Example Assembly Problems

Problem 1:
Consider the following pairs of C functions and assembly code. Fill in the missing instructions in the assembly code (one instruction per a blank). Your answers should be syntactically correct assembly.

```c
int goose()
{
    return -4;
}
```

goose:
```
pushl %ebp
movl %esp, %ebp
___________________________
popl %ebp
ret
```

```c
int cow(int a, int b)
{
    return a - b;
}
```
cow:
```
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
___________________________
popl %ebp
ret
```

```c
int pig(int a)
{
    return a*3;
}
```
pig:
```
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
movl %eax, %eax
_____________________________________
popl %ebp
ret
```
int sheep(int c) {
    if (c < 0)
        return 1;
    else
        return 0;
}

int duck(int a) {
    if (sheep(a))
        return -a;
    else
        return a;
}
Problem 2:
This problem tests your understanding of IA32 condition codes.

A. Consider the instruction:

\[ \text{cmp} \ a, b \]

Write in the values (0 if clear, 1 if set) of the condition flags if this instruction is executed with the given values of \( a \) and \( b \).

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>Zero Flag (ZF)</th>
<th>Sign Flag (SF)</th>
<th>Carry Flag (CF)</th>
<th>Overflow Flag (OF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>0xffffffffc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0xffffffffc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0x80000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffffff</td>
<td>0x80000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80000000</td>
<td>0x7ffffffff</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x7ffffffff</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80000000</td>
<td>0x80000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7ffffffff</td>
<td>0xffffffffc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B. On an IA32 architecture, compare and test instructions aren’t the only instructions which set the condition codes and conditional branches aren’t the only instructions which read the condition codes. Specifically, the \texttt{adc} instruction sets the condition codes based on the result and the \texttt{add with carry} instruction (\texttt{adc}) computes the sum of its two operands and the carry flag. That is, \texttt{adc} \%edx, \%eax computes \( eax = eax + edx + CF \). Briefly describe a specific instance where the compiler can make use of this combination of instructions.
### Problem 3:
Consider the following C functions and assembly code:

```c
int fun1(int i, int j)
{
    if(i+3 != j)
        return i+3;
    else
        return j*16;
}

int fun2(int i, int j)
{
    if(i+3 != (unsigned)j)
        return i;
    else
        return j*4;
}

int fun3(int i, int j)
{
    if(i+3 <= (unsigned)j)
        return i;
    else
        return j>>2;
}
```

Which of the functions compiled into the assembly code shown?
Problem 4:
Consider the following C function and assembly code fragments:

```c
int woohoo(int a, int r)
{
    int ret = 0;
    switch(a)
    {
        case 11:
            ret = 4;
            break;
        case 22:
        case 55:
            ret = 7;
            break;
        case 33:
        case 44:
            ret = 11;
            break;
        default:
            ret = 1;
    }
    return ret;
}
```

```assembly
Fragment 1

woohoo:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl $0, %ecx
    cmpl $11, %edx
    jne .L2
    movl $4, %ecx
    jmp .L3
.L2:
    cmpl $22, %edx
    jne .L3
    movl $7, %ecx
.L3:
    cmpl $55, %edx
    jne .L5
    movl $7, %ecx
.L5:
    cmpl $33, %edx
    sete %al
    cmpl $44, %edx
    sete %dl
    orl %edx, %eax
    testb $1, %al
    je .L6
    .L6:
    movl %ecx, %eax
    popl %ebp
    ret
```
Which of the assembly code fragments matches the C function shown?
Problem 5:
This problem tests your understanding of how for loops in C relate to IA32 machine code. Consider the following IA32 assembly code for a procedure dog():

dog:
    pushl  %ebp
    movl  %esp, %ebp
    movl  12(%ebp), %ecx
    movl  $1, %eax
    movl  8(%ebp), %edx
    cmpl  %ecx, %edx
    jge   .L7
.
.L5:
    imull  %edx, %eax
    addl  $2, %edx
    cmpl  %ecx, %edx
    jl    .L5
.
.L7:
    popl  %ebp
    ret

