**CSCI 2021 Machine Architecture and Organization, Fall 2018, Written Assignment #2**

Moti Begna, begna002, sec: 007

# Problem 1 (20 points)

Consider the following assembly code for a function with a while loop:

Prob1:

jmp .L2

.L5: testb $1, %dil # %dil is the lowest byte of %edi

jne .L3

leal 4(%rsi, %rsi, 2), %eax

addl $5, %esi

sall $3, %edi

jmp .L2

.L3: leal 7(,%rdi,8), %eax addl $9, %esi

sall $2, %edi

.L2: cmpl %esi, %edi ja .L5

ret

Based on the assembly code above, fill in the blanks below in its corresponding C source code. You may only use the source-level C variable names such as n, m, and result. Don’t use register names!

unsigned prob1(unsigned n, unsigned m) { unsigned result;

while(n > m) {

if(n = 1) {

result = result\*3 + 4;

m = m + 5;

n = n<<3;

}

else {

result = result\*8 + 7;

m = m + 9;

n = n << 2;

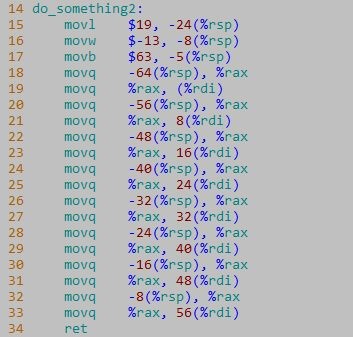
} }

return result;

}

# Problem 2 (20 points)

The following questions are based on the **C code on the next page**​ .​



1. What is the output of the two print statements in lines 41 and 44? Explain this result.

Output:

do\_something1 result: 0 0 0

do\_something2 result: 19 13 ?

Explanation:

In do\_something2, the actual addresses of the bar variables are being passed through, and the values at those addresses are being changed. Thus, when b.x, b.s, and b.b are being called to print, the values that are printed are the value that were changed in those memory addresses.

In do\_something1, the values associated with b.x, b.s, and b.b were only changed locally within the function. Thus, when they are being called to print in main, the values have been unchanged and what is printed is what they were originally set to in main.

1. In Part A, if the expected result differed from the actual result, how might you change the existing code to work as intended?

In order to print the values in do\_something2, we can change the function so that it resembles do\_something1 and pass in the reference of bar when calling the function.

1. Show the partitioning of the bar\_t struct (i.e. show the number of bytes dedicated to each field, as well as any gaps inserted between fields).

foo\_stuffs[5] = 40

x = 4 bytes

4 bytes in gap

f = 8

s = 2

a = 1

b = 1

4 bytes in gap

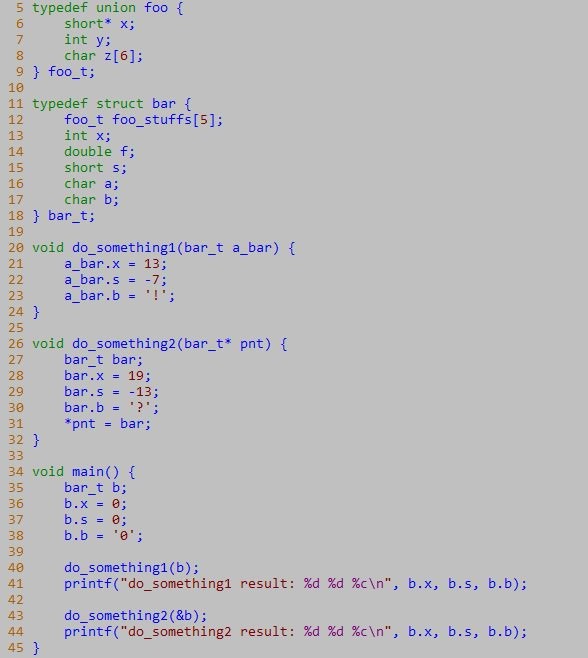
64 bytes in total

1. How many bytes does the foo\_t union require in total?

Because short\*x requires the most bytes out of the other variables, foo\_t requires 88 bytes (40 for foo\_stuffs[5] and 48 for z[6]).

1. Using the assembly code to the left, compiled by GCC, draw the stack frame for do\_something2. You must show each member field for any composite structures.

|  |  |
| --- | --- |
| α | Caller |
| α + 4 | Return address |
| α + 8 | bar\_t bar |
| α + 12 | bar.x = 19 |
| α + 14 | bar.s = -13 |
| α + 18 | bar.b = ‘!’ |
| α + 26 | \*pnt = bar |
|  |  |



# Problem 3 (20 points)

For a C function prob3 with the general structure shown later, gcc generates the following assembly code, including a jump table:

prob3:

cmpq $8, %rdx

ja .L2

jmp\*.L4(,%rdx,8)

.L4:

.quad .L2

.quad .L3

.quad .L5

.quad .L2

.quad .L2

.quad .L6

.quad .L5

.quad .L2

.quad .L7

.L3: leaq (%rsi, %rsi, 2), %rax leaq (%rax, %rax), %rsi addq (%rdi), %rsi

jmp .L8

.L5: leaq (%rsi, %rsi, 2), %rax movq %rdx, %rax

salq $6, %rax

addq %rax, %rsi

jmp .L8

.L6: leaq 80(%rsi), %rax movq %rax, (%rdi)

.L7: movq (%rdi), %rax

leaq (%rax, %rsi, 4), %rsi

jmp .L8

.L2: addq $11, %rsi

.L8: movq %rsi, (%rdi) ret

Based on the assembly code above, fill in the blanks below in its corresponding C source code.

You may only use the source-level C variables x, m, result, and value: don’t use register names!

void prob3(long\* value, long x, long m) {

long result;

switch(m) {

case 1:

result = value + result\*6;

break;

case 2 :

case 6 :

result += (result\*3)<<6;

break;

case 5:

\*value = 80 + result;

case 8:

result = value + result\*4 ;

break;

default:

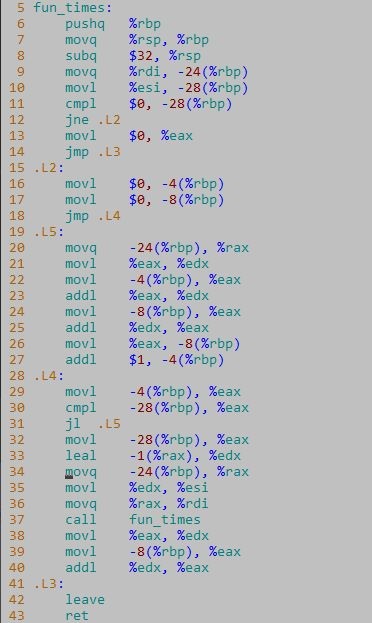
result += 11;

}

\*value = result;

}

# Problem 4 (20 points)



Using the assembly code to the left, answer the following questions.

1. What does fun\_times do? Demonstrate the logic either in C code or in pseudocode.

def fun\_times(int sum, int i)

if (i != 0):

if(i > 0):

sum += i

i++

fun\_times(i - 1, sum)

else:

return sum

1. How many bytes are

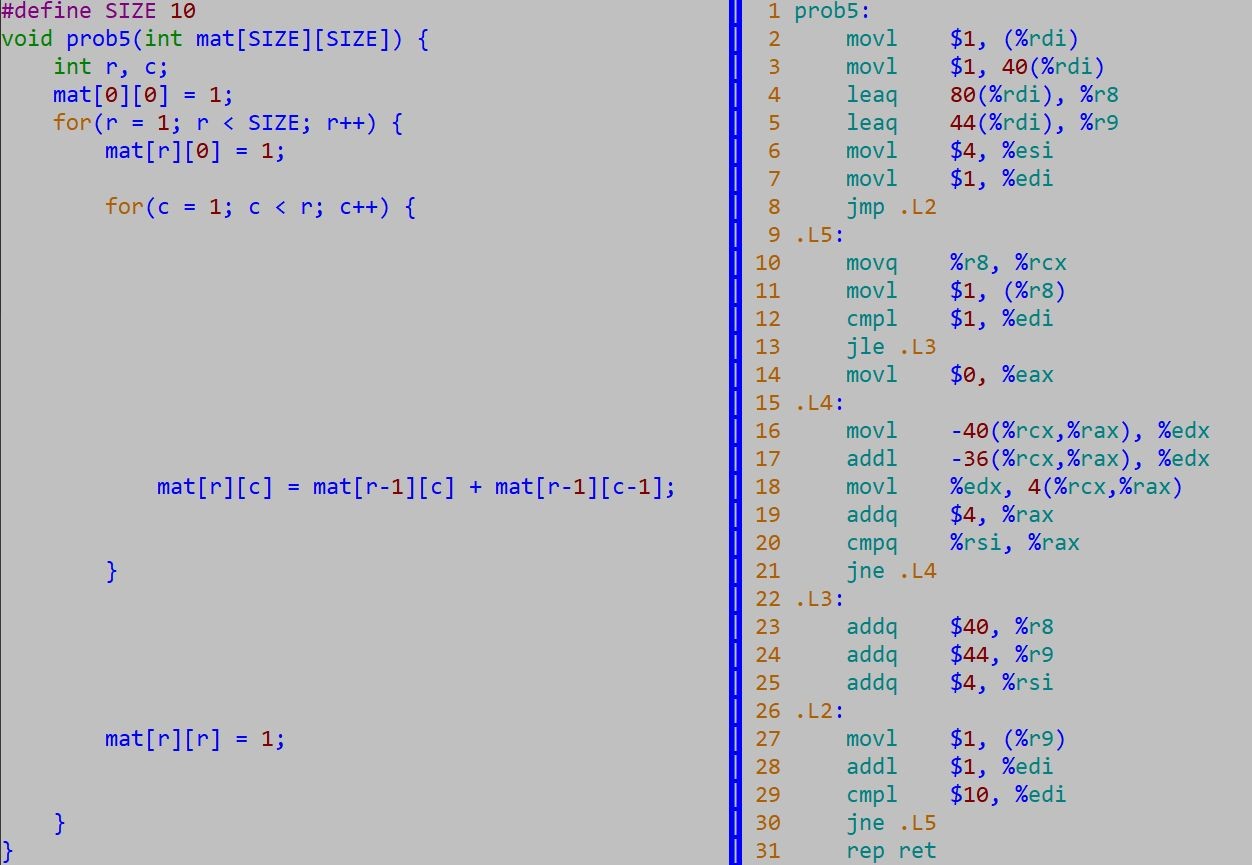
allocated on the stack with each call to fun\_times?

3 bytes

1. Where is the accumulator (i.e. the value calculated by the loop) stored?

The accumulator sum is stored in L3

# Problem 5 (20 points)



Because the compiler has optimized some of the accesses to the array, the registers don’t all correspond exactly to variables in the source code. (And the statements and instructions don’t line up exactly one-to-one either, so don’t put too much significance in the way we’ve spaced the lines). For each of the following registers, as it is used in a particular range of instructions (shown by their assembly code line number), write a C expression that corresponds to the value in the register. Your expressions should be written using the C variables mat, r, and c, together with C operators and constants; don’t use register names.

|  |  |
| --- | --- |
| **Register** | **C expression** |
| %edi, lines 10-30 | mat[0][0] = 1, mat[r][0] = 1 |
| %r8, lines 10-21 | mat[r-1][0] |
| %eax, lines 14-21 | c++ |
| %r9, lines 24-27 | mat[r][0] |
| %esi, lines 10-25 | r++ |