

# Wstęp do programowania, potok imperatywny (Info, I rok) 16/17, laboratorium

Moja strona domowa ► Informatyka ► Informatyczne studia I stopnia ► I rok ► WPI.LAB.INFO.I.16/17 ► Zadanie 2 ► Zadanie 2: MIK



## NAWIGACJA

Moja strona domowa

■ Strona główna

Strony

Bieżący

przedmiot

WPI.LAB.INFO

.I.16/17

Uczestnicy

Odznaki

Główne


składowe


Zadanie 0

(treningowe  
)

Zadanie 1

Zadanie 2

 **Zadanie  
2: MIK**

 Zadanie  
2:

poprawn  
ość

Zadanie 3

Zadanie 4  
(poprawkow  
e)

Moje kursy



## ADMINISTRACJA

Administracja

1 z 13 kursem

## Zadanie 2: MIK

### Wprowadzenie

Realizacja podzbioru języka C na maszynie wirtualnej (abstrakcyjny komputer) o nazwie MIK, składa się z

- kompilatora `mikc`, tłumaczącego program w C na kod maszyny MIK w reprezentacji symbolicznej,
- asemblera `mika`, przekształcającego kod MIK z reprezentacji symbolicznej na wykonywalną,
- interpretera `miki`, wykonującego program.

Program, którego kod źródłowy jest w pliku `program.c`, możemy uruchomić poleceniem

```
( < program.c mikc | mika ; cat ) | miki
```

Skompiluje ono program do kodu wykonywalnego, połączy go z danymi z wejścia i przekaże do wykonania interpreterowi.

W tym zadaniu rozważamy implementację interpretera `miki`.

### Deksarny zapis liczb

Deksarny (ang. *dexary*) zapis nieujemnej liczby całkowitej jest niepustym ciągiem cyfr dziesiętnych oraz liter. Tam, gdzie konieczne jest wyróżnienie zapisu liczb, możemy wymagać, by pierwszym jego znakiem była cyfra dziesiętna lub zabronić stosowania w nim małych liter. Poniżej przyjmujemy to drugie ograniczenie.

Cyfry dziesiętne i litery są cyframi deksarnego zapisu liczby. Każda z nich ma dwa atrybuty - bazę i wartość:

- cyfry dziesiętne od `'0'` do `'9'` mają bazę 10 i wartości od 0 do 9,
- cyfry od `'A'` do `'P'`, nazywane szesnastkowymi, mają bazę 16 i wartości od 0 do 15,

14.01.2017 02:24

- cyfry od '0' do '7', nazywane ósemkowymi, mają bazę 8 i wartości od 0 do 7,
- cyfry 'Y' i 'Z', nazywane binarnymi, mają bazę 2 i wartości, odpowiednio, 1 oraz 0.

Wartość liczby zapisanej jedną cyfrą jest równa wartości tej cyfry. Ciąg cyfr powstały przez dopisanie na koniec niepustego ciągu cyfr  $s$  cyfry  $c$  reprezentuje liczbę o wartości  $xb + w$ , gdzie  $x$  to wartość liczby reprezentowanej przez ciąg cyfr  $s$ ,  $b$  to baza cyfry  $c$  a  $w$  to jej wartość.

Przykładowo, każdy z poniższych ciągów cyfr jest deksarnym zapisem liczby o wartości tysięcy

```
1000
DOI
RXVQ
YYYYYZZZZ
01000
0DOI
0RXVQ
0YYYYYZZZZ
PYZYQ
K00
ZWE0
1PZI
```

## Maszyna wirtualna

Maszyna MIK ma tzw. *architekturę harwardzką*. Inaczej niż w przypadku maszyn o *architekturze von Neumanna*, które mają wspólną pamięć dla danych i instrukcji, MIK ma oddzielne pamięci dla instrukcji programu oraz dla jego danych.

MIK jest maszyną typu *load/store*. Większość operacji na danych wymaga wczytania ich z pamięci do znajdujących się poza pamięcią *rejestrów*.