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

int dog(int x, int y)
{
    int i, result;

    result = ________;

    for (i = ________; _____________; ________) {
        result = ________________;
    }
}

return result;
}
Problem 6:
This problem tests your understanding of how while loops in C relate to IA32 machine code. Consider the following IA32 assembly code for a procedure cat():

```assembly
cat:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ecx
    pushl %ebx
    xorl %ebx, %ebx
    movl 12(%ebp), %eax
    decl %ecx
    cmpl $-1, %ecx
    je .L6
    movl %ecx, %edx
    imull %eax, %edx
    negl %eax
    .p2align 4,,15
    .L4:
    decl %ecx
    addl %edx, %ebx
    addl %eax, %edx
    cmpl $-1, %ecx
    jne .L4
    .L6:
    movl %ebx, %eax
    popl %ebx
    popl %ebp
    ret
```

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

```c
int cat(int x, int y) {
    int i, ret;
    ret = ________;
    i = ________;
    while(_____________)
    {
        ret = ___________________;  
    }
    return ret;
}
```
Problem 7:
This problem tests your understanding of how switch statements in C relate to IA32 machine code. Consider the following IA32 assembly code for a procedure frog():

```assembly
frog:
pushl  %ebp
movl  %esp, %ebp
movl  8(%ebp), %edx
movl  12(%ebp), %eax
cmpl  $7, %edx
ja   .L8
jmp   *.L9(%edx,4)
.section .rodata
.align 4
.align 4
.L9:
.long  .L8
.long  .L4
.long  .L8
.long  .L5
.long  .L8
.long  .L4
.long  .L6
.long  .L2
.text
.L4:
movl  $7, %eax
jmp   .L2
.L5:
decl  %eax
jmp   .L2
.L6:
incl  %eax
jmp   .L2
.L8:
movl  $-1, %eax
.L2:
popl  %ebp
ret
```
Based on the assembly code, fill in the blanks below in its corresponding C source code. (Note: you may only use symbolic variables \(a\), \(b\), and \(result\), from the source code in your expressions below — do not use register names.)

```c
int frog(int a, int b)
{
    int result;

    switch(_______)
    {
        case ______:
            ______;
            break;
        case ______:
            ______;
            break;
        case ______:
            ______;
            break;
        case ______:
            ______;
            break;
        case 7:
            ______;
            break;
        default:
            ______;
    }
    return result;
}
```
Problem 8:
This problem tests your understanding of the stack discipline and byte ordering. Consider the following C functions and assembly code:

```c
void top_secret(int len)
{
    char buf[8];
    scanf("%s", buf);
    if(strlen(buf) != len)
        exit(1);
}

int main()
{
    printf("Enter a passphrase: ");
    top_secret(8);
    printf("The chicken flies at midnight!\n");
    return 0;
}
```

```assembly
08048530 <top_secret>:
8048530: 55 push %ebp
8048531: 89 e5 mov %esp,%ebp
8048533: 83 ec 08 sub $0x8,%esp
8048536: 8d 45 f8 lea 0xfffffff8(%ebp),%eax
8048539: 50 push %eax
804853a: 68 40 86 04 08 push $0x8048640
804853f: e8 44 fe ff ff call 8048388 <scanf>
8048544: 8d 45 f8 lea 0xffffffff8(%ebp),%eax
8048547: 50 push %eax
8048548: e8 5b fe ff ff call 80483a8 <strlen>
804854d: 83 c4 0c add %0xc,%esp
8048550: 3b 45 08 cmp 0x8(%ebp),%eax
8048553: 74 0b je 8048560 <top_secret+0x30>
8048555: 6a 01 push 50x1
8048557: e8 8c fe ff ff call 80483e8 <exit>
804855c: 8d 74 26 00 lea 0x0(%esi,1),%esi
8048560: 89 ec mov %ebp,%esp
8048562: 5d pop %ebp
8048563: c3 ret
```
Here are some notes to help you work the problem:

- `scanf("%s", buf)` reads an input string from the standard input stream (stdin) and stores it at address `buf` (including the terminating `\0` character). It does **not** check the size of the destination buffer.

- `strlen(s)` returns the length of the null-terminated string `s`.

- `exit(1)` halts execution of the current process without returning.

