Assignment 5- SOLUTION

Assignment 5 - Objectives:

- Investigating the size of some x86-64 machine instructions
- x86-64 function call and stack
- Recursion in x86-64 assembly code

For each of our team-based assignments, you can either:

- Create a group with the same partner,
- Change your group (select a different partner), or
- Decide to work on you own.

Group of two (2):

- You can do Assignment 5 in a group of two (2) or you can work on your own.
- If you choose to work in a group of two (2):
  - Crowdmark will allow you to select your teammate (1). Both of you will only need to submit one assignment and when the TAs mark this one assignment, the marks will be distributed to both of you automatically (just like on CourSys).
  - I doubt Crowdmark will allow you to team up with a student from the other section, but try it and let me know if it works.
  - You can always work with a student from the other section, but both of you will need to submit your assignment separately, i.e., Crowdmark will not consider the both of you as a group.
  - You will need to form the same group of 2 on CourSys.

- If you choose to work on your own:
  - You will need to form a group of 1 on CourSys. (This is the way CourSys works.)

Requirements for this Assignment 5:

- Always show your work (as illustrated in lectures), if appropriate, and
- Make sure the pdf/jpeg/png documents you upload are of good quality, i.e., easy to read, therefore easy to mark! :) If the TA cannot read your work, the TA cannot mark your work and you will get 0. :(

Marking scheme:

- This assignment will be marked as follows:
  - Questions 1, 2 and 3 will be marked for correctness.

- The amount of marks for each question is indicated as part of the question.
- A solution will be posted on Monday after the due date.

Deadline:

- Friday March 3 at 23:59:59 on Crowdmark and CourSys.
- Late assignments will receive a grade of 0, but they will be marked (if they are submitted before the solutions are posted on Monday) in order to provide feedback to the student.

Enjoy!

Q1 (8 points)

Investigating the size of some x86-64 machine instructions

1. First, download the **A5_Q1_Tables.pdf** or **A5_Q1_Tables.docx** from our course web site (under Assignment 5) and open the file with the format you would like to work with: **pdf** or **Word**.

   In this file, you will find five tables. Each of these five tables contain two columns. The first column (the column on the left-hand side) lists x86-64 assembly instructions. The second column (the column on the right-hand side) is intended to list the size (in bytes) of the corresponding x86-64 machine code instruction of each of these x86-64 assembly instructions. As you can see, this column is currently empty.

2. First, have a look at the x86-64 assembly instructions listed in each table. They are all "synonyms" of the same action. This is to say that each of these x86-64 assembly instructions are all doing the same thing. For each table, can you figure out what this action is? Note that in Table 4 and Table 5, the first two x86-64 assembly instructions together are "synonym" of the third instruction.

3. Once you have identified the action associated with each table, you now need to complete its right-hand side column by entering the size (in bytes) of the x86-64 machine code instruction corresponding to each of the listed x86-64 assembly instructions.

   In order to figure out the size (in bytes) of these x86-64 machine code instructions, you must:

- Write one assembly program called **`main.s`** which must include all, and only, the x86-64 assembly instructions found in these five tables. Of course, this program is going to be rather nonsensical and perhaps dangerous. This is not a problem because we shall not be executing it.
- Compile **`main.s`** into the object file **`main.o`** using the provided **`makefile`** (see **makefile for Q1** under Assignment 5 on our course web site).
- Disassemble **`main.o`** using the **`objdump`** tool.
- Looking at the result you obtained from the **`objdump`** tool, compute the size of each of the x86-64 machine code instructions (corresponding to each of the x86-64 assembly instructions listed in **`main.s`**) in bytes and report this size in the right-hand column of the appropriate table.

4. Once you have entered all sizes in all tables, clearly indicate (e.g., by using **bold**) the assembly instruction(s) that has/have the most space efficient corresponding x86-64 machine code instruction (i.e., the shortest x86-64 machine code instruction(s) - least number of bytes) in Table 1, Table 2 and Table 3. In Table 4 and Table 5, clearly indicate which of the two: the first two x86-64 assembly instructions together versus the third instruction, produce the most space efficient corresponding x86-64 machine code instruction.