*Rozmiarem* maszyny MIK jest nieujemna parzysta liczba całkowita  $n$ . Maszyna MIK rozmiaru  $n$  ma  $2^n$  komórek pamięci danych, pamięć programu mieszczącą  $2^n$  instrukcji oraz  $2^{n/2}$  rejestrów. Dane w pamięci i w rejestrach, nazywane *słowami*, są nieujemnymi liczbami całkowitymi od 0 do  $2^n - 1$ , zapisanymi na  $n$  bitach.

Pamięci danych i instrukcji programu są indeksowane słowami. Każdy element tych pamięci jest jednoznacznie identyfikowany przez *adres*, będący liczbą całkowitą od 0 do  $2^n - 1$ . Przyjmujemy też, że rejestry są numerowane liczbami całkowitymi od 0 do  $2^{n/2} - 1$ .

Maszyna MIK rozmiaru 8, której implementacja jest naszym celem w tym zadaniu, ma 16 rejestrów o numerach od 0 do 15, 256 komórek pamięci danych oraz pamięć na 256 instrukcji programu. Adresy oraz dane w rejestrach i komórkach pamięci danych są liczbami całkowitymi od 0 do 255.

MIK jest maszyną *sekwencyjną*. Interpreter jej kodu wykonuje na raz jedną instrukcję. Adres kolejnej instrukcji do wykonania jest nazywany *licznikiem rozkazów* (ang. *program counter*).

Interpreter działa w potencjalnie nieskończonej petli. Po wykonaniu aktualnej instrukcji przechodzi do następnej chyba, że wykonał instrukcję przerywającą wykonanie programu.

Poniższa tabela przedstawia instrukcje maszyny MIK:

Symbol	Kod	Warunek	Zapis	Nazwa	Znaczenie
A	0	b != c	div a b c	divide	<pre>trc = reg[c]; if (trc != 0) {     trb = reg[b];     reg[a] = trb / trc;     reg_[a] = trb % trc; }</pre>
B	1	a != b && a != c	ret a b c	return	<pre>tpc = pc; pc = mem[reg[ a]]; reg[a] += reg [c] + 1; reg[b] = tpc;</pre>
C	2	b != c	cmp a b c	compare	<pre>reg[a] = (reg [b] &lt; reg[c]) ? 1 : 0;</pre>
D	3	b != c	sub a b c	subtract	<pre>reg[a] = reg[ b] - reg[c];</pre>
E	4	b <= c	ldi a b c	load indexed	<pre>reg[a] = mem[ reg[b] + reg[ c]];</pre>
F	5	b <= c	sti a b c	store indexed	<pre>mem[reg[b] + reg[c]] = reg [a];</pre>
G	6	b <= c	mul a b c	multiply	<pre>(reg_[a] : re g[a]) = reg[b ] * reg[c];</pre>

Symbol	Kod	Warunek	Zapis	Nazwa	Znaczenie
H	7	b <= c	cli a b c	call indexed	<pre>tpc = pc; pc = mem[reg[ b] + reg[c]]; reg[a] = tpc;</pre>
I	8		jze a bc	jump if zero	<pre>if (reg[a] == 0) {     pc = bc; }</pre>
J	9		jnz a bc	jump if not zero	<pre>if (reg[a] != 0) {     pc = bc; }</pre>
K	10		cls a bc	call subroutine	<pre>reg[a] = pc; pc = bc;</pre>
L	11		cal a bc	call	<pre>mem[--reg[a]] = pc; pc = bc;</pre>
M	12		ldr a bc	load register	<pre>reg[a] = mem[ bc];</pre>
N	13		str a bc	store register	<pre>mem[bc] = reg [a];</pre>
O	14		ldc a bc	load constant	<pre>reg[a] = bc;</pre>

Symbol	Kod	Warunek	Zapis	Nazwa	Znaczenie
P	15		sys a bc	system call	<pre>switch (bc) { case CORE_DUM P: /* 0 */     core_dump (reg[a]);     break; case GET_INT:     /* 1 */     if (scanf ("%d", &amp;tra) != 1) {         reg_[ a] = 0;     } else {         reg_[ a] = 1;         reg[a ] = tra;     }     break; case PUT_INT:     /* 2 */     printf("% d", reg[a]);     break; case GET_CHAR : /* 3 */     tra = get char();     if (tra = = EOF) {         reg_[ a] = 0;     } else {         reg_[ a] = 1;         reg[a ] = tra;     }     break; case PUT_CHAR : /* 4 */     putchar(r eg[a]);     break; case PUT_STRI NG: /* 5 */     printf("% s", reg[a]); }</pre>

