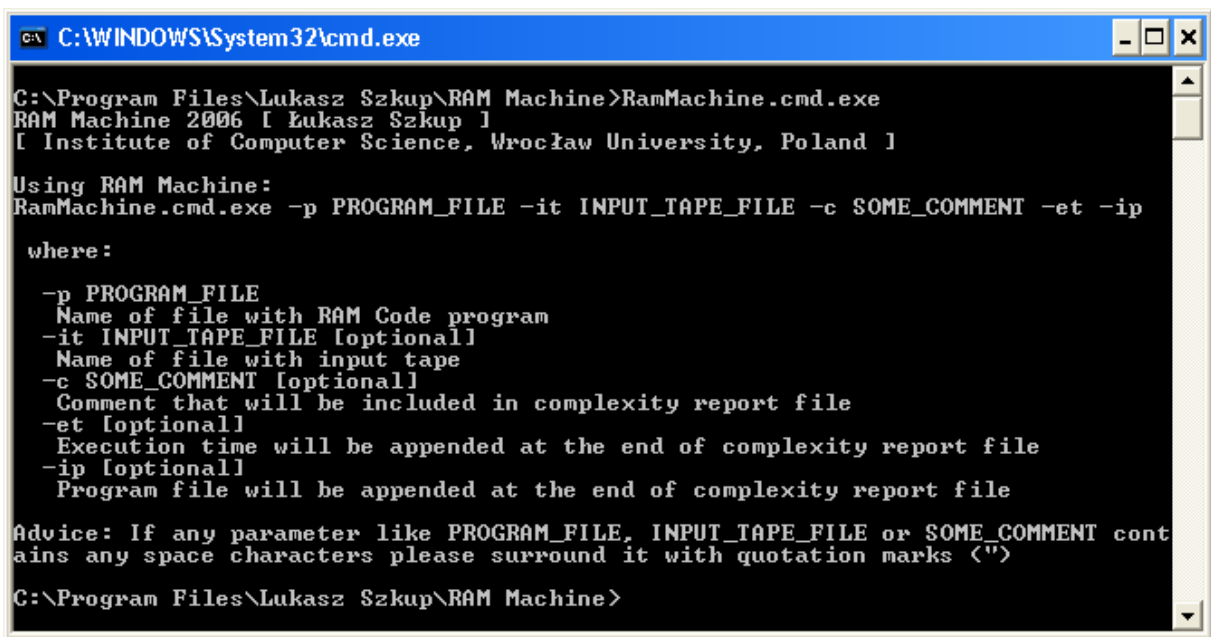
PROGRAM_FILE_Report.txt

gdzie:

PROGRAM_FILE – nazwa pliku programu RAM (lecz bez rozszerzenia), podana jako parametr przy uruchamianiu maszyny.

Pierwszy plik to zapis taśmy wyjściowej. Drugi plik to raport złożoności programu (wraz z zawartym komentarzem, czasem wykonywania oraz programem RAM, gdy zostały podane odpowiednie parametry podczas uruchamiania z linii poleceń). Wszystkie trzy pliki, jakie program wczytuje/zapisuje mają formaty identyczne jak podczas eksportu/importu, opisane w jednym z rozdziałów powyżej. Dzięki temu możemy napisać program w kodzie RAM w okienkowej wersji „Maszyny RAM” (oraz przygotować i wyeksportować taśmę wejściową), a następnie uruchomić program z poziomu konsolowej wersji maszyny.