5. Lastly, copy and paste the **entire** content of your program **`main.s`** after your tables, i.e. in the same document containing your tables and label it **Assembly Code**. Next, copy and paste the **entire** result you obtained from the **`objdump`** tool below your assembly code, in the same document containing your tables, and label it **Disassembled Code**. Create a **pdf** document from this document and upload this **pdf** document at the end of this question on Crowdmark.

SOLUTION:

Table 1

x86-64 instruction (represented here as assembly instructions as opposed to machine instructions)	**The size (in bytes) of the corresponding x86-64 machine instruction**
xorq %rax, %rax	3 bytes
xorl %eax, %eax	**2 bytes**
movq $0, %rax	7 bytes
movl $0, %eax	5 bytes
subl %eax, %eax	**2 bytes**
imull $0, %eax	3 bytes
andl $0, %eax	3 bytes

Table 2

x86-64 instruction (represented here as assembly instructions as opposed to machine instructions)	**The size (in bytes) of the corresponding x86-64 machine instruction**
addl $1, %eax	3 bytes
leal 1(%eax), %eax	4 bytes
incl %eax	**2 bytes**
subl $-1, %eax	3 bytes

Table 3

x86-64 instruction (represented here as assembly instructions as opposed to machine instructions)	**The size (in bytes) of the corresponding x86-64 machine instruction**
addl $8, %eax	**3 bytes**
leal 8(%eax), %eax	4 bytes

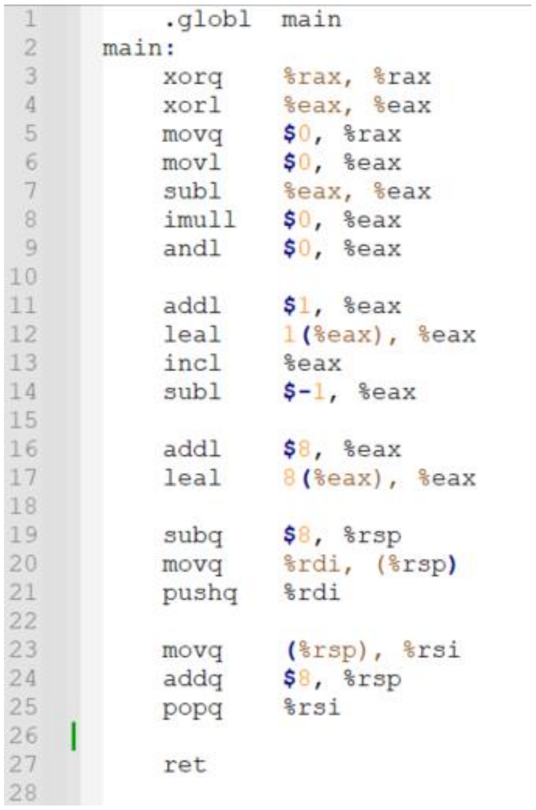
Table 4

x86-64 instruction (represented here as an assembly instruction as opposed to a machine instruction)	**The size (in bytes) of the corresponding x86-64 machine instruction**
subq $8, %rsp	4 bytes
movq %rdi, (%rsp)	4 bytes
pushq %rdi	**1 byte**

