# Chương 5

# Lập trình hợp ngữ MIPS

# Nội dung

- Abc
- Def
- Ghk
- …

# Xử lý vòng lặp (tiếp)

- Xét mảng int A[]. Giả sử ta có vòng lặp trong C:

```
do {
        g = g + A[i];
        i  = i + j;
} while (i != h);
```

- Ta có thể viết lại:

```
Loop:   g = g + A[i];
        i  = i + j;
        if (i != h) goto Loop;
```

→ Sử dụng lệnh rẽ có điều kiện để biểu diễn vòng lặp!

# Xử lý vòng lặp

- Ánh xạ biến vào các thanh ghi như sau:

| g | h | i | j | base address of A |
|---|---|---|---|---|
| $s1 | $s2 | $s3 | $s4 | $s5 |

- Trong ví dụ trên có thể viết lại thành lệnh MIPS như sau:

```
Loop:   sll   $t1, $s3, 2      # $t1 = i * 2^2
        add   $t1, $t1, $s5    # $t1 = addr A[i]
        lw    $t1, 0 ($t1)     # $t1 = A[i]
        add   $s1, $s1, $t1    # g = g + A[i]
        add   $s3, $s3, $s4    # i = i + j
        bne   $s3, $s2, Loop   # if (i != j) goto Label
```

# Xử lý vòng lặp

- Tương tự cho các vòng lặp phổ biến khác trong C:
  - while
  - for
  - do…while
- Nguyên tắc chung:
  - Viết lại vòng lặp dưới dạng goto
  - Sử dụng các lệnh MIPS rẽ nhánh có điều kiện

# So sánh không bằng

- beq và bne được sử dùng để so sánh bằng (== và != trong C)
- → Muốn so sánh lớn hơn hay nhỏ hơn?
- MIPS hỗ trợ lệnh so sánh không bằng:
  - slt  opr1, opr2, opr3
  - slt: Set on Less Than
  - if (opr2 < opr3)

    opr1 = 1;

    else

    opr1 = 0;

# So sánh không bằng

- Trong C, câu lệnh sau:

  **if (g < h) goto Less;** # g: $s0, h: $s1

- Được chuyển thành lệnh MIPS như sau:

```
slt   $t0, $s0, $s1      # if (g < h) then $t0 = 1
bne   $t0, $0, Less      # if ($t0 != 0) goto Less
                         # if (g < h) goto Less
```

- Nhận xét: Thanh ghi $0 luôn chứa giá trị 0, nên lệnh bne và bep thường dùng để so sánh sau lệnh slt

# Các lệnh so sánh khác

- Các phép so sánh còn lại như >, ≥, ≤ thì sao?
- MIPS không trực tiếp hỗ trợ cho các phép so sánh trên, tuy nhiên dựa vào các lệnh slt, bne, beq ta hoàn toàn có thể biểu diễn chúng!

# a: $s0, b: $s1

- **a < b**

```
slt   $t0, $s0, $s1     # if (a < b) then $t0 = 1
bne   $t0, $0, Label    # if (a < b) then goto Label
<do something>          # else then do something
```

- **a > b**

```
slt   $t0, $s1, $s0     # if (b < a) then $t0 = 1
bne   $t0, $0, Label    # if (b < a) then goto Label
<do something>          # else then do something
```

- **a ≥ b**

```
slt   $t0, $s0, $s1     # if (a < b) then $t0 = 1
beq   $t0, $0, Label    # if (a ≥ b) then goto Label
<do something>          # else then do something
```

- **a ≤ b**

```
slt   $t0, $s1, $s0     # if (b < a) then $t0 = 1
beq   $t0, $0, Label    # if (b ≥ a) then goto Label
<do something>          # else then do something
```

# Nhận xét

- So sánh == → Dùng lệnh beq
- So sánh != → Dùng lệnh bne
- So sánh < và > → Dùng cặp lệnh (slt → bne)
- So sánh ≤ và ≥ → Dùng cặp lệnh (slt → beq)

# So sánh với hằng số

- So sánh bằng: beq / bne
- So sánh không bằng: MIPS hỗ trợ sẵn lệnh slti
  - slti  opr, opr1, const
  - Thường dùng cho switch…case, vòng lặp for

# Ví dụ: switch ... case

- **switch (k) {**

  **case 0: f = i + j; break;**

  **case 1: f = g + h; break;**

  **case 2: f = g - h; break;**

  **}**

- Ta có thể viết lại thành các lệnh if lồng nhau:

  **if (k == 0)  f = i + j;**

  **else if (k == 1)  f = g + h;**

  **else if (k == 2)  f = g – h;**

- Ánh xạ giá trị biến vào các thanh ghi:

| f | g | h | i | j | k |
|---|---|---|---|---|---|
| $s0 | $s1 | $s2 | $s3 | $s4 | $s5 |

