Computer Architecture CSF342

Lab sheet 4

Topic: Introduction to Function Calls

1. In MIPS assembly, function calls involve two main components:

- **Caller**: The function that initiates the call
- **Callee**: The function being called

Key instructions for function calls:

- `jal label`: Jump and Link - saves return address in $ra and jumps to label
- `jalr $register`: Jump and Link Register - jumps to address in register and saves return address
- `jr $register`: Jump Register - returns to address stored in register (typically $ra)

2. Argument and Return Registers

MIPS uses a specific convention for passing arguments and return values:

- Arguments: $a0-$a3 (first four arguments)
- Return values: $v0-$v1
- Additional arguments are passed on the stack

Example 1: Power function (a^b)

```
.data
prompt_base:     .asciiz "Enter the base: "
prompt_exponent: .asciiz "Enter the exponent: "
result_msg:      .asciiz "Result: "
.text
main:
    # Print prompt for base
    li $v0, 4
    la $a0, prompt_base
    syscall

    # Read base
    li $v0, 5
    syscall
    move $s0, $v0          # Store base in $s0

    # Print prompt for exponent
    li $v0, 4
    la $a0, prompt_exponent
    syscall

    # Read exponent
    li $v0, 5
    syscall
    move $s1, $v0          # Store exponent in $s1

    # Prepare arguments and call pow function
    move $a0, $s0          # First argument: base
    move $a1, $s1          # Second argument: exponent
    jal pow

    # Store result
```

```
    move $s2, $v0

    # Print result message
    li $v0, 4
    la $a0, result_msg
    syscall

    # Print result
    li $v0, 1
    move $a0, $s2
    syscall

    # Exit program
    li $v0, 10
    syscall

# Function: pow(a, b)
# Input: $a0 = base (a), $a1 = exponent (b)
# Output: $v0 = a^b
pow:
    li $v0, 1           # Initialize result to 1
    li $t0, 0           # Initialize counter to 0
pow_loop:
    beq $t0, $a1, pow_end  # If counter == exponent, done
    mul $v0, $v0, $a0   # result = result * base
    addi $t0, $t0, 1    # Increment counter
    j pow_loop
pow_end:
    jr $ra              # Return to caller
```

Example 2: C = P + K mod 26, where P is the input and K is a hardcoded key, pass P as an argument and return C, and print it. Check the exception that both P and C are in the English alphabet.

```
.data
    prompt: .asciiz "Enter a character: "
    result_msg: .asciiz "\nEncrypted Character: "
    newline: .asciiz "\n"

.text
    .globl main

main:
    # Print prompt
    li $v0, 4
    la $a0, prompt
    syscall

    # Read character
    li $v0, 12
    syscall
    move $a0, $v0         # Pass character as argument
    li $a1, 5             # Hardcoded key K=5

    jal ceaser            # Call ceaser function
    move $t0, $v0         # Save returned character
```

```
    # Print result message
    li $v0, 4
    la $a0, result_msg
    syscall

    move $a0, $t0        # Move saved result to $a0 for printing
    li $v0, 11           # Print character
    syscall

    # Print newline
    li $v0, 4
    la $a0, newline
    syscall

    # Exit program
    li $v0, 10
    syscall

ceaser:
    # Check if uppercase
    li $t0, 'A'
    li $t1, 'Z'
    blt $a0, $t0, not_upper
    bgt $a0, $t1, not_upper
    # Handle uppercase
    sub $t2, $a0, 'A'   # Convert to 0-25
    add $t2, $t2, $a1   # Add key
    li $t3, 26
    div $t2, $t3        # Divide by 26
    mfhi $t2            # Get remainder
    add $v0, $t2, 'A'   # Convert back to uppercase
    jr $ra

not_upper:
    # Check if lowercase
    li $t0, 'a'
    li $t1, 'z'
    blt $a0, $t0, not_alpha
    bgt $a0, $t1, not_alpha
    # Handle lowercase
    sub $t2, $a0, 'a'   # Convert to 0-25
    add $t2, $t2, $a1   # Add key
    li $t3, 26
    div $t2, $t3        # Divide by 26
    mfhi $t2            # Get remainder
    add $v0, $t2, 'a'   # Convert back to lowercase
    jr $ra

not_alpha:
    # Not an alphabet character, return as is
    move $v0, $a0
    jr $ra
```