Symbol	Kod	Warunek	Zapis	Nazwa	Znaczenie
					<pre>break; default:     /* nic nie rób */     break; }</pre>
Q	0	<code>a != b &amp;&amp; b == c</code>	<code>psh a b</code>	<i>push</i>	<code>mem[--reg[a]] = reg[b];</code>
R	1	<code>a != b &amp;&amp; a == c</code>	<code>pop a b</code>	<i>pop</i>	<code>reg[b] = mem[reg[a]++];</code>
S	2	<code>b == c</code>	<code>shl a b</code>	<i>shift left</i>	<code>reg[a] = reg[b] &lt;&lt; 1;</code>
T	3	<code>b == c</code>	<code>shr a b</code>	<i>shift right</i>	<code>reg[a] = reg[b] &gt;&gt; 1;</code>
U	4	<code>b &gt; c</code>	<code>add a b c</code>	<i>add</i>	<code>reg[a] = reg[b] + reg[c];</code>
V	5	<code>b &gt; c</code>	<code>orr a b c</code>	<i>bitwise or</i>	<code>reg[a] = reg[b]   reg[c];</code>
W	6	<code>b &gt; c</code>	<code>and a b c</code>	<i>bitwise and</i>	<code>reg[a] = reg[b] &amp; reg[c];</code>
X	7	<code>b &gt; c</code>	<code>xor a b c</code>	<i>bitwise exclusive or</i>	<code>reg[a] = reg[b] ^ reg[c];</code>
Y	1	<code>a == b</code>	<code>rts a c</code>	<i>return from subroutine</i>	<pre>tpc = pc; pc = reg[c]; reg[a] = tpc;</pre>
Z	0	<code>a == b &amp;&amp; b == c</code>	<code>hlt a</code>	<i>halt</i>	<code>exit(reg[a]);</code>

Opis tabeli:

- W kolumnie **Symbol** jest symboliczna, jednoliterowa nazwa instrukcji.
- Każda instrukcja składa się z czterech elementów: **Kodu** operacji oraz trzech *pól* argumentów nazywanych `a`, `b` i `c`.

- W kolejnych kolumnach jest **Zapis** symbolicznej instrukcji w assemblerze oraz jej angielska **Nazwa**.
- Ostatnia kolumna podaje w *pseudokodzie* **Znaczenie** instrukcji.

W pseudokodzie instrukcji:

- `reg` i `mem` to, przedstawione w formie tablic, rejestry i pamięć danych.
- `pc` to licznik rozkazów.
- `tpc`, `tra`, `trb` i `trc` są zmiennymi pomocniczymi.
- `bc` jest "sklejeniem" wartości pól `b` oraz `c` w jedno słowo. Jeśli `n` jest rozmiarem maszyny, to sklejenie `b` i `c` można obliczyć jako  $(b \ll (n / 2)) \mid c$ . Na Mik 8 jest ono równe  $b * 16 + c$ .
- Zapis `reg[a]` oznacza rejestr następny za rejestrem o numerze `a`. Jeśli `reg[a]` jest rejestrem o najwyższym numerze, to następnym za nim jest `reg[0]`, w przeciwnym przypadku jest nim rejestr o numerze o 1 większym od `a`.
- Instrukcja `mul` mnoży dwa słowa. Na maszynie rozmiaru `n` wynik `x` jest wartością z przedziału od 0 do  $2^{2n} - 1$ . Instrukcja zapisuje bardziej znaczącą jego część, równą  $x \gg n$ , w rejestrze `reg[a]` a część mniej znaczącą, równą  $x \& ((1 \ll n) - 1)$ , w rejestrze `reg[a]`. Na Mik 8 będą to, odpowiednio,  $x / 256$  i  $x \% 256$ .
- Z wyjątkiem instrukcji `mul`, wszystkie operacje dają na maszynie rozmiaru `n` wynik liczony modulo  $2^n$ , gdzie przez `a` modulo `b`, dla dodatniego `b`, z pomocniczą wartością `c = a % b`, rozumiemy  $(c < 0) ? c + b : c$ .
- Stałe `CORE_DUMP`, `GET_INT`, `PUT_INT`, `GET_CHAR`, `PUT_CHAR`, `PUT_STRING` mają wartości, odpowiednio, od 0 do 5.
- Funkcja `core_dump()` wypisuje stan maszyny na wyjście diagnostyczne `stderr`. Prawdziwą wartość licznika rozkazów zastępuje tam swoim argumentem.
- Standardowa funkcja `exit()` z opisu instrukcji `hlt` przerywa wykonanie programu, przekazując systemowi operacyjnemu swój argument jako kod wyjścia.
- Jeżeli instrukcja nie ustali nowej wartości licznika rozkazów przez przypisanie na `pc`, wartością tą będzie adres następnej instrukcji za aktualną, modulo rozmiar pamięci.

## Polecenie

Napisz interpreter kodu maszyny MIK 8, czytający zapis początkowego stanu maszyny z wejścia. To, co jest na wejściu za zapisem stanu maszyny, ma być traktowane jako dane dla interpretowanego programu. Wynik działania programu jest wypisywany na standardowe wyjście lub, w przypadku `CORE_DUMP`, na wyjście diagnostyczne.

# Zapis stanu maszyny

Zapis stanu maszyny jest ciągiem deksarnych reprezentacji liczb oraz czterech znaków `'%'`, dowolnie podzielonym na wiersze oraz sformatowanym spacjami. Bezpośrednio za ostatnim `'%'` powinien być koniec wiersza.

Zapis dzieli się na cztery sekcje, każda zakończona przez `'%'`.

- Liczby pierwszej sekcji to zawartość poszczególnych rejestrów, zaczynając od rejestru o numerze 0.
- W drugiej sekcji są wartości kolejnych komórek pamięci danych, od komórki o adresie 0.
- Trzecia i czwarta sekcja opisuje stan pamięci instrukcji, z uwzględnieniem wartości licznika rozkazów. W trzeciej są instrukcje o adresach mniejszych od wartości licznika rozkazów a w czwartej instrukcje o adresach nie mniejszych od licznika rozkazów. Instrukcje uporządkowane są w kolejności rosnących adresów, od 0.

Jeśli w pierwszej sekcji jest mniej niż 16 liczb, pozostałe rejestry są wypełniane zerami. Podobnie, jeżeli w drugiej sekcji jest mniej niż 256 liczb, pozostałe komórki pamięci danych wypełniane są zerami. Jeśli łącznie w sekcji trzeciej i czwartej jest mniej niż 256 instrukcji przyjmujemy, że pozostałe mają kod operacji oraz wszystkie trzy pola argumentów równe 0 czyli, że to instrukcje `hlt 0`.

Liczba `x` w sekcji trzeciej lub czwartej jest zapisem instrukcji o kodzie operacji równym `x >> 12` i polach argumentów `a`, `b`, `c` równych, odpowiednio, `(x >> 8) & 15`, `(x >> 4) & 15` oraz `x & 15`.

Zapis stanu maszyny na wyjściu diagnostycznym operacją `CORE_DUMP` ma być zgodny z tym opisem. Podział na wiersze i formatowanie spacjami może być dowolne.

## Przykłady

- Kompilator `mikc` dla programu `kwadrat.c` podnoszącego liczbę do kwadratu wygenerował kod assemblerowy `kwadrat.s`. Assembler `mika` przekształcił go do pliku `kwadrat.mik`. Interpreter `miki`, uruchomiony poleceniem

```
cat kwadrat.mik kwadrat.in | ./miki > kwadrat.out
```

wykonał program na danych `kwadrat.in` i dał wynik `kwadrat.out`.

- Kompilator `mikc` dla programu `hello.c` wypisującego `Hello` wygenerował kod assemblerowy `hello.s`. Assembler `mika` przekształcił go do pliku `hello.mik`. Interpreter `miki`, uruchomiony poleceniem

```
< hello.mik ./miki > hello.out
```

wykonał program i dał wynik `hello.out`.

- Kompilator `mikc` dla programu `euklides.c` liczącego największy wspólny



dzielnik dwóch liczb algorytmem Euklidesa wygenerował kod asemblerowy euklides.s. Asembler mika przekształcił go do pliku euklides.mik. Interpreter miki, uruchomiony poleceniem

```
cat euklides.mik euklides.in | ./miki > euklides.out
```

wykonał program na danych euklides.in i dał wynik euklides.out.

- Kompilator mikc dla programu suma.c sumującego ciąg liczb zakończony zerem wygenerował kod asemblerowy suma.s. Asembler mika przekształcił go do pliku suma.mik. Interpreter miki, uruchomiony poleceniem

```
cat suma.mik suma.in | ./miki > suma.out
```

wykonał program na danych suma.in i dał wynik suma.out.

- Kompilator mikc dla programu choinka.c wypisującego za pomocą spacji i gwiazdek rysunek choinki o zadanej wysokości wygenerował kod asemblerowy choinka.s. Asembler mika przekształcił go do pliku choinka.mik. Interpreter miki, uruchomiony poleceniem

```
cat choinka.mik choinka.in | ./miki > choinka.out
```

wykonał program na danych choinka.in i dał wynik choinka.out.

- Kompilator mikc dla programu hanoi.c rozwiązującego problem wież Hanoi wygenerował kod asemblerowy hanoi.s. Asembler mika przekształcił go do pliku hanoi.mik. Interpreter miki, uruchomiony poleceniem

```
< hanoi.mik ./miki > hanoi.out
```

wykonał program i dał wynik hanoi.out.

- Kompilator mikc dla programu odwrotnie.c odwracającego ciąg liczb długości 10 wygenerował kod asemblerowy odwrotnie.s. Asembler mika przekształcił go do pliku odwrotnie.mik. Interpreter miki, uruchomiony poleceniem

```
cat odwrotnie.mik odwrotnie.in | ./miki > odwrotnie.out
```

wykonał program na danych odwrotnie.in i dał wynik odwrotnie.out.

- Kompilator mikc dla programu zadanie0.c, będącego rozwiązaniem zadania treningowego o iloczynie wielomianów, wygenerował kod asemblerowy zadanie0.s. Asembler mika przekształcił go do pliku zadanie0.mik. Interpreter miki, uruchomiony poleceniem

```
cat zadanie0.mik zadanie0.in | ./miki > zadanie0.out
```

wykonał program na danych zadanie0.in i dał wynik zadanie0.out.

- Kompilator mikc dla korzystającego z niestandardowego rozszerzenia języka C programu podzbiory.c wygenerował kod asemblerowy

```
< podzbiory0.mik ./miki > podzbiory0.out 2> podzbiory1.mik  
< podzbiory1.mik ./miki > podzbiory1.out 2> podzbiory2.mik  
< podzbiory2.mik ./miki > podzbiory2.out 2> podzbiory3.mik  
< podzbiory3.mik ./miki > podzbiory3.out 2> podzbiory4.mik  
< podzbiory4.mik ./miki > podzbiory4.out 2> podzbiory5.mik  
< podzbiory5.mik ./miki > podzbiory5.out 2> /dev/null
```

sześć razy uruchomił interpreter `miki` i stworzył pliki `podzbiory0.out`, `podzbiory1.out`, `podzbiory2.out`, `podzbiory3.out`, `podzbiory4.out`, `podzbiory5.out`. W każdym z nich zapisał inny czteroelementowy ciąg zer i jedynek.

- Plik `podzbiory_.mik` jest poprawną reprezentacją tego samego stanu maszyny, co plik `podzbiory0.mik` z poprzedniego przykładu. Interpreter, będący rozwiązaniem zadania, powinien umieć go wczytać.

## Uwagi i wskazówki

- Wolno założyć, że dane są poprawne.
- Tekst na wyjście diagnostyczne `stderr` można wypisać funkcjami `fputc()` i `fprintf()`, wywołując `fputc(c, stderr)` zamiast `putc(c)` oraz `fprintf(stderr, ...)` zamiast `printf(...)`.
- Deklaracja funkcji `exit()`, przerywającej wykonanie programu, jest w `stdlib.h`.
- Program do testów będzie kompilowany poleceniem:

```
gcc -std=c89 -pedantic -Wall -Wextra -Werror nazwa.c -o nazwa
```










































Wszystkie wymienione opcje kompilatora są obowiązkowe i nie wolno dodawać do nich żadnych innych.








- Przyjmujemy, że wynik funkcji `main()` inny niż 0 informuje o błędzie wykonania programu.
- Do treści zadania dołączone są pliki z danymi przykładowymi i z wynikami wzorcowymi.
- Poprawność wyniku można sprawdzić, przekierowując na wejście programu zawartość pliku z przykładowymi danymi i porównując rezultat, za pomocą programu `diff`, z plikiem zawierającym wynik wzorcowy, np.:

```
< przyklad.dane ./miki | diff - przyklad.wynik
```


Wynik uznamy za poprawny tylko, jeśli jest identyczny z wynikiem wzorcowym.

- W zapisie stanu maszyny **Mik 8** na wejściu interpretera dane w rejestrach i pamięci są liczbami całkowitymi od 0 do 255 a instrukcje są reprezentowane przez liczby całkowite od 0 do 65535.
- Pseudokod z kolumny **Znaczenie** w tabeli opisującej instrukcje zakłada, że licznik rozkazów `pc`, po rozpoznaniu instrukcji a przed rozpoczęciem jej wykonania, został przestawiony na następną instrukcję, modulo rozmiar pamięci.



 choinka.c  
 choinka.in  
 choinka.mik  
 choinka.out  
 choinka.s  
 euklides.c  
 euklides.in  
 euklides.mik  
 euklides.out  
 euklides.s  
 hanoi.c  
 hanoi.mik  
 hanoi.out  
 hanoi.s  
 hello.c  
 hello.mik  
 hello.out  
 hello.s  
 kwadrat.c  
 kwadrat.in  
 kwadrat.mik  
 kwadrat.out  
 kwadrat.s  
 odwrotnie.c  
 odwrotnie.in  
 odwrotnie.mik  
 odwrotnie.out  
 odwrotnie.s  
 podzbiory\_.mik  
 podzbiory.c  
 podzbiory.s  
 podzbiory0.mik  
 podzbiory0.out  
 podzbiory1.out  
 podzbiory2.out  
 podzbiory3.out  
 podzbiory4.out  
 podzbiory5.out  
 suma.c  
 suma.in  
 suma.mik

 suma.out  
 suma.s  
 zadanie0.c  
 zadanie0.in  
 zadanie0.mik  
 zadanie0.out  
 zadanie0.s

## Status przesłanego zadania

Numer próby	To jest próba nr 1.
Status przesłanego zadania	Przesłane do oceny
Stan oceniania	Ocenione
Termin oddania	środa, 14 grudzień 2016, 10:00
Pozostały czas	Zadanie zostało złożone 11 godz. 59 min. przed terminem
Ostatnio modyfikowane	wtorek, 13 grudzień 2016, 22:00
Przesyłane pliki	 PIOTR_SZUBERSKI_ZADANIE2.c
Komentarz do przesłanego zadania	► Komentarze (0)

## Informacja zwrotna

Ocena	10,00 / 10,00
Ocenione dnia	środa, 21 grudzień 2016, 11:03
Ocenione przez	 Artur Zaroda
Komentarz zwrotny	 Ocena jakości: 10/10 pkt  Między instrukcjami warunkowymi, których warunki wykluczają się, jak w funkcjach `jaka_liczba()` i `mnoznik()`, ...

Jesteś zalogowany(a) jako Piotr Szuberski (Wyloguj)  
WPI.LAB.INFO.I.16/17

Moodle, wersja 3.1.1 | [moodle@mimuw.edu.pl](mailto:moodle@mimuw.edu.pl)