Table 5

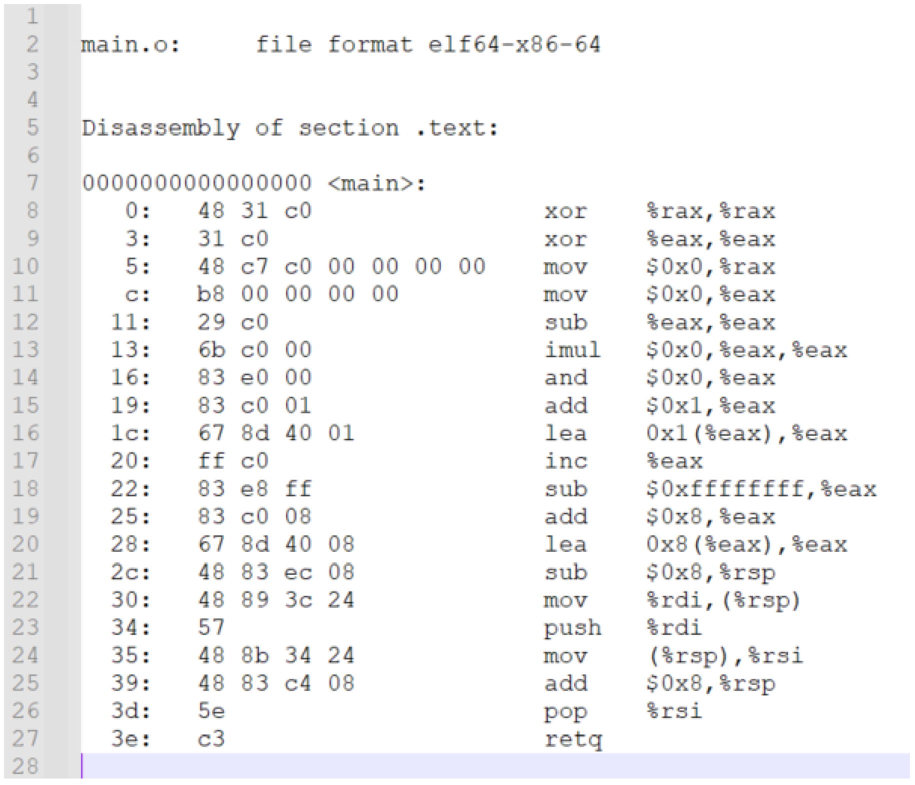
x86-64 instruction (represented here as an assembly instruction as opposed to a machine instruction)	**The size (in bytes) of the corresponding x86-64 machine instruction**
movq (%rsp), %rsi	4 bytes
addq $8, %rsp	4 bytes
popq %rsi	**1 byte**

Assembly Code



```
        .globl  main
main:
        xorq    %rax, %rax
        xorl    %eax, %eax
        movq    $0, %rax
        movl    $0, %eax
        subl    %eax, %eax
        imull   $0, %eax
        andl    $0, %eax

        addl    $1, %eax
        leal    1(%eax), %eax
        incl    %eax
        subl    $-1, %eax

        addl    $8, %eax
        leal    8(%eax), %eax

        subq    $8, %rsp
        movq    %rdi, (%rsp)
        pushq   %rdi

        movq    (%rsp), %rsi
        addq    $8, %rsp
        popq    %rsi

        ret
```

Disassembled Code



```

main.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   48 31 c0                xor    %rax,%rax
   3:   31 c0                   xor    %eax,%eax
   5:   48 c7 c0 00 00 00 00    mov    $0x0,%rax
   c:   b8 00 00 00 00          mov    $0x0,%eax
  11:   29 c0                   sub    %eax,%eax
  13:   6b c0 00                imul   $0x0,%eax,%eax
  16:   83 e0 00                and    $0x0,%eax
  19:   83 c0 01                add    $0x1,%eax
  1c:   67 8d 40 01             lea    0x1(%eax),%eax
  20:   ff c0                   inc    %eax
  22:   83 e8 ff                sub    $0xffffffff,%eax
  25:   83 c0 08                add    $0x8,%eax
  28:   67 8d 40 08             lea    0x8(%eax),%eax
  2c:   48 83 ec 08             sub    $0x8,%rsp
  30:   48 89 3c 24             mov    %rdi,(%rsp)
  34:   57                      push   %rdi
  35:   48 8b 34 24             mov    (%rsp),%rsi
  39:   48 83 c4 08             add    $0x8,%rsp
  3d:   5e                      pop    %rsi
  3e:   c3                      retq
```

Q2 (12 points)

x86-64 function call and stack

Part 1

1. Do Lab 4 on Monday Feb. 27 and/or Tuesday Feb. 28 in CSIL (during our lab sessions) before.

2. First, download **A5_Q2_Stack_Diagram.pdf** or **A5_Q2_Stack_Diagram.docx** from our course web site (under Assignment 5) and open this file with the format you would like to work with: **pdf** or **Word**.

3. Using the **stack diagram** found in the above file, hand trace the code given in our **Lab 4** (`main.c`, `main.s`, `p1.c`, `p1.s`, `p2.c` and `p2.s`) using the test case, i.e., `x = 6`, `y = 9`, `buf[40]` and reflect the result of your hand tracing on this stack diagram.
   Note: Hand trace the entire program until you reach (but have not yet executed) the `ret` instruction of the `main` function.

