Metody Kompilacji
Wykłady 14-15
Generacja kodu maszynowego
Symulator SPIM

Język Asembler

Język Asembler jest symboliczną reprezentacją binarnego kodu komputerowego, jest językiem maszynowym.

Asembler jest bardziej czytelny niż kod binarny, ponieważ używa symboli zamiast bitów.

Język Asembler

Inną rolą asemblera jest możliwość pisania programu komputerowego.

Funkcje systemowe są pisane w oparciu o asembler celem optymalizacji kodu.

Asembler jako program

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

Symulator SPIM

SPIM jest to symulator, który wykonuje programy napisane w języku asemblera dla procesorów, które implementują architekturę MIPS32.

SPIM jest odwróceniem porządku liter nazwy MIPS.

Symulator SPIM

Kod i dokumentacja są dostępne pod adresem:

http://spimsimulator.sourceforge.net/

MIPhpS: Online MIPS Simulator:
http://alanhogan.com/asu/simulator.php

Symulator SPIM

MIPS (Microprocessor without Interlocked Piped Stages)

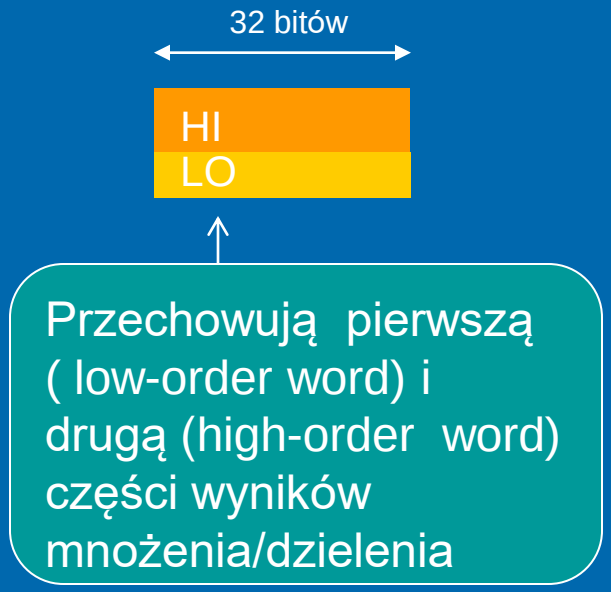
jest to architektura komputerowa (w szczególności procesor typu RISC) rozwijana przez firmę MIPS Technologies. Istnieje zarówno w wersji 32-jak i 64-bitowej.

Symulator SPIM

- Posiada 32 rejestry całkowitoliczbowe oraz 32 rejestry zmiennoprzecinkowe. Pierwszy rejestr całkowitoliczbowy jest pseudo rejestrem zawierającym zawsze 0 ($zero), co w praktyce upraszcza wiele operacji.
- Trzydziesty pierwszy rejestr ($ra) całkowitoliczbowy jest adresem powrotu przy wywołaniach funkcji.

Rejestry MIPS

Rejestry ogólnego przeznaczenia



Rejestry specjalnego przeznaczenia

Uwagi

- SPIM wymaga etykiety `main:` w miejscu startu
- Dane muszą być poprzedzone dyrektywą `".data"`
- Kod wykonywalny musi być poprzedzony dyrektywą `".text"`

Uwagi

- Dane i kod mogą być przeplatane.
- Nie można mieć nazw zmiennych, które są takie same, jak nazwy rozkazów.

Dyrektywy

.text
- Poprzedza kod

.data
- Poprzedza dane

.global
- Informuje, że symbol jest zmienną globalna

.asciiz
- Informuje, że kolejne znaki tworzą "ciąg (łańcuch)”

Dyrektywy SPIM

Plik add.s

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

Rozmiar danych jest znany przed kompilacją, czas życia - cały czas wykonywania programu

Każda dyrektywa zaczyna się od kropki

Dyrektywy SPIM

Plik add.s

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

Pamięć dla danych dynamicznych jest alokowana w czasie wykonywania programu

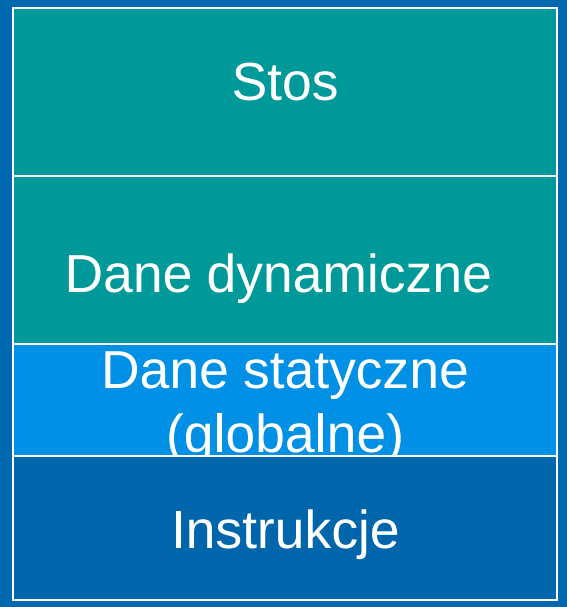
Każda dyrektywa zaczyna się od kropki

Dyrektywy SPIM

Plik add.s

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

Podobnie jak dla danych dynamicznych rozmiar danych nie jest znany przed kompilacją, przykładowo parametry funkcji są odkładane na stosie

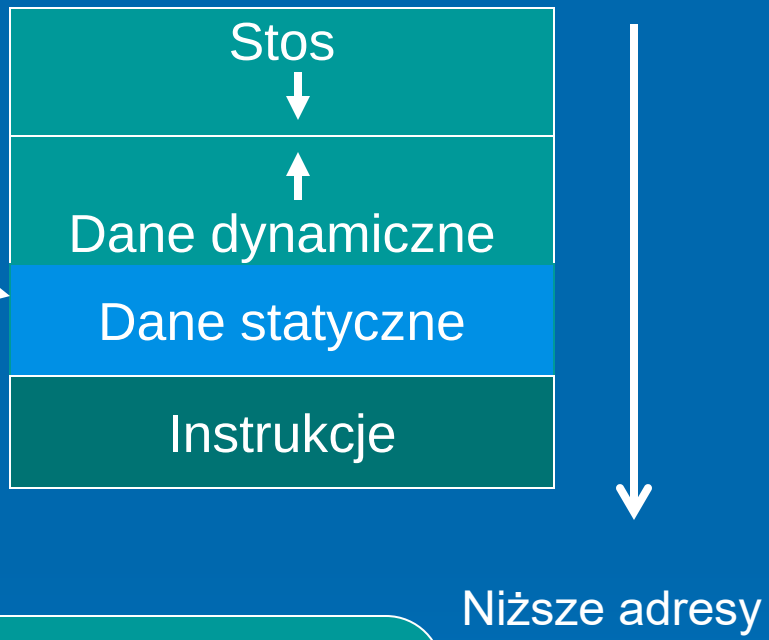


Każda dyrektywa zaczyna się od kropki

Dyrektywy SPIM

Plik add.s

```
.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
```

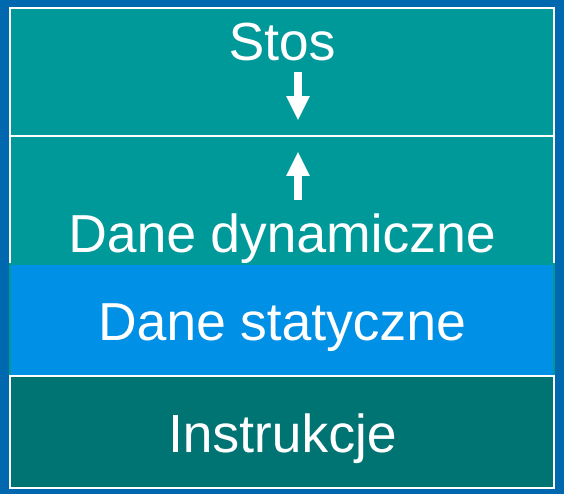


oznacza, że main jest symbolem globalnym, widocznym dla kodu zapisanym w innych plikach.

Etykiety

Plik add.s

```
.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
```



Etykiety

Symulator SPIM

Instrukcje:

Ładowanie / Zapisywanie

`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

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

```
li $t2,5
```

Wartość bezpośrednia

– przenosi wartość 5 do rejestru t2

Przykłady

– przenosi wartość pod adresem "x" do rejestru t3

Przykłady

```
la $t3,x
```

– przenosi adres pod adresem "x„ do rejestru t3

Przykłady

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

– przenosi wartość, której adresem w pamięci jest zawartość rejestru t2, do rejestru t0

Przykłady

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

– przenosi wartość, której adresem w pamięci jest wartość rejestru 2 plus wartość „8”, do rejestru t1

Zastosowanie rejestrów

Będziemy głównie korzystać 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)

- Dla jednoargumentowych operatorów arytmetycznych korzystamy z reg1, reg2 jak niżej

reg1 = op reg2

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

Przykład generowania kodu
a := b *-c + b*-c

```
lw $t0,b
```

przenosi wartość b do rejestru t0

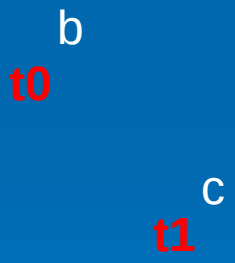
b

t0

Przykład generowania kodu
a := b *-c + b*-c

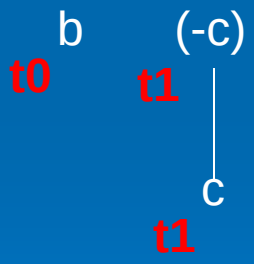
```
lw $t0,b
lw $t1,c
```

przenosi wartość c do rejestru t1



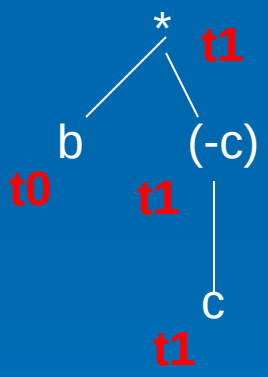
Przykład generowania kodu
a := b *-c + b*-c

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



Neguje wartość c, wynik zapisuje do rejestru t1

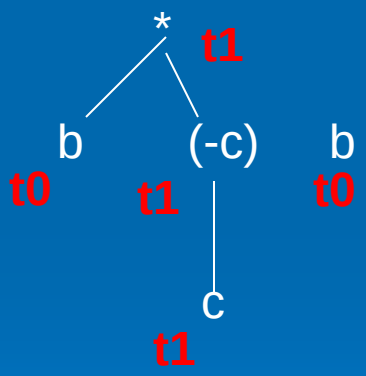
Przykład generowania kodu a := b *-c + b*-c




