

Problem 1. (10 points):

Consider the source code below, where M and N are constants declared with #define.

```
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j)
{
    return mat2[i][j] += mat1[j][i];
}
```

A. Suppose the above code generates the following assembly code:

```
sum_element:
    movslq    %edi, %rdi
    movslq    %esi, %rsi
    leaq      0(,%rdi,8), %rax
    subq      %rdi, %rax
    leaq      (%rdi,%rax,4), %rdx
    leaq      (%rsi,%rsi,8), %rax
    leaq      (%rsi,%rax,2), %rax
    addq      %rsi, %rdx
    leaq      (%rax,%rdi), %rdi
    movl      mat2(,%rdx,4), %eax
    addl      mat1(,%rdi,4), %eax
    movl      %eax, mat2(,%rdx,4)
    ret
```

What are the values of M and N?

M =

N =

Problem 2. (10 points):

Consider the following assembly code for a C `for` loop:

```
decode_me:
    cmpl    %esi, %edi
    jle     .L5
    leal    (%rdi,%rdi), %edx
    movl    $1, %eax
    subl    %esi, %edx
.L4:
    subl    $1, %edi
    addl    $4, %esi
    addl    %edx, %eax
    subl    $6, %edx
    cmpl    %esi, %edi
    jg      .L4
    addl    $46, %eax
    ret
.L5:
    movl    $47, %eax
    ret
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: you may only use the symbolic variables `x`, `y`, and `result` in your expressions below — *do not use register names.*)

```
int decode_me(int x, int y)
{
    int result;

    for (result = 1; _____; x--, y = y + 4 ) {
        _____;
    }

    return _____;
}
```

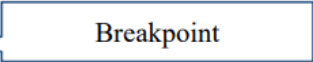
Problem 3. Stack Discipline (20 points)

Examine the following recursive function:

```
long sunny(long a, long *b) {
    long temp;
    if (a < 1) {
        return *b - 8;
    } else {
        temp = a - 1;
        return temp + sunny(temp - 2, &temp);
    }
}
```

Here is the x86_64 assembly for the same function:

```
0000000000400536 <sunny>:
400536:    test    %rdi,%rdi
400539:    jg      400543 <sunny+0xd>
40053b:    mov     (%rsi),%rax
40053e:    sub     $0x8,%rax
400542:    retq
400543:    push    %rbx
400544:    sub     $0x10,%rsp
400548:    lea     -0x1(%rdi),%rbx
40054c:    mov     %rbx,0x8(%rsp)
400551:    sub     $0x3,%rdi
400555:    lea     0x8(%rsp),%rsi
40055a:    callq   400536 <sunny>
40055f:    add     %rbx,%rax
400562:    add     $0x10,%rsp
400566:    pop     %rbx
400567:    retq
```



We call **sunny** from **main()**, with registers **%rsi = 0x7ff...ffad8** and **%rdi = 6**. The value stored at address **0x7ff...ffad8** is the long value 32 (0x20). We set a breakpoint at “**return *b - 8**” (i.e. we are just about to return from **sunny()** without making another recursive call). We have executed the **sub** instruction at **40053e** but have not yet executed the **retq**.

Fill in the register values on the next page and draw what the stack will look like when the program hits that breakpoint. Give both a description of the item stored at that location and the value stored at that location. If a location on the stack is not used, write “unused” in the Description for that address and put “----” for its Value. You may list the Values in hex or decimal. Unless preceded by **0x** we will assume decimal. It is fine to use **f...f** for sequences of **f**’s as shown above for **%rsi**. Add more rows to the table as needed. Also, fill in the box on the next page to include the value this call to **sunny** will *finally* return to **main**.

Register	Original Value	Value <u>at Breakpoint</u>
rsp	0x7ff...ffad0	
rdi	6	
rsi	0x7ff...ffad8	
rbx	4	
rax	5	

DON'T
FORGET



What value is **finally** returned to **main** by this call?



Memory address on stack	Name/description of item	Value
0x7fffffffffffffffad8	Local var in main	0x20
0x7fffffffffffffffad0	Return address back to main	0x400827
0x7fffffffffffffffac8		
0x7fffffffffffffffac0		
0x7fffffffffffffffab8		
0x7fffffffffffffffab0		
0x7fffffffffffffffaa8		
0x7fffffffffffffffaa0		
0x7fffffffffffffff998		
0x7fffffffffffffff990		
0x7fffffffffffffff988		
0x7fffffffffffffff980		
0x7fffffffffffffff978		
0x7fffffffffffffff970		
0x7fffffffffffffff968		
0x7fffffffffffffff960		

Problem 4. (4+6 points):

A. Consider an implementation of a processor where the combinational circuit latency is α ns (nanosecond). You are going to pipeline this implementation, and in your system the latency of each pipeline register is β ns. If you are to divide this entire combinational circuit in k stages, what is the throughput of your pipelined implementation? What about the total latency of a single instruction?

B. Now, assume that the parameters α , β and k described above have the following relation:

$$\beta = \begin{cases} \alpha/10 & \text{if } k \leq 5, \\ \alpha/10 + k/50 \bullet 9\alpha/10 & \text{if } k > 5. \end{cases} \quad (1)$$

Based on the equation above, clearly present an analysis when would you pipeline this processor. Please consider three different values of k in your analysis: (i) $k = 5$; (ii) $k = 50$; and (iii) $k = 500$.