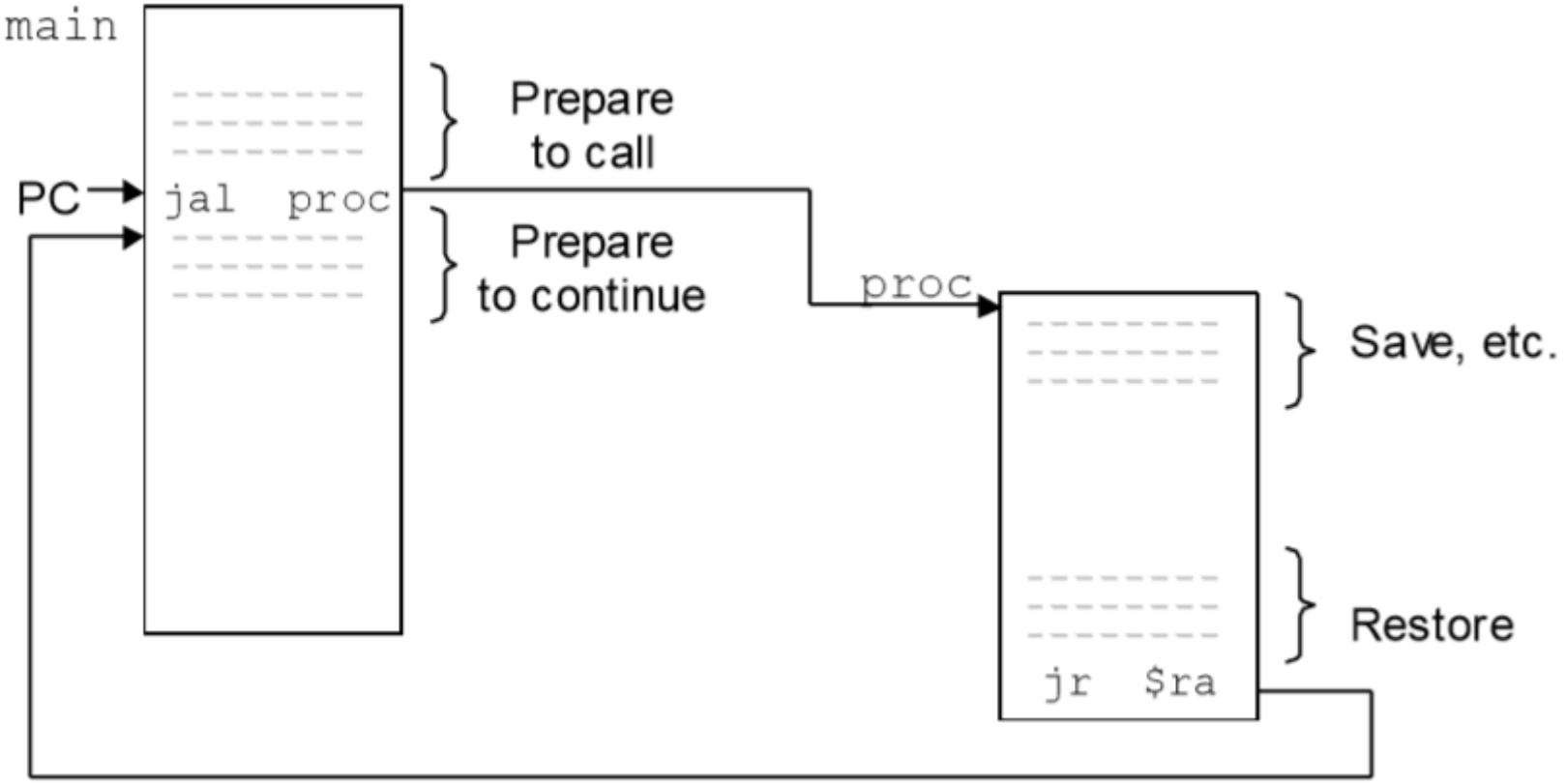
- Chuyển thành lệnh hợp ngữ MIPS:

```
        bne     $s5, $0, L1     # if (k != 0) then goto L1
        add     $s0, $s3, $s4   # else (k == 0) then f = i + j
        j       Exit            # end of case → Exit (break)
L1:     addi    $t0, $s5, -1    # $t0 = k – 1
        bne     $t0, $0, L2     # if (k != 1) then goto L2
        add     $s0, $s1, $s2   # else (k == 1) then f = g+ h
        j       Exit            # end of case → Exit (break)
L2:     addi    $t0, $s5, -2    # $t0 = k – 2
        bne     $t0, $0, Exit   # if (k != 2) then goto Exit
        sub     $s0, $s1, $s2   # else (k == 2) then f = g - h
Exit:   ….
```

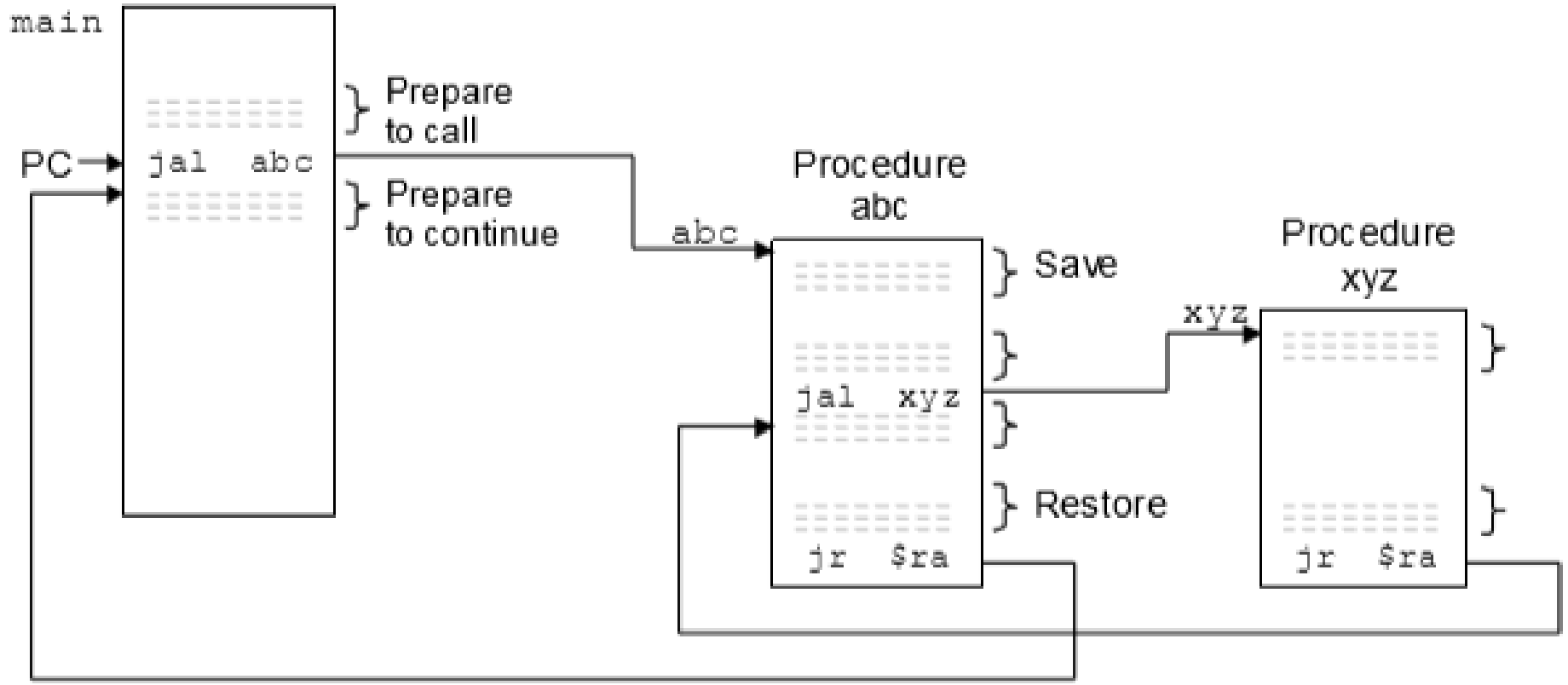
# Thủ tục (chương trình con)