Przygotowując ręcznie plik z programem RAM, należy pamiętać, że po etykiecie zawsze powinien się znaleźć znak dwukropka. Jeśli jednak etykieta jest argumentem instrukcji skoku, wtedy podajemy samą nazwę etykiety. Dodatkowo, w każdej linii programu możemy wstawić komentarz. Znakiem specjalnym oznaczającym początek linii z komentarzem jest znak '#'. Tekst zawarty pomiędzy tym znakiem, a znakiem końca linii jest zawsze traktowany jako komentarz i nie ma wpływu na kompilację i działanie programu. Po pozostałe informacje na temat formatu kodu RAM odsyłam do jednego z początkowych rozdziałów temu poświęconych. Program „Maszyna RAM” (wersja okienkowa i konsolowa) podczas odczytu i zapisu pliku z programem RAM, zawsze stosuje w/w format.



```
C:\WINDOWS\System32\cmd.exe

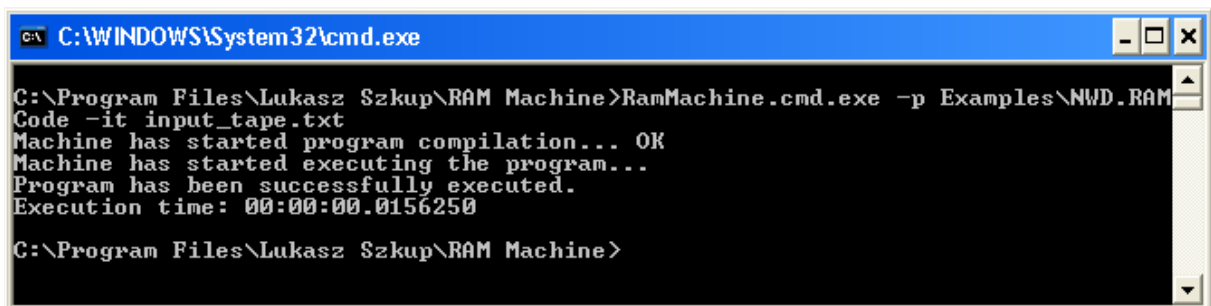
C:\Program Files\Lukasz Szkup\RAM Machine>RamMachine.cmd.exe
RAM Machine 2006 [ Lukasz Szkup ]
[ Institute of Computer Science, Wrocław University, Poland ]

Using RAM Machine:
RamMachine.cmd.exe -p PROGRAM_FILE -it INPUT_TAPE_FILE -c SOME_COMMENT -et -ip
where:

-p PROGRAM_FILE
  Name of file with RAM Code program
-it INPUT_TAPE_FILE [optional]
  Name of file with input tape
-c SOME_COMMENT [optional]
  Comment that will be included in complexity report file
-et [optional]
  Execution time will be appended at the end of complexity report file
-ip [optional]
  Program file will be appended at the end of complexity report file

Advice: If any parameter like PROGRAM_FILE, INPUT_TAPE_FILE or SOME_COMMENT contains any space characters please surround it with quotation marks ">
```

Rysunek 23 – uruchomienie bez parametrów

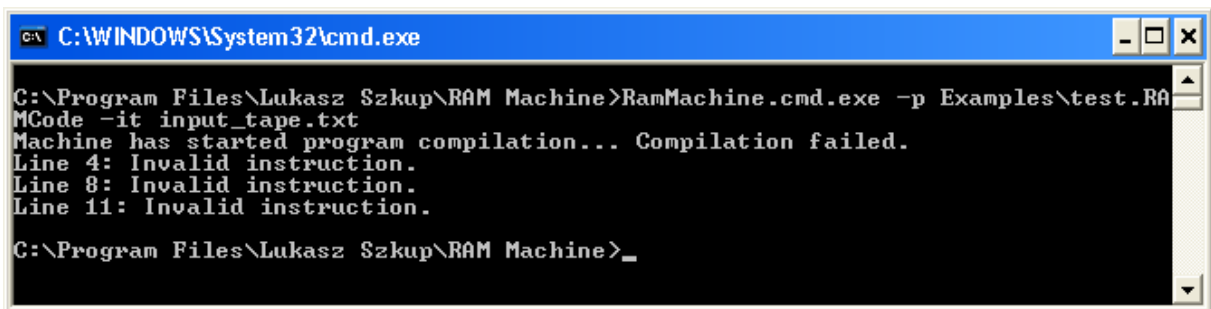


```
C:\WINDOWS\System32\cmd.exe

C:\Program Files\Lukasz Szkup\RAM Machine>RamMachine.cmd.exe -p Examples\NWD.RAM
Code -it input_tape.txt
Machine has started program compilation... OK
Machine has started executing the program...
Program has been successfully executed.
Execution time: 00:00:00.0156250

C:\Program Files\Lukasz Szkup\RAM Machine>
```

