WI

Metody kompilacji

Wykład 12, Generowanie kodu maszynowego I

Włodzimierz Bielecki, Piotr Błaszyński

Wydział Informatyki Zachodniopomorskiego Uniwersytetu Technologicznego

26 maja 2019

Asembler

Język Asembler jest symboliczną reprezentacją binarnego kodu komputerowego; jest językiem maszynowym. Asembler jest bardziej czytelny niż kod binarny, ponieważ używa symboli zamiast bitów. Inną jego rolą jest możliwość pisania programu komputerowego. Funkcje systemowe są pisane na podstawie języka Asembler w celu optymalizacji kodu.

Asembler

Asembler jako program. Assembler jest programem, który przekłada symboliczną wersję kodu na kod binarny.

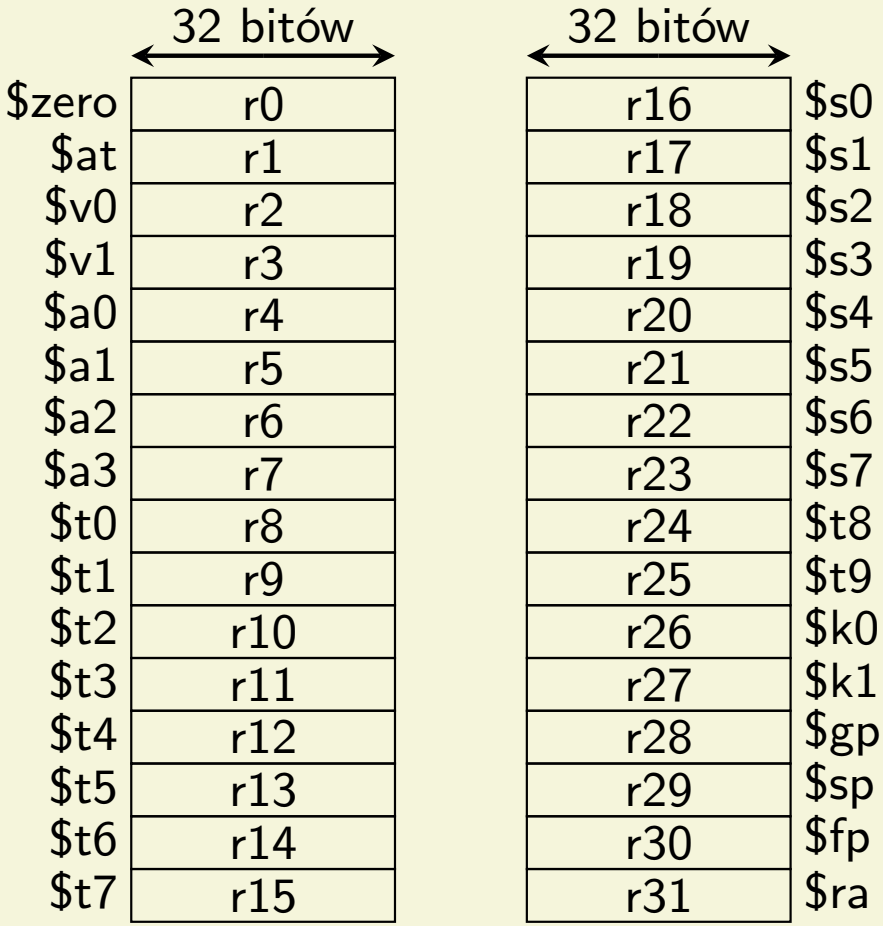
Asembler

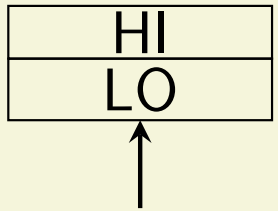
Symulator SPIM. Jest to symulator, który wykonuje programy napisane w języku Asembler dla procesorów, które implementują architekturę MIPS32. SPIM jest odwróceniem liter nazwy MIPS.
Kod i dokumentacja są dostępne pod adresem:
http://spimsimulator.sourceforge.net/.
Inne przydatne narzędzie to MIPhpS – Online MIPS Simulator:
http://alanhogan.com/asu/simulator.php.
Alternatywny emulator procesora MIPS - MARS
`http://courses.missouristate.edu/KenVollmar/mars/download.htm`