   The use of the **Register Table** is optional: use it only if you find it useful. You do not have to include it as part of your answer to this question.

4. As you are hand tracing and updating the stack diagram, make sure you attend to the following details:
   - Indicate the movement of `%rsp` along the left-hand side of the stack diagram (under the column named **Base + Displacement**) by crossing its old location and rewriting `%rsp` to indicate its new location (as we have done in our lectures and in Lab 4).
   - Strikethrough the content of the stack that has been popped and/or dealt with.
   - When updating the content of a stack location, strikethrough its old value and write the new value in the same stack location.
   - Include the content of `buf[ ]`, i.e., the actual value stored in `buf[ ]`, in your stack diagram. Do not simply write `buf[ ]` over this section of the stack as we did in Lab 4. This will help you answering the question in Part 2. Remember that each character is a byte.
   - Draw a line just below each **stack frame** to clearly indicate where each of them ends. Also, under the column named **Purpose** on the right-hand side, label each stack frame using the name of its associated function.

5. When drawing your stack diagram, you do not have to show the effect on the stack of the five (5) call instructions at Line 33, Line 43, Line 45 and Line 57 in `main.s` and at Line 47 in `p1.s`. These are calls to `printf(…)`, `puts(…)` and `sprintf(…)`. In other words, you do not have to add the return addresses associated with these five (5) calls onto the stack.

Part 2

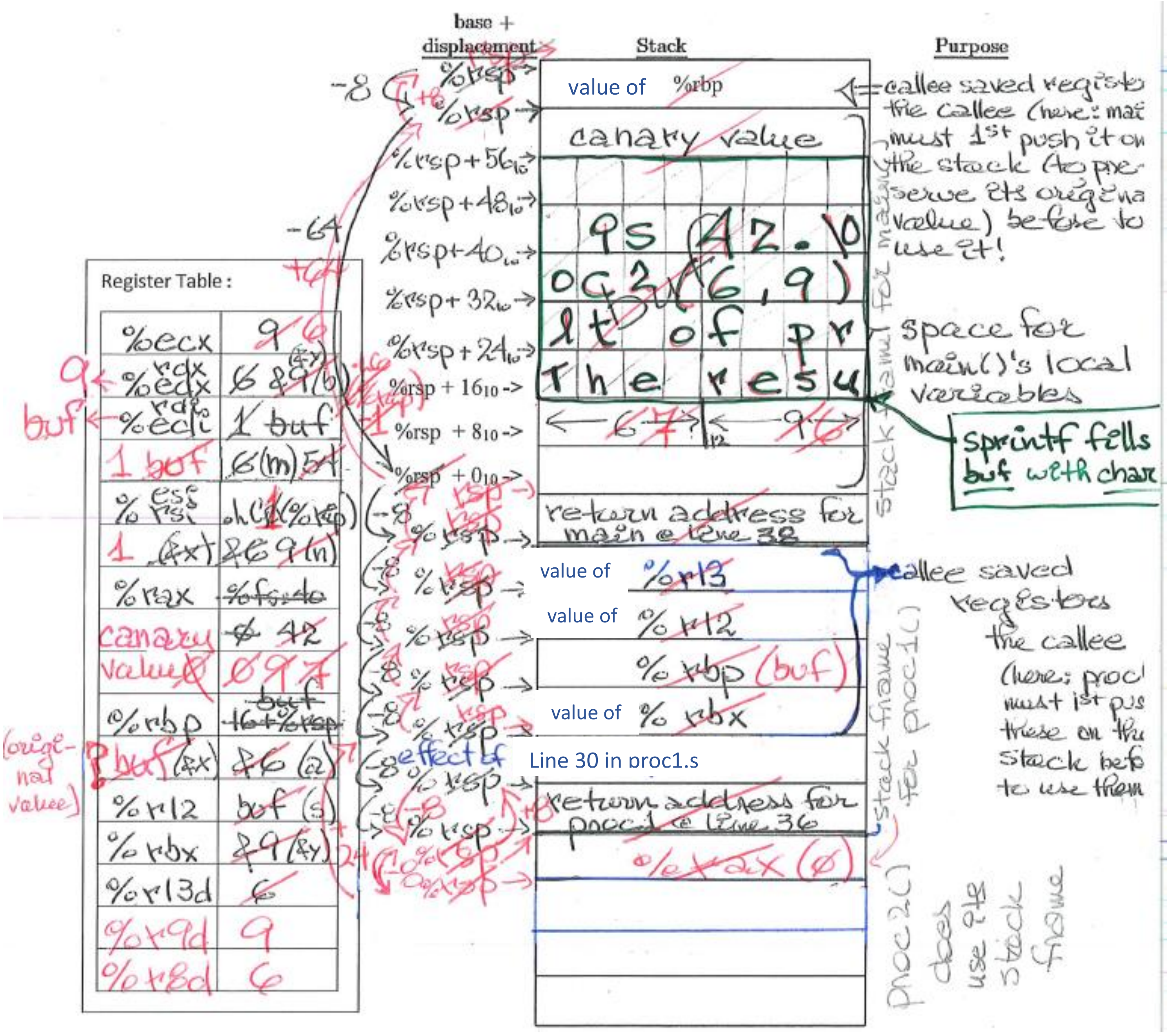
In this second part, we shall investigate what happens to the **canary value** when we change the size of the array called `buf[ ]`.

