Procedury

```
main() {
    int i,j,k,m;
    ...
    i = mult(j,k); ...
    m = mult(i,i); ...
  }
```

```
int mult (int mcand, int mlier){
      int product = 0;
      while (mlier > 0) {
      product = product + mcand;
      mlier = mlier -1; }
      return product;
}
```

Jak wykonywane są obliczenia ?
Jak zapewniamy komunikację ?

Procedury – kolejne kroki

- **Umieść argumenty tak aby były dostępne dla procedury**
- **Przenieś sterowanie do procedury**
- **Zaalokuj pamięć dla procedury**
- **Wykonaj instrukcje ciała procedury**
- **Umieść wyniki tak aby były dostępne dla programu ją wywołującego**
- **Przenieść sterowanie do „wymaganego” miejsca w programie wywołującym**

- Przyporządkowanie rejestrów
  - Adres powrotu: **$ra**
  - Argumenty: **$a0, $a1, $a2, $a3**
  - Wyniki: **$v0, $v1**
  - Rejestry robocze (lokalne): **$s0, $s1, … , $s7**

Procedury - przykład

C

```
... sum(a,b);...            /* a,b:$s0,$s1 */
}
.....
int sum(int x, int y) {
return x+y;
}
```

MIPS

```
1000    add $a0,$s0,$zero       # x = a
1004    add $a1,$s1,$zero       # y = b
1008    addi $ra,$zero,1016     #$ra=1016
1012    j sum                   #jump to sum
1016    ...
2000    sum: add $v0,$a0,$a1
2004    jr $ra                  # nowa instrukcja
```

Procedury - przykład

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
    int sum(int x, int y) {
    return x+y;
}
```

MIPS

```
2000 sum: add $v0,$a0,$a1
2004 jr $ra                    # nowa instrukcja
```

Dlaczego używamy nowej instrukcji zamiast znanej „j” (jump) ?

Procedury - przykład

Ale przydałaby się jeszcze jedna nowa instrukcja:

realizuje skok oraz zapamiętuje adres powrotu

jal (jump and link)

Bez tej instrukcji

```
1008 addi $ra,$zero,1016    # $ra=1016
1012 j sum                  # goto sum
```

Z tą instrukcją

```
1008 jal sum                # $ra=1012, goto sum
```

Instrukcje „jal” oraz „jr”

Dla „jal” składnia taka sama jak dla „j”

jal etykieta

Znaczenie

Krok 1 zapamiętaj adres kolejnej instrukcji w rejestrze $ra

Dlaczego kolejnej, a nie bieżącej ?

Krok 2 przenieś sterowanie zgodnie z etykietą.

Składnia dla instrukcji „jr” odmienna

jr rejestr

Zamiast etykiety podajemy nazwę rejestru w którym znajduje się adres instrukcji do wykonania.

Obie instrukcje stanowią parę instrukcji wykorzystywanych do implementacji procedur

jal zapamiętuje adres powrotu w rejestrze, jr realizuje skok do tego adresu

Zagnieżdżone wywołanie procedur

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

Co zrobić gdy procedura sum Square jest wywoływana przez procedurę mult ?

Przecież rejestr adresu powrotu $ra ustawiony podczas wywołania procedury sumSquare zostanie nadpisany przez procedurę mult.

Trzeba jakoś zachować adres powrotu dla sumSquare.

7

Zagnieżdżone wywołanie procedur

W czasie wykonania programu w C mamy trzy następujące możliwości:

- Obszar statyczny programu, przykładowo zmienne globalne
- Sterta (heap) - zmienne deklarowane dynamicznie poprzez malloc
- Stos (stack) - obszar wykorzystywany przez procedury (funkcje) podczas ich wykonania - tutaj możemy przechować rejestry

Zagnieżdzone wywołanie procedur

- **Rejestr $sp zawsze wskazuje na szczyt stosu – wskaźnik szczytu stosu (ostatni zajęty obszar)**
- **Aby użyć stosu należy odpowiednio zmniejszyć wartość wskaźnika a następnie zapamiętać niezbędne dane.**

Jak zaimplementować poniższy kod w asemblerze ?

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
  }
```


9

Zagnieżdzone wywołanie procedur

```
sumSquare:
        addi $sp,$sp,-8          # space on stack
        sw $ra, 4($sp)           # save ret addr
        sw $a1, 0($sp)           # save y
        add $a1,$a0,$zero        # mult(x,x)
        jal mult                 # call mult
        lw $a1, 0($sp)           # restore y
        add $v0,$v0,$a1          # mult()+y
        lw $ra, 4($sp)           # get ret addr
        addi $sp,$sp,8           # restore stack
        jr $ra
mult: ...
```