Procesor MIPS

MIPS (Microprocessor without Interlocked Piped Stages). Jest to architektura komputerowa (w szczególności procesor typu RISC) rozwijana przez firmę MIPS Technologies. Istnieje zarówno w wersji 32-, jak i 64-bitowej. Ma 32 rejestry całkowitoliczbowe oraz 32 rejestry zmiennoprzecinkowe. Pierwszy rejestr całkowitoliczbowy jest pseudorejestrem zawierającym zawsze 0 ($zero), co upraszcza wiele operacji. Trzydziesty pierwszy rejestr ($ra) całkowitoliczbowy jest adresem powrotu, przy wywołaniach funkcji. Rejestry MIPS są zaprezentowane na rysunku. Natomiast na kolejnym rysunku został przedstawiony schemat ogranizacji pamięci.

Procesor MIPS – Rejestry



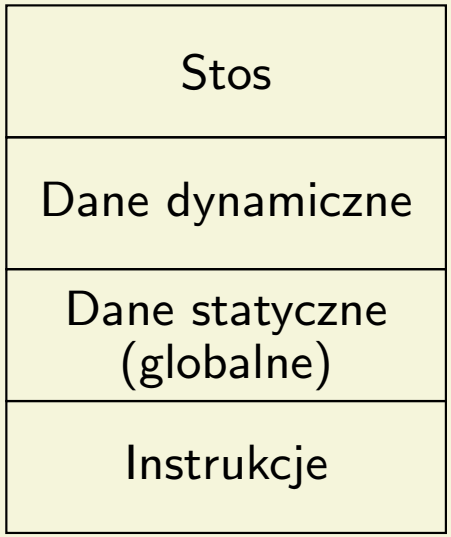


Przechowują pierwszą (*low-order word*) i drugą (*high-order word*) części wyników mnożenia / dzielenia

Rejestry ogólnego przeznaczenia

Rejestry specjalnego przeznaczenia

Procesor MIPS – organizacja pamięci



Generowanie kodu MIPS – Uwagi

Uwagi:

- SPIM wymaga etykiety *main*: w miejscu startu.
- Dane muszą być poprzedzone dyrektywą *.data*.
- Kod wykonywalny musi być poprzedzony dyrektywą *.text*.
- Dane i kod mogą być przeplatane.
- Nie można mieć nazw zmiennych, które są takie same jak nazwy rozkazów.

Generowanie kodu MIPS – Dyrektywy

Dyrektywy:

- *.text* – poprzedza kod,
- *.data* – poprzedza dane,
- *.global* – informuje, że symbol jest zmienną globalna,
- *.asciiz* – informuje, że kolejne znaki tworzą ciąg (łańcuch).

Generowanie kodu MIPS – Dyrektywy

Dyrektywy SPIM działają również w innych emulatorach, w tym w MARS. Poniżej przedstawiono przykładowy kod z zastosowaniem dyrektyw:

```
.text
.globl main
main:
        addi $t0, $zero, 5
        addi $t1, $zero, 7
        add $t2, $t0, $t1
        ...
        jal swap_proc
        jr $ra
```

Generowanie kodu MIPS – Dyrektywy

Wielkość bloku danych statycznych jest znana przed kompilacją; czas życia to cały czas wykonywania programu. Pamięć dla danych dynamicznych jest alokowana w czasie realizacji programu. Podobnie jak w przypadku danych dynamicznych wielkość danych na stosie nie jest znana przed kompilacją, np. parametry funkcji są odkładane na stosie, powodując jego powiększenie. Etykieta *.text* zawiera instrukcje programu. Każda dyrektywa w SPIM (również MARS) zaczyna się od kropki.

