Problem 1 (20 points) Optimization

Perform at least five standard compiler optimizations on the following C code fragment by writing the optimized version (in C) to the right. Assume \( f \) and \( g \) are pure functions that each return an integer with no side effects to other data structures.

```c
int mycode(int w, int z) {
    int x = 256;
    int y = 1;
    while (y<x+z) {
        if (x)
            z = f(w*x, y, z+w*x);
        else
            z = g(z+w*x, y, w*x);
        printf("y:%d, z:%d\n",y,z);
        y += z;
    }
    while (x>0)
        printf("%d\n", g(y, --x, z));
    return y;
}
```

Briefly describe which standard compiler optimizations you applied:

1. constant propagation (x=256)
2. strength reduction (w*256 to w<<8)
3. common subexpression elimination (temp = w<<8)
4. dead code elimination (if (256) … always nonzero, so reduce to then clause only)
5. loop invariant removal (temp=w<<8 moved outside loop)
Problem 2 (2 parts, 20 points) Conditionals: Compound Predicates

Part A (8 points) Consider the following MIPS code fragment. The comment indicates which variable each register holds. These variables are of type int and are initialized elsewhere.

<table>
<thead>
<tr>
<th>Label</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># $2: I, $3: C, $9: Count, $8: temp</td>
</tr>
<tr>
<td></td>
<td>slt $8, $3, $0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne $8, $0, Next</td>
<td></td>
</tr>
<tr>
<td></td>
<td>slti $8, $3, 26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>beq $8, $0, Next</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addi $9, $9, 1</td>
<td></td>
</tr>
<tr>
<td>Next:</td>
<td>addi $2, $2, 1</td>
<td></td>
</tr>
</tbody>
</table>

What is the equivalent C code fragment? For maximum credit, use a compound logical predicate wherever possible.

```c
if ((c >= 0) && (c < 26))
    count++;
    i++;
```

Part B (12 points) Turn this C code fragment into the equivalent MIPS code. Assume $1 holds A, $2 holds B, $3 holds C and $4 holds D. For maximum credit, include comments and use a minimal number of instructions.

```c
if (A && B)
    C = C | D;
else
    C = C & D;
    D = C * 8;
```

<table>
<thead>
<tr>
<th>Label</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>beq $1, $0, Else</td>
<td># if !A, branch to Else</td>
</tr>
<tr>
<td></td>
<td>beq $2, $0, Else</td>
<td># else if !B, branch to Else</td>
</tr>
<tr>
<td></td>
<td>or $3, $3, $4</td>
<td># else (if A&amp;&amp;B), C=C</td>
</tr>
<tr>
<td></td>
<td>j End</td>
<td># jump over Else</td>
</tr>
<tr>
<td></td>
<td>and $3, $3, $4</td>
<td># if !A</td>
</tr>
<tr>
<td></td>
<td>sll $4, $3, 3</td>
<td># D = C*8</td>
</tr>
</tbody>
</table>

End:
Problem 3 (3 parts, 24 points) 

Part A (8 points) Suppose we have an associative set of 125 (key, value) pairs implemented as a sorted singly linked list. An application performs 1500 lookups of various keys: 1200 of the lookups find the key in the list and 300 lookups fail to find the key. The keys that are found are distributed throughout the list so that each position is equally likely to be where a key is found. What is the average number of key comparisons that would be needed for a lookup in this list implementation? (Show work. Note: you may not have to use all data provided.)

$L = 125$

Number comparisons: \( \frac{(125+1)}{2} = 63 \)

| number of comparisons: | 63 |

Part B (8 points) Suppose the associative set is reimplemented as an open hash table. The same 125 (key, value) pairs are stored in the hash table and are evenly distributed across 25 buckets, each implemented as an unsorted singly linked list. An application performs the same 1500 lookups in which 1200 find the key being searched for and 300 do not. The keys that are found are distributed throughout the bucket lists so that each bucket and each position in the bucket lists is equally likely to be where a key is found. What is the average number of key comparisons that would be needed for a lookup in this hash table implementation? (Show work. Note: you may not have to use all data provided.)

$L = \frac{125}{5}$ elements/bucket

Number comparisons = \( \frac{(1200/1500)(5+1)}{2} + \frac{(300/1500)*5}{2} \)

= \( \frac{4}{5}(3) + \frac{1}{5}*5 = 3.4 \)

| number of comparisons: | 3.4 |

Part C (8 points) Suppose we have a video snippet containing $L$ image frames, where each frame has width $w$ and height $h$ pixels. Complete the following procedure which sets a pixel at position $(x, y)$ in frame number $f$ to $Color$, where $y$ gives the row and $x$ gives the column, with $(0, 0)$ at the top lefthand corner of the image frame, as in Project 3. Assume $L$, $w$ and $h$ are globally defined. $VideoPixels$ is a pointer to the base of the video pixel array containing all $L$ image frames in a contiguous linear sequence starting with the first pixel in the first row of frame 0 and ending with the last pixel in the last row of frame $L-1$.

```c
void SetPixel(int x, int y, int f, uint32_t* VideoPixels, uint32_t Color){
    VideoPixels[f*w*h + y*w + x] = Color;
}
```
Problem 4 (4 parts, 21 points)

Below is a snapshot of heap storage. Values that are pointers are denoted with a “$”. The heap pointer is $6188$. The heap has been allocated contiguously beginning at $6000$, with no gaps between objects.

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6000</td>
<td>8</td>
<td>6032</td>
<td>12</td>
<td>6064</td>
<td>0</td>
<td>6096</td>
<td>16</td>
<td>6128</td>
<td>12</td>
</tr>
<tr>
<td>6004</td>
<td>33</td>
<td>6036</td>
<td>28</td>
<td>6068</td>
<td>4</td>
<td>6100</td>
<td>$6052$</td>
<td>6132</td>
<td>$6120$</td>
</tr>
<tr>
<td>6008</td>
<td>$6132$</td>
<td>6040</td>
<td>$6120$</td>
<td>6072</td>
<td>$6132$</td>
<td>6104</td>
<td>$6016$</td>
<td>6136</td>
<td>$6016$</td>
</tr>
<tr>
<td>6012</td>
<td>16</td>
<td>6044</td>
<td>80</td>
<td>6076</td>
<td>8</td>
<td>6108</td>
<td>5</td>
<td>6140</td>
<td>72</td>
</tr>
<tr>
<td>6016</td>
<td>$6100$</td>
<td>6048</td>
<td>16</td>
<td>6080</td>
<td>24</td>
<td>6112</td>
<td>148</td>
<td>6144</td>
<td>20</td>
</tr>
<tr>
<td>6020</td>
<td>$6172$</td>
<td>6052</td>
<td>0</td>
<td>6084</td>
<td>$6172$</td>
<td>6116</td>
<td>8</td>
<td>6148</td>
<td>6046</td>
</tr>
<tr>
<td>6024</td>
<td>25</td>
<td>6056</td>
<td>$6100$</td>
<td>6088</td>
<td>4</td>
<td>6120</td>
<td>32</td>
<td>6152</td>
<td>8</td>
</tr>
<tr>
<td>6028</td>
<td>30</td>
<td>6060</td>
<td>0</td>
<td>6092</td>
<td>80</td>
<td>6124</td>
<td>$6080$</td>
<td>6156</td>
<td>26</td>
</tr>
</tbody>
</table>

Part A (10 points) Suppose the stack holds a local variable whose value is the memory address $6080$. No other registers or static variables currently hold heap memory addresses. List the addresses of all objects in the heap that are not garbage.

Addresses of 6080, 6172, 6016, 6100, 6052

Non-Garbage Objects:

Part B (3 points) If a reference counting garbage collection strategy is being used, what would be the reference count of the object at address $6016$?

Reference count of object at $6016 = 2$

Part C (5 points) If the local variable whose value is the address $6080$ is popped from the stack, which addresses from Part A will be reclaimed by mark and sweep garbage collection strategy, but not by a reference counting strategy? If none, write “none.”

Addresses: 6172, 6016, 6100, 6052

Part D (3 points) What benefit does old-new space (copying) garbage collection provide that a mark and sweep garbage collection strategy does not provide?

Benefit: It consolidates memory to reduce fragmentation, creating larger contiguous blocks of available memory.
Problem 5 (2 parts, 20 points)

Part A (5 points) Write a single MIPS instruction that is equivalent to the following MIPS fragment.

<table>
<thead>
<tr>
<th>Original</th>
<th>Equivalent MIPS statement:</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $1, $0, 0xFF</td>
<td></td>
</tr>
<tr>
<td>sll $1, $1, 16</td>
<td></td>
</tr>
<tr>
<td>lw $4, 0($8)</td>
<td></td>
</tr>
<tr>
<td>and $4, $1, $4</td>
<td></td>
</tr>
<tr>
<td>srl $4, $4, 16</td>
<td></td>
</tr>
<tr>
<td>lbu $4, 2($8)</td>
<td></td>
</tr>
</tbody>
</table>

Part B (15 points) Consider a singly linked list whose elements are Student_t structs defined as:

```c
typedef struct STUDENT
{
    struct STUDENT* next; // Next pointer for linked list
    char* fname;
    char* mname;
    char* lname;
    double average;
    char letterGrade;
} Student_t;

Student_t* head;
Student_t* tail;
```

The global variables head and tail are initially NULL and they hold the head and tail of the list, respectively. Complete the C function AddToList below that adds the student record s to the end of the linked list pointed to by head and tail. This list might or might not be empty. Be sure to update head and tail properly. (The list is unsorted.)

```c
void AddToList(Student_t* s)
{
    if (head == NULL) {
        head = s;
        tail = s;
        return;
    }
    tail->next = s;
    tail = s;
}
```
Problem 6 (40 points)

The function `Bar` (below left) calls function `