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```


10

Kroki podczas wykonywania procedury

1. Zapisanie niezbędnych danych na stosie.
2. Przypisanie odpowiednich argumentów.
3. **jal** call
4. Odczytanie danych ze stosu

11

Zasady „organizacji” procedur

- Wywołanie procedury instrukcją **jal** oraz powrót instrukcją **jr $ra**
- Do czterech argumentów: **$a0, $a1, $a2** oraz **$a3**
- Wynik zawsze w **$v0**, ewentualnie jeśli potrzeba to w **$v1**
- Zawsze należy stosować się do przyjętej konwencji

12

Struktura funkcji

```
entry_label:
        addi $sp,$sp, -niezbedny_rozmiar
        sw $ra, niezbedny_rozmiar-4($sp)   # save $ra
        Zapamiętaj inne rejestry jeśli potrzeba
Ciało funkcji.....
        Otworz rejetry
        lw $ra, niezbedny_rozmiar-4($sp)   # restore $ra
        addi $sp,$sp, niezbedny_rozmiar
        jr $ra
```


13

Jeszcze raz konwencja stosowania rejestrów

The constant 0	**$0**	**$zero**
Reserved for Assembler	**$1**	**$at**
Return Values	**$2 - $3**	**$v0 - $v1**
Arguments	**$4 - $7**	**$a0 - $a3**
Temporary	**$8 - $15**	**$t0 - $t7**
Saved	**$16 - $23**	**$s0 - $s7**
More Temporary	**$24 - $25**	**$t8 - $t9**
Used by Kernel	**$26 - $27**	**$k0 - $k1**
Global Pointer	**$28**	**$gp**
Stack Pointer	**$29**	**$sp**
Frame Pointer	**$30**	**$fp**
Return Address	**$31**	**$ra**

14

Przykład 1/3

```
main() {
      int i,j,k,m;   /* i-m : $s0-$s3 */
      ...
      i = mult(j,k); ...
      m = mult(i,i); ...
  }
int mult (int mcand, int mlier){
     int product;
     product = 0;
     while (mlier > 0) {
          product += mcand;
          mlier -= 1; }
     return product;
 }
```


15

Przykład 2/3

```
start:
    ...
    add $a0,$s1,$0      # arg0 = j
    add $a1,$s2,$0      # arg1 = k
    jal mult            # call mult
    add $s0,$v0,$0      # i = mult()
    ...
    add $a0,$s0,$0      # arg0 = i
    add $a1,$s0,$0      # arg1 = i
    jal mult            # call mult
    add $s3,$v0,$0      # m = mult()
    ...
    j   exit
```

```
main() {
   int i,j,k,m;      /* i-m:$s0-$s3 */
   ...
   i = mult(j,k); ...
   m = mult(i,i); ... }
```


16

Przykład 3/3

```
mult:
    add $t0,$0,$0       # prod=0
Loop:
    slt $t1,$0,$a1      # mlr > 0?
    beq $t1,$0,Fin      # no=>Fin
    add $t0,$t0,$a0     # prod+=mc
    addi $a1,$a1,-1     # mlr-=1
    j Loop              # goto Loop
Fin:
    add $v0,$t0,$0      # $v0=prod
    jr $ra              # return
```

```
int mult (int mcand, int mlier){
        int product = 0;
        while (mlier > 0) {
        product += mcand;
        mlier -= 1; }
        return product;
 }
```


17

Lista instrukcji

• **add** Rd, Rs, Rt	Rd = Rs + Rt (signed)
• **addu** Rd, Rs, Rt	Rd = Rs + Rt (unsigned)
• **addi** Rd, Rs, Imm	Rd = Rs + Imm (signed)
• **sub** Rd, Rs, Rt	Rd = Rs - Rt (signed)
• **subu** Rd, Rs, Rt	Rd = Rs - Rt (unsigned)
• **div** Rs, Rt	lo = Rs/Rt, hi = Rs **mod** Rt (integer division, signed)
• **divu** Rs, Rt	lo = Rs/Rt, hi = Rs **mod** Rt (integer division, unsigned)
div Rd, Rs, Rt	Rd = Rs/Rt (integer division, signed)
divu Rd, Rs, Rt	Rd = Rs/Rt (integer division, unsigned)
rem Rd, Rs, Rt	Rd = Rs **mod** Rt (signed)
remu Rd, Rs, Rt	Rd = Rs **mod** Rt (unsigned)
mul Rd, Rs, Rt	Rd = Rs * Rt (signed)
• **mult** Rs, Rt	hi, lo = Rs * Rt (signed, hi = high 32 bits, lo = low 32 bits)

Lista instrukcji

• **multu** Rd, Rs	hi, lo = Rs * Rt (unsigned, hi = high 32 bits, lo = low 32 bits)
• **and** Rd, Rs, Rt	Rd = Rs • Rt
• **andi** Rd, Rs, Imm	Rd = Rs • Imm
neg Rd, Rs	Rd = -(Rs)
• **nor** Rd, Rs, Rt	Rd = (Rs + Rt)’
not Rd, Rs	Rd = (Rs)’
• **or** Rd, Rs, Rt	Rd = Rs + Rt
• **ori** Rd, Rs, Imm	Rd = Rs + Imm
• **xor** Rd, Rs, Rt	Rd = Rs Rt
• **xori** Rd, Rs, Imm	Rd = Rs Imm
• **sll** Rd, Rt, Sa	Rd = Rt left shifted by Sa bits
• **sllv** Rd, Rs, Rt	Rd = Rt left shifted by Rs bits
• **srl** Rd, Rs, Sa	Rd = Rt right shifted by Sa bits
• **srlv** Rd, Rs, Rt	Rd = Rt right shifted by Rs bits
move Rd, Rs	Rd = Rs

19

Lista instrukcji

- **mfhi** Rd — Rd = hi
- **mflo** Rd — Rd = lo

li Rd, Imm — Rd = Imm

- **lui** Rt, Imm — Rt[31:16] = Imm, Rt[15:0] = 0
- **lb** Rt, Address(Rs) — Rt = byte at M[Address + Rs] (sign extended)
- **sb** Rt, Address(Rs) — Byte at M[Address + Rs] = Rt (sign extended)
- **lw** Rt, Address(Rs) — Rt = word at M[Address + Rs]
- **sw** Rt, Address(Rs) — Word at M[Address + Rs] = Rt
- **slt** Rd, Rs, Rt — Rd = 1 if Rs < Rt, Rd = 0 if Rs >= Rt (signed)
- **slti** Rd, Rs, Imm — Rd = 1 if Rs < Imm, Rd = 0 if Rs >= Imm (signed)
- **sltu** Rd, Rs, Rt — Rd = 1 if Rs < Rt, Rd = 0 if Rs >= Rt (unsigned)
- **beq** Rs, Rt, Label — Branch to Label if Rs == Rt

beqz Rs, Label — Branch to Label if Rs == 0

bge Rs, Rt, Label — Branch to Label if Rs >= Rt (signed)

- **bgez** Rs, Label — Branch to Label if Rs >= 0 (signed)

· **bgezal** Rs, Label — Branch to Label and Link if Rs >= Rt (signed)

20

Lista instrukcji

• **bgt** Rs, Rt, Label	Branch to Label if Rs > Rt (signed)
bgtu Rs, Rt, Label	Branch to Label if Rs > Rt (unsigned)
• **bgtz** Rs, Label	Branch to Label if Rs > 0 (signed)
ble Rs, Rt, Label	Branch to Label if Rs <= Rt (signed)
bleu Rs, Rt, Label	Branch to Label if Rs <= Rt (unsigned)
• **blez** Rs, Label	Branch to Label if Rs <= 0 (signed)
• **bgezal** Rs, Label	Branch to Label and Link if Rs >= 0 (signed)
• **bltzal** Rs, Label	Branch to Label and Link if Rs < 0 (signed)
blt Rs, Rt, Label	Branch to Label if Rs < Rt (signed)
bltu Rs, Rt, Label	Branch to Label if Rs < Rt (unsigned)
• bltz Rs, Label	Branch to Label if Rs < 0 (signed)
• **bne** Rs, Rt, Label	Branch to Label if Rs ≠ Rt
bnez Rs, Label	Branch to Label if Rs ≠ 0
• **j** Label	Jump to Label unconditionally
• **jal** Label	Jump to Label and link unconditionally
• **jr** Rs	Jump to location in Rs unconditionally
• **jalr** Label	Jump to location in Rs and link unconditionally

21