1 x86 Assembly

1.1 Modular Exponentation

Alice wants to implement her own version of **modular exponentiation**, an important primitive operation in public-key cryptography.

She was able to complete most of the subroutine in assembly language:

```
modexp:
        pushl   %r12d
        movl    %edi, %eax
        movl    %edx, %r8d
        movl    $1, %r12d
        testl   #1, #1
        je      .L1
        movl    $0, %edx
        divl    %r8d                ;; eax = eax / r8d; edx = eax % r8d
        movl    %edx, %ecx
        jmp     .L4
.L3:
        imull   %ecx, %ecx          ;; ecx = ecx * ecx
        movl    %ecx, %eax
        movl    $0, %edx
        divl    %r8d                ;; eax = eax / r8d; edx = eax % r8d
        movl    %edx, %ecx
        #2      %esi
        testl   %esi, %esi
        je      .L1
.L4:
        movl    %esi, %edi
        andl    $3, %edi
        cmpl    $1, %edi
        jne     .L3
.L2:
        movl    %r12d, %eax
        imull   %ecx, %eax          ;; eax = ecx * eax
        movl    $0, %edx
        divl    %r8d
        movl    %edx, %r12d
        jmp     .L3
.L1:
        movl    %r12d, #3
        ________#4________
        ret
```

Alice dropped 6.110 last week, so she does not know how to implement the rest of the subroutine. The final program should be equivalent to the following C code:

```
unsigned modexp(unsigned base, unsigned exp, unsigned mod) {
    unsigned result = 1;
    base = base % mod;

    while (exp > 0) {
        if ((exp & 3) == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp = exp >> 1;
    }
    return result;
}
```

She wants her code to follow the x86 calling convention. She knows that **#1** should be a register, that **#2** should be an x86 instruction mnemonic, that **#3** should be a register, and that **#4** should be a missing instruction.

Answer.

1. `%esi`
2. `shrl`
3. `%eax`
4. `popl %r12d`

1.2 One-Time Pad

Eve is trying to reverse engineer Alice's source code. However, Eve has skipped the 6.110 lectures on code generation and has not yet watched the relecture. Help Eve by writing Decaf code that could have resulted in the following x86 code. Assume that Decaf has an XOR (^) operator which works similarly to how it works in C.

Hint. Your solution should include some global declarations.

```
.LC0:
        .string "output[%d] = %d"
key:
        .zero   64
input:
        .zero   64
output:
        .zero   64
main:
        pushq   %rbx
        movl    $0, %eax
.L2:
        movslq  %eax, %rdx
        movl    input(,%rdx,4), %ecx
        movl    key(,%rdx,4), %esi
        xorl    %esi, %ecx
        movl    %ecx, output(,%rdx,4)
        addl    $1, %eax
        cmpl    $16, %eax
        jne     .L2
        movl    $0, %ebx
.L3:
        movslq  %ebx, %rax
        movl    output(,%rax,4), %edx
        movl    %ebx, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        addl    $1, %ebx
        cmpl    $16, %ebx
        jne     .L3
        popq    %rbx
        ret
```

Answer. Your code does not need to match exactly, but should be close enough.

```
import printf;

int output[16];
int input[16];
int key[16];

void main() {
    int i;
    int j;
    for (i = 0; i < 16; i++) {
        output[i] = input[i] ^ key[i];
    }

    for (j = 0; j < 16; j++) {
        printf(”output[%d] = %d”, j, output[j]);
    }
}
```

2 Control-Flow Graphs

2.1 The Pulverizer

Bob is implementing a key exchange system that requires computing the modular inverse of an integer a modulo b. To do this, Bob writes a procedure implementing the **Extended Euclidean Algorithm** which computes,

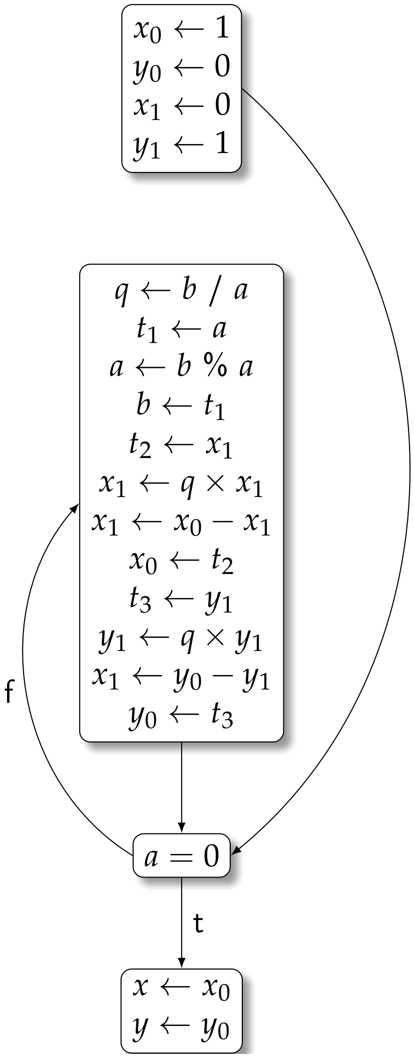
$$a \cdot x + b \cdot y = \gcd(a, b)$$

Bob wrote the following implementation in Decaf:

```
void extended_gcd(int a, int b) {
    int x, y;
    int x0, y0, x1, y1;
    x = 0;
    y = 0;
    x0 = 1;
    y0 = 0;
    x1 = 0;
    y1 = 1;
    while (a != 0) {
        int q, t1, t2, t3;
        q = b / a;
        t1 = a;
        a = b % a;
        b = t1;
        t2 = x1;
        x1 = x0 - q * x1;
        x0 = t2;
        t3 = y1;
        y1 = y0 - q * y1;
        y0 = t3;
    }
    x = x0;
    y = y0;
    printf("x = %d; y = %d\n", x, y);
}
```

Construct the control-flow graph for Bob's `extended_gcd` procedure. Clearly indicate all basic blocks and control flow edges.

Answer.



$x_0 \leftarrow 1$
$y_0 \leftarrow 0$
$x_1 \leftarrow 0$
$y_1 \leftarrow 1$

$q \leftarrow b \;/\; a$
$t_1 \leftarrow a$
$a \leftarrow b \;\%\; a$
$b \leftarrow t_1$
$t_2 \leftarrow x_1$
$x_1 \leftarrow q \times x_1$
$x_1 \leftarrow x_0 - x_1$
$x_0 \leftarrow t_2$
$t_3 \leftarrow y_1$
$y_1 \leftarrow q \times y_1$
$x_1 \leftarrow y_0 - y_1$
$y_0 \leftarrow t_3$

$a = 0$

$x \leftarrow x_0$
$y \leftarrow y_0$