- Các bước yêu cầu:
  1. Đặt các tham số vào các thanh ghi
  2. Chuyển điều khiển đến thủ tục
  3. Thực hiện các thao tác của thủ tục
  4. Đặt kết quả vào thanh ghi cho chương trình đã gọi thủ tục
  5. Trở về vị trí đã gọi

# Minh họa gọi thủ tục



# Gọi thủ tục lồng nhau



# Thủ tục (chương trình con)

- Hàm (fucntion) trong C → (Biên dịch) → Trình con (Thủ tục) trong hợp ngữ
- Giả sử trong C, ta viết như sau:

```
void main()
{
    int a, b;
    ...
    sum(a, b);
    ...
}

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

- Hàm được chuyển thành lệnh hợp ngữ như thế nào ?
- Dữ liệu được lưu trữ ra sao ?

**C**

```
... sum (a, b); ...                /* a: $s0, b: $s1 */
[Làm tiếp thao tác khác...]
}
int sum (int x, int y) {
    return x + y;
}
```

**MIPS**

| Địa chỉ | Lệnh | | |
|---|---|---|---|
| 1000 | add | $a0, $s0, $zero | # x = a |
| 1004 | add | $a1, $s1, $zero | # y = b |
| 1008 | addi | $ra, $zero, 1016 | # lưu địa chỉ lát sau quay về vào $ra = 1016 |
| 1012 | j | sum | # nhảy đến nhãn sum |
| 1016 | [Làm tiếp thao tác khác...] | | |
| .... | | | |
| 2000 | sum: | add $v0, $a0, $a1 | # thực hiện thủ tục "sum" |
| 2024 | jr | $ra | # nhảy tới địa chỉ trong $ra |

**C**