1. Modify the code in `main.c` by reducing the size of `buf[ ]` from `40` down to `24` as you did in Lab 4. Compile and execute this modified program. What happens?

2. On the second page of the file you downloaded in Part 1 (the file containing the stack diagram) answer the question: What happens to the **canary** value when you reduce the size of `buf[ ]` from `40` down to `24`, and `x = 6`, `y = 9`?
   To figure out what happens, you can hand trace the assembly code of this modified program creating a second drawing of the stack diagram. If you follow the instructions in Part 1 and attend to details as you are drawing this second stack diagram, it should reveal what is happening to the **canary** value.

Lastly, save your **A5_Q2_Stack_Diagram** document (with answers to Part 1 and Part 2) as a **pdf** document and upload it at the end of this question on Crowdmark.

Part 1 - SOLUTION


base + displacement
Stack
Purpose
value of %rbp
canary value
95 (42.1
0C2 (6, 9)
lt of pr
The resu
← 6 7 → ← 9 6 →
return address for main @ line 38
value of %r13
value of %r12
%rbp (buf)
value of %rbx
effect of Line 30 in proc1.s
return address for proc1 @ line 36
%eax (0)
%rsp
%rsp + 56
%rsp + 48
%rsp + 40
%rsp + 32
%rsp + 24
%rsp + 16
%rsp + 8
%rsp + 0
-8
-64
+64
callee saved registers the callee (here: main) must 1st push it on the stack (to preserve its original value) before to use it!
space for main()'s local variables
sprintf fills buf with char
stack frame for main
callee saved registers the callee (here: proc1) must 1st push these on the stack before to use them
stack frame for proc1()
proc2() does use its stack frame

Register Table :

%ecx	9 ~~6~~
%edx / %rdx	~~6~~ &9 (&y) (b)
%edi / %rdi	~~1~~ ~~buf~~
~~1 buf~~	6 (m) ~~54~~
%esi / %rsi	~~.h C(1(%rsp)~~
1 (&x)	&6 ~~9~~ (n)
%rax	~~%fs:40~~
canary value	~~0~~ ~~42~~
value ~~0~~	~~0~~ ~~9~~ ~~7~~
%rbp	~~buf~~ ~~16+%rsp~~
~~buf~~ (&x) (original value)	~~&6~~ (a)
%r12	~~buf~~ (s)
%rbx	~~&9~~ (&y)
%r13d	~~6~~
%r9d	9
%r8d	6

Part 2

What happens to the **canary** value when you reduce the size of `buf[ ]` from `40` down to `24`, and `x = 6`, `y = 9`?

SOLUTION:

Asnwer: The *canary value* gets overwritten by the last few characters (“` is 42.\0`”) of the variable `buf`.