Rysunek 24 – przykład uruchomienia



```
C:\WINDOWS\System32\cmd.exe

C:\Program Files\Lukasz Szkup\RAM Machine>RamMachine.cmd.exe -p Examples\test.RA
MCode -it input_tape.txt
Machine has started program compilation... Compilation failed.
Line 4: Invalid instruction.
Line 8: Invalid instruction.
Line 11: Invalid instruction.

C:\Program Files\Lukasz Szkup\RAM Machine>_
```

Rysunek 25 – przykład próby uruchomienia błędnego programu

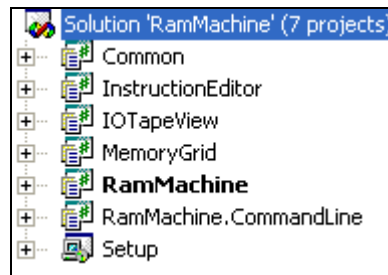
Implementacja i informacje dla kontynuatorów programu

Program „Maszyna RAM” został w całości napisany w C# z użyciem framework .NET w wersji 1.1. Łącznie posiada ok. 22.000 linii kodu. Przy implementacji aplikacji została użyta biblioteka NetAdvantage 2005 vol.1 firmy Infragistics (<http://www.infragistics.com/>). Z biblioteki tej zostały użyte m.in. następujące kontrolki:

- kontrolka tabeli (UltraWinGrid) – użyta przy tworzeniu kontrolki edytora kodu RAM,
- kontrolka menu (UltraToolbarManager),
- mechanizm do zarządzania wieloma dokumentami (UltraWinTabbedMdi),

- kontrolka umożliwiająca dokowanie okna (UltraWinDock) – użyta przy tworzeniu kontrolki wyświetlającej raport złożoności,
- kontrolka drukująca i wyświetlająca podgląd wydruku tabeli (UltraWinPrintPreviewDialog) – użyta przy wszelkich wydrukach

Projekt programu został podzielony na 7 modułów, przy czym ostatni z nich jest projektem odpowiedzialnym za tworzenie instalacji aplikacji i nie zawiera kodu źródłowego.



Rysunek 26 - podział projektu na moduły

Poniżej przedstawię kolejne moduły, na które składa się cała aplikacja, wraz z wprowadzającym opis modułu.

Common – moduł, który zawiera wszelkie struktury danych (między innymi arytmetykę dużych liczb całkowitych) oraz mechanizmy do parsowania, weryfikowania i uruchamiania programów napisanych w kodzie RAM. Dodatkowo zawiera mnóstwo przydatnych klas, m.in. do eksportu/importu danych (taśma wejścia/wyjścia, pamięć, raport złożoności), zarządzania zasobami programu (wielojęzyczne tablice stringów, ikonki, dźwięki) oraz do zarządzania dokumentami XML. Moduł jest niezależny od pozostałych modułów, dzięki czemu można go użyć w innym projekcie, np.: przy tworzeniu nowej wersji Maszyny RAM, zawierającej zupełnie inny interfejs użytkownika lub przy tworzeniu programu, który skorzysta z implementacji arytmetyki dużych liczb.

InstructionEditor – kontrolka edytora do pisania programów w kodzie RAM. Zawiera kompletną obsługę interfejsu użytkownika i jako moduł niezależny (wymaga jedynie powyższej biblioteki *Common*), może zostać użyty bezproblemowo w dowolnym innym projekcie.

IOTapeView – kontrolka taśmy wejściowej i wyjściowej. Także, poza wymaganiem biblioteki *Common*, jest zupełnie niezależna. Posiada kompletną obsługę interfejsu użytkownika i jest gotowa do użycia w innym projekcie.

MemoryGrid – kontrolka pamięci maszyny RAM. Również do swojej niezależności wymaga jedynie biblioteki *Common* i po spełnieniu tego warunku może być wcielona do dowolnego innego projektu.

