MIPS Assembly Language Programming

ICS 233

Computer Architecture & Assembly Language

Prof. Muhamed Mudawar

College of Computer Sciences and Engineering

King Fahd University of Petroleum and Minerals

Presentation Outline

- Assembly Language Statements
- Assembly Language Program Template
- Defining Data
- Memory Alignment and Byte Ordering
- System Calls
- Procedures
- Parameter Passing and the Runtime Stack

Assembly Language Statements

- ❖ Three types of statements in assembly language
  - ✧ Typically, one statement should appear on a line
1. Executable Instructions
   - ✧ Generate machine code for the processor to execute at runtime
   - ✧ Instructions tell the processor what to do
2. Pseudo-Instructions and Macros
   - ✧ Translated by the assembler into real instructions
   - ✧ Simplify the programmer task
3. Assembler Directives
   - ✧ Provide information to the assembler while translating a program
   - ✧ Used to define segments, allocate memory variables, etc.
   - ✧ Non-executable: directives are not part of the instruction set

Instructions

- ❖ Assembly language instructions have the format:

```
[label:]   mnemonic   [operands]   [#comment]
```

- ❖ Label: (optional)
  - ✧ Marks the address of a memory location, must have a colon
  - ✧ Typically appear in data and text segments
- ❖ Mnemonic
  - ✧ Identifies the operation (e.g. **add**, **sub**, etc.)
- ❖ Operands
  - ✧ Specify the data required by the operation
  - ✧ Operands can be registers, memory variables, or constants
  - ✧ Most instructions have three operands

```
L1:   addiu $t0, $t0, 1         #increment $t0
```

Comments

❖ Comments are very important!
- ✧ Explain the program's purpose
- ✧ When it was written, revised, and by whom
- ✧ Explain data used in the program, input, and output
- ✧ Explain instruction sequences and algorithms used
- ✧ Comments are also required at the beginning of every procedure
  - Indicate input parameters and results of a procedure
  - Describe what the procedure does

❖ Single-line comment
- ✧ Begins with a hash symbol # and terminates at end of line

Next . . .

❖ Assembly Language Statements

❖ Assembly Language Program Template

❖ Defining Data

❖ Memory Alignment and Byte Ordering

❖ System Calls

❖ Procedures

❖ Parameter Passing and the Runtime Stack

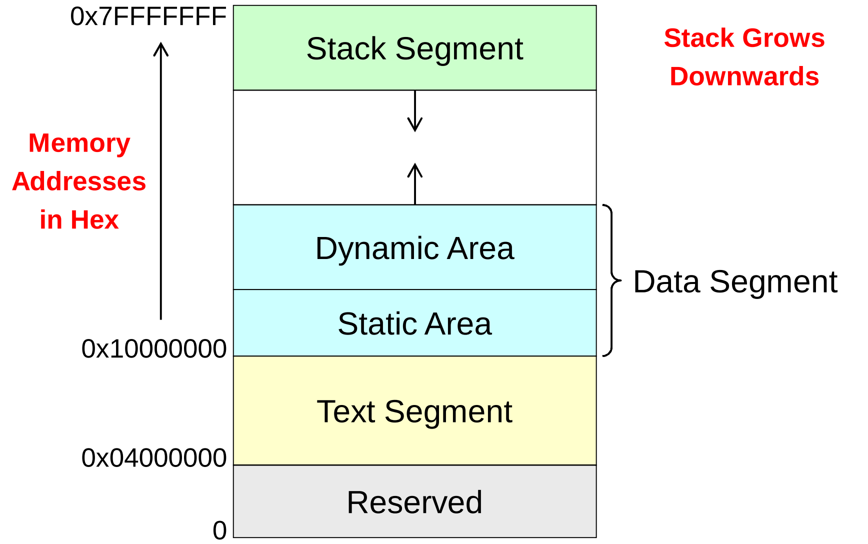
Program Template

```
# Title:                        Filename:
# Author:                       Date:
# Description:
# Input:
# Output:
################# Data segment #####################
.data
 . . .
################# Code segment #####################
.text
.globl main
main:                           # main program entry
 . . .
li $v0, 10                      # Exit program
syscall
```


.DATA, .TEXT, & .GLOBL Directives

- **.DATA** directive
  - Defines the data segment of a program containing data
  - The program's variables should be defined under this directive
  - Assembler will allocate and initialize the storage of variables
- **.TEXT** directive
  - Defines the code segment of a program containing instructions
- **.GLOBL** directive
  - Declares a symbol as global
  - Global symbols can be referenced from other files
  - We use this directive to declare *main* procedure of a program

Layout of a Program in Memory



Next . . .