The program prints:

```
Original values are: x=6, y=9.
Final values are: x=7, y=6.
The result of proc2(6,9) is 42.
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Q3 (6 points)

Recursion in x86-64 assembly code

In this question, you are asked to rewrite the `mul` function you wrote in Assignment 4. This time, instead of using a loop, you are to use recursion. You must use the **stack**, either by pushing/popping or by getting a section of it, e.g.,:

```
subq $factor_of_8, %rsp
```

and releasing it, e.g.,:

```
addq $factor_of_8, %rsp
```

at the end of your function.
Use your files from Assignment 4: `main.c`, `makefile` and `calculator.s`. Keep the content of your `calculator.s` as is, i.e., keep the code of all the functions you dealt with in Assignment 4. Then copy the following and paste it over (replace) your entire `mul` function in `calculator.s`:

```
mul: # performs integer multiplication - when both operands are non-negative!
# x in edi, y in esi
# Requirements:
# - cannot use imul* instruction
# - you must use recursion (no loop) and the stack
```

Then implement this recursive version of `mul`. While doing so, you must satisfy its new requirements. You must also satisfy the requirements below.

Requirements:

- Your code must be commented such that others (i.e., TA's) can read your code and understand what each instruction does.
  - About comments:
    - Comment of Type 1: Here is an example of a useful comment:

      ```
      cmpl  %edx, %r8d         # if j (%r8d) < N (%edx), continue looping
      ```

    - Comment of Type 2: Here is an example of a **not so** useful comment:

      ```
      cmpl  %edx, %r8d         # compare %r8d to %edx
      ```

      Make sure you write comments of Type 1. :)

- You cannot modify the prototype of the function `mul`. The reason is that your code may be tested using a test driver built based on this function prototype.
- Make sure you update the header comment block in `calculator.s`.
- You are free to name your label(s) as you wish, however, make sure they are descriptive.
- You must follow the **x86-64 function call** and **register saving** conventions described in class and in the textbook.
- Do not push/pop registers unless you make use of them in your function. Memory accesses are expensive!
- Your code must compile using `gcc` and execute on our ***target machine***.

Lastly, submit your modified `calculator.s` on **CourSys** and write the word **Done!** in the answer space below so that Crowdmark does not think you haven't completed this question of Assignment 5. Do not upload your file in the answer space below.

Possible Solution 1:

```
mul: # performs integer multiplication - when both operands are
non-negative!
# x in edi, y in esi
# Requirements:
# - cannot use imul* instruction
# - you must use recursion (no loop) and the stack

#####################
# Check base case
     xorl      %eax, %eax
     cmpl      $0, %esi
     jle       done
# Save caller saved registers - parameter values - before the
recursive call
     pushq     %rdi
     pushq     %rsi
# Call setup - Prep'ing parameters for recursive call - %rdi
already contained x
     decl      %esi           # y is decremented
# Call myself
     call      mul
# Get original value of x and y for this invocation of mul
     popq      %rsi           # not necessary since y is not
used, but we are practicing!
     popq      %rdi           # not necessary since x unchanged,
but we are practicing!
```

```
# Actual computation
     addl      %edi, %eax     # add x to running total
# Done!
done:
     retq
```

```
Possible Solution 2:

mul: # performs integer multiplication - when both operands are
non-negative!
# x in edi, y in esi
# Requirements:
# - cannot use imul* instruction
# - you must use recursion (no loop) and the stack

#####################
# Check base case
     xorl      %eax, %eax
     cmpl      $0, %esi
     jle       done
# Get stack space
     subq      $32, %rsp      # 32 to create padding (i.e.,
empty spaces on stack)
# Save caller saved registers - parameter values before the
recursive call
     movl      %edi, 16(%rsp)
     movl      %esi, 8(%rsp)
# Call setup - Prep'ing parameters for recursive call
     movl      16(%rsp), %edi # x unchanged - not necessary, but
we are practicing!
     movl      8(%rsp), %esi  # y changed, but not used - not
necessary, but we are practicing!
     decl      %esi           # y is decremented
# Call myself
     call      mul
# Get original value of x for this invocation of mul
     movl      16(%rsp), %edi # not necessary since x unchanged,
but we are practicing!
# Actual computation
     addl      %edi, %eax     # add x to running total
# Clean up stack
     addq $32, %rsp
done:
     retq
```

Base + Displacement	`M[]` **Stack Segment**	**Purpose**

Register Table:
