**Homework 9: Compilers**

**CS 200 • 100 Points Total  
Due Wednesday, November 22, 2023**

**Assignment**

This one is fairly tough. It’s almost like a project. For this one, you are the compiler. You need to convert the C program given below into a MIPS program. You’ll want to use MARS and test the code you write to be sure it works correctly. This one problem demonstrates all of Section 9 and is worth the entire 100 points.

Here are the things I’m looking for in your solution:

* There are no global variables other than I/O strings. There are a few number constants (-1, 1, 2, and 20) which can be handled with instructions that use immediate values.
* You should be able to do a while loop and the if/else constructs in assembly. I presented them a couple of sections ago and the text details them in Section 9.
* The subroutine call to fib must pass values on the stack. In fact, because it is recursive, a stack frame must be used. See my suggested layout for the stack frame.
* Local variables for fib are required (result and temp). Don’t just use registers because recursive calls to fib will alter those registers and your code won’t work correctly. Again, see my suggested layout for the stack frame.
* If you use a register in main to hold ‘number’, you should preserve it before calling fib and restore it after fib returns.
* If you use registers in fib, you’ll use them in recursive calls, so preserve them in the stack frame. I suggest you use the ‘T’ registers, so you don’t need to worry about the ‘S’ registers.

Suggested stack frame:

|  |  |  |  |
| --- | --- | --- | --- |
| Pointers | Usage | $sp offset | $fp offset |
| stack pointer | temp | 0 | ? |
|  | result | 4 | ? |
|  | any registers | 8 ꜜ | -20 ꜛ |
|  | original $fp | ? | -16 |
|  | $ra | ? | -12 |
|  | number | ? | -8 |
|  | return | ? | -4 |
| frame pointer |  |  |  |

Depending on how many registers you use in fib, the stack frame size may vary, which is why $sp offsets below the registers or $fp offsets above the registers cannot be determined. Once you determine which registers you will use, you can finish the stack frame diagram.

The C program is on the following page.

#include <stdio.h> // not necessary for the assembly; you’ll use system calls

int fib(int); // forward declaration. Again, no assembly equivalent.

void main(void)  
{  
 int number = -1;

while ((number < 1) || (number > 20))  
{  
 printf("Enter a number (1 - 20): ");  
 scanf("%d", &number);

if (number < 1)  
 {  
 printf("Number must be > 0, try again.\n");  
 }

if (number > 20)  
 {  
 printf("Number must be < 21, try again.\n");  
 }  
 }

int value = fib(number);  
 printf("The fibonacci value is: %d\n", value);  
}

int fib(int n)  
{  
 int result;  
 int temp;

if (n < 2)  
 {  
 result = n;  
 }  
 else  
 {  
 temp = fib(n-2);  
 result = fib(n-1) + temp;  
 }

return result;  
}

# MIPS Assembly equivalent of the C program

# Data segment (global variables, constants)

.data

prompt: .asciiz "Enter a number (1 - 20): "

less\_than\_zero: .asciiz "Number must be > 0, try again.\n"

greater\_than\_twenty: .asciiz "Number must be < 21, try again.\n"

# Text segment (main program)

.text

# Main function

main:

# Initialize stack frame

addi $sp, $sp, -24 # Allocate space on stack for local variables and saved registers

sw $ra, 0($sp) # Save return address

sw $s0, 4($sp) # Save $s0 (if used to hold 'number')

sw $s1, 8($sp) # Save $s1 (if used for calculations)

sw $fp, 12($sp) # Save previous frame pointer

move $fp, $sp # Set up new frame pointer

input\_loop:

# Prompt for input

li $v0, 4

la $a0, prompt

syscall

# Read integer input

li $v0, 5

syscall

move $s0, $v0 # Store 'number' in $s0

# Check input range

bltz $s0, invalid\_input1 # Branch if 'number' < 1

bgt $s0, 20, invalid\_input2 # Branch if 'number' > 20

j input\_validated # Jump if input is valid

invalid\_input1:

# Number < 1

li $v0, 4

la $a0, less\_than\_zero

syscall

j input\_loop

invalid\_input2:

# Number > 20

li $v0, 4

la $a0, greater\_than\_twenty

syscall

j input\_loop

input\_validated:

# Call fib function

move $a0, $s0 # Pass 'number' as argument

jal fib # Jump and link to fib function

move $s1, $v0 # Store return value in $s1

# Display result

li $v0, 1

move $a0, $s1 # Print the result stored in $s1

syscall

# Restore saved registers and return

lw $ra, 0($sp) # Restore return address

lw $s0, 4($sp) # Restore $s0

lw $s1, 8($sp) # Restore $s1

lw $fp, 12($sp) # Restore previous frame pointer

addi $sp, $sp, 24 # Deallocate stack frame

jr $ra # Jump back to caller (return from main)

# Fibonacci function

fib:

# Set up stack frame for fib function

addi $sp, $sp, -8 # Allocate space on stack for local variables

sw $ra, 0($sp) # Save return address

sw $s0, 4($sp) # Save $s0 (for result)

# Retrieve argument 'n' from stack

lw $s0, 12($fp) # Load 'n' from stack frame

# Base case: fib(0) or fib(1)

blt $s0, 2, fib\_base\_case

addi $s0, $s0, -2 # Calculate fib(n-2)

jal fib

move $s1, $v0 # Store fib(n-2) in $s1

lw $s0, 12($fp) # Reload 'n'

addi $s0, $s0, -1 # Calculate fib(n-1)

jal fib

add $s0, $v0, $s1 # result = fib(n-1) + fib(n-2)

fib\_base\_case:

move $v0, $s0 # Store result in $v0

# Restore saved registers and return

lw $ra, 0($sp) # Restore return address

lw $s0, 4($sp) # Restore $s0

addi $sp, $sp, 8 # Deallocate stack frame

jr $ra # Return to caller