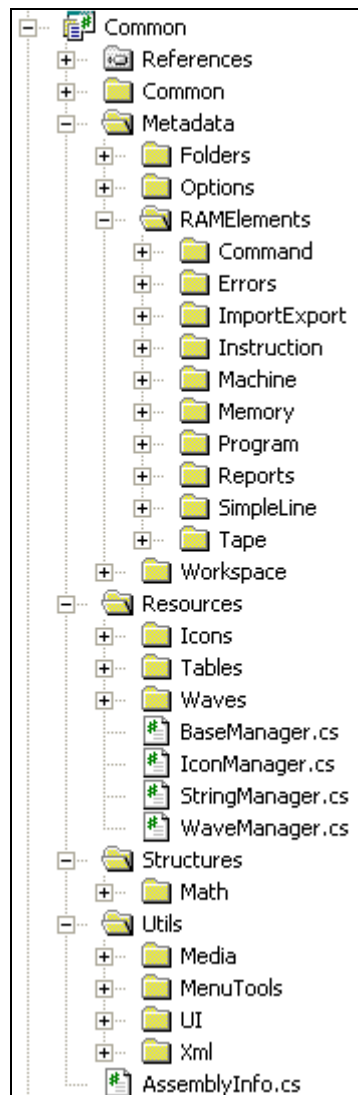
RamMachine – moduł uruchamialny, który spina w całość powyższe moduły, które tworzą ostatecznie aplikację – „Maszynę RAM”. Zawiera on także klasy odpowiedzialne za animację.

RamMachine.CommandLine – wersja „Maszyny RAM” uruchamiana z linii poleceń. Jedyne moduły, jakie używa do swojego działania to moduł *Common*, który zawiera komplet klas potrzebnych do weryfikacji i uruchamiania programów napisanych w kodzie RAM.

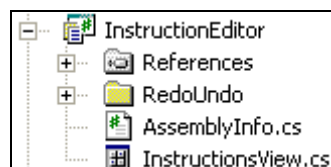
Jest to przy okazji bardzo dobry przykład obrazujący jak łatwo można stworzyć nowy program implementujący interfejs dla maszyny RAM, gdy użyjemy biblioteki *Common*, która posiada wszystkie gotowe mechanizmy z tym związane. Moduł ten składa się na zaledwie 200 linii kodu, bo tylko tyle wystarczy, by używając biblioteki *Common* napisać najprostszy, działający interfejs dla maszyny RAM.

Maszyna Ram została napisana w oparciu o wzorzec projektowy Dokument – Widok. Wszelkie kontrolki, m.in. takie jak InstructionEditor, IOTapeView i MemoryGrid służą jedynie do wyświetlania danych znajdujących się w dostarczonym dokumencie. Klasy opakowujące dokumenty używane w aplikacji znajdują się w module *Common*. Taki podział na warstwy zapewnia nam przede wszystkim niezależność interfejsu użytkownika (a więc przede wszystkim graficzny sposób prezentowania danych) od danych aplikacji. Wniosek jaki płynie z zastosowania takiego podziału jest taki, jak już wcześniej wspominałem, że w każdym momencie możemy „wymienić” interfejs użytkownika na zupełnie inny, nie musząc od nowa implementować lub poprawiać warstwy zawierającej dokumenty (dane programu).

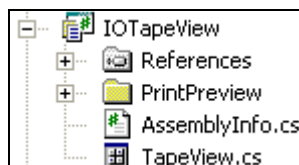
Projekt maszyny RAM został napisany na tyle elastycznie, że dość łatwo jest wdrożyć do programu kolejne języki programowania. Można przykładowo zaprogramować nowy język, oparty na składni języka C, który podczas kompilacji będzie konwertowany do kodu RAM. W bibliotece *Common* proponowałbym wówczas zaimplementowanie leksera i parsera, a w projekcie RamMachine napisanie nowej klasy okna, bazującej na klasie RAMEditorBase (z której dziedziczy klasa RAMCodeEditor, odpowiedzialna za wyświetlanie edytora zawierającego kontrolkę procesora, pamięci, taśm wejścia/wyjścia oraz edytora kodu RAM). Gdyby czytelnik postanowił na poważnie wdrożyć kolejny (lub kolejne) język programowania do niniejszej aplikacji, autor chętnie udzieli pomocnych wskazówek.



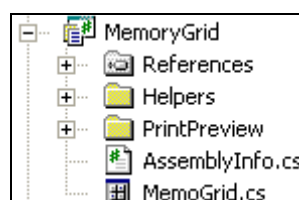
Rysunek 27 - moduł Common – ogólny widok



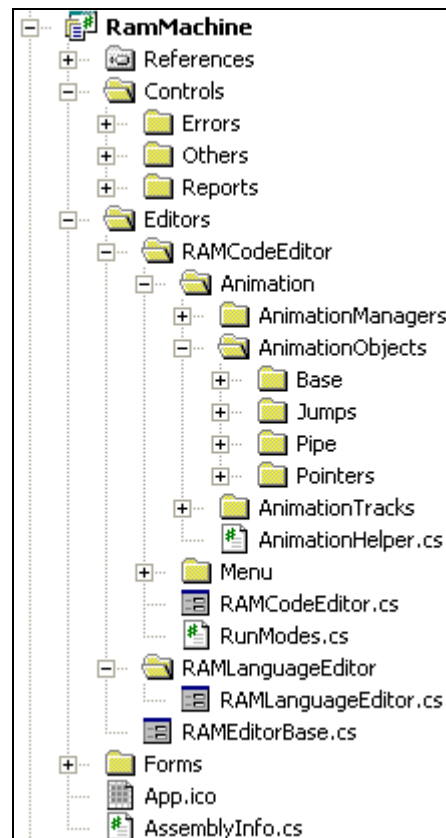
Rysunek 28 - moduł InstructionEditor



Rysunek 29 - moduł IOTapeView



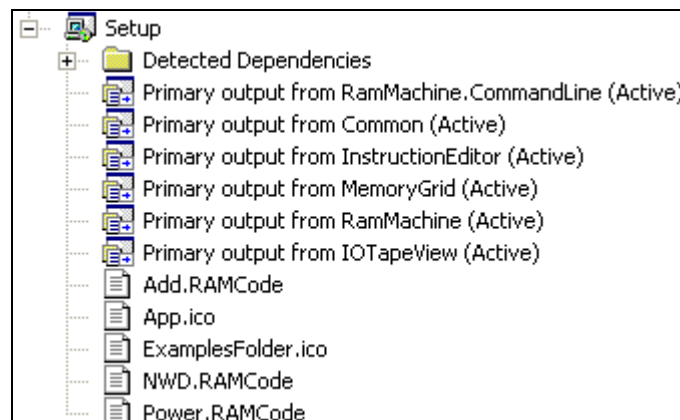
Rysunek 30 - moduł MemoryGrid



Rysunek 31 - moduł RamMachine



Rysunek 32 - moduł RamMachine.CommandLine



Rysunek 33 - moduł Setup

Podsumowanie

Program „Maszyna RAM” został napisany głównie z myślą o zastosowaniach edukacyjnych. Program może stać się jedną z pozycji w programie nauczania w szkołach średnich i wyższych jako narzędzie wprowadzające do świata programowania. Osoby, które już znają jakikolwiek wysokopoziomowy język programowania (np.: Java, C++, C#) mają okazję poznać kod RAM, który najbardziej przypomina języki typu assembler. Assembler jest jednym z najniższych w hierarchii językiem (choćby z tego powodu, że jego instrukcje są bezpośrednio wykonywane przez procesor, bez jakiegokolwiek konwersji), na którym bazują pozostałe języki i bez którego by nie istniały. Dlatego też, ucząc się kodu RAM, posiadamy dobry fundament do nauki assemblera, który przede wszystkim różni się (od kodu RAM) jedynie większą ilością instrukcji.

Wszystkie wysokopoziomowe języki programowania są tłumaczone, na etapie kompilacji, do języka niskopoziomowego. Kod RAM jest już sam w sobie językiem niskopoziomowym, co zmusza użytkownika do kreatywnego myślenia, ponieważ aby dany algorytm przenieść do kodu RAM, należy włożyć w to nieco więcej wysiłku niż w zaprogramowanie go w języku C++ lub Java. Musimy przeskoczyć wiele warstw na raz, zaczynając myśleć o jakimś algorytmie w języku naturalnym (zrozumiałym jedynie dla człowieka), by zakodować go w kodzie RAM skończywszy. Dlatego też programowanie w kodzie RAM może być nie tylko niezłym wyzwaniem intelektualnym, ale także dobrą rozrywką – odskoczną od aktualnie panujących języków programowania. Po dłuższym okresie obcowania z kodem RAM następuje etap, że zaprogramowanie dowolnego algorytmu staje się na tyle płynne i naturalne, jakby użytkownik... myślał w kodzie RAM(!).

Autor dołożył największych starań, by praca z „Maszyną RAM” była możliwie przyjemna i bezproblemowa. Każdy z elementów aplikacji był wielokrotnie testowany, co pozwala mieć nadzieję, że ostateczny produkt stoi na możliwie najwyższym poziomie.

Dodatek - Maszyny Turinga

Deterministyczna maszyna Turinga (DTM)

Formalnie deterministyczna maszyna Turinga jest zbiorem $\{Q, \Sigma, \delta, q_0, F\}$, gdzie:

Q - zbiór stanów sterowania maszyny;

Σ - alfabet (zbiór symboli) taśmy;

δ – funkcja przejścia, gdzie $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{R, L, N\}$;

q_0 - początkowy stan sterowania, $q_0 \in Q$;

F - zbiór końcowych stanów sterowania, $F \subseteq Q$.

Posługując się językiem opisowym, DTM jest automatem, który może być rozumiany jako dowolne urządzenie przetwarzające dane. Urządzenie to składa się z elementu sterującego, głowicy oraz taśmy, z której automat będzie zarówno odczytywał, jak i zapisywał dane (symbole z alfabetu Σ). W każdym kroku automat odczytuje z taśmy jeden symbol, nad którym jest ustawiony aktualnie czytnik, następnie, posługując się funkcją przejścia, określa, na podstawie odczytanego symbolu i aktualnego stanu, kolejny, nowy stan maszyny. Następnie zapisuje symbol na taśmie, oraz przesuwa głowicę czytnika w lewo (L), w prawo (R) lub pozostawia ją w niezmienionej pozycji (N). Maszyna na samym początku znajduje się w stanie q_0 , a jak tylko znajdzie się w jednym ze stanów ze zbioru F , kończy swoje działanie. Może się także zdarzyć taki przypadek, że maszyna będzie w nieskończoność lawirować pomiędzy stanami, które nie należą do F , co będziemy nazywać zapętleniem się maszyny.

Funkcję przejścia możemy też nazywać programem, jaki wykonuje maszyna, która jest rzeczywiście sekwencją decyzji, jakie ma podejmować maszyna w każdym kroku, co ostatecznie prowadzi do zmiany stanu maszyny po uruchomieniu każdej instrukcji.

Niedeterministyczna maszyna Turinga (NDTM)

Niedeterministyczna wersja maszyny Turinga jest, poza jednym szczegółem, identyczna z deterministyczną maszyną Turinga. Jedyna różnica polega na tym, że funkcją przejścia może spowodować przejście maszyny w więcej niż jeden kolejny stan. Mówiąc bardziej kolokwialnie, NDTM „zgaduje” swoje kolejne przejście w następny stan, na podstawie aktualnego stanu i przeczytanego z taśmy symbolu.

Innym sposobem na przybliżenie sobie działania niedeterministycznej maszyny, jest wyobrażenie kilku deterministycznych maszyn uruchomionych jednocześnie. Początkowo każda znajduje się w stanie q_0 , jednak w kolejnym kroku każda z nich przechodzi w innych stan, który opisuje nasza „magiczna” funkcja przejścia. Po kolejnym kroku odpowiednie maszyny zachowują się ponownie w ten sam sposób, wyznaczając jednocześnie pewne drzewo obliczeń. NDTM jest na tyle „mądra”, że wybiera tylko jedną ścieżkę w takim drzewie (którego wierzchołkami są stany ze zbioru Q), by dotrzeć ostatecznie do jednego ze stanów F . Wynika stąd także, że działanie NDTM możemy zasymulować za pomocą DTM, przeprowadzając mozolne sprawdzenie wszystkich ścieżek obliczeń w w/w drzewie, by sprawdzić, która (lub: które) ostatecznie biegnie do osiągnięcia stanu ze zbioru F .

Klasy P i NP

Aby przeprowadzić obliczenia dla pewnego problemu potrzebujemy ustalić algorytm. Algorytm jest pewnym przepisem obliczania. Inaczej mówiąc, algorytm jest sekwencją instrukcji, które ma wykonać maszyna, aby obliczyć rozwiązanie dla pewnych danych wejściowych. Dla danego problemu może być wiele różnych algorytmów, różniących się przede wszystkim czasem działania i ilością wykorzystanej pamięci (w naszym przypadku: ilością komórek taśmy maszyny). Z pewnością chcielibyśmy wybrać najlepszy algorytm dla każdego problemu. Okazuje się, że istnieją problemy, dla których nie ma dobrego, szybkiego algorytmu. Dla lepszego rozróżnienia skali trudności problemów powstały definicje klas P i NP .

Zbiór problemów należących do klasy P , określa się mówiąc, że istnieje wielomianowy algorytm dla tych problemów. Oznacza to, że istnieje taki ciąg instrukcji dla maszyny DTM, że dla danego problemu wykona ona wielomianową ilość zmian stanów, zanim dotrze stanu ze zbioru F , a na taśmie znajdzie się zapisany wynik obliczeń.

Natomiast zbiór problemów należących do klasy NP określa się mianem problemów trudnych, ponieważ nie istnieje dla nich algorytm wielomianowy, który jest w stanie obliczyć dla pewnych danych rozwiązanie. Definiuje się jednakże, że problemy z tej klasy są obliczalne w czasie wielomianowym przez NDTM. Otóż niedeterministyczna maszyna Turinga najpierw „zgaduje” rozwiązanie problemu, a następnie w czasie wielomianowych sprawdza, że rozwiązanie jest prawidłowe.

Do klasycznych przykładów problemów należących do klasy NP , są: problem komiwojażera, problem plecakowy czy problem SAT. Nie istnieje dla nich algorytm wielomianowy. Pomimo klasyfikacji problemów na P i NP nikt dotychczas nie udowodnił, że $P = NP$ lub, że $P \neq NP$. Jednakże przyjmuje się na razie, że $P \neq NP$. Gdyby ktoś natomiast dla jakiegokolwiek problemu z klasy NP przedstawił wielomianowy algorytm, wówczas $P = NP$. Wynika to z faktu, że pomiędzy problemami tej samej klasy możemy w czasie wielomianowych dokonać redukcji, transformując dane jednego problemu w dane drugiego problemu, następnie uruchamiając algorytm (dla drugiego problemu), a wyniki z powrotem przełożyć na wynik odpowiedni dla pierwszego problemu. Więc umiemy rozwiązać choćby jeden problem z klasy NP w czasie wielomianowym, potrafilibyśmy rozwiązać pozostałe problemu z tej klasy także w tym czasie. Byłoby to oczywiście wielkim zagrożeniem choćby dla takiej dziedziny informatyki, jaką jest kryptografia, ponieważ wiele metod szyfrowania (przykładowo: RSA stosowana w kluczach PGP), opiera się na rozkładzie długich liczb na czynniki pierwsze, co jest problemem trudnym. Jednak na razie możemy spać spokojnie, ponieważ nikt (pomimo wielu dziesiątek lat poszukiwań) nie podał żadnego algorytmu, który by rozkładał liczby w czasie wielomianowym i wiele wskazuje na to, że nigdy do takiego momentu nie dojdzie.

Bibliografia

1. „*Projektowanie i analiza algorytmów*” – Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ulman, wydawnictwo Helion 2003
2. „*Wprowadzenie do algorytmów*” – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, wydawnictwo WNT, 1997, 1998
3. http://pl.wikipedia.org/wiki/Maszyna_RAM
4. http://pl.wikipedia.org/wiki/Maszyna_Turinga