- Recall that Linux/x86 machines are Little Endian.

You may find the following diagram helpful to work out your answers.

![Diagram]

```plaintext
ebp →
0x0c
0x08
0x04
0x00
0xfc
0xf8
0xf4
0xf0
0xec
0xe8
0xe4
0xe0
0xdc
0xd8
0xd4
0xd0
```
A. **Circle the address** (relative to ebp) of the following items. Assume that the code has executed up to (but not including) the call to `scanf` at 0x804853f).

- **return address**: 0xc 0x08 0x04 0x00 0xfc 0xf8 0xf4 0xf0 0xec 0xe8 0xe4 0xe0
- **saved %ebp**: 0xc 0x08 0x04 0x00 0xfc 0xf8 0xf4 0xf0 0xec 0xe8 0xe4 0xe0
- **len**: 0xc 0x08 0x04 0x00 0xfc 0xf8 0xf4 0xf0 0xec 0xe8 0xe4 0xe0
- **&buf**: 0xc 0x08 0x04 0x00 0xfc 0xf8 0xf4 0xf0 0xec 0xe8 0xe4 0xe0
- **%esp**: 0xc 0x08 0x04 0x00 0xfc 0xf8 0xf4 0xf0 0xec 0xe8 0xe4 0xe0
- **&“%s”**: 0xc 0x08 0x04 0x00 0xfc 0xf8 0xf4 0xf0 0xec 0xe8 0xe4 0xe0

B. Let us enter the string “chickenstonight” (not including the quotes) as a password. Inside the top_secret function `scanf` will read this string from stdin, writing its value into `buf`. Afterwards what will be the value in the 4-byte word pointed to by %ebp? You should answer in hexadecimal notation.

The following table shows the hexadecimal value for relevant ASCII characters.

<table>
<thead>
<tr>
<th>Character</th>
<th>Hex value</th>
<th>Character</th>
<th>Hex value</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘c’</td>
<td>0x63</td>
<td>‘h’</td>
<td>0x68</td>
</tr>
<tr>
<td>‘i’</td>
<td>0x69</td>
<td>‘k’</td>
<td>0x6b</td>
</tr>
<tr>
<td>‘e’</td>
<td>0x65</td>
<td>‘n’</td>
<td>0x6e</td>
</tr>
<tr>
<td>‘s’</td>
<td>0x73</td>
<td>‘t’</td>
<td>0x74</td>
</tr>
<tr>
<td>‘o’</td>
<td>0x6f</td>
<td>‘g’</td>
<td>0x67</td>
</tr>
<tr>
<td>‘h’</td>
<td>0x68</td>
<td>\0</td>
<td>0x00</td>
</tr>
</tbody>
</table>

(%ebp) = 0x________________________
C. The function `top_secret` is called from a 5-byte `call` instruction at the address 0x804857f inside of `main`. Before the first instruction of `top_secret` (0x08048530) is executed, the registers contain the following values:

<table>
<thead>
<tr>
<th>Register</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x14</td>
</tr>
<tr>
<td>ecx</td>
<td>0x0</td>
</tr>
<tr>
<td>edx</td>
<td>0x0</td>
</tr>
<tr>
<td>ebx</td>
<td>0x40157770</td>
</tr>
<tr>
<td>esp</td>
<td>0xbffff98c</td>
</tr>
<tr>
<td>ebp</td>
<td>0xbffff998</td>
</tr>
<tr>
<td>esi</td>
<td>0x40015e8c</td>
</tr>
<tr>
<td>edi</td>
<td>0xbffffa04</td>
</tr>
<tr>
<td>eip</td>
<td>0x8048530</td>
</tr>
</tbody>
</table>

The program continues to execute until it hits the `lea` instruction at 0x8048544 (right after the call to `tt scanf`). The user inputs ‘chickens’. Fill in the full 4-byte hexadecimal values for the following memory locations. If a value is cannot be computed from the information given, write “unknown”.

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffff990</td>
<td>unknown</td>
</tr>
<tr>
<td>0xbffff98c</td>
<td>unknown</td>
</tr>
<tr>
<td>0xbffff88</td>
<td>unknown</td>
</tr>
<tr>
<td>0xbffff84</td>
<td>unknown</td>
</tr>
<tr>
<td>0xbffff80</td>
<td>unknown</td>
</tr>
<tr>
<td>0xbffff7c</td>
<td>unknown</td>
</tr>
<tr>
<td>0xbffff78</td>
<td>unknown</td>
</tr>
</tbody>
</table>
Problem 9:
This problem tests your understanding of the IA32 calling convention. Consider the following C code and corresponding assembly. Fill in the missing instructions (one instruction per a blank line).

```c
int global;

int bear(int i, int j, int k)
{
    for( ; i < j; i++)
    {
        global += k*i;
    }
    return global;
}
```

```assembly
bear:
pushl  %ebp
movl  %esp, %ebp

__________________
__________________
movl  8(%ebp), %edx
movl  12(%ebp), %ebx
movl  16(%ebp), %esi
cmpl  %ebx, %edx
jge   .L7
movl  global, %ecx

.L5:
    movl  %esi, %eax
    imull  %edx, %eax
    leal  (%ecx,%eax), %ecx
    incl  %edx
cmpl  %ebx, %edx
    jl    .L5
movl  %ecx, global

.L7:

__________________
__________________
__________________
popl   %ebp
ret
```
**Problem 10:**
The following problem will test your understanding of stack frames. It is based on the following function:

```c
int scrat(int val, int n)
{
    int result = 0;
    if(n > 0)
        result = val + scrat(val, n-1);
    return result;
}
```

A compiler on an IA-32 Linux machine produces the following object code for this function, which we have disassembled (using `objdump`) back into assembly code:

```
08048390 <scrat>:
  8048390: 55 push %ebp
  ->8048391: 89 e5 mov %esp,%ebp
  8048393: 53 push %ebx
  8048394: 83 ec 08 sub $0x8,%esp
  8048397: 8b 5d 08 mov 0x8(%ebp),%ebx
  804839a: 8b 45 0c mov 0xc(%ebp),%eax
  804839d: ba 00 00 00 00 mov $0x0,%edx
  80483a2: 85 c0 test %eax,%eax
  80483a4: 7e 10 jle 80483b6 <scrat+0x26>
  80483a6: 48 dec %eax
  80483a7: 89 44 24 04 mov %eax,0x4(%esp,1)
  80483ab: 89 1c 24 mov %ebx,(%esp,1)
  80483ae: e8 dd ff ff ff call 8048390 <scrat>
  80483b3: 8d 14 18 lea (%eax,%ebx,1),%edx
  80483b6: 80 48 08 add $0x8,%esp
  80483bb: 5b pop %ebx
  80483bc: 5d pop %ebp
  80483bd: c3 ret
```

A. On the next page, you have the diagram of the stack immediately after some function makes a call to `scrat` and the very first instruction of `scrat` has executed (the next instruction to be executed is denoted with an arrow (->). The value of register %esp at this point is 0xbffff998. For each of the numeric values shown in the table, give a short description of the value. If the value has a corresponding variable in the original C source code, use the name of the variable as its description.

B. Assume that `scrat` runs until it reaches the position denoted with an arrow (->) again. In the table on the next stage, fill in the updated stack. Use a numeric value (if possible, else write n/a) and provide a short description of the value. Cross out any stack space not used.

C. Which instruction (give its address) computes the result of addition?

```
0x______________________
```
<table>
<thead>
<tr>
<th>Address</th>
<th>Numeric Value</th>
<th>Comments/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffff9a4</td>
<td>0x00000003</td>
<td></td>
</tr>
<tr>
<td>0xbffff9a0</td>
<td>0x00000021</td>
<td></td>
</tr>
<tr>
<td>0xbffff99c</td>
<td>0x080483db</td>
<td></td>
</tr>
<tr>
<td>0xbffff998</td>
<td>0xbfffffa8</td>
<td></td>
</tr>
<tr>
<td>0xbffff994</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff990</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff98c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff988</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff984</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff980</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff97c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff978</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff974</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff970</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff97c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff978</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xbffff974</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>