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ Defining Data
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

Data Definition Statement

- ❖ Sets aside storage in memory for a variable
- ❖ May optionally assign a name (label) to the data
- ❖ Syntax:

[*name:*] *directive* *initializer* [, *initializer*] . . .

```
var1: .WORD   10
```

- ❖ All initializers become binary data in memory

Data Directives

- ❖ **.BYTE** Directive
  - ✧ Stores the list of values as 8-bit bytes
- ❖ **.HALF** Directive
  - ✧ Stores the list as 16-bit values aligned on half-word boundary
- ❖ **.WORD** Directive
  - ✧ Stores the list as 32-bit values aligned on a word boundary
- ❖ **.FLOAT** Directive
  - ✧ Stores the listed values as single-precision floating point
- ❖ **.DOUBLE** Directive
  - ✧ Stores the listed values as double-precision floating point

String Directives

- **.ASCII** Directive
  - Allocates a sequence of bytes for an ASCII string
- **.ASCIIZ** Directive
  - Same as **.ASCII** directive, but adds a NULL char at end of string
  - Strings are null-terminated, as in the C programming language
- **.SPACE** Directive
  - Allocates space of *n* uninitialized bytes in the data segment

Examples of Data Definitions

```
.DATA
var1:  .BYTE    'A', 'E', 127, -1, '\n'
var2:  .HALF    -10, 0xffff
var3:  .WORD    0x12345678:100
var4:  .FLOAT   12.3, -0.1
var5:  .DOUBLE  1.5e-10
str1:  .ASCII   "A String\n"
str2:  .ASCIIZ  "NULL Terminated String"
array: .SPACE   100
```

Array of 100 words

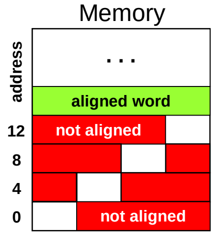
100 bytes (not initialized)

Next . . .

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ Defining Data
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

Memory Alignment

- ❖ Memory is viewed as an array of bytes with addresses
  - ✧ Byte Addressing: address points to a byte in memory
- ❖ Words occupy 4 consecutive bytes in memory
  - ✧ MIPS instructions and integers occupy 4 bytes
- ❖ Alignment: address is a multiple of size
  - ✧ Word address should be a multiple of **4**
    - ▪ Least significant 2 bits of address should be **00**
  - ✧ Halfword address should be a multiple of **2**



- ❖ **.ALIGN n** directive
  - ✧ Aligns the next data definition on a 2^n byte boundary

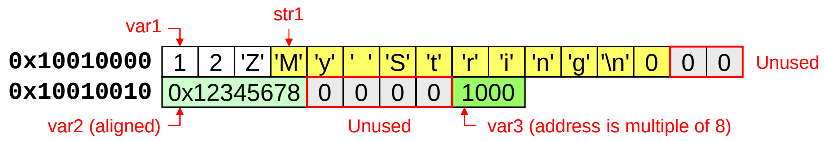
Symbol Table

- ❖ Assembler builds a symbol table for labels (variables)
  - ✧ Assembler computes the address of each label in data segment
- ❖ Example

```
.DATA
var1:   .BYTE    1, 2,'Z'
str1:   .ASCIIZ "My String\n"
var2:   .WORD    0x12345678
.ALIGN   3
var3:   .HALF    1000
```

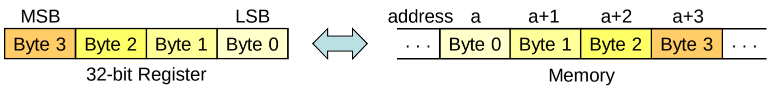
Symbol Table

Label	Address
var1	0x10010000
str1	0x10010003
var2	0x10010010
var3	0x10010018

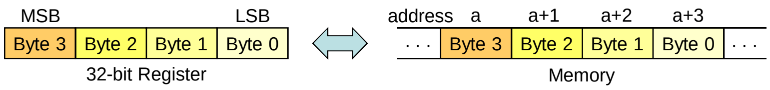


Byte Ordering and Endianness

- ❖ Processors can order bytes within a word in two ways
- ❖ Little Endian Byte Ordering
  - ✧ Memory address = Address of **least significant byte**
  - ✧ Example: Intel IA-32, Alpha



- ❖ Big Endian Byte Ordering
  - ✧ Memory address = Address of **most significant byte**
  - ✧ Example: SPARC, PA-RISC



- ❖ MIPS can operate with both byte orderings

Next . . .

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ Defining Data
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

System Calls

- ❖ Programs do input/output through system calls
- ❖ MIPS provides a special **`syscall`** instruction
  - ✧ To obtain services from the operating system
  - ✧ Many services are provided in the SPIM and MARS simulators
- ❖ Using the **`syscall`** system services
  - ✧ Load the service number in register $v0
  - ✧ Load argument values, if any, in registers $a0, $a1, etc.
  - ✧ Issue the **`syscall`** instruction
  - ✧ Retrieve return values, if any, from result registers

Syscall Services

Service	$v0	Arguments / Result
Print Integer	1	$a0 = integer value to print
Print Float	2	$f12 = float value to print
Print Double	3	$f12 = double value to print
Print String	4	$a0 = address of null-terminated string
Read Integer	5	Return integer value in $v0
Read Float	6	Return float value in $f0
Read Double	7	Return double value in $f0
Read String	8	$a0 = address of input buffer $a1 = maximum number of characters to read
Allocate Heap memory	9	$a0 = number of bytes to allocate Return address of allocated memory in $v0
Exit Program	10	

Syscall Services – Cont'd

Print Char	11	$a0 = character to print
Read Char	12	Return character read in $v0
Open File	13	$a0 = address of null-terminated filename string $a1 = flags (0 = read-only, 1 = write-only) $a2 = mode (ignored) Return file descriptor in $v0 (negative if error)
Read from File	14	$a0 = File descriptor $a1 = address of input buffer $a2 = maximum number of characters to read Return number of characters read in $v0
Write to File	15	$a0 = File descriptor $a1 = address of buffer $a2 = number of characters to write Return number of characters written in $v0
Close File	16	$a0 = File descriptor

Reading and Printing an Integer

```
################# Code segment #####################
.text
.globl main
main:                        # main program entry
  li    $v0, 5               # Read integer
  syscall                    # $v0 = value read

  move  $a0, $v0             # $a0 = value to print
  li    $v0, 1               # Print integer
  syscall

  li    $v0, 10              # Exit program
  syscall
```


Reading and Printing a String

```
################# Data segment #####################
.data
  str: .space  10          # array of 10 bytes
################# Code segment #####################
.text
.globl main
main:                      # main program entry
  la    $a0, str           # $a0 = address of str
  li    $a1, 10            # $a1 = max string length
  li    $v0, 8             # read string
  syscall
  li    $v0, 4             # Print string str
  syscall
  li    $v0, 10            # Exit program
  syscall
```

Program 1: Sum of Three Integers

```
# Sum of three integers
#
# Objective: Computes the sum of three integers.
#     Input: Requests three numbers.
#    Output: Outputs the sum.
################### Data segment ###################
.data
prompt:   .asciiz     "Please enter three numbers: \n"
sum_msg:  .asciiz     "The sum is: "
################### Code segment ###################
.text
.globl main
main:
      la    $a0,prompt          # display prompt string
      li    $v0,4
      syscall
      li    $v0,5               # read 1st integer into $t0
      syscall
      move  $t0,$v0
```


Sum of Three Integers – Slide 2 of 2

```
      li    $v0,5               # read 2nd integer into $t1
      syscall
      move  $t1,$v0

      li    $v0,5               # read 3rd integer into $t2
      syscall
      move  $t2,$v0

      addu  $t0,$t0,$t1         # accumulate the sum
      addu  $t0,$t0,$t2

      la    $a0,sum_msg         # write sum message
      li    $v0,4
      syscall

      move  $a0,$t0             # output sum
      li    $v0,1
      syscall

      li    $v0,10              # exit
      syscall
```

Program 2: Case Conversion

```
# Objective: Convert lowercase letters to uppercase
#      Input: Requests a character string from the user.
#     Output: Prints the input string in uppercase.
################### Data segment #####################
.data
name_prompt:  .asciiz         "Please type your name: "
out_msg:      .asciiz         "Your name in capitals is: "
in_name:      .space 31       # space for input string
################### Code segment #####################
.text
.globl main
main:
      la    $a0,name_prompt  # print prompt string
      li    $v0,4
      syscall
      la    $a0,in_name      # read the input string
      li    $a1,31           # at most 30 chars + 1 null char
      li    $v0,8
      syscall
```


Case Conversion - Slide 2 of 2

```
      la    $a0,out_msg      # write output message
      li    $v0,4
      syscall
      la    $t0,in_name
loop:
      lb    $t1,($t0)
      beqz  $t1,exit_loop    # if NULL, we are done
      blt   $t1,'a',no_change
      bgt   $t1,'z',no_change
      addiu $t1,$t1,-32      # convert to uppercase: 'A'-'a'=-32
      sb    $t1,($t0)
no_change:
      addiu $t0,$t0,1        # increment pointer
      j     loop
exit_loop:
      la    $a0,in_name      # output converted string
      li    $v0,4
      syscall
      li    $v0,10           # exit
      syscall
```

Example of File I/O

```
# Sample MIPS program that writes to a new text file
.data
file:      .asciiz "out.txt"     # output filename
buffer:    .asciiz "Sample text to write"

.text
li   $v0, 13         # system call to open a file for writing
la   $a0, file       # output file name
li   $a1, 1          # Open for writing (flags 1 = write)
li   $a2, 0          # mode is ignored
syscall              # open a file (file descriptor returned in $v0)
move $s6, $v0        # save the file descriptor
li   $v0, 15         # Write to file just opened
move $a0, $s6        # file descriptor
la   $a1, buffer     # address of buffer from which to write
li   $a2, 20         # number of characters to write = 20
syscall              # write to file
li   $v0, 16         # system call to close file
move $a0, $s6        # file descriptor to close
syscall              # close file
```


Next . . .

- Assembly Language Statements
- Assembly Language Program Template
- Defining Data
- Memory Alignment and Byte Ordering
- System Calls
- Procedures
- Parameter Passing and the Runtime Stack

Procedures

- A procedure (or function) is a tool used by programmers
  - Allows the programmer to focus on just one task at a time
  - Allows code to be reused
- Procedure Call and Return
  - Put parameters in a place where procedure can access
    - Four argument registers: **$a0** thru **$a3** in which to pass parameters
  - Transfer control to the procedure and save return address
    - Jump-and-Link instruction: **jal** (Return Address saved in **$ra**)
  - Perform the desired task
  - Put results in a place where the calling procedure can access
    - Two value registers to return results: **$v0** and **$v1**
  - Return to calling procedure: **jr $ra** (jump to return address)

Procedure Example

- Consider the following swap procedure (written in C)
- Translate this procedure to MIPS assembly language

```
void swap(int v[], int k)
{  int temp;
   temp = v[k]
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Parameters:

$a0 = Address of **v[]**
$a1 = **k**, and
Return address is in **$ra**

```
swap:
 sll $t0,$a1,2    # $t0=k*4
 add $t0,$t0,$a0  # $t0=v+k*4
 lw  $t1,0($t0)   # $t1=v[k]
 lw  $t2,4($t0)   # $t2=v[k+1]
 sw  $t2,0($t0)   # v[k]=$t2
 sw  $t1,4($t0)   # v[k+1]=$t1
 jr  $ra          # return
```

Call / Return Sequence

- Suppose we call procedure swap as: **swap(a,10)**
  - Pass **address** of array **a** and **10** as arguments
  - Call the procedure swap saving **return address** in **$31 = $ra**
  - Execute procedure swap
  - Return control to the point of origin (return address)

Registers

	. . .
$a0=$4	addr a
$a1=$5	10
	. . .
$ra=$31	ret addr

Caller

```
   la    $a0, a
   li    $a1, 10
   jal   swap
# return here
   . . .
```

```
swap:
 sll $t0,$a1,2
 add $t0,$t0,$a0
 lw  $t1,0($t0)
 lw  $t2,4($t0)
 sw  $t2,0($t0)
 sw  $t1,4($t0)
 jr  $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	ori $a1,$0,10
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

Next . . .

- Assembly Language Statements
- Assembly Language Program Template
- Defining Data
- Memory Alignment and Byte Ordering
- System Calls
- Procedures
- Parameter Passing and the Runtime Stack

Parameter Passing

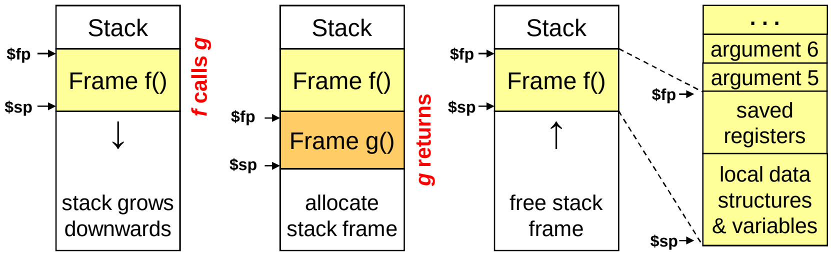
- Parameter passing in assembly language is different
  - More complicated than that used in a high-level language
- In assembly language
  - Place all required parameters in an accessible storage area
  - Then call the procedure
- Two types of storage areas used
  - Registers: general-purpose registers are used (register method)
  - Memory: stack is used (stack method)
- Two common mechanisms of parameter passing
  - Pass-by-value: parameter **value** is passed
  - Pass-by-reference: **address** of parameter is passed

Parameter Passing - cont'd

- By convention, register are used for parameter passing
  - **`$a0 = $4 .. $a3 = $7`** are used for **passing arguments**
  - **`$v0 = $2 .. $v1 = $3`** are used for **result values**
- Additional arguments/results can be placed on the stack
- Runtime stack is also needed to …
  - Store variables / data structures when they cannot fit in registers
  - Save and restore registers across procedure calls
  - Implement recursion
- Runtime stack is implemented via software convention
  - The **stack pointer `$sp = $29`** (points to top of stack)
  - The **frame pointer `$fp = $30`** (points to a procedure frame)

Stack Frame

- Stack frame is the segment of the stack containing …
  - Saved arguments, registers, and local data structures (if any)
- Called also the activation frame
- Frames are pushed and popped by adjusting …
  - Stack pointer `$sp = $29` and Frame pointer `$fp = $30`
  - Decrement `$sp` to allocate stack frame, and increment to free



Procedure Calling Convention

- The Caller should do the following:

1. Pass Arguments
   - First four arguments are passed in registers $a0 thru $a3
   - Additional arguments are pushed on the stack
2. Save Registers $a0 - $a3 and $t0 - $t9 if needed
   - Registers $a0 - $a3 and $t0 - $t9 should be saved by Caller
   - To preserve their value if needed after a procedure call
   - Called procedure is free to modify $a0 to $a3 and $t0 to $t9
3. Execute JAL Instruction
   - Jumps to the first instruction inside the procedure
   - Saves the return address in register $ra

Procedure Calling Convention - 2

❖ The Called procedure (Callee) should do the following:

1. Allocate memory for the stack frame
   - ✧ $sp = $sp – n (n bytes are allocated on the stack frame)
   - ✧ The programmer should compute n
   - ✧ A simple leaf procedure might not need a stack frame (n = 0)
2. Save registers $ra, $fp, $s0 - $s7 in the stack frame
   - ✧ $ra, $fp, $s0 - $s7 should be saved inside procedure (callee)
   - ✧ Before modifying their value and only if needed
   - ✧ Register $ra should be saved only if the procedure makes a call
3. Update the frame pointer $fp (if needed)
   - ✧ For simple procedures, the $fp register is not be required

Procedure Return Convention

❖ Just before returning, the called procedure should:

1. Place the returned results in $v0 and $v1 (if any)
2. Restore all registers that were saved upon entry
   - ✧ Load value of $ra, $fp, $s0 - $s7 if saved in the stack frame
3. Free the stack frame
   - ✧ $sp = $sp + n (if n bytes are allocated for the stack frame)
4. Return to caller
   - ✧ Jump to the return address: jr $ra

Preserving Registers

- Need to preserve registers across a procedure call
  - Stack can be used to preserve register values
- Caller-Saved Registers
  - Registers **$a0** to **$a3** and **$t0** to **$t9** should be saved by Caller
  - Only if needed after a procedure call
- Callee-Saved Registers (Saved inside procedure)
  - Registers **$s0** to **$s7**, **$sp**, **$fp**, and **$ra** should be saved
  - Only if used and modified inside procedure
  - Should be saved upon procedure entry before they are modified
  - Restored at end of procedure before returning to caller

Example on Preserving Register

- A function **f** calls **g** twice as shown below. We don't know what **g** does, or which registers are used in **g**.
- We only know that function **g** receives two integer arguments and returns one integer result.
- Translate **f**:

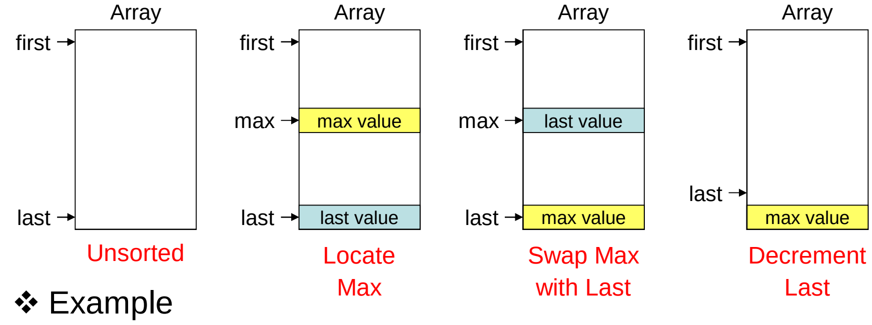
```
int f(int a, int b) {
  int d = g(b, g(a, b));
  return a + d;
}
```

Example on Preserving Registers

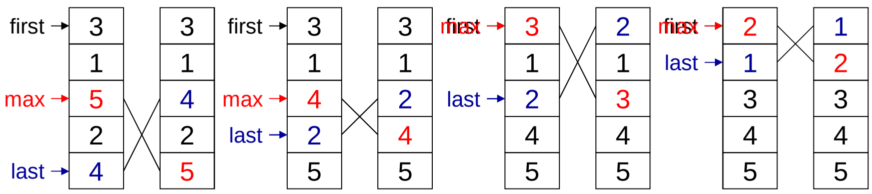
```
int f(int a, int b) {
  int d = g(b, g(a, b)); return a + d;
}

f: addiu  $sp, $sp, -12      # frame = 12 bytes
   sw     $ra, 0($sp)        # save $ra
   sw     $a0, 4($sp)        # save argument a
   sw     $a1, 8($sp)        # save argument b
   jal    g                  # call g(a,b)
   lw     $a0, 8($sp)        # $a0 = b
   move   $a1, $v0           # $a1 = g(a,b)
   jal    g                  # call g(b, g(a,b))
   lw     $a0, 4($sp)        # $a0 = a
   addu   $v0, $a0, $v0      # $v0 = a + d
   lw     $ra, 0($sp)        # restore $ra
   addiu  $sp, $sp, 12       # free stack frame
   jr     $ra                # return to caller
```


Selection Sort



❖ Example



Selection Sort (Leaf Procedure)

```
# Input:  $a0 = pointer to first, $a1 = pointer to last
# Output: array is sorted in place
##########################################################
sort:  beq    $a0, $a1, ret     # if (first == last) return
top:   move   $t0, $a0          # $t0 = pointer to max
       lw     $t1, ($t0)        # $t1 = value of max
       move   $t2, $t0          # $t2 = array pointer
max:   addiu  $t2, $t2, 4       # $t2 = pointer to next A[i]
       lw     $t3, 0($t2)       # $t3 = value of A[i]
       ble    $t3, $t1, skip    # if (A[i] <= max) then skip
       move   $t0, $t2          # $t0 = pointer to new maximum
       move   $t1, $t3          # $t1 = value of new maximum
skip:  bne    $t2, $a1, max     # loop back if more elements
       sw     $t1, 0($a1)       # store max at last address
       sw     $t3, 0($t0)       # store last at max address
       addiu  $a1, $a1, -4      # decrement pointer to last
       bne    $a0, $a1, top     # more elements to sort
ret:   jr     $ra               # return to caller
```


Example of a Recursive Procedure

int fact(int n) { if (n<2) return 1; else return (n*fact(n-1)); }

```
fact:  slti    $t0,$a0,2      # (n<2)?
       beq     $t0,$0,else    # if false branch to else
       li      $v0,1          # $v0 = 1
       jr      $ra            # return to caller

else:  addiu   $sp,$sp,-8     # allocate 2 words on stack
       sw      $a0,4($sp)     # save argument n
       sw      $ra,0($sp)     # save return address
       addiu   $a0,$a0,-1     # argument = n-1
       jal     fact           # call fact(n-1)
       lw      $a0,4($sp)     # restore argument
       lw      $ra,0($sp)     # restore return address
       mul     $v0,$a0,$v0    # $v0 = n*fact(n-1)
       addi    $sp,$sp,8      # free stack frame
       jr      $ra            # return to caller
```