Problem 1. *Address generation.

* The following C code fragment transposes array A into array B.

```
#define M 40
#define N 50
void transpose(short A[M][N], B[N][M]) {
  for (int i = 0; i < M; i++) // 40
    for (int j = 0; i < N; j++) // 50
      B[j][i] = A[i][j];
 }
```

(a) code for the assignment statement `B[j][i] = A[i][j]`;

```
[%rdi: A, %rsi: B, %r8d: i, %r9d: j]
Calculation:
[i][j] = i*50 + j = i*2*(8*3+1) + j
[j][i] = j*40 + i = j*2*(8 + 2) + i
Code:
// calculate [i][j] into %rax
LEAQ (, %r8d, 8), %rax // i*(8)
LEAQ (%rax, %rax, 2), %rax // i*(8*3)
LEAQ (%r8d, %rax), %rax // i*(8*3+1)
LEAQ (%r8d, %rax, 2), %rax // i*2*(8*3+1) + j

// calculate [j][i] into %rbx
LEAQ (, %r9d, 8), %rbx // j*(8)
LEAQ (%rbx, %r9d, 2), %rbx // i*(8 + 2)
LEAQ (%r9d, %rbx, 2), %rbx // j*2*(8 + 2) + i

MOVQ, (%rdi, %rax), %rax //  A[i][j] to %rax
MOVQ, %rax, (%rsi, rbx) // %rax to B[j][i]
```

(b) Simplify the generated code by exploiting regularities in the sequence of accesses to arrays A and B. Use additional registers if needed.

```
[%rdi: A, %rsi: B, %r8d: i, %r9d: j, %r10: i*50 + j from last loop, %r11: j*40 + i from last loop]

MOVQ, (%rdi, %rax), %rax //  A[i][j] to %rax
MOVQ, %rax, (%rsi, rbx) // %rax to B[j][i]

for incrementing j or i:
LEAQ 1(%r11 or %r10) %r10 or %r11
LEAQ 40 or 50(%r11 or %r10) %r10 or %r11
```

Problem 2. *Code generation.

* The following C code counts the number of 1-bits in the variable x. Hand-translate it into x86-64 assembly language code. The input parameter x is provided to you in register %rdi, and the result needs to be returned in register %rax.

```c
int pop(unsigned x) {
  x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  x = (x + (x >> 4)) & 0x0F0F0F0F;
  x += x >> 8;
  x += x >> 16;
  return x & 0x0000003F;
}
```

Also, it might be instructive to examine the assembly code that gcc generates for these code fragments, and see whether it looks anything like the assembly code you generated by hand.

```
[%rdi: x, %rax: result]

// x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
MOVQ %rdi, %rax
ANDQ $0x55555555, %rax
SARQ 1, %rbx
ANDQ %0x55555555, %rbx
ADDQ %rbx, %rax

        movl    -4(%rbp), %eax
        andl    $1431655765, %eax
        movl    %eax, %edx
        movl    -4(%rbp), %eax
```

```
        shrl    %eax
        andl    $1431655765, %eax
        addl    %edx, %eax
        movl    %eax, -4(%rbp)

// x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
MOVQ %rdi, %rax
ANDQ $0x33333333, %rax
SARQ 2, %rbx
ANDQ %0x33333333, %rbx
ADDQ %rbx, %rax

        movl    -4(%rbp), %eax
        andl    $858993459, %eax
        movl    %eax, %edx
        movl    -4(%rbp), %eax
        shrl    $2, %eax
        andl    $858993459, %eax
        addl    %edx, %eax
        movl    %eax, -4(%rbp)

// x = (x + (x >> 4)) & 0x0F0F0F0F;
MOVQ %rax, %rbx
SARQ 4, %rbx
ADDQ %rax %rbx
ANDQ $0x0F0F0F0F, %rbx
ADDQ %rbx, %rax

        movl    -4(%rbp), %eax
        shrl    $4, %eax
        movl    %eax, %edx
        movl    -4(%rbp), %eax
        addl    %edx, %eax
        andl    $252645135, %eax
        movl    %eax, -4(%rbp)

// x += x >> 8;
MOVQ %rax, %rbx
SARQ 8, %rbx
ADDQ %rbx, %rax

        movl    -4(%rbp), %eax
        shrl    $8, %eax
        addl    %eax, -4(%rbp)

// x += x >> 16;
```

```
MOVQ %rax, %rbx
SARQ 16, %rbx
ADDQ %rbx, %rax

        movl    -4(%rbp), %eax
        shrl    $16, %eax
        addl    %eax, -4(%rbp)

// x & 0x0000003F
ANDQ $0x0F0F0F0F, %rax

        movl    -4(%rbp), %eax
        andl    $63, %eax
```

Problem 3. *Reverse-engineering stack frames.

* The assembly code generated by gcc for a C function with the prototype long bar(long); is as follows.

```
bar:
# prologue
pushq %rbp # callee saved register -- 4 (word)
movq %rsp, %rbp # link
subq $32, %rsp # 0-8: old rbp, 8-32: 3 local vars

# body
movq %rdi, -8(%rbp) # var1 = param1
cmpq $1, -8(%rbp) # cmpq: set flag based on src1 and src2
jg LBB0_2 # jump if 1 > var1
movq $1, -16(%rbp) # var2 = 1
jmp LBB0_3
LBB0_2:
movq -8(%rbp), %rax # rax = var1
movq -8(%rbp), %rcx # rcx = var1
subq $1, %rcx # rcx = rcx - 1

# pre call
movq %rcx, %rdi # param1 = rcx, prepare param for call
movq %rax, -24(%rbp) # var3 = rax, store rax value

# call
```

```
callq bar // recursive call to bar

# post call
movq -24(%rbp), %rcx # rcx = var3, restore return vlaue
imulq %rax, %rcx # rcx = rcx * rax
movq %rcx, -16(%rbp) # var2 = rcx

LBB0_3:
# epilogue
movq -16(%rbp), %rax # rax = var2, store return value
addq $32, %rsp # clear local var on stack
popq %rbp # pop fbr

# return
retq
```

(a) Show the layout of the stack frame for this function. Indicate each of the four areas of the stack frame, how much each area takes, and the total size of the stack frame.

```
old rbp --4
var1 -- 8
var2 -- 8
ret addr -- 8
[tot 32]
```

(b) The function is invoked with an argument value of 2. Show the state of the stack just before program execution reaches the recursive call to bar.

```
bar:
# prologue
pushq %rbp
movq %rsp, %rbp
subq $32, %rsp

# body
movq %rdi, -8(%rbp) # var1 = param1 = 2
cmpq $1, -8(%rbp)
jg LBB0_2 # jump if 1 > var1 =2
movq $1, -16(%rbp)
jmp LBB0_3
LBB0_2:
movq -8(%rbp), %rax # rax = var1 = 2
movq -8(%rbp), %rcx # rcx = var1 = 2
```

```
subq $1, %rcx # rcx = rcx - 1 = 1

# pre call
movq %rcx, %rdi # param1 = rcx = 1
movq %rax, -24(%rbp) # var3 = rax = 2

rbp: old
var1: 2
var2:
ret: 2
```

(c) Show the state of the stack when the function is in the recursive call to bar.

```
bar:
# prologue
pushq %rbp # new
movq %rsp, %rbp
subq $32, %rsp

# body
movq %rdi, -8(%rbp) # var1 = param1 = 1
cmpq $1, -8(%rbp)
jg LBB0_2 # jump if 1 > var1 =1
movq $1, -16(%rbp) # var2 = 1
jmp LBB0_3

rbp: new
var1: 1
var2: 1
var3: 1
```

(d) What value does the function return when invoked with an argument value of 2?

when invoked with 1, return 1:

```
bar:
# prologue
pushq %rbp
movq %rsp, %rbp
subq $32, %rsp

# body
```

```
movq %rdi, -8(%rbp) # var1 = param1 = 1
cmpq $1, -8(%rbp)
jg LBB0_2 # jump if 1 > var1 =1
movq $1, -16(%rbp) # var2 = 1
jmp LBB0_3

LBB0_3:
# epilogue
movq -16(%rbp), %rax # rax = var2 = 1, store return value
addq $32, %rsp # clear local var on stack
popq %rbp # pop fbr

# return
retq # return 1
```

continue on 2, return 2:

```
rax: 1
var1: 2
var2:
var3: 2

# call
callq bar // recursive call to bar

# post call
movq -24(%rbp), %rcx # rcx = var3 = 1
imulq %rax, %rcx # rcx = rcx * rax = 2
movq %rcx, -16(%rbp) # var2 = rcx = 2

LBB0_3:
# epilogue
movq -16(%rbp), %rax # rax = var2 = 2
addq $32, %rsp # clear local var on stack
popq %rbp # pop fbr

# return
retq # return 2
```

- - - -- - - -- -

Problem 4. *Caller-saved and callee-saved registers.

* For each of the C procedures below, identify the minimal sets of caller-saved and callee-saved registers that will be saved/restored in the assembly code generated for the procedure. The normal x86-64 procedure call/return linkage conventions are followed, and each procedure is compiled separately.

```
unsigned long fn1(long x, long y){
  return x*x + y*y;
}

caller-saved:
0, //not calling other functions

callee-saved:
%rbp, // base pointer
%rbx, %rcx // for calculation
```

```
unsigned long fn2(long x, long y){
  return fn1(x+y, y-2);
}

caller-saved:
0, //not using values after function call

callee-saved:
%rbp, // base pointer
%rbx, %rcx // for calculation
```

```
unsigned long fn3(long x, long y){
  return fn1(x, y) - x*y;
}

caller-saved:
%r1, //use after function call

callee-saved:
%rbp, // base pointer
%rbx  // for calculation
```

```
unsigned long fn4(long x, long y){
  y = fn1(y, x);
  return fn2(x,y);
}

caller-saved:
%r1, //use after function call

callee-saved:
%rbp, // base pointer


```

```
unsigned long fn5(long x, long y){
  return fn1(x,y) + fn2(x,y);
}

caller-saved:
%r1, %r2, //use after function call

callee-saved:
%rbp, // base pointer
```