```
... sum (a, b); ...                /* a: $s0, b: $s1 */
[Làm tiếp thao tác khác...]
}
int sum (int x, int y) {
    return x + y;
}
```

**M I P S**

| Địa chỉ | Lệnh | | |
|---|---|---|---|
| 1000 | add | $a0, $s0, $zero | |
| 1004 | add | $a1, $s1, $zero | |
| 1008 | addi | $ra, $zero, 1016 | |
| 1012 | j | sum | |
| 1016 | [Làm tiếp thao tác khác...] | | |
| .... | | | |
| 2000 | sum: | add | $v0, $a0, |
| 2024 | jr | $ra | |

- Thay vì dùng 2 lệnh để lưu địa chỉ quay về vào thanh ghi $ra và nhảy đến thủ tục "sum":

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

→MIPS hỗ trợ lệnh mới: jal (jump and link) để thực hiện 2 công việc trên:

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

# Các lệnh nhảy mới

- jr (jump register)
  - Cú pháp: **jr register**
  - Diễn giải: Nhảy đến địa chỉ nằm trong thanh ghi register thay vì nhảy đến 1 nhãn như lệnh j (jump)
- jal (jump and link)
  - Cú pháp: **jal label**
  - Diễn giải: Thực hiện 2 bước:
    - Bước 1 (link): Lưu địa chỉ của lệnh kế tiếp vào thanh ghi $ra (Tại sao không phải là địa chỉ của lệnh hiện tại ?)
    - Bước 2 (jump): Nhảy đến nhãn label
- Hai lệnh này được sử dụng hiệu quả trong thủ tục
  - jal: tự động lưu địa chỉ quay về chương trình chính vào thanh ghi $ra và nhảy đến thủ tục con
  - jr $ra: Quay lại thân chương trình chính bằng cách nhảy đến địa chỉ đã được lưu trước đó trong $ra

# Details of JAL and JR

| Address | Instructions | Assembly Language |
|---|---|---|
| 00400020 | lui $1, 0x1001 | la $a0, a |
| 00400024 | ori $4, $1, 0 | |
| 00400028 | ori $5, $0, 10 | li $a1,10 |
| 0040002C | jal 0x10000f | jal swap |
| 00400030 | . . . | # return here |
| | | swap: |
| 0040003C | sll $8, $5, 2 | sll $t0,$a1,2 |
| 00400040 | add $8, $8, $4 | add $t0,$t0,$a0 |
| 00400044 | lw $9, 0($8) | lw $t1,0($t0) |
| 00400048 | lw $10,4($8) | lw $t2,4($t0) |
| 0040004C | sw $10,0($8) | sw $t2,0($t0) |
| 00400050 | sw $9, 4($8) | sw $t1,4($t0) |
| 00400054 | jr $31 | jr $ra |

**Pseudo-Direct Addressing**

PC = imm26<<2

0x10000f << 2

= 0x0040003C

$31 | 0x00400030

Register $31 is the return address register

# Instructions for Procedures

- JAL (Jump-and-Link) used as the call instruction
  - Save return address in **$ra = PC+4** and jump to procedure
  - Register **$ra = $31** is used by **JAL** as the **return address**
- JR (Jump Register) used to return from a procedure
  - Jump to instruction whose address is in register Rs (PC = Rs)
- JALR (Jump-and-Link Register)
  - Save return address in Rd = PC+4, and
  - Jump to procedure whose address is in register Rs (PC = Rs)
  - Can be used to call methods (addresses known only at runtime)

| Instruction | | Meaning | Format | | | | | |
|---|---|---|---|---|---|---|---|---|
| jal | label | $31=PC+4, jump | $op^6 = 3$ | $imm^{26}$ | | | | |
| jr | Rs | PC = Rs | $op^6 = 0$ | $rs^5$ | 0 | 0 | 0 | 8 |
| jalr | Rd, Rs | Rd=PC+4, PC=Rs | $op^6 = 0$ | $rs^5$ | 0 | $rd^5$ | 0 | 9 |

# Thanh ghi lưu trữ dữ liệu trong thủ tục

- MIPS hỗ trợ 1 số thanh ghi để lưu trữ dữ liệu cho thủ tục:
  - Đối số input (argument input): $a0 $a1 $a2 $a3
  - Kết quả trả về (return ...): $v0 $v1
  - Biến cục bộ trong thủ tục: $s0 $s1 ... $s7
  - Địa chỉ quay về (return address): $ra
- Nếu có nhu cầu lưu nhiều dữ liệu (đối số, kết quả trả về, biến cục bộ) hơn số lượng thanh ghi kể trên?

→ Bao nhiêu thanh ghi là đủ ?

→ Sử dụng ngăn xếp (stack)

# Thủ tục lồng nhau

- Vấn đề đặt ra khi chuyển thành mã hợp ngữ của đoạn lệnh sau:

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

- Thủ tục sumSquare sẽ gọi thủ tục mult trong thân hàm của nó
- Vấn đề:
  - Địa chỉ quay về của thủ tục sumSquare lưu trong thanh ghi $ra sẽ bị ghi đè bởi địa chỉ quay về của thủ tục mult khi thủ tục này được gọi!
  - Như vậy cần phải lưu lại (backup) trong bộ nhớ chính địa chỉ quay về của thủ tục sumSquare (trong thanh ghi $ra) trước khi gọi thủ tục mult

→ Sử dụng ngăn xếp (Stack)

# Ngăn xếp (Stack)

- Được truy cập theo cơ chế "vào trước ra sau" (LIFO – Last in First out)
- $sp (stack pointer) đóng vai trò con trỏ ngăn xếp, luôn chỉ đến đỉnh của ngăn xếp.

- Là ngăn xếp gồm nhiều ô nhớ kết hợp (vùng nhớ) nằm trong bộ nhớ chính
- Cấu trúc dữ liệu lý tưởng để chứa tạm các giá trị trong thanh ghi
  - Thường chứa địa chỉ trả về, các biến cục bộ của trình con, nhất là các biến có cấu trúc (array, list…) không chứa vừa trong các thanh ghi trong CPU
- Được định vị và quản lý bởi **stack pointer**
- Có 2 tác vụ hoạt động cơ bản:
  - push: Đưa dữ liệu từ thanh ghi vào stack
  - pop: Lấy dữ liệu từ stack chép vào thanh ghi
- Trong MIPS dành sẵn 1 thanh ghi **$sp** để lưu trữ stack pointer
- Để sử dụng Stack, cần khai báo kích vùng Stack bằng cách tăng (push) giá trị con trỏ ngăn xếp stack pointer (lưu trữ trong thanh ghi $sp)
  - Lưu ý: Stack pointer tăng theo chiều **giảm địa chỉ** (đỉnh của stack luôn có địa chỉ thấp)

# Ngăn xếp (Stack)

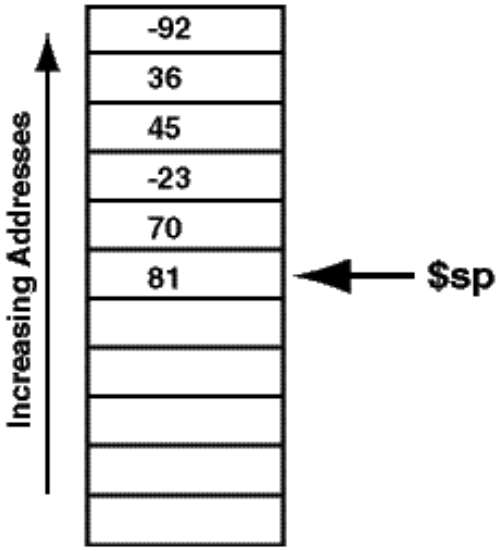
- Cơ chế hoạt động:
  - Push: giảm $sp đi 4, lưu giá trị vào ô nhớ mà $sp chỉ đến.
    - **Ví dụ:** push vào stack gtri trong thanh ghi $t0

      subu $sp, $sp, 4

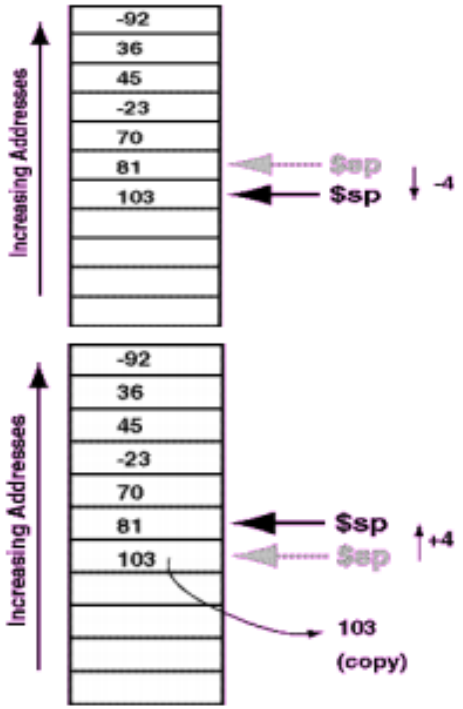
      sw $t0, ($sp)
  - Pop: copy giá trị trong vùng nhớ được chỉ đến bởi $sp, cộng 4 vào $sp.
    - **Ví dụ:** pop từ stack ra $t0

      lw $t0, ($sp)

      addu $sp, $sp, 4




Cấu trúc stack trong bộ nhớ,
mỗi phần tử có kích thước 1 word



**C**

```
int sumSquare (int x, int y) { return mult (x, x) + y; }
/* x: $a0, y: $a1 */
```

**M I P S**

```
sumSquare:
        addi    $sp, $sp, -8        # khai báo kích thước stack cần dùng = 8 byte
        sw    $ra, 4 ($sp)          # cất địa chỉ quay về của thủ tục sumSquare đưa vào stack
        sw    $a1, 0 ($sp)          # cất giá trị y vào stack
        add   $a1, $a0, $zero       # gán tham số thứ 2 là x (ban đầu là y) để phục vụ cho thủ tục mult sắp gọi
        jal   mult                  # nhảy đến thủ tục mult
        lw    $a1, 0 ($sp)          # sau khi thực thi xong thủ tục mult , khôi phục lại tham số thứ 2 = y
                                    # dựa trên giá trị đã lưu trước đó trong stack
        add   $v0, $v0, $a1         # mult() + y
        lw    $ra, 4 ($sp)          # khôi phục địa chỉ quay về của thủ tục sumSquare từ stack, đưa lại vào $ra
        addi    $sp, $sp, 8         # khôi phục 8 byte giá trị $sp ban đầu đã "mượn", kết thúc stack
        jr    $ra                   # nhảy đến đoạn lệnh ngay sau khi gọi thủ tục sumSquare trong chương
                                    # trình chính, để thao tác tiếp các lệnh khác.
mult:
        ...                         # lệnh xử lý cho thủ tục mult
        jr  $ra                     # nhảy lại đoạn lệnh ngay sau khi gọi thủ tục mult trong thủ tục sumSquare
```

# Một số nguyên tắc khi thực thi thủ tục

- Nhảy đến thủ tục bằng lệnh jal và quay về nơi trước đó đã gọi nó bằng lệnh jr $ra
- 4 thanh ghi chứa đối số của thủ tục: $a0, $a1, $a2, $a3
- Kết quả trả về của thủ tục chứa trong thanh ghi $v0 (và $v1 nếu cần)
- Phải tuân theo nguyên tắc sử dụng các thanh ghi (register conventions)

# Nguyên tắc sử dụng các thanh ghi

- $0: (Không thay đổi) Luôn bằng 0
- $s0 - $s7: (Khôi phục lại nếu thay đổi) Rất quan trọng, nếu thủ tục được gọi (callee) thay đổi các thanh ghi này thì nó phải khôi phục lại giá trị các thanh ghi này trước khi kết thúc
- $sp: (Khôi phục lại nếu thay đổi) Thanh ghi con trỏ stack phải có giá trị không đổi trước và sau khi gọi lệnh "jal", nếu không thủ tục gọi (caller) sẽ không quay về được.

# Nguyên tắc sử dụng các thanh ghi

- $ra: (Có thể thay đổi) Khi gọi lệnh "jal" sẽ làm thay đổi giá trị thanh ghi này. Thủ tục gọi (caller) lưu lại (backup) giá trị của thanh ghi $ra vào stack nếu cần
- $v0 - $v1: (Có thể thay đổi) Chứa kết quả trả về của thủ tục
- $a0 - $a1: (Có thể thay đổi) Chứa đối số của thủ tục
- $t0 - $t9: (Có thể thay đổi) Đây là các thanh ghi tạm nên có thể bị thay đổi bất cứ lúc nào

# Tóm tắt

- Nếu thủ tục R gọi thủ tục E:
  - R phải lưu vào stack các thanh ghi tạm có thể bị sử dụng trong E trước khi gọi lệnh jal E (goto E)
  - E phải lưu lại giá trị các thanh ghi lưu trữ ($s0 - $s7) nếu nó muốn sử dụng các thanh ghi này → trước khi kết thúc E sẽ khôi phục lại giá trị của chúng
  - Nhớ: Thủ tục gọi R (caller) và Thủ tục được gọi E (callee) chỉ cần lưu các thanh ghi tạm / thanh ghi lưu trữ mà nó muốn dùng, không phải tất cả các thanh ghi!

# Xem lại chức năng các thanh ghi

| Name | Register number | Usage | Preserved on call? |
|---|---|---|---|
| $zero | 0 | the constant value 0 | n.a. |
| $v0–$v1 | 2–3 | values for results and expression evaluation | no |
| $a0–$a3 | 4–7 | arguments | no |
| $t0–$t7 | 8–15 | temporaries | no |
| $s0–$s7 | 16–23 | saved | yes |
| $t8–$t9 | 24–25 | more temporaries | no |
| $gp | 28 | global pointer | yes |
| $sp | 29 | stack pointer | yes |
| $fp | 30 | frame pointer | yes |
| $ra | 31 | return address | yes |

# System Call

| Dịch vụ | Giá trị trong $v0 | Đối số | Kết quả |
|---|---|---|---|
| print_int | 1 | $a0 = integer | |
| print_float | 2 | $f12 = float | |
| print_double | 3 | $f12 = double | |
| print_string | 4 | $a0 = string | |
| read_int | 5 | | integer (trong $v0) |
| read_float | 6 | | float (trong $f0) |
| read_double | 7 | | double (trong $f0) |
| read_string | 8 | $a0 = buffer, $a1 = length | |
| sbrk | 9 | $a0 = amount | address (trong $v0) |
| exit | 10 | | |
| print_character | 11 | $a0 = char | |
| read_character | 12 | | char (trong $v0) |

# Cấu trúc của một chương trình hợp ngữ MIPS

```
#Title:                                    File name:
#Author:                                   Date:
#Description:
#Input:
#Output:
```

# Data segment

```
        .data                   #Khai báo biến sau chỉ thị này
label1: <kiểu lưu trữ> <giá trị khởi tạo>
label2: <kiểu lưu trữ> <giá trị khởi tạo>
…
```

# Code segment

```
        .text                   #Viết các lệnh sau chỉ thị này
        .globl  <các text label toàn cục, có thể truy xuất từ các file khác>
        .globl  main            # Đây là text label toàn cục bắt buộc của program
        …
main:                           #Điểm bắt đầu của chương trình
…
```

# Cách khai báo biến

| tên_biến: | kiểu_lưu_trữ | giá_trị |
|---|---|---|

- Các kiểu lưu trữ hỗ trợ: .word, .byte, .ascii, .asciiz, .space

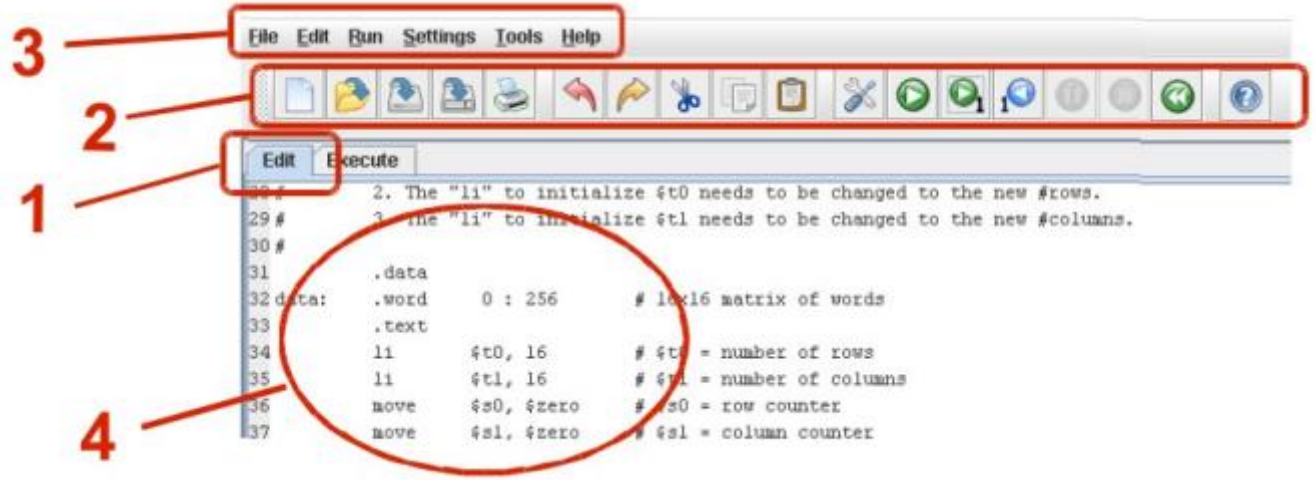
- *Lưu ý:* tên_biến (nhãn) phải theo sau bởi dấu hai chấm (:)
- Ví dụ:

```
var1:   .word       3       # số nguyên 4-byte có giá trị khởi tạo là 3
var2:   .byte       'a','b' # mảng 2 phần tử, khởi tạo là a và b
var3:   .space      40      # cấp 40-byte bộ nhớ, chưa được khởi tạo
char_array:  .byte  'A':10  # mảng 10 ký tự được khởi tạo là 'A', có thể thay 'A' bằng 65
int_array:   .word  0:30    # mảng 30 số nguyên được khởi tạo là 0
```

# Ví dụ: Hello.asm

```
        .data                   # data segment
str:    .asciiz  "Hello asm !"
        .text                   # text segment
        .globl   main
main:                           # starting point of program
        addi   $v0, $0, 4       # $v0 = 0 + 4 = 4 → print str syscall
        la  $a0, str            # $a0 = address(str)
        syscall                 # excute the system call
```

# MARS

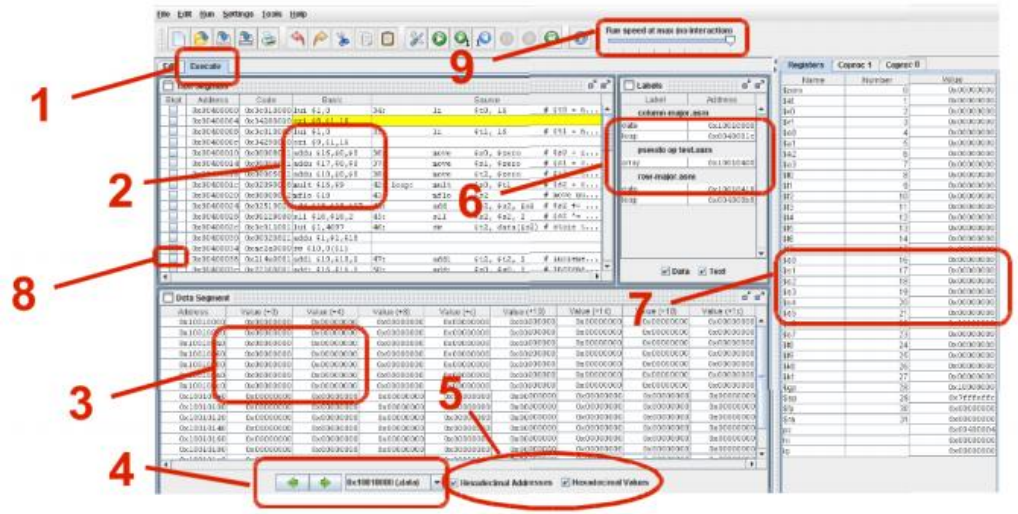



1. Đang ở chế độ soạn thảo

2,3. Thanh menu và thanh công cụ hỗ trợ các chức năng của CT

4. Nơi soạn thảo CT

# MARS (...)




1. Cho ta biết đang ở chế độ thực thi.
2. Khung thực thi cho ta biết địa chỉ lệnh, mã máy, lệnh hợp ngữ MIPS, dòng lệnh trong file source tương ứng.
3. Các giá trị trong bộ nhớ, có thể chỉnh sửa được.
4. Cho phép ta duyệt bộ nhớ và đi đến các phân đoạn bộ nhớ thông dụng.
5. Bật, tắt việc xem địa chỉ và giá trị ô nhớ ở dạng thập phân hay hexa.
6. Địa chỉ của các khai báo nhãn và dữ liệu.
7. Các dữ liệu trong thanh ghi, có thể chỉnh sửa được.
8. Điểm đặt breakpoint dùng cho việc debug chương trình.
9. Điều chỉnh tốc độ chạy CT, cho phép người dùng có thể xem những gì diễn ra thay vì kết thúc CT ngay.

# Bài tập

Hãy viết chương trình hợp ngữ MIPS (không dùng lệnh giả) để giải quyết các bài toán sau:

1. Nhập vào một chuỗi, xuất lại chuỗi đó ra màn hình (echo).

Ví dụ:
Nhap mot chuoi: Hello
Chuoi da nhap: Hello

2. Nhập vào một ký tự, xuất ra ký tự liền trước và liền sau.

Ví dụ:
Nhap mot ky tu: b
Ky tu lien truoc: a
Ky tu lien sau: c

3. Nhập vào một ký tự hoa, in ra ký tự thường.

Ví dụ:
Nhap mot ky tu: A
Ky tu thuong: a

4. Nhập từ bàn phím 2 số nguyên, tính tổng, hiệu, tích, thương của 2 số.

Ví dụ:
Nhap so thu nhat: 7
Nhap so thu hai: 4
Tong: 11
Hieu: 3
Tich: 28
Thuong: 1 du 3

# Bài tập (...)

5. Nhập vào 2 số nguyên, xuất ra số lớn hơn.

Ví dụ:
Nhap so thu nhat: 6
Nhap so thu hai: 9
So lon hon la: 9

6. Nhập một ký tự từ bàn phím. Nếu ký tự vừa nhập thuộc [0-9], [a-z], [A-Z] thì xuất ra màn hình ký tự đó và loại của ký tự đó (số, chữ thường, chữ hoa).

Ví dụ:
Nhập vào một ký tự: 5
Ký tự vừa nhập: 5 là số
Nhập vào một ký tự : f
Ký tự vừa nhập : f là chữ thường
Nhập vào một ký tự : D
Ký tự vừa nhập : D là chữ hoa

7. Nhập một mảng các số nguyên n phần tử, xuất mảng đó ra màn hình.

Ví dụ:
Nhap n: 5
[ 0 ] = 4
[ 1 ] = 2
[ 2 ] = 7
[ 3 ] = 9
[ 4 ] = 3
Mang vua nhap: 4 2 7 9 3

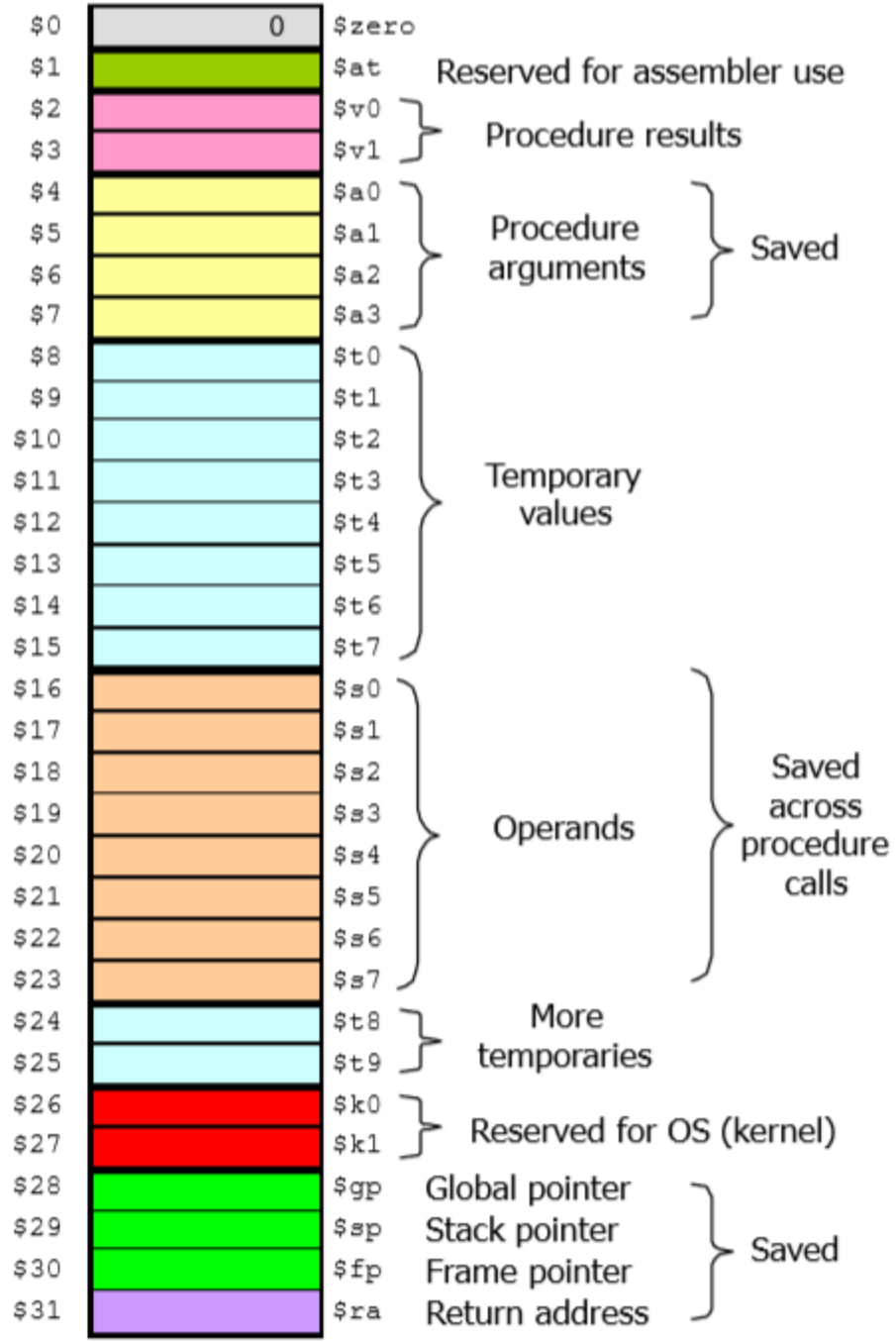
8. Nhập vào một số nguyên n, tính tổng từ 1 đến n.

Ví dụ:
Nhap mot so: 4
Tong tu 1 den 4 la: 10

9. Nhập vào một chuỗi. Tính chiều dài của chuỗi.

Ví dụ:
Nhap mot chuoi: HCMUS
Chieu dai cua chuoi: 5

# Phụ lục


$0
0
$zero
$1
$at
Reserved for assembler use
$2
$v0
$3
$v1
Procedure results
$4
$a0
$5
$a1
$6
$a2
$7
$a3
Procedure arguments
Saved
$8
$t0
$9
$t1
$10
$t2
$11
$t3
$12
$t4
$13
$t5
$14
$t6
$15
$t7
Temporary values
$16
$s0
$17
$s1
$18
$s2
$19
$s3
$20
$s4
$21
$s5
$22
$s6
$23
$s7
Operands
Saved across procedure calls
$24
$t8
$25
$t9
More temporaries
$26
$k0
$27
$k1
Reserved for OS (kernel)
$28
$gp
Global pointer
$29
$sp
Stack pointer
$30
$fp
Frame pointer
$31
$ra
Return address
Saved

# 40 lệnh MIPS cơ bản

| Instruction | Usage |
|---|---|
| Load upper immediate | `lui rt,imm` |
| Add | `add rd,rs,rt` |
| Subtract | `sub rd,rs,rt` |
| Set less than | `slt rd,rs,rt` |
| Add immediate | `addi rt,rs,imm` |
| Set less than immediate | `slti rd,rs,imm` |
| AND | `and rd,rs,rt` |
| OR | `or rd,rs,rt` |
| XOR | `xor rd,rs,rt` |
| NOR | `nor rd,rs,rt` |
| AND immediate | `andi rt,rs,imm` |
| OR immediate | `ori rt,rs,imm` |
| XOR immediate | `xori rt,rs,imm` |
| Load word | `lw rt,imm(rs)` |
| Store word | `sw rt,imm(rs)` |
| Jump | `j L` |
| Jump register | `jr rs` |
| Branch less than 0 | `bltz rs,L` |
| Branch equal | `beq rs,rt,L` |
| Branch not equal | `bne rs,rt,L` |

| Instruction | Usage |
|---|---|
| Move from Hi | `mfhi rd` |
| Move from Lo | `mflo rd` |
| Add unsigned | `addu rd,rs,rt` |
| Subtract unsigned | `subu rd,rs,rt` |
| Multiply | `mult rs,rt` |
| Multiply unsigned | `multu rs,rt` |
| Divide | `div rs,rt` |
| Divide unsigned | `divu rs,rt` |
| Add immediate unsigned | `addiu rs,rt,imm` |
| Shift left logical | `sll rd,rt,sh` |
| Shift right logical | `srl rd,rt,sh` |
| Shift right arithmetic | `sra rd,rt,sh` |
| Shift left logical variable | `sllv rd,rt,rs` |
| Shift right logical variable | `srlv rd,rt,rs` |
| Shift right arith variable | `srav rd,rt,rs` |
| Load byte | `lb rt,imm(rs)` |
| Load byte unsigned | `lbu rt,imm(rs)` |
| Store byte | `sb rt,imm(rs)` |
| Jump and link | `jal L` |
| System call | `syscall` |

# Lệnh giả

- "Lệnh giả": Mặc định không được hỗ trợ bởi MIPS
- Là những lệnh cần phải biên dịch thành rất nhiều câu lệnh thật trước khi được thực hiện bởi phần cứng

  → Lệnh giả = Thủ tục
- Dùng để hỗ trợ lập trình viên thao tác nhanh chóng với những thao tác phức tạp gồm nhiều bước

# Một số lệnh giả phổ biến của MIPS

| Name | instruction syntax | meaning |
|---|---|---|
| Move | move rd, rs | rd = rs |
| Load Address | la rd, rs | rd = address (rs) |
| Load Immediate | li rd, imm | rd = 32 bit Immediate value |
| Branch greater than | bgt rs, rt, Label | if(R[rs]>R[rt]) PC=Label |
| Branch less than | blt rs, rt, Label | if(R[rs]<R[rt]) PC=Label |
| Branch greater than or equal | bge rs, rt, Label | if(R[rs]>=R[rt]) PC=Label |
| branch less than or equal | ble rs, rt, Label | if(R[rs]<=R[rt]) PC=Label |
| branch greater than unsigned | bgtu rs, rt, Label | if(R[rs]<=R[rt]) PC=Label |
| branch greater than zero | bgtz rs, Label | if(R[rs] >=0) PC=Label |