Dyrektywy SPIM - przykład


```
.data
        .word 5
        .word 7
        .byte 25
        .asciiz "the answer is"
.text
.globl main
main:
        lw $t0, 0($gp)
        lw $t1, 4($gp)
        add $t2, $t0, $t1
        ...
        jal swap_proc
        jr $ra
```

Etykiety


```
.data
        In1: .word 5
        in2 : .word 7
        C1: .byte 25
        str : .asciiz "the answer is"
.text
.globl main
main:
        lw $t0, in1
        lw $t1, in2
        add $t2, $t0, $t1
        ...
        jal swap_proc
        jr $ra
```

Dyrektywy SPIM

W powyższych przykładach *.globl main* wskazuje, że *main* jest symbolem globalnym, widocznym dla kodu zapisanego w innych plikach. Nazwy *In*1, *in*2, *C*1, *str* pełnią funkcję etykiet. Blok *.data* wskazuje początek danych statycznych.

Instrukcje

Poniżej wymieniono instrukcje ładowania i zapisywania:

```
lw register,addr   – przenosi wartość do rejestru
li register,num    – przenosi stałą do rejestru
la register,addr   – przenosi adres do rejestru
sw register,addr   – zapisuje wartość z rejestru
```

We fragmencie powyżej poszczególne elementy mają następujące znaczenie: litera *l* (instrukcje *lw, li, la*) oznacza *load*, czyli ładowanie wartości (*w* – *word*), stałej bezpośredniej (*i* – *immediate*) lub adresu (*a* – *address*). Litera *s* oznacza *store*, czyli zapis wartości.

Sposoby adresowania

Format	Adres w pamięci
register	zawartość rejestru
imm	bezpośrednia wartość (*immediate*)
imm(register)	bezpośrednia + zawartość rejestru
symbol	adres symbolu
symbol + / − imm	adres symbolu + lub − bezpośrednia
symbol + / − imm(register)	adres symbolu + lub − (bezpośrednia + zawartość rejestru)

Przykłady

Poniższa instrukcja przenosi wartość 5 do rejestru $t2$:

```
li $t2,5
```

Poniższa instrukcja przenosi wartość przechowywaną pod adresem x do rejestru $t3$:

```
lw $t3,x
```

Poniższa instrukcja przenosi adres przechowywany pod adresem x do rejestru $t3$. x oznacza w tym wypadku adres symbolu w tablicy symboli, czyli miejsce w pamięci w danych statycznych.

```
la $t3,x
```

Przykłady

Poniższa instrukcja przenosi wartość, której adresem w pamięci jest zawartość rejestru $t2$, do rejestru $t0$:

```
lw $t0,($t2)
```

Poniższa instrukcja przenosi wartość, której adresem w pamięci jest wartość rejestru $t2$ plus wartość 8, do rejestru $t1$:

```
lw $t1,8($t2)
```

Stosowanie rejestrów

Będziemy korzystać głównie z 8 rejestrów ($\$t0 - \$t7$) do generowania kodu w asemblerze. Dla binarnych operatorów arytmetycznych korzystamy z rejestrów $reg1, reg2, reg3$, jak niżej ($reg1 = reg2\ op\ reg3$):

```
add reg1,reg2,reg3    (dodawanie)
sub reg1,reg2,reg3    (odejmowanie)
mul reg1,reg2,reg3    (mnożenie)
div reg1,reg2,reg3    (dzielenie)
```

Stosowanie rejestrów

Dla jednoargumentowych operatorów arytmetycznych korzystamy z $reg1, reg2$, jak niżej ($reg1 = op\ reg2$):

```
neg reg1, reg2 ##negowanie wartości
```

Generowanie kodu - wyrażenia

Przykład generowania kodu dla instrukcji $a := b * -c + b * -c$
Poniższa instrukcja przenosi wartość b do rejestru $t0$:

```
lw $t0, b
```

Generowanie kodu - wyrażenia

Następujący fragment kodu przenosi wartość c do rejestru $t1$:

```
lw $t0,b
lw $t1,c
```

Generowanie kodu - wyrażenia

Poniższy fragment neguje wartość $c (c = -c)$; wynik zapisuje do rejestru $t1$:

```
lw  $t0 , b
lw  $t1 , c
neg  $t1 , $t1
```

Generowanie kodu - wyrażenia

Poniższe instrukcje obliczają wartość $b * -c$; wynik zapisywany jest do rejestru $t1$:

```
lw  $t0 ,b
lw  $t1 ,c
neg  $t1 ,$t1
mul  $t1 ,  $t1 ,$t0
```

Generowanie kodu - wyrażenia

Poniższy fragment kodu przenosi wartość *b* do rejestru *t*0:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
```

Generowanie kodu - wyrażenia

Następujący fragment kodu przenosi wartość *c* do rejestru *t2*:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
```

Generowanie kodu - wyrażenia

Poniższy fragment kodu neguje wartość c, wynik zapisuje do rejestru $t2$:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t2
```

Generowanie kodu - wyrażenia

Następujący fragment kodu oblicza wartość $b * -c$; wynik zapisuje do rejestru $t0$:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t0
mul $t0,$t0,$t2
```

Generowanie kodu - wyrażenia

W kolejnym fragmencie kodu obliczana jest wartość $b * -c + b * -c$; wynik zapisywany jest do rejestru $t1$:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t0
mul $t0,$t0,$t2
add $t1,$t0,$t1
```

Generowanie kodu - wyrażenia

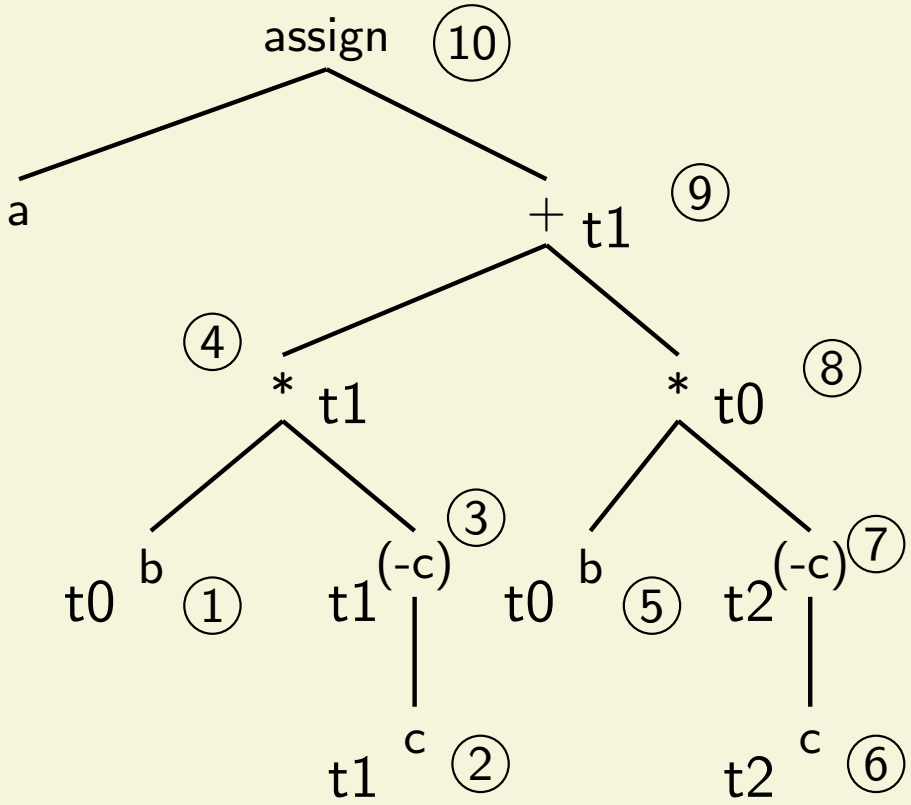
Poniższy kod zapisuje wynik końcowy z rejestru $t1$ do pamięci pod etykietą a:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t0
mul $t0,$t0,$t2
add $t1,$t0,$t1
sw $t1,a
```

Generowanie kodu - wyrażenia

Na rysunku oznaczono za pomocą liczb w okręgach poszczególne linie kodu wynikowego (a zarazem kroki parsowania wyrażenia).

Generowanie kodu - drzewo dla wyrażenia

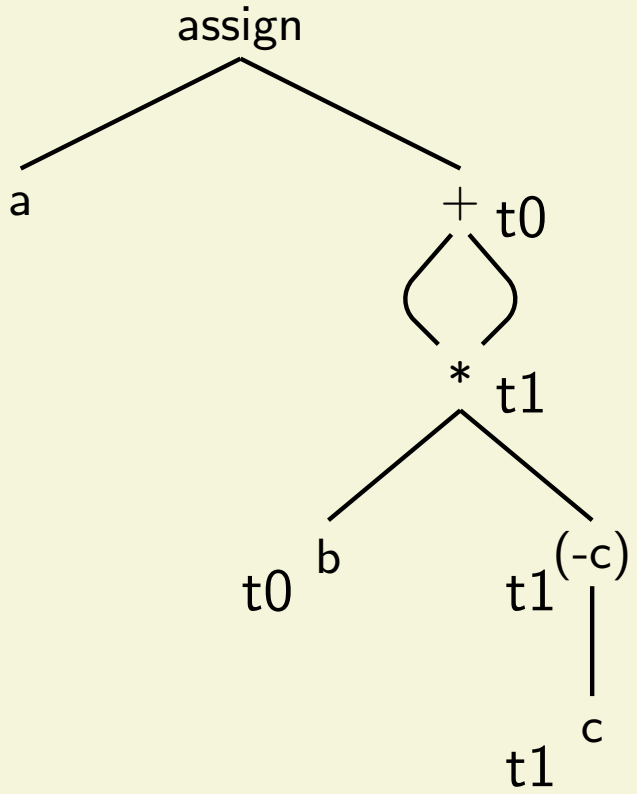


Generowanie kodu - zoptymalizowane wyrażenie

Na kolejnym rysunku i w poniższym fragmencie kodu przedstawiono kod zoptymalizowany, w którym nie obliczamy po raz drugi wartości $b * -c$; korzystamy z wcześniej obliczonej wartości:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1,$t1,$t0
add $t0,$t1,$t1
sw $t0,a
```

Generowanie kodu - zoptymalizowane wyrażenie



Generowanie kodu - drzewo dla wyrażenia

Operatory porównania

Operatory porównania mają następującą strukturę: $temp1 = temp2\ xxx\ temp3$, gdzie *xxx* oznacza warunek: $sgt(>), sge(>=), slt(<), sle(<=), seq(==)$, a $temp1$ służy do przechowywania wyniku i wynosi 0 dla *false*; wartość niezerowa oznacza *true*. Znaczenie poszczególnych instrukcji:

- *sgt* – set greater than,
- *sge* – set greater than or equal,
- *slt* – set less than,
- *sle* – set less than or equal,
- *seq* – set equal.

Operatory porównania

Przykład użycia operatora porównania:

```
sgt reg1,reg2,reg3
slt reg1,reg2,reg3
```

Skoki

Skoki:

1. *b label* – bezwarunkowy skok do etykiety (b – *branch to label*);
2. *bxxx temp*, *label* – warunkowy skok do etykiety, xxx = warunek, np.:
   - eqz(=0) (*branch on equal to zero*),
   - neq(/=) (*branch on not equal*),
   - le(¡=) (*branch on less or equal*);
3. *jal label* – skok i zapisanie adresu powrotu (jal – *jump and link*);
4. *jr register* – skok pod adres przechowywany w rejestrze (jr – *jump register*);

Skoki

Na przykładzie kolejnych fragmentów kodu omówiono przepływ sterowania dla pętli *while*:

```
while x <= 100 do
    x := x + 1
end while
```

Skoki

Kod powyższej pętli jest zamieniany na następujący fragment kodu wynikowego:

```
                lw $t0,x
                li $t1,100
L25:    sle $t2,$t0,$t1 ;Set less than or equal
                beqz $t2,L26
                addi $t0,$t0,1 ;Addition immediate
                sw $t0,x
                b L25
L26:
```

Skoki

W powyższym kodzie linia 4 powoduje skok, jeżeli wartość to fałsz, a linie 5 i 6 są ciałem pętli.
Poniżej podano przykład generowania przez kompilator kodu dla pętli i wyrażeń dla prostej operacji generowania liczb pierwszych.

Skoki


```
print 2  print  blank  #drukuj 2, drukuj spacje
for i = 3 to 100
   divides = 0
   for j = 2 to i/2
      if  "reszta␣z␣␣dzielenia␣␣i␣␣przez␣j␣␣jest␣0"
          then
           divides = 1
   end for
   if divides = 0 then
            print i
            print blank
  end for
  exit
```

Skoki

Na początku generujemy kod dla pętli z linii 2–9. Poniżej znajduje się przetworzona przez kompilator pętla zewnętrzna *for* $i = 3$ *to* 100.

```
        li $t0, 3           # variable i=3 in t0
        li $t1,100          # max loop counter in t1
l1:     sle $t7,$t0,$t1     # i <= 100
        beqz $t7, l2
        ...
        ...
        addi $t0,$t0,1      # increment i
        b l1
l2:
```

Skoki

Poniższy fragment przedstawia pętlę wewnętrzną *for* $j = 2$ *to* $i/2$ (linie 4–7).

```
          li $t2,2              # j = 2 in t2
                div $t3,$t0,2    # i/2 in t3
l3:    sle $t7,$t2,$t3    # j <= i/2
       beqz $t7,l4
       ...
       ...
           addi $t2,$t2,1     # increment j
       b l3
l4:
```

Skoki

Natomiast ostatni fragment dotyczy instrukcji warunkowych (linie 5–6 i 8).

```
    rem $t7,$t0,$t2      # reszta  i/j
    bnez $t7,l5          #
                         #
    li $t4,1             # divides=1
l5:
   ....
   bnez $t4,l6           # if divides = 0,
                            #then print i
   print i
l6:
```

Skoki

W powyższym kodzie skok (*bnez*) jest wykonywany, jeśli wartość w rejestrze nie jest zerem.

Wywołania systemowe

SPIM zapewnia kilka usług SO: Najbardziej przydatne są operacje I/O: czytania, pisania, otwierania i zamykania plików. Argumenty dla procedury *syscall* są umieszczone w rejestrach $\$a0 - \$a3$. Typ procedury *syscall* jest identyfikowany przez umieszczenie odpowiedniego numeru w rejestrze $\$v0$:

- 1 dla *print_int*,
- 4 dla *print_string*,
- 5 dla *read_int*.

Wywołania systemowe

Rejestr $\$v0$ może przechowywać także adres do zwracania wartości przez wywołanie systemowe.

Wywołania systemowe

Poniższe wywołanie odpowiada instukcji *Print*(*i*):

```
li $v0,1
lw $a0,i
syscall
```

Wywołania systemowe

W powyższym kodzie liczba 1 ładowana jest do rejestru $\$v0$ i określa funkcję *print_int*. Instrukcja *lw* przenosi wartość z adresu *i* do rejestru *a*0; wartość ta jest argumentem funkcji *print_int*.

Wywołania systemowe

Poniższe wywołanie odpowiada instukcji *Read*(*i*):

```
li $v0,5
syscall
sw $v0,i
```

Wywołania systemowe

W powyższym kodzie liczba 5 ładowana jest do rejestru $\$v0$ i określa funkcję *read_int*. Instrukcja *sw* zapisuje w pamięci wartość przechowywaną w $\$v0$ pod adresem i.

Wywołania systemowe

Poniższe wywołanie odpowiada zakończeniu programu (*exit*):

```
li $v0,10
syscall
```

Wywołania systemowe

W powyższym kodzie liczba 10 jest ładowana do rejestru $\$v0$ i określa funkcję *exit*, która po wywołaniu *syscall* kończy wykonywanie kodu.

Wywołania systemowe

W tabeli na kolejnym slajdzie zostały przedstawione podstawowe funkcje systemowe wraz ze sposobem przekazywania do nich parametrów i pobierania danych wyjściowych.

Wywołania systemowe

Funkcja	Kod	Wejście/Wyjście
print_int	$v0 = 1	$a0 = integer to print prints $a0 to standard output
print_float	$v0 = 2	$f12 = float to print prints $f12 to standard output
print_double	$v0 = 3	$f12 = double to print prints $f12 to standard output
print_string	$v0 = 4	$a0 = address of first character prints a character string to standard output
read_int	$v0 = 5	integer read from standard input placed in $v0
read_float	$v0 = 6	float read from standard input placed in $f0
read_double	$v0 = 7	double read from standard input placed in $f0
read_string	$v0 = 8	$a0 = address to place string, $a1 = max string length reads standard input into address in $a0

Wywołania systemowe

Funkcja	Kod	Wejście/Wyjście
sbrk	$v0 = 9	$a0 = number of bytes required $v0 = address of allocated memory. Allocates memory from the heap
exit	$v0 = 10	
print_char	$v0 = 11	$a0 = character (low 8 bits) -
read_char	$v0 = 12	 $v0 = character (no line feed) echoed

Wywołania systemowe

Funkcja	Kod	Wejście/Wyjście
file_open	$v0 = 13	$a0 = full path (zero terminated string with no line feed), $a1 = flags, $a2 = UNIX octal file mode (0644 for rw-r–r–) $v0 = file descriptor
file_read	$v0 = 14	$a0 = file descriptor, $a1 = buffer address, $a2 = amount to read in bytes $v0 = amount of data in buffer from file (-1 = error, 0 = end of file)
file_write	$v0 = 15	$a0 = file descriptor, $a1 = buffer address, $a2 = amount to write in bytes $v0 = amount of data in buffer to file (-1 = error, 0 = end of file)
file_close	$v0 = 16	$a0 = file descriptor

Wywołania systemowe

Poniżej podano jeszcze raz przykład generowania liczb pierwszych, wraz z pełnym kodem wynikowym (w kolejnej ramce).

```
print 2  print blank
for i = 3 to 100
   divides = 0
   for j = 2 to i/2
      if j divides i evenly then divides = 1
   end for
  if divides = 0 then print i print blank
end for
exit
```

Wywołania systemowe

Kod wynikowy:

```
.data
        blank:   .asciiz "␣"
.text
main:
        li $v0,1
        li $a0,2
        syscall
        li $v0,4
        la $a0,blank
        syscall
        li $t0,3      # i in t0
        li $t1,100    # max in t1
l1: sle $t7,$t0,$t1
        beqz $t7,l2
        li $t4,0
        li $t2,2     # jj in t2
        div $t3,$t0,2 # max in t3
```

Wywołania systemowe

Kod wynikowy, ciąg dalszy:

```
l3:     sle $t7,$t2,$t3
        beqz $t7,l4
        rem $t7,$t0,$t2
        bnez $t7,l5
        li $t4,1
l5: addi $t2,$t2,1
        b l3                #end of inner loop
l4:     bnez $t4,l6
        li $v0,1
        move $a0,$t0
        syscall # print i
        li $v0,4
        la $a0,blank
        syscall
l6: addi $t0,$t0,1
        b l1                #end of outer loop
l2:     li $v0,10
        syscall
```

Wywołania systemowe

Komentarz do powyższego kodu:

- Dyrektywa *.asciiz*" " zapisuje znaki łańcucha w pamięci.
- Pętla zewnętrzna rozpoczyna się od linii 28.
- W linii 38 jest skok warunkowy do etykiety l6.
- Pętla zewnętrzna kończy się w linii 53.