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

Oblicza wartość b *-c , wynik zapisuje do rejestru t1

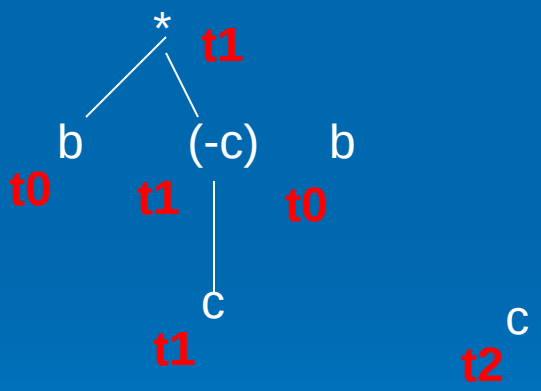
Przykład generowania kodu a := b *-c + b*-c




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

przenosi wartość b do rejestru t0

Przykład generowania kodu
a := b *-c + b*-c

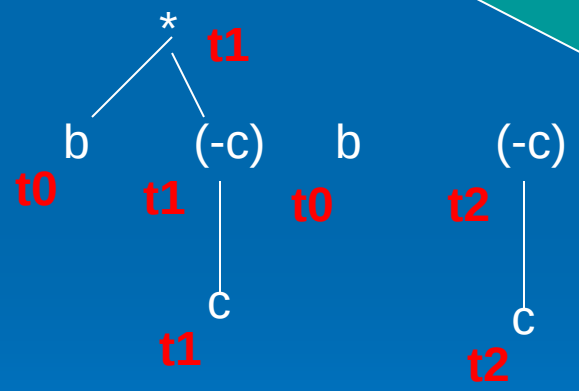



```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
```

przenosi wartość c do rejestru t2

Przykład generowania kodu a := b *-c + b*-c

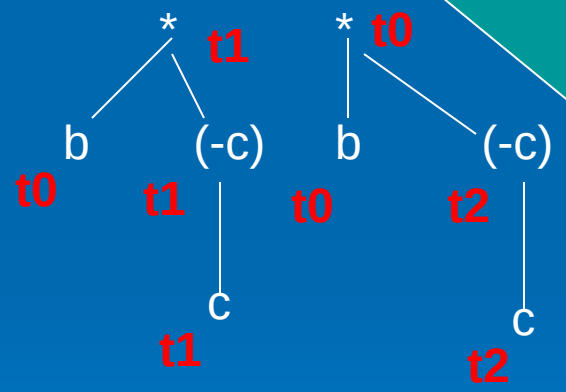
Neguje wartość c, wynik zapisuje do rejestru t2




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

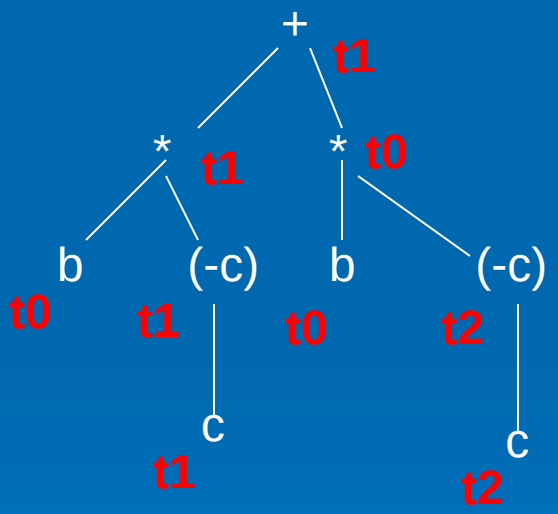
Przykład generowania kodu a := b *-c + b*-c

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
```

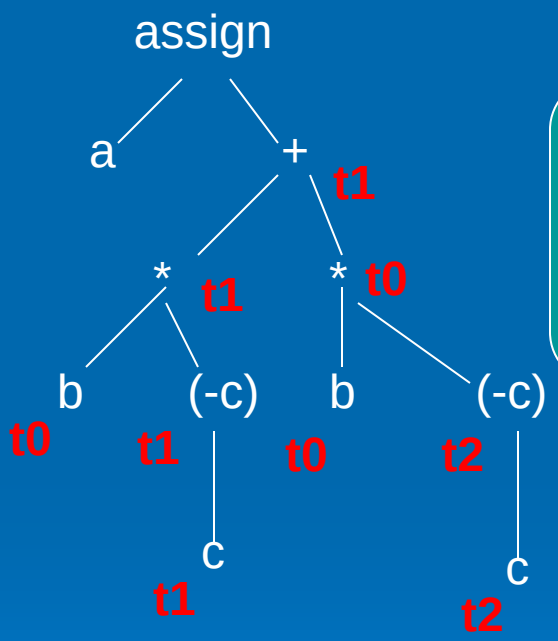
Przykład generowania kodu
a := b *-c + b*-c



Oblicza wartość b *-c+b*-c , wynik zapisuje do rejestru t1

```
l     t2,c
n     $t2,$t0
m  1  $t0,$t0,$t2
add   $t1,$t0,$t1
```

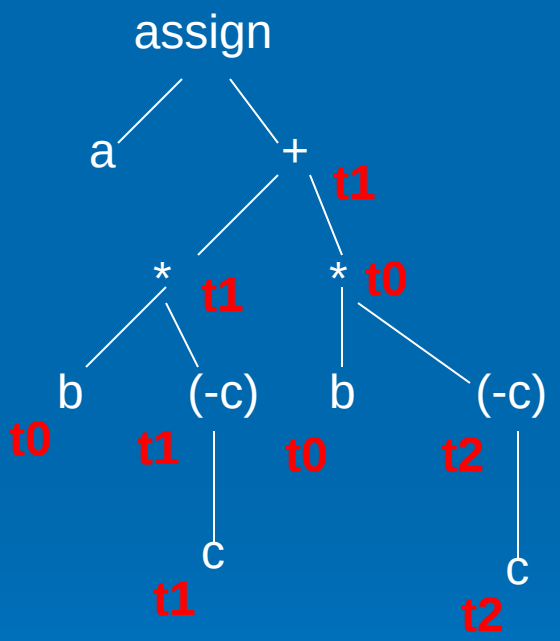
Przykład generowania kodu a := b *-c + b*-c




```
lw $t0,b
                    ,$t0
    $t2,c
  eg $t2,$t0
  ul $t0,$t0,$t2
  dd $t1,$t0,$t1
sw $t1,a
```

zapisuje wartość a przechowywaną w t1

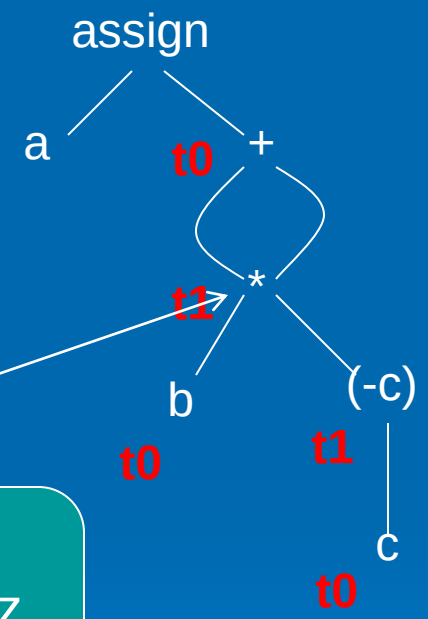
Wynik końcowy
a := b *-c + b*-c




```
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
```

Kod zoptymalizowany
a := b * -c + b * -c

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



Nie obliczamy po raz drugi wartości b * -c, korzystamy z wcześnie obliczonej wartości

Operatory porównania

temp1 = temp2 xxx temp3,

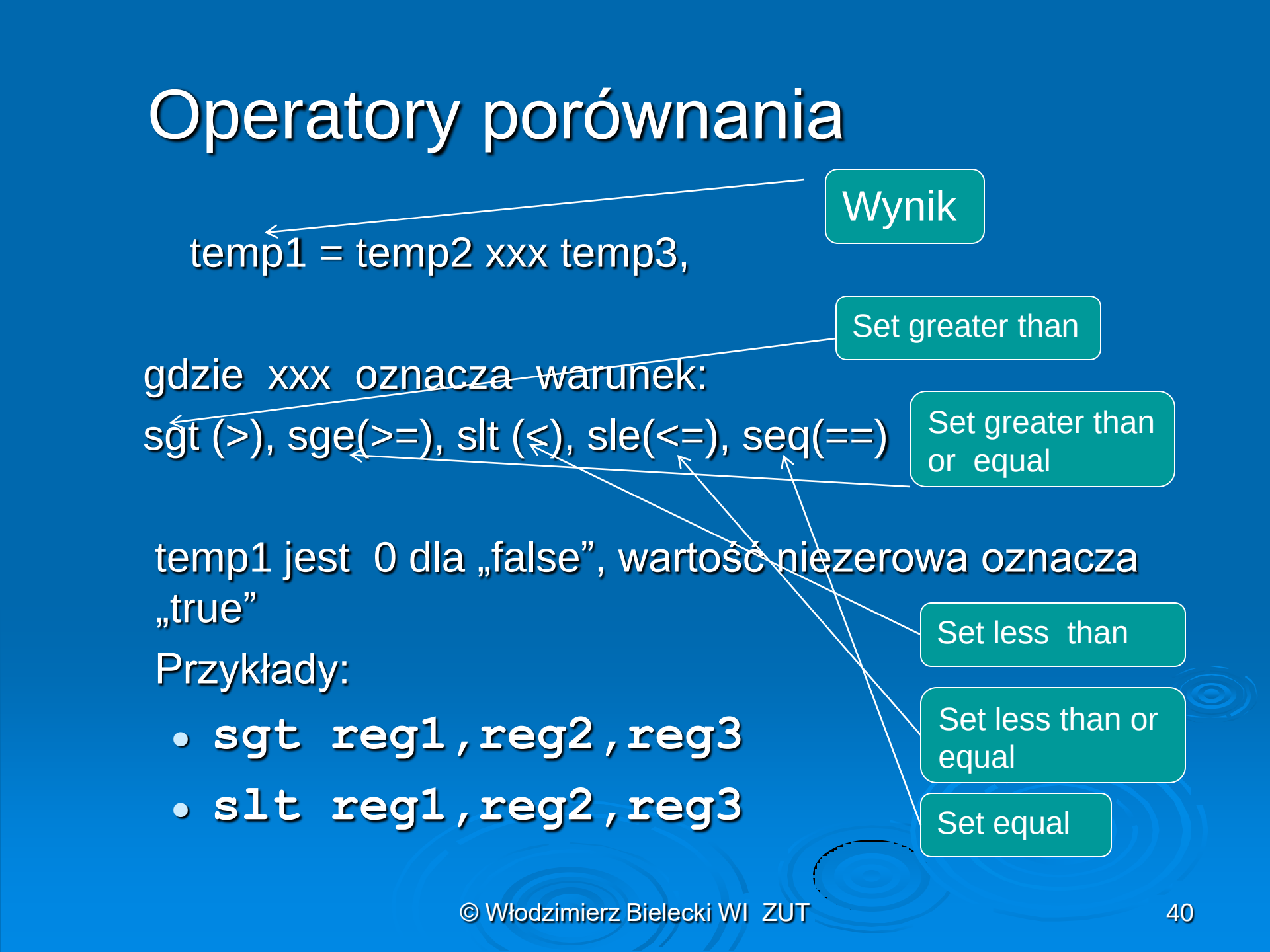
gdzie xxx oznacza warunek:

sgt (>), sge(>=), slt (<), sle(<=), seq(==)

temp1 jest 0 dla „false”, wartość niezerowa oznacza „true”

Przykłady:

- **`sgt reg1,reg2,reg3`**
- **`slt reg1,reg2,reg3`**



SKOKI

branch to label

- **`b label`** - bezwarunkowy skok do etykiety

- **`bxxx temp, label`** – warunkowy skok do etykiety, **`xxx`** = warunek: eqz(=0), neq(/=), le(<=), …

Branch on less or equal

Branch on equal to zero

Branch on not equal

SKOKI

Jump and link

Jump register

- **`jal label`** – skok i zapisanie adresu powrotu
- **`jr register`** – skok pod adres przechowywany w rejestrze

Przepływ sterowania

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

```
     lw $t0,x
     li $t1,100
L25: sle $t2,$t0,$t1
     beqz $t2,L26
     addi $t0,$t0,1
     sw $t0,x
     b L25
L26:
```

Set less than or equal

Skok jeśli FALSE(0)

Addition immediate

Ciało pętli

Przykład: Generowanie liczb pierwszych

```
print 2  print  blank  #drukuj  2, drukuj spację
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
```

Pętle

```
print 2  print blank
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
```

Generujemy najpierw kod dla zaznaczonego fragmentu

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:
```

Pętla wewnętrzna: for j = 2 to i/2

```
        li $t2,2            # j = 2 in t2
        div $t3,$t0,2       # i/2 in t3
13:     sle $t7,$t2,$t3     # j <= i/2
        beqz $t7,14
        ...
        addi $t2,$t2,1      # increment j
        b 13
14:
```

Instrukcje warunkowe

```
print 2  print blank
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
```

Instrukcje warunkowe

```
    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:
```

Skok 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

Wywołania systemowe

Typ procedury syscall jest identyfikowany przez umieszczenie odpowiedniego numeru w rejestrze $v0:

1 dla print_int,
4 dla print_string,
5 dla read_int

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

Wywołania systemowe

Print(i)

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

Określa funkcję print_int

Przenosi wartość pod adresem I do rejestru a0; wartość ta jest argumentem funkcji print_int

Wywołania systemowe

- Read(i)

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

Określa funkcję read_int

Zapisuje w pamięci wartość przechowywaną w $v0 pod adresem i

Wywołania systemowe

- Exiting

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

Kończy wykonywanie kodu

Funkcje 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
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
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	

Przykład: Generowanie liczb pierwszych

```
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
```

```
    .data
blank:   .asciiz " "
    .text
    li $v0,1
    li $a0,2
    syscall             # print 2
    li $v0,4
    la $a0,blank    # print blank
    syscall

    li $v0,1
    lw $a0,i
    syscall     # print i

    li $v0,10
    syscall      # exit
```

Dyrektywa .asciiz “ zapisuje znaki łańcucha w pamięci

```
        .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
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
```

Pętla wewnętrzna

Cały program

```
        .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
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
```

Print „2”

Pętla wewnętrzna

Cały program

```
        .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
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
```

Print „blank”

Pętla wewnętrzna

Cały program

```
        .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
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
l4:
        bnez $t4,l6
        li $v0,1
        move $a0,$t0
        syscall   # print i
        li $v0,4
        la $a0,blank
        syscall
```

Początek kodu pętli zewnętrznej

```
l6:     addi $t0,$t0,1
        b l1
l2:                     #end of outer loop
```

Cały program

```
        .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
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:
```

Pętla wewnętrzna

bnez $t4,l6

Skok warunkowy do 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
```

Cały program

```
        .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
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
        li $v0,4
        la $a0,blank
        syscall

l6:
        addi $t0,$t0,1
        b l1                    #end of outer loop
l2:     li $v0,10
        syscall
```

Przenosi wynik z t0 do a0

print i

Pętla wewnętrzna

Cały program

```
        .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
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
```

Print blank

Pętla wewnętrzna

Cały program

```
        .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
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
```

Pętla wewnętrzna

Koniec pętli zewnętrznej

Cały program

```
        .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
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
```

Pętla wewnętrzna

Zakończenie programu

Cały program

Generowanie kodu

- Żeby wygenerować kod w asemblerze,
- Potrzebne są:
  - Deklaracje
  - Wyrażenia
  - Przepływ sterowania
  - Wywołanie procedur

Przetwarzanie deklaracji

- Zmienna lokalna czy globalna?
- Przydział pamięci dla zmiennych
- Podstawowe typy: integer, boolean …
- Złożone typy: records, arrays …

Przydział pamięci. SPIM

przydział 4 bajtów do każdego słowa

inicjalizacja początkową wartością

Generowanie kodu dla deklaracji

```
    .data
var_name1:      .word   0
var_name2:      .word 29,10
var_name3:      .space 40
```

Może również przydzielić większy obszar

Dyrektywy SPIM

.data	poprzedza dane
.ascii "str"	zapisuje "str“ w pamięci bez znaku końca wiersza '\0'
.asciiz "str"	to samo jak poprzednia, ale z '\0'
.byte 3,4,16	zapisuje 3 wartości, każda zajmuje jeden bajt
.double 3.14, 2.72	zapisuje 2 wartości zmiennoprzecinkowe o podwójnej dokładności

Dyrektywy SPIM

.float 3.14, 2.72	zapisuje 2 wartości zmiennoprzecinkowe,
.word 3,4,16	zapisuje 3 wartości, każda zajmuje 32 bitów,
.space 100	rezerwuje 100 bajtów,
.text	zaczyna segment tekstu z instrukcjami.

Przetwarzanie wyrażeń

- Generowanie poprawnego kodu
- Kontrola typów
- Obliczenie adresów elementów tablicy
- Wyrażenia w konstrukcjach sterowania

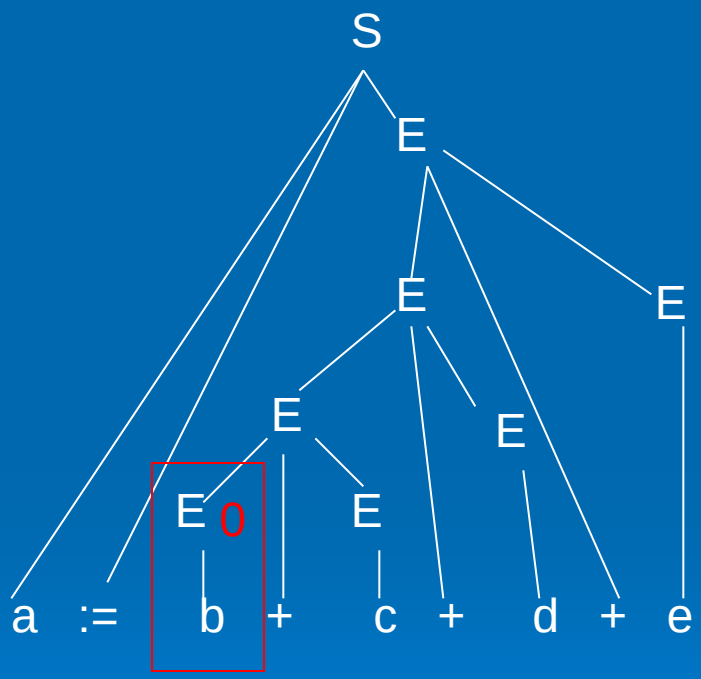
Wyrażenia

Gramatyka:

S → id := E

E → E + E

E → id



Kod:

```
lw $t0,b
```

Atrybuty przekazują informacje o zmiennych tymczasowych w górę drzewa

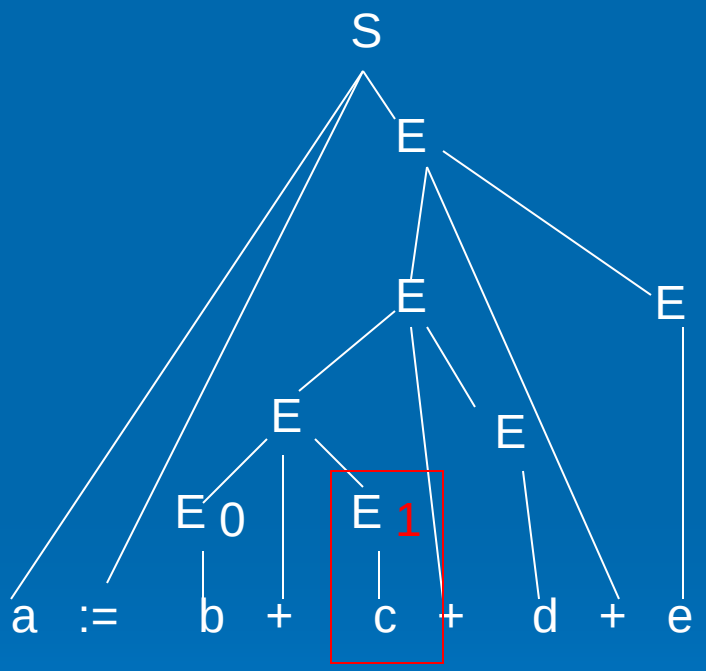
Wyrażenia

Gramatyka :

S → id := E

E → E + E

E → id



Kod:

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

Każdy numer odpowiada zmiennej tymczasowej.

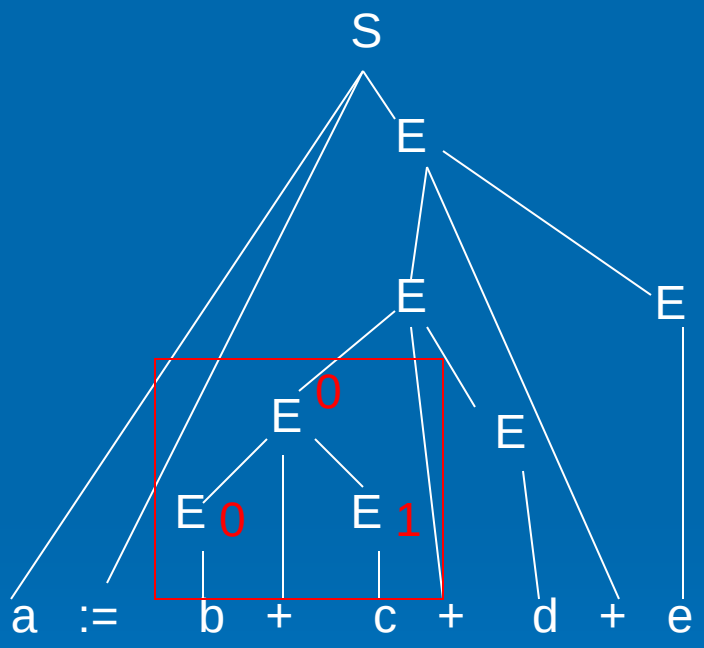
Wyrażenia

Gramatyka :

S → id := E

E → E + E

E → id




```
kod:
   lw $t0,b
   lw $t1,c
   add $t0,$t0,$t1
   sw  $t0,tmp1
```

Każdy numer odpowiada zmiennej tymczasowej.

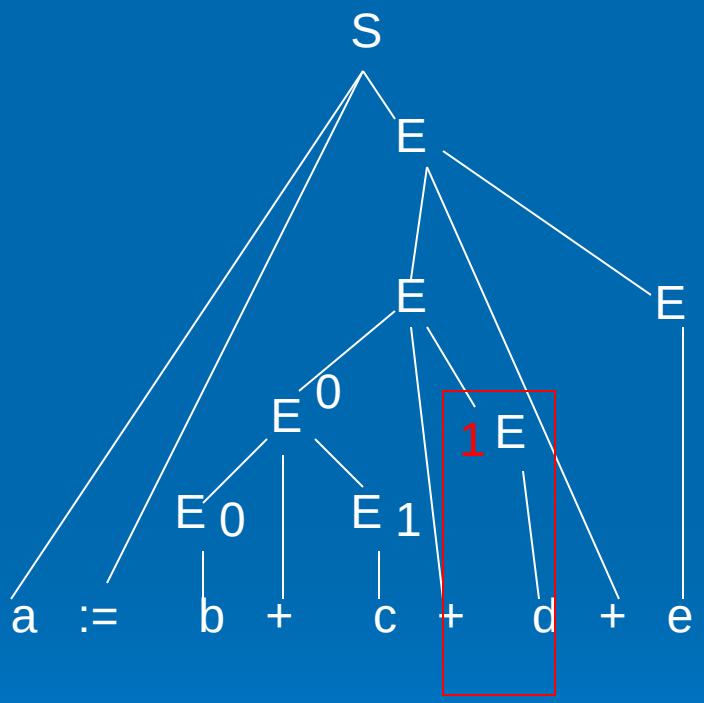
Wyrażenia

Gramatyka :

S → id := E

E → E + E

E → id



Każdy numer odpowiada zmiennej tymczasowej.

```
Kod:
   lw   $t0,b
   lw   $t1,c
   add  $t0,$t0,$t1
   sw   $t0,tmp1
   lw   $t0,tmp1
   lw   $t1,d
```

Wyrażenia

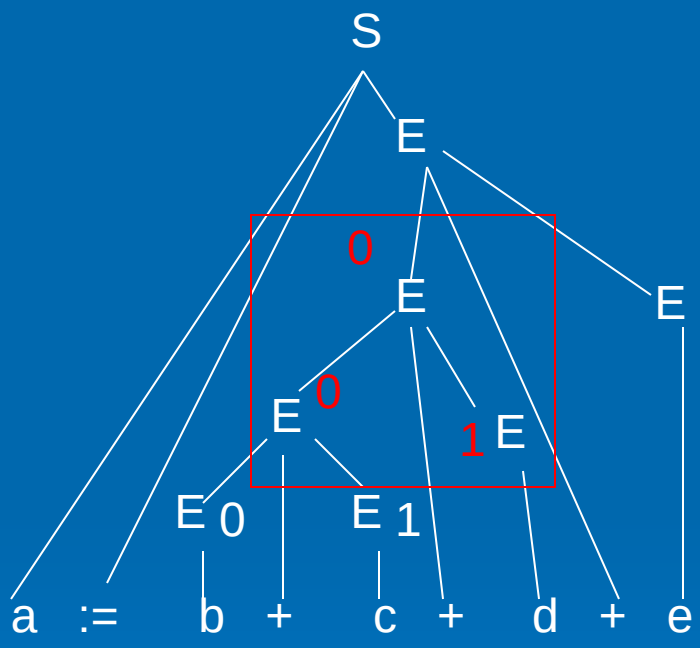
Gramatyka :

S → id := E

E → E + E

E → id

```
 instrukcje:
  sw tmp1,$t0
  lw $t0,tmp1,
są redundatne,
więc nie ma
konieczności ich
pisania
```



Każdy numer odpowiada zmiennej tymczasowej.

Kod:

```
   lw   t0,b
   lw   t1,c
   add  $t0,$t0,$t1
   sw   $t0,tmp1
   lw   $t0,tmp1
   lw   t1,d
   add  $t0,$t0,$t1
```

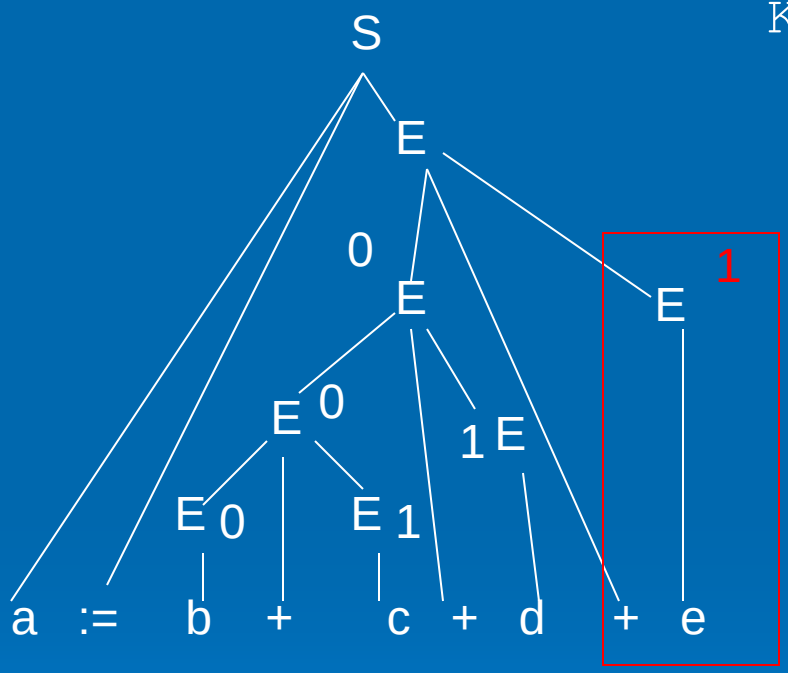
Wyrażenia

Gramatyka :

S → id := E

E → E + E

E → id



Każdy numer odpowiada zmiennej tymczasowej.

Kod:

```
lw $t0,b
lw $t1,c
add $t0,$t0,$t1
lw $t1,d
add $t0,$t0,$t1
lw $t1,e
```

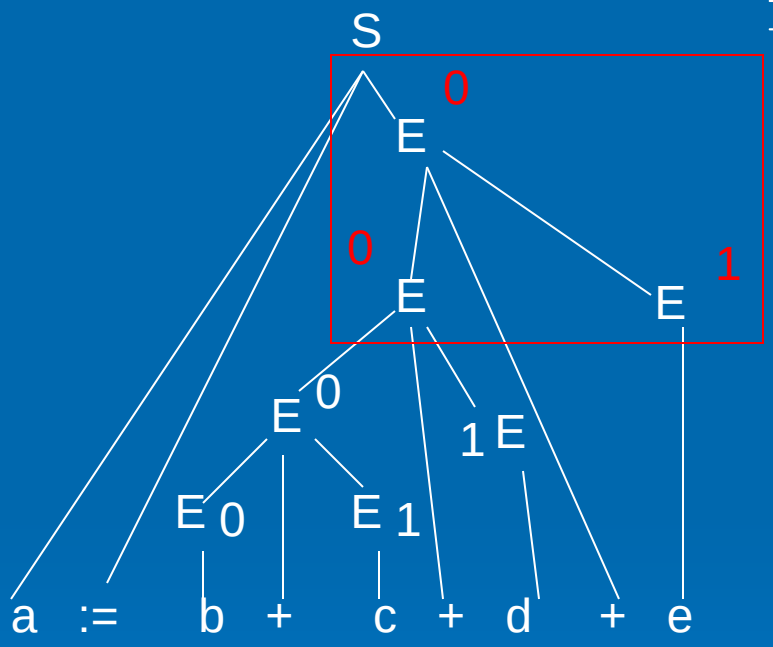
Wyrażenia

Gramatyka :

S → id := E

E → E + E

E → id



Kod:

```
lw $t0,b
lw $t1,c
add $t0,$t0,$t1
lw $t1,d
add $t0,$t0,$t1
lw $t1,e
add $t0,$t0,$t1
```

Każdy numer odpowiada zmiennej tymczasowej.

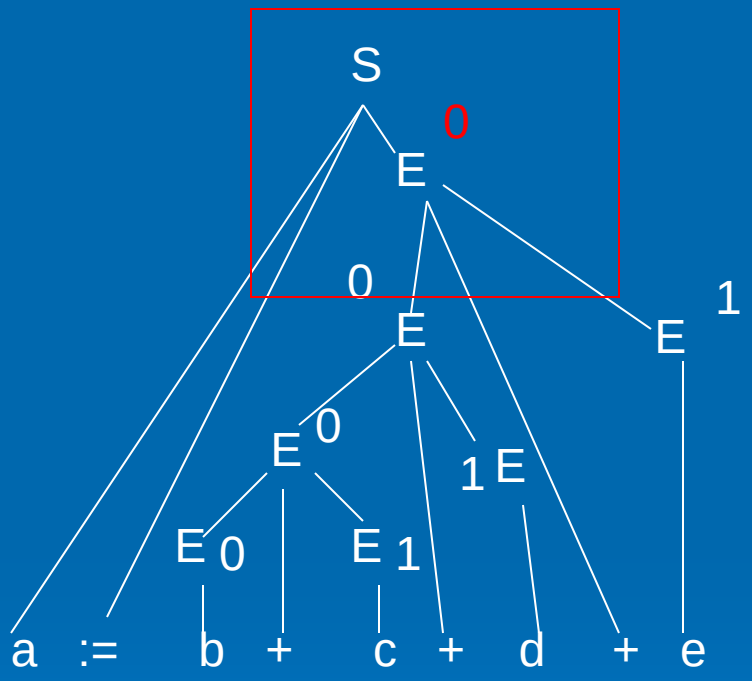
Wyrażenia

Gramatyka :

S → id := E

E → E + E

E → id



Każdy numer odpowiada zmiennej tymczasowej.

```
Kod:
   lw   $t0,b
   lw   $t1,c
   add  $t0,$t0,$t1
   lw   $t1,d
   add  $t0,$t0,$t1
   lw   $t1,e
   add  $t0,$t0,$t1
   sw   $t0,a
```

Akcje semantyczne dla wyrażeń, SPIM

Zapisuje wartość przechowywaną w rejestrze `$3.reg` w pamięci pod adresem `$1`

S → id := E	{printf("sw $t%d,%s\n", $3.reg,$1);
	free_reg($3.reg); //zwalnia //rejestr
	}

Symbol $3 odwołuje się do wartości skojarzonej z 3-tym symbolem gramatyki po prawej stronie.

Akcje semantyczne dla wyrażeń, SPIM

Rejestr, który przechowuje wartość pierwszej nazwy

Rejestr, który przechowuje wartość drugiej nazwy

$E \rightarrow E + E$	`{ $$.reg = $1.reg;`
	`printf("add $t%d, $t%d,`
	`$t%d\n",$$.reg,$1.reg, $3.reg,`
	`$3.reg);`
	
	`free_reg($3.reg); //zwalnia rejestr`
	`}`

Symbol $$ odwołuje się do wartości atrybutu skojarzonej z nieterminalem po lewej stronie

Akcje semantyczne dla wyrażeń, SPIM

Przenosi wartość do rejestru

Zwraca wolny rejestr

E → id	{ $$.reg = get_register(); printf("lw $t%d,%s\n", $$.reg,$1); }

Adres, pod którym w pamięci jest przechowywana wartość zmiennej (id)

Obliczenie adresów elementów tablic

b oznacza adres bazowy, od którego zaczyna się obszar pamięci, zarezerwowany do przechowywania elementów tablicy

Tablice jednowymiarowe

a[*l*…*h*],każdy element zajmuje *s* bajtów

- Liczba elementów: $e = h - l + 1$
- Rozmiar tablicy: $e * s$
- Adres elementu a[i], zakładając, że obszar zaczyna się od adresu *b*, $l <= i <= h$:

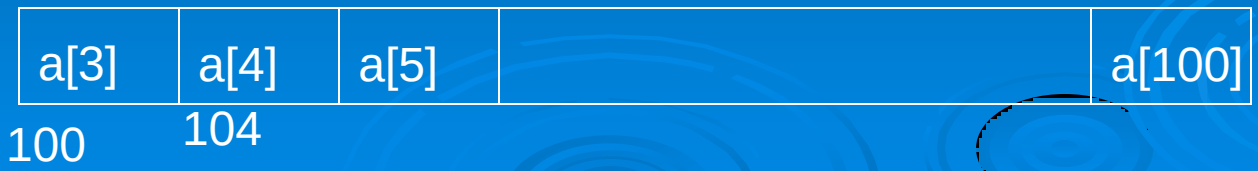
$b + (i - l) * s$

Przykład

a[3…100], każdy element jest reprezentowany przez 4 bajty.

- Liczba elementów: 100 – 3 + 1 = 98
- Rozmiar tablicy: 98 * 4 = 392
- Adres elementu a[50], zakładając, że obszar zaczyna się od adresu 100

100 + (50 - 3) * 4 = 288



Tablice jednowymiarowe

- Kod C: A[8] = h + A[8];
- Kod MIPS:
  - Założenia:
    - **$s3** zawiera adres pierwszego elementu **A (adres bazowy b)**
    - **$s2** zawiera wartość **h**

```
lw      $t0,32($s3)     # $t0 gets A[8]
                        # i-l =8, s=4,  (i-l)xs =8 x 4 =32
add     $t0,$s2,$t0     # add h
sw      $t0,32($s3)     # store value back in A[8]
```

Tablice jednowymiarowe

- Kod C :

  g = h + A [i];

- Kod MIPS:
  - Założenie: **$s4** zawiera **i**

```
        # zapisz wartość w $t1
add $t1, $s4, $s4       # $t1 = 2 *i
add $t1, $t1, $t1       # $t1 = 4 *i
   # Baza jest przechowywana  w $s3
   # Adres  A[i]
add    $t1, $t1, $s3    #$t1=Adres(A[i])
   #zapisz A[i]
 lw   $t0, 0($t1)       # $t0 = A[i]
   # dodaj  A[i] do h
add $s1, $s2, $t0       # $s1 = h + A[i]
```

Tablice

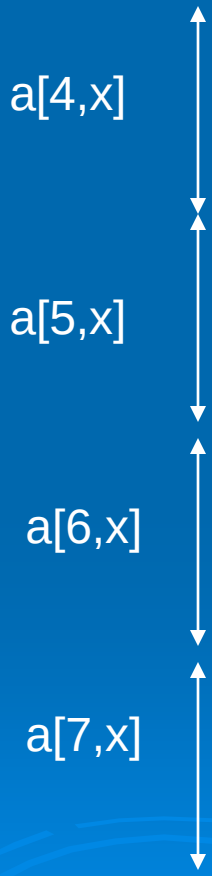
Przechowywanie wierszami i kolumnami dla a[4..6,3..4]

s oznacza liczbę bajtów

Adres	Wiersz	Kolumna
b + 0s	a[4,3]	a[4,3]
b + 1s	a[4,4]	a[5,3]
b + 2s	a[5,3]	a[6,3]
b + 3s	a[5,4]	a[4,4]
b + 4s	a[6,3]	a[5,4]
b + 5s	a[6,4]	a[6,4]

Tablice dwuwymiarowe: przechowywanie wierszami

A[4..7,3..4]



Adres	Wiersz
b + 0s	a[4,3]
b + 1s	a[4,4]
b + 2s	a[5,3]
b + 3s	a[5,4]
b + 4s	a[6,3]
b + 5s	a[6,4]
b + 6s	a[7,3]
b + 7s	a[7,4]

Tablice dwuwymiarowe: przechowywanie wierszami

a[$l_1..h_1, l_2..h_2$] każdy element jest reprezentowany przez s bajtów

Liczba elementów: $e = e_1 * e_2$, gdzie

$e_1 = (h_1 - l_1 + 1)$ i $e_2 = (h_2 - l_2 + 1)$

Rozmiar tablicy: $e * s$

Tablice dwuwymiarowe: przechowywanie wierszami

Rozmiar każdego wymiaru:

$d_1 = e_2 * d_2$, $e_2 = (h_2 - l_2 + 1)$

$d_2 = s$

Adres elementu a[i, j] z bazą *b*,

$l_1 <= i <= h_1$, $l_2 <= j <= h_2$:

$b + (i - l_1) * d_1 + (j - l_2) * s$

e_2 określa liczbę słów w jednym wierszu

d_1 określa liczbę bajtów, które zajmuje jeden wiersz

Liczba bajtów, które zajmują ($i - l_1$) wierszy

Liczba bajtów, które zajmują ($j - l_2$) słów

Przykład

A[3…100,4…50], każdy element jest reprezentowany przez 4 bajty.

- 98*47 = 4606 elementów
- 4606 * 4 = 18424 bajtów
- $d_2 = 4$, $d_1 = 47 * 4 = 188$
- Dla b=100, adres a[5,5]:

  100+(5-3) * 188 + (5 – 4) * 4 = 720

Tablice dwuwymiarowe, SPIM

a[3,5] : przechowywanie wierszami

Adres=b + (i - l_1) * d_1 + (j – l_2) * s

- Przydział

```
        .data
a:      .space 60       # 3x5=15 word-size elements * 4
```

- Obliczenie adresu:

```
#calculate the address of a[x,y] word size elements
la $t0,a                #baza b w  $t0
lw $t1,x                # x w $t1
mul $t1,$t1,20          # (x - l1) * d1 , d1=20
add $t0,$t0,$t1         # b+ (x- l1)*d1 w $t0
lw $t1,y                # y w $t1
mul $t1,$t1,4           # (j – l2) * s, s=4
add $t0,$t0,$t1         # Adres dla a[x,y]: b + (i - l1) * d1 + (j – l2) * s
lw $t1,($t0)            #t1 zawiera  a[x,y]
```

$d_1 = e_2 * d_2 = 5*4=20$
$e_2 = (h_2 - l_2 + 1)=5$
d_2 = s=4

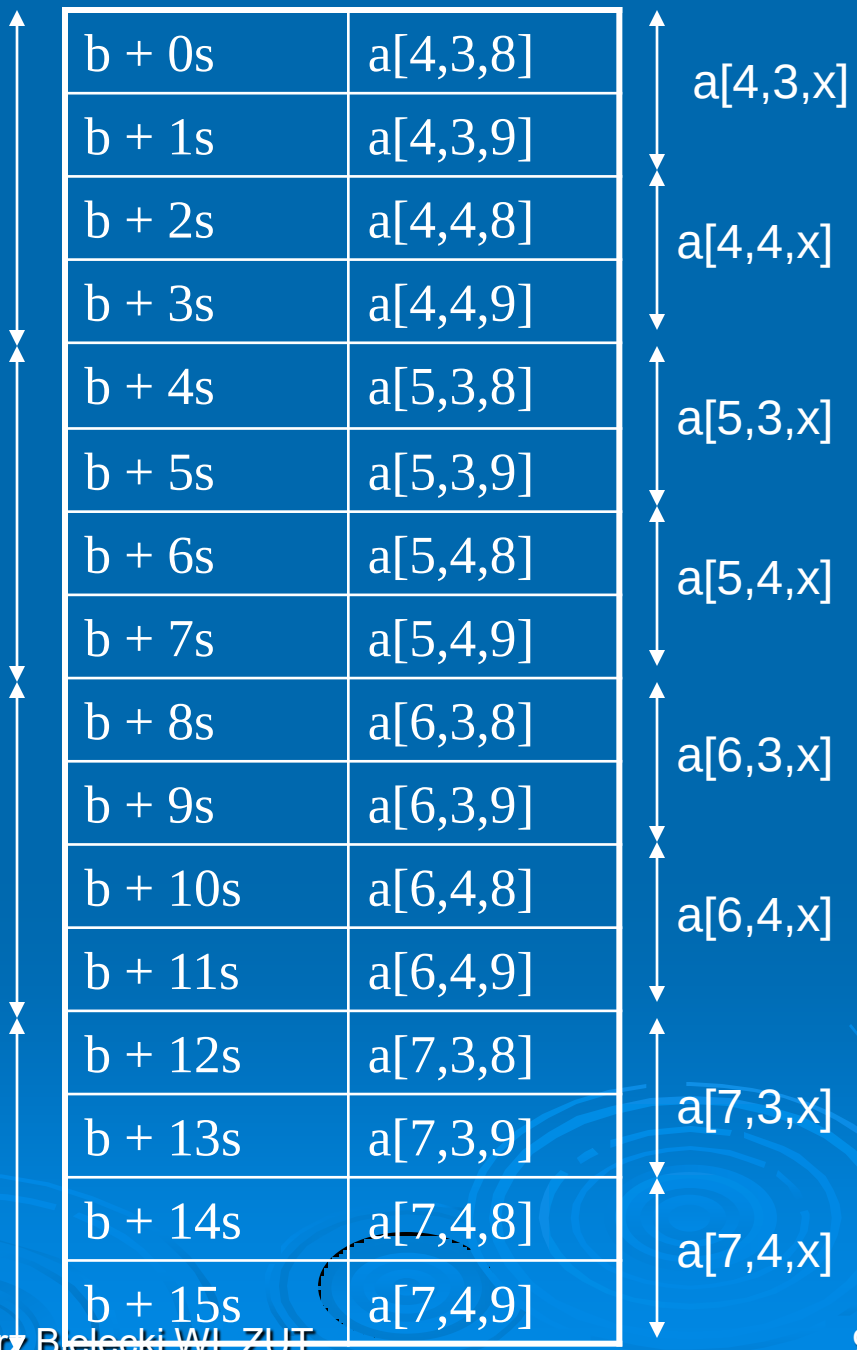
Tablice 3D

a[4..7,3..4,8..9]

Rozmiar trzeciego wymiaru= s

Rozmiar drugiego wymiaru= s*2

Rozmiar pierwszego wymiaru= s * 2 * 2



Tablice 3D. Przechowywanie wierszami

a[$l_1..h_1, l_2..h_2, l_3..h_3$] każdy element zajmuje *s* bajtów

Liczba elementów: $e = e_1 * e_2 * e_3$, where $e_i = (h_i - l_i + 1)$

Rozmiar tablicy: $e * s$

Rozmiar poszczególnych wymiarów:

$d_1 = e_2 * d_2$

$d_2 = e_3 * d_3$

$d_3 = s$

Tablice 3D. Przechowywanie wierszami

Adres elementu a[i,j,k] z bazą *b*,
$l_1 <= i <= h_1$ and $l_2 <= j <= h_2$:

$$b + (i - l_1) * d_1 + (j - l_2) * d_2 + (k - l_3) * s$$

Przykład:

A[3…100,4…50,1..4] każdy element zajmuje 4 bajty

98*47* 4 = 18424 elementów

18424 * 4 = 73696 bajtów

$d_3 = 4$, $d_2 = 4 * 4 = 16$, $d_1 = 16 * 47 = 752$

Dla b=100, adres elementu a[5,5,2] :

100+(5-3) * 752 + (5 – 4) * 16 + (2 – 1)*4 = 1624

Przetwarzanie konstrukcji sterowania

- Konstrukcje:
  - If
  - While
  - Repeat
  - For
  - case
- Generacja etykiet - wszystkie etykiety muszą być unikatowe

Instrukcje warunkowe

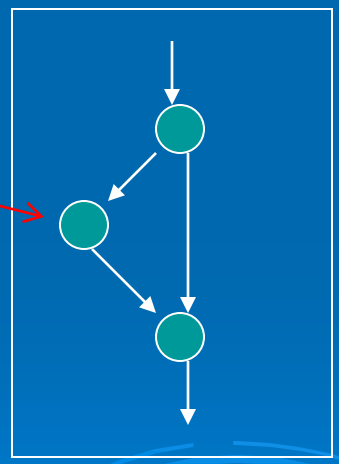
if (y > 0) then begin

…body…

end

```
lw $t0,y
li $t1,0
sgt $t2,$t0,$t1   # = 1 if true
beqz $t2,L2
…body…

L2:
```



Przepływ danych

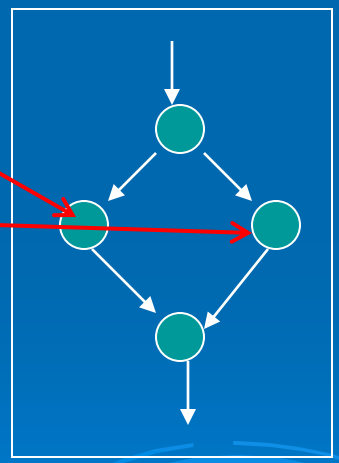
Instrukcje warunkowe

if (y > 0) then begin

… body_1 …

end else
 …body_2 …
end

```
lw $t0,y
li $t1,0
sgt $t2,$t0,$t1   # = 1 if true
beqz $t2,L2
…body_1…
b L3
L2:
…body_2 …
L3:
```



Przepływ danych

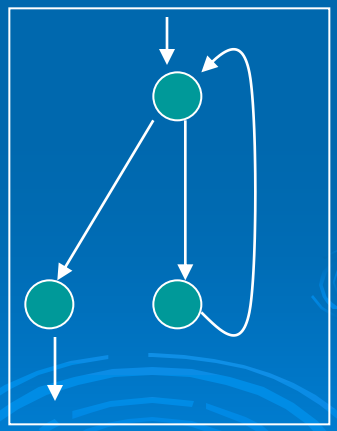
Pętle

while x < 100 do

… body …

end

```
L25:    lw $t0,x
        li $t1,100
        sgt $t2,$t0,$t1
        beqz $t2,L26
        … body …
        b L25
L26:
```



Przepływ danych

Schemat ogólny dla instrukcji warunkowych

Wartość expr

if_stmt → IF expr THEN

```
kod do obliczenia  expr ($2),
utwórz dwie nowe etykiety: L1, L2,
jeśli expr=flase ($2=false), to skok do L1,
ciało if_stmt
skok do L2
```

ELSE

```
L1:ciało else_stmt }
```

ENDIF

```
L2:...
```

Schemat ogólny dla pętli

for_stmt → FOR id = start TO stop

```
{ kod do obliczenia start ($1) i stop ($2),
   utwórz 2 etykiety L1, L2,
   utwórz kod dla instrukcji id = start,
   L1:kod  do  porównania id i stop ($3),
   jeśli ($3)=false, to skok do L2,
   kod dla ciała pętli,
   kod dla inkrementacji  id,
   skok do L1
END FOR
   L2:…..
```

Wywołanie procedur

Założenie: jedna funkcja(wywołująca) wywołuje drugą funkcję(wywoływaną).

Jakie są czynności realizowane przez pierwszą i drugą funkcje?

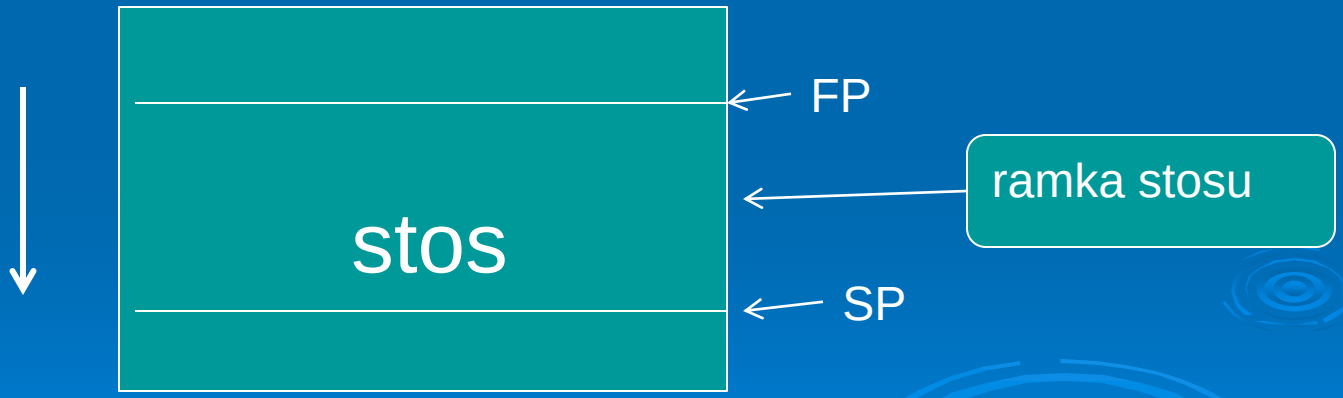
Wskaźniki SP i FP

Dla każdej funkcji trzeba przydzielić ramkę stosu.

FP wskazuje na początek bieżącej aktywacji(pierwsze słowo ramki stosu).

SP wskazuje na ostanie słowo ramki stosu.

Stos rośnie w kierunku niższych adresów, dlatego żeby zwiększyć stos trzeba zastosować operator !!! odejmowania



Wywołanie procedur

Funkcja wywołująca zapisuje argumenty funkcji wywoływanej w standardowych miejscach i wykonuje następujące czynności:

1. Przekazuje argumenty.

Zgodnie z konwencją, pierwsze cztery argumenty przekazywane są do rejestrów $a0–$a3.

Wszystkie pozostałe argumenty są odkładane na stos i pojawiają się na początku stosu.

Wywołanie procedur

2. Procedura wywoływana może korzystać z następujących rejestrów ($a0–$a3 i $t0–$t9). Jeśli funkcja wywołująca zamierza korzystać z tych rejestrów, to musi zapisać zawartość tych rejestrów w pamięci przed wywołaniem.
3. Wykonuje instrukcję jal, która przekazuje sterowanie do pierwszej instrukcji funkcji wołanej i zapisuje adres powrotu w rejestrze **$ra**.

Wywołanie procedur

Przed wykonywaniem obliczeń, funkcja wywoływana musi wykonać następujące kroki

1. Przydzielić obszar pamięci(ramkę) poprzez odjęcie wielkości ramki od wskaźnika stosu FS(stos zaczyna się od większych adresów).

Wywołanie procedur

2. Zapisać zawartość rejestrów funkcji wywoływanej w przydzielonym obszarze pamięci (ramce).

   Funkcja wywoływana musi zapisać w ramce dane przechowywane w rejestrach ($s0–$s7, $fp, $ra) przed korzystaniem z tych rejestrów ponieważ funkcja wywołująca spodziewa się korzystać z danych w tych rejestrach po zakończeniu wykonania funkcji wywoływanej.

Wywołanie procedur

2. Cd

Zawartość rejestru $fp musi być zapisana w pamięci przez każdą procedurę, która przydziela obszar pamięci dla stosu.

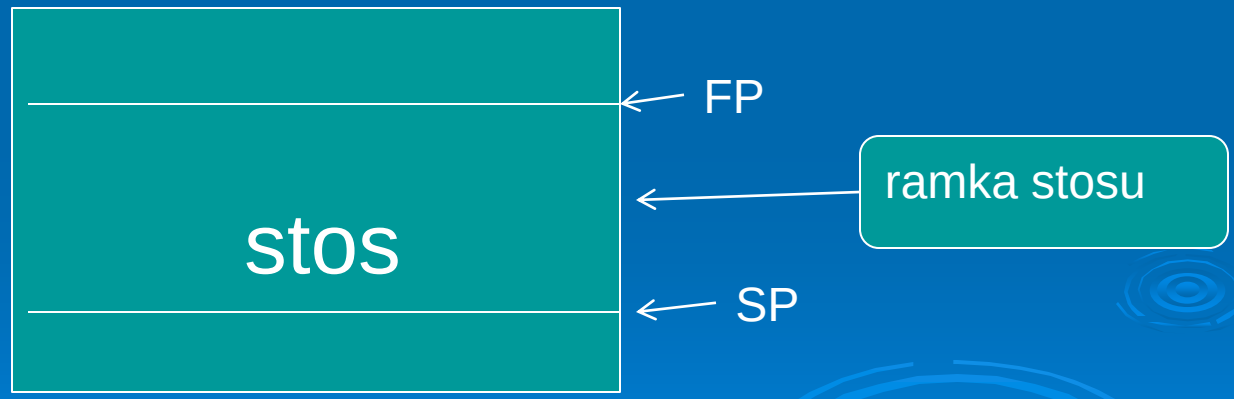
Natomiast zawartość rejestru $ra musi być zapisywana tylko wtedy gdy funkcja wywoływana sama wywołuje inną funkcję.

Zawartość pozostałych rejestrów, z których korzysta funkcja wywoływana, muszą być także zapisane.

Wywołanie procedur

3. Ustawić wskaźnik stosu dodając rozmiar ramki minus 4 (zwiększa stos o 4 bajty) do zawartości rejestru $sp i zapisać wynik w rejestrze $fp.

Stos rośnie w kierunku niższych adresów, dlatego żeby zwiększyć stos trzeba zastosować operator !!! odejmowania



Wywołanie procedur

Zakończenie:

1. Jeśli funkcja wywoływana zwraca wartość, to zapisuje ją w rejestrze $v0.
2. Przywraca zawartość wszystkich rejestrów, które zostały zapisane w momencie wywołania procedury.
3. Zdejmuje ramkę stosu dodając rozmiar ramki do $sp.
4. Powrót do adresu podanego w rejestrze $ra.

Wywołanie procedur

Przykład w C

```
int main()
{
  x=addthem(a,b);
}
 int addthem(int a, int b)
{
   return a+b;
 }
```

Wywołanie procedur

```
Kod   SPIM:
.text
main:  #założenia: a jest  w $t0, b jest  w $t1
   add $a0,$0, $t0
   add $a1,$0,$t1
   jal addthem    # wywołanie funkcji addthem
  add $t3,$0,$v0    # gdy funkcja wywoływana zwróci
                    # wartość do $v0, jest ona   przesłana do $t3
```

Przekazanie argumentów do rejestrów $a0, $a1

Miejsce powrotu

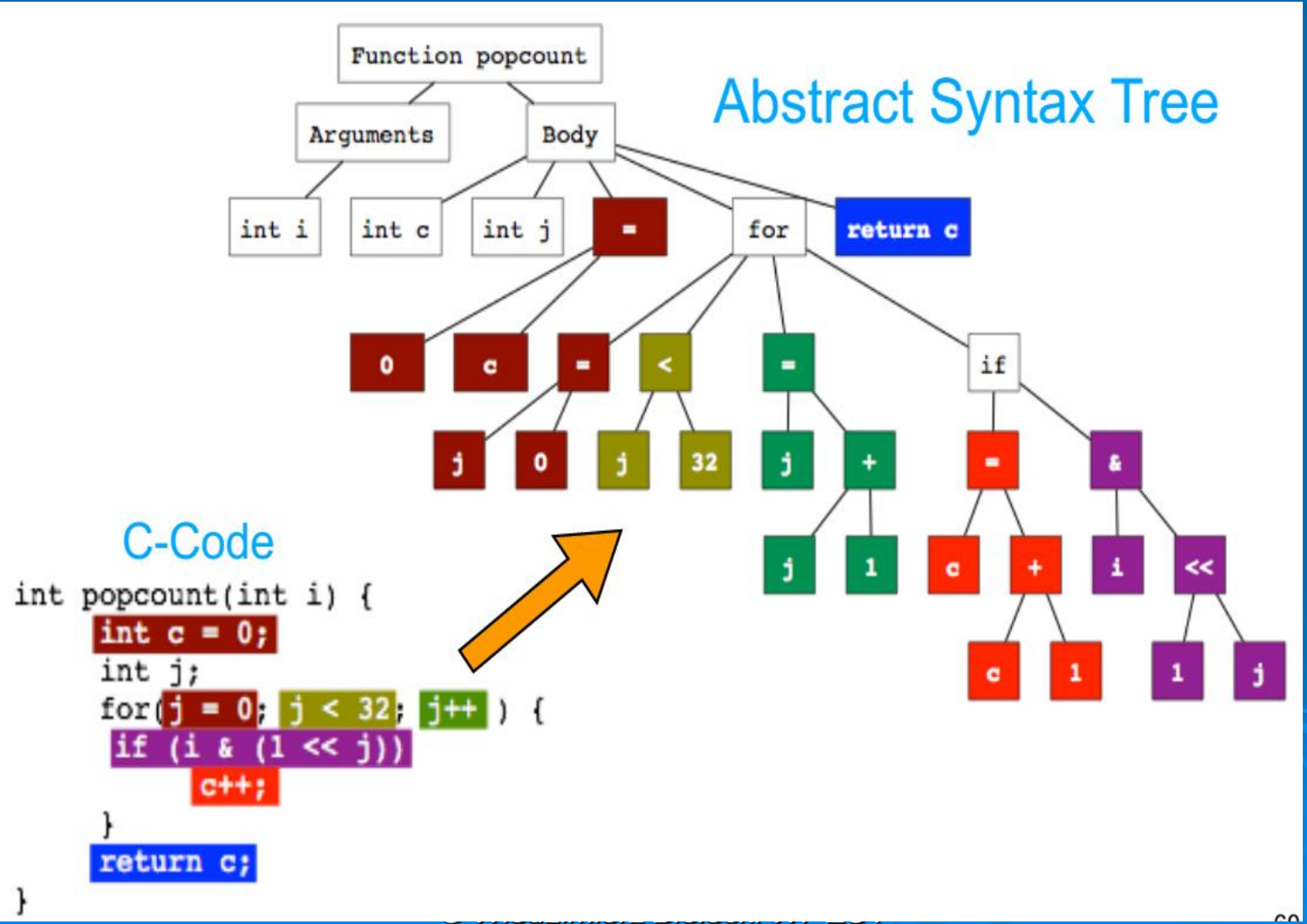
Wywołanie procedur

Kod SPIM: cd

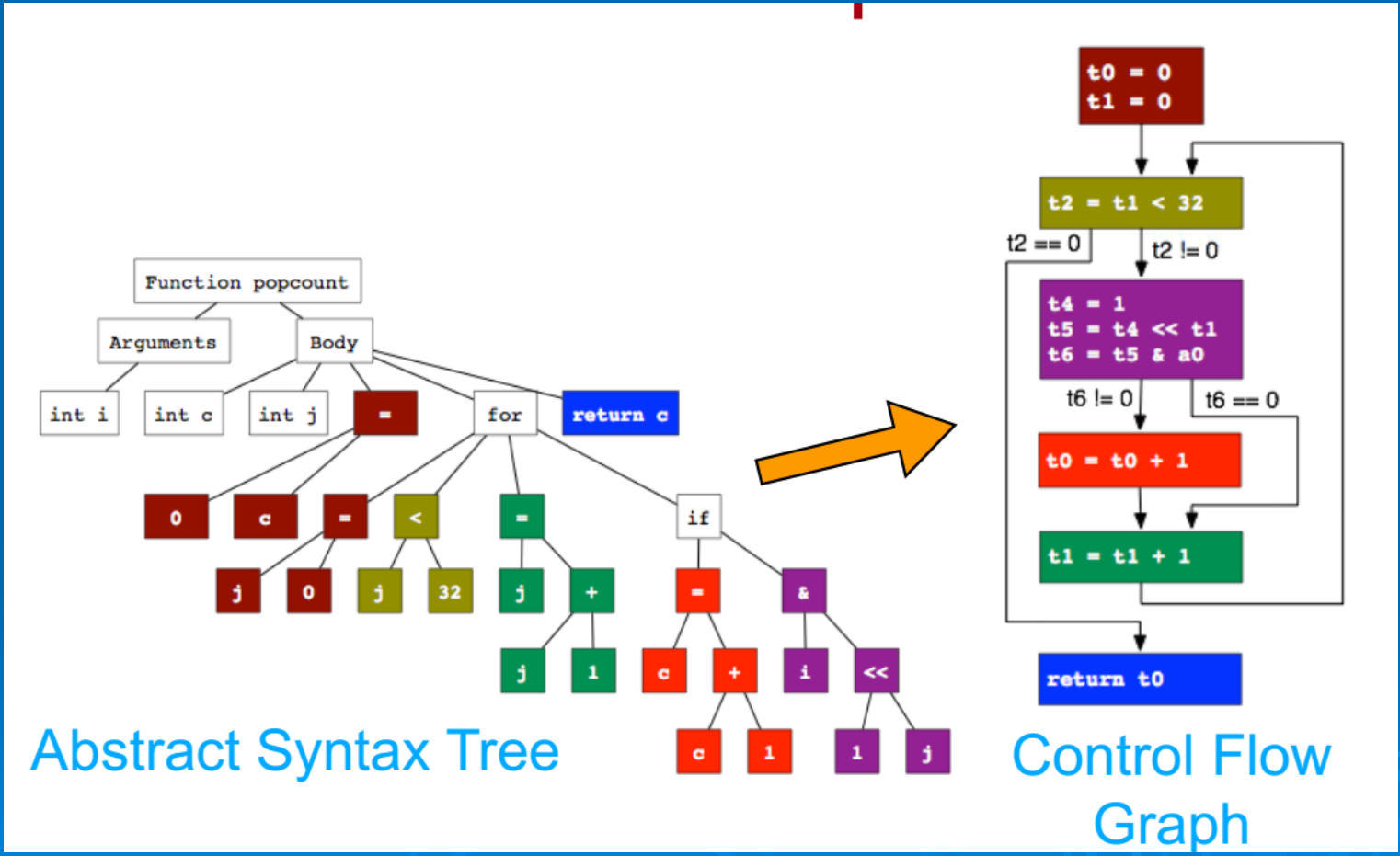
```
syscall
    addthem:
   addi $sp,$sp,-4   # zarezerwowanie ramki stosu
    sw $t0, 0($sp)   # zapis poprzedniej wartości ($t0)
    add $t0,$a0,$a1 # instrukcja implementująca ciało
                     # funkcji
    add $v0,$0,$t0   # wynik
    lw $t0, 0($sp)   # ładowanie poprzedniej wartości
    addi $sp,$sp,4   # Zdejmij  ramkę ze stosu
    jr $ra           # powrót
```

Funkcja wymaga ramki w stosie (4 bajty) do zapisania wartości rejestru t0, którą należy przywrócić po zakończeniu obliczeń funkcji

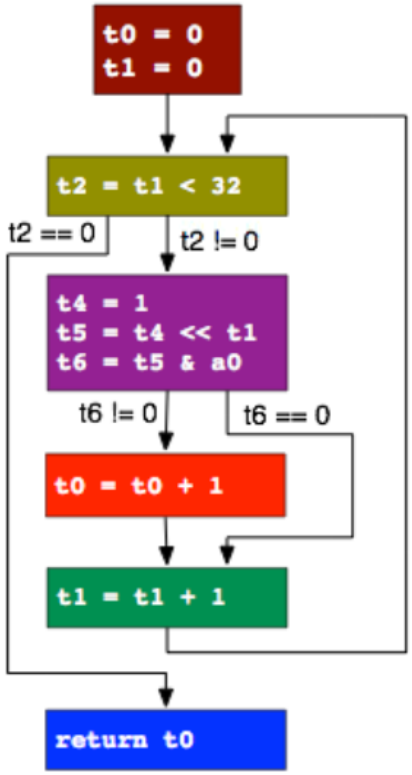
Przykład generowania kodu



Przykład generowania kodu



Przykład generowania kodu



ori -Operator OR

nop –pusty operator

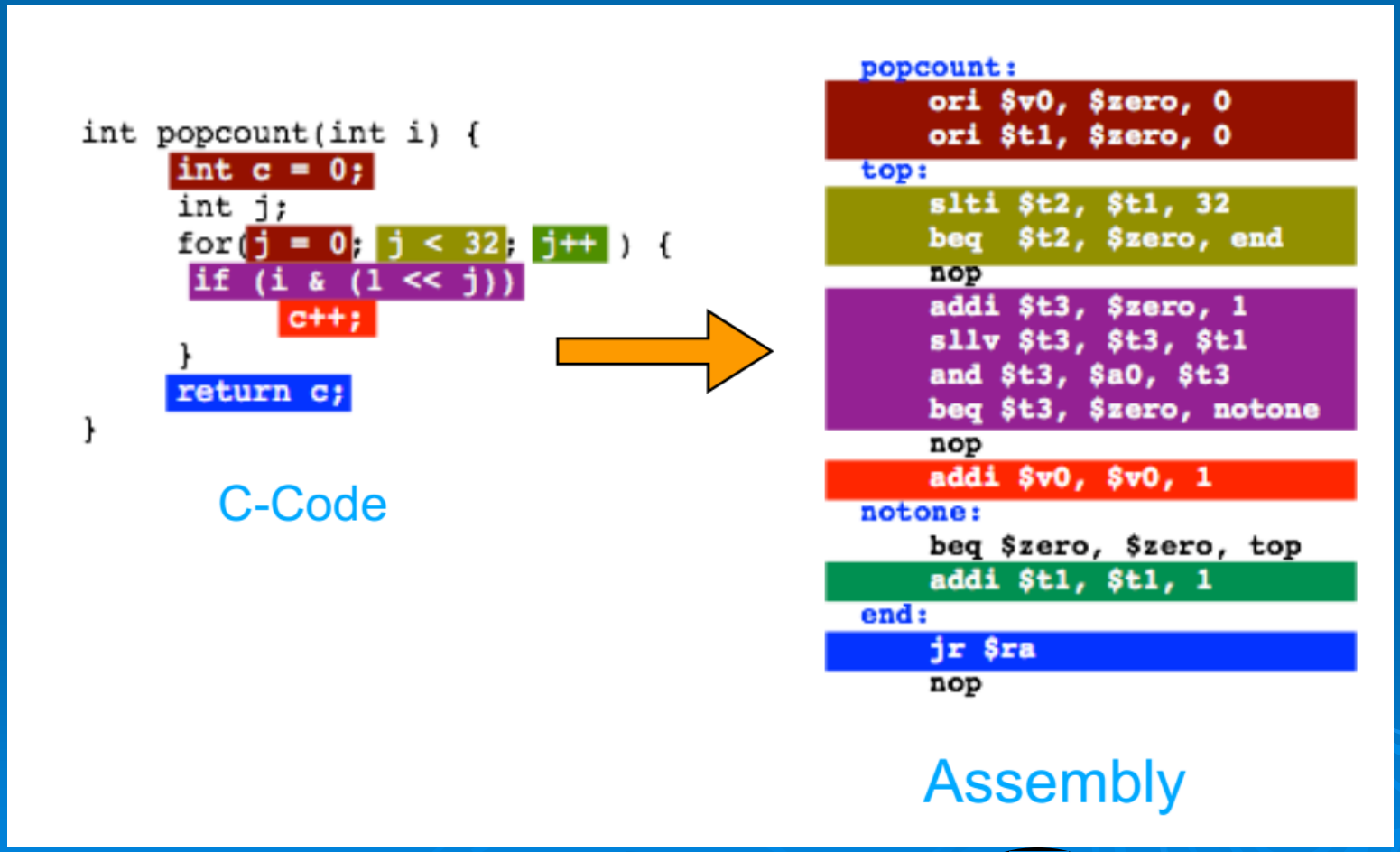
sllv - shift left logical variable

```
popcount:
        ori $v0, $zero, 0
        ori $t1, $zero, 0
top:
        slti $t2, $t1, 32
        beq  $t2, $zero, end
        nop
        addi $t3, $zero, 1
        sllv $t3, $t3, $t1
        and $t3, $a0, $t3
        beq $t3, $zero, notone
        nop
        addi $v0, $v0, 1
notone:
        beq $zero, $zero, top
        addi $t1, $t1, 1
end:
        jr $ra
        nop
```

Control flow graph

Assembly

Przykład generowania kodu



Przykłady kodu

Kolejne 2 slajdy przedstawiają przykłady kodów w asemblerze.

Dodawanie 2 liczb

```
#       $t2     - used to hold the sum of the $t0 and $t1.
#       $v0     - syscall number, and syscall return value.
#       $a0     - syscall input parameter.

        .text                   # Code area starts here
main:
        li      $v0, 5          # read number into $v0
        syscall                 # make the syscall read_int
        move    $t0, $v0        # move the number read into $t0

        li      $v0, 5          # read second number into $v0
        syscall                 # make the syscall read_int
        move    $t1, $v0        # move the number read into $t1

        add     $t2, $t0, $t1

        move    $a0, $t2        # move the number to print into $a0
        li      $v0, 1          # load syscall print_int into $v0
        syscall                 #

        li      $v0, 10         # syscall code 10 is for exit
        syscall                 #
# end of main
```

Dodawanie N liczb

```
# Input: number of inputs, n, and n integers;   Output: Sum of integers
            .data                       # Data memory area.
prmpt1:     .asciiz "How many inputs? "
prmpt2:     .asciiz "Next input: "
sumtext:    .asciiz "The sum is "
            .text                       # Code area starts here
main:       li   $v0, 4                 # Syscall to print prompt string
            la   $a0, prmpt1            # li and la are pseudo instr.
            syscall
            li   $v0, 5                 # Syscall to read an integer
            syscall
            move $t0, $v0               # n stored in $t0

            li   $t1, 0                 # sum will be stored in $t1
while:      blez $t0, endwhile          # (pseudo instruction)
            li   $v0, 4                 # syscal to print string
            la   $a0, prmpt2
            syscall
            li   $v0, 5
            syscall
            add  $t1, $t1, $v0          # Increase sum by new input
            sub  $t0, $t0, 1            # Decrement n
            j    while

endwhile:   li   $v0, 4                  # syscal to print string
            la   $a0, sumtext
            syscall
            move $a0, $t1                # Syscall to print an integer
            li   $v0, 1
            syscall
            li   $v0, 10                 # Syscall to exit
            syscall
```

Dziękuję za uwagę