3. Call Stack and Stack Frames

The call stack is used to:

- Save return addresses
- Store function arguments beyond the first four
- Preserve register values across function calls
- Allocate space for local variables

Stack operations:

- $sp: Stack Pointer (points to top of stack)
- $fp: Frame Pointer (points to current stack frame)

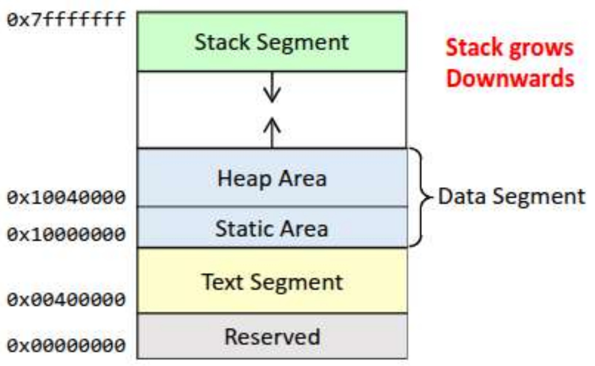

Figure 7.4: The text, data, and stack segments of a program

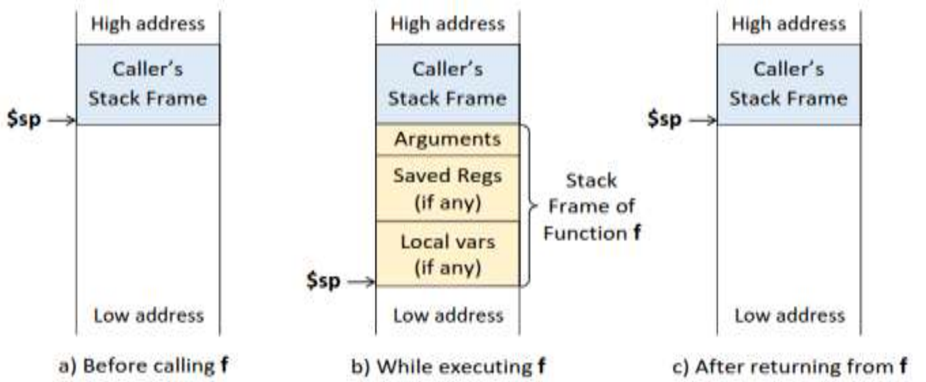

Figure 7.5: Stack allocation (a) before (b) while executing, and (c) after returning from function **f**

4. Recursive Functions with Complete Examples

Factorial Example (Complete C code with main):

```c
#include <stdio.h>

int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n-1);
}

int main() {
    int n, result;

    printf("Enter a positive integer: ");
    scanf("%d", &n);
```

```c
    result = factorial(n);

    printf("Factorial of %d = %d\n", n, result);

    return 0;
}
```

Factorial Example (Complete MIPS with main):

```
.data
prompt:     .asciiz "Enter a positive integer: "
result_msg: .asciiz "Factorial = "

.text
main:
    # Print prompt
    li $v0, 4
    la $a0, prompt
    syscall

    # Read integer input
    li $v0, 5
    syscall
    move $a0, $v0      # Store input in $a0

    # Call factorial function
    jal factorial

    # Store result
    move $t0, $v0

    # Print result message
    li $v0, 4
    la $a0, result_msg
    syscall

    # Print result
    li $v0, 1
    move $a0, $t0
    syscall

    # Exit program
    li $v0, 10
    syscall

factorial:
    addi $sp, $sp, -8   # Allocate stack space
    sw $ra, 4($sp)      # Save return address
    sw $a0, 0($sp)      # Save argument n

    li $v0, 1           # Base case: if n <= 1
    ble $a0, 1, fact_done

    addi $a0, $a0, -1   # n-1
    jal factorial        # Recursive call
```

```
    lw $a0, 0($sp)      # Restore n
    mul $v0, $a0, $v0   # n * factorial(n-1)

fact_done:
    lw $ra, 4($sp)      # Restore return address
    addi $sp, $sp, 8    # Deallocate stack space
    jr $ra
```

Fibonacci Example (Complete C code with main):

```c
#include <stdio.h>

int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}

int main() {
    int n, result;

    printf("Enter a non-negative integer: ");
    scanf("%d", &n);

    result = fib(n);

    printf("Fibonacci(%d) = %d\n", n, result);

    return 0;
}
```

Fibonacci Example (Complete MIPS with main):

```
.data
prompt:     .asciiz "Enter a non-negative integer: "
result_msg: .asciiz "Fibonacci = "

.text
main:
    # Print prompt
    li $v0, 4
    la $a0, prompt
    syscall

    # Read integer input
    li $v0, 5
    syscall
    move $a0, $v0      # Store input in $a0

    # Call fib function
    jal fib

    # Store result
    move $t0, $v0
```

```
    # Print result message
    li $v0, 4
    la $a0, result_msg
    syscall

    # Print result
    li $v0, 1
    move $a0, $t0
    syscall

    # Exit program
    li $v0, 10
    syscall

fib:
    addi $sp, $sp, -12   # Allocate stack space
    sw $ra, 8($sp)       # Save return address
    sw $a0, 4($sp)       # Save n

    # Base cases
    li $v0, 0
    beq $a0, 0, fib_done
    li $v0, 1
    beq $a0, 1, fib_done

    # fib(n-1)
    addi $a0, $a0, -1
    jal fib
    sw $v0, 0($sp)       # Save fib(n-1)

    # fib(n-2)
    lw $a0, 4($sp)       # Restore n
    addi $a0, $a0, -2
    jal fib

    lw $t0, 0($sp)       # Restore fib(n-1)
    add $v0, $t0, $v0    # fib(n-1) + fib(n-2)

fib_done:
    lw $ra, 8($sp)       # Restore return address
    addi $sp, $sp, 12    # Deallocate stack space
    jr $ra
```

5. MIPS Register Usage Convention

Register	Name	Purpose	Preserved?
$0	$zero	Constant value 0	N/A
$1	$at	Assembler temporary	No
$2-$3	$v0-$v1	Function return values	No
$4-$7	$a0-$a3	Function arguments	No

Register	Name	Purpose	Preserved?
$8-$15	$t0-$t7	Temporary registers	No
$16-$23	$s0-$s7	Saved registers	Yes
$24-$25	$t8-$t9	More temporary registers	No
$26-$27	$k0-$k1	Kernel registers	No
$28	$gp	Global pointer	Yes
$29	$sp	Stack pointer	Yes
$30	$fp	Frame pointer	Yes
$31	$ra	Return address	Yes

6. Student Task 1: Convert the example of square root and Sin(θ) using function calls and then find the square root of a positive real number from user input by function call, do similar for converting a user given angle to radian and finding the sin. (code has to be shown to verify function is implemented)

7. Student Task 2: Make two functions nCr and nPr (combination and permutation) from user given numbers, call and use nPr from nCr.

Appendix: C to MIPS Compilation Protocols

General Compilation Steps:

1. **Function Prologue**:

    - Save return address ($ra) and frame pointer ($fp) on stack
    - Adjust stack pointer to allocate space for local variables
    - Set frame pointer to current stack position

2. **Function Body**:

    - Map local variables to stack locations relative to $fp
    - Pass arguments in $a0-$a3, additional arguments on stack
    - Use $t registers for temporary values (caller-saved)
    - Use $s registers for values that persist across calls (callee-saved)

3. **Function Epilogue**:

    - Place return value in $v0 (and $v1 if needed)
    - Restore saved registers from stack
    - Restore return address and frame pointer
    - Adjust stack pointer to deallocate stack frame
    - Return using jr $ra

Stack Frame Structure:

```
□Higher addresses
|---------------|
| ...           |
| Argument 5    | - $fp + 16
| Argument 4    | - $fp + 12
| Saved $ra     | - $fp + 8
| Saved $fp     | - $fp + 4
| Local var 1   | - $fp
| Local var 2   | - $fp - 4
| ...           |
|---------------| - $sp
Lower addresses
```

Calling Convention:

1. Caller saves any $t registers that need to be preserved
2. Caller places first 4 arguments in $a0-$a3, remaining on stack
3. Caller uses jal to jump to function
4. Callee saves $ra, $fp, and any $s registers it will modify
5. Callee executes function code
6. Callee places result in $v0-$v1
7. Callee restores saved registers and stack pointer
8. Callee returns with jr $ra

Important Tips:

- Always preserve $s registers across function calls
- $t registers can be used without preservation but are not guaranteed to persist across calls
- Keep track of stack pointer adjustments to maintain proper stack alignment
- Use the frame pointer to access arguments and local variables consistently