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**.

```
do_something2:
    movl    $19, -24(%rsp)
    movw    $-13, -8(%rsp)
    movb    $63, -5(%rsp)
    movq    -64(%rsp), %rax
    movq    %rax, (%rdi)
    movq    -56(%rsp), %rax
    movq    %rax, 8(%rdi)
    movq    -48(%rsp), %rax
    movq    %rax, 16(%rdi)
    movq    -40(%rsp), %rax
    movq    %rax, 24(%rdi)
    movq    -32(%rsp), %rax
    movq    %rax, 32(%rdi)
    movq    -24(%rsp), %rax
    movq    %rax, 40(%rdi)
    movq    -16(%rsp), %rax
    movq    %rax, 48(%rdi)
    movq    -8(%rsp), %rax
    movq    %rax, 56(%rdi)
    ret
```

A. 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.

B. 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.

C. 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

D. 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]).

E. 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

```c
typedef union foo {
    short* x;
    int y;
    char z[6];
} foo_t;

typedef struct bar {
    foo_t foo_stuffs[5];
    int x;
    double f;
    short s;
    char a;
    char b;
} bar_t;

void do_something1(bar_t a_bar) {
    a_bar.x = 13;
    a_bar.s = -7;
    a_bar.b = '!';
}

void do_something2(bar_t* pnt) {
    bar_t bar;
    bar.x = 19;
    bar.s = -13;
    bar.b = '?';
    *pnt = bar;
}

void main() {
    bar_t b;
    b.x = 0;
    b.s = 0;
    b.b = '0';

    do_something1(b);
    printf("do_something1 result: %d %d %c\n", b.x, b.s, b.b);

    do_something2(&b);
    printf("do_something2 result: %d %d %c\n", b.x, b.s, b.b);
}
```

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)

```
fun_times:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $32, %rsp
    movq    %rdi, -24(%rbp)
    movl    %esi, -28(%rbp)
    cmpl    $0, -28(%rbp)
    jne .L2
    movl    $0, %eax
    jmp .L3
.L2:
    movl    $0, -4(%rbp)
    movl    $0, -8(%rbp)
    jmp .L4
.L5:
    movq    -24(%rbp), %rax
    movl    %eax, %edx
    movl    -4(%rbp), %eax
    addl    %eax, %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    movl    %eax, -8(%rbp)
    addl    $1, -4(%rbp)
.L4:
    movl    -4(%rbp), %eax
    cmpl    -28(%rbp), %eax
    jl  .L5
    movl    -28(%rbp), %eax
    leal    -1(%rax), %edx
    movq    -24(%rbp), %rax
    movl    %edx, %esi
    movq    %rax, %rdi
    call    fun_times
    movl    %eax, %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
.L3:
    leave
    ret
```

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

A. 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
```

B. How many bytes are allocated on the stack with each call to fun_times?

3 bytes

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

The accumulator sum is stored in L3

Problem 5 (20 points)

```c
#define SIZE 10
void prob5(int mat[SIZE][SIZE]) {
    int r, c;
    mat[0][0] = 1;
    for(r = 1; r < SIZE; r++) {
        mat[r][0] = 1;

        for(c = 1; c < r; c++) {









            mat[r][c] = mat[r-1][c] + mat[r-1][c-1];


        }





        mat[r][r] = 1;


    }
}
```

```
1 prob5:
2     movl    $1, (%rdi)
3     movl    $1, 40(%rdi)
4     leaq    80(%rdi), %r8
5     leaq    44(%rdi), %r9
6     movl    $4, %esi
7     movl    $1, %edi
8     jmp .L2
9 .L5:
10    movq    %r8, %rcx
11    movl    $1, (%r8)
12    cmpl    $1, %edi
13    jle .L3
14    movl    $0, %eax
15 .L4:
16    movl    -40(%rcx,%rax), %edx
17    addl    -36(%rcx,%rax), %edx
18    movl    %edx, 4(%rcx,%rax)
19    addq    $4, %rax
20    cmpq    %rsi, %rax
21    jne .L4
22 .L3:
23    addq    $40, %r8
24    addq    $44, %r9
25    addq    $4, %rsi
26 .L2:
27    movl    $1, (%r9)
28    addl    $1, %edi
29    cmpl    $10, %edi
30    jne .L5
31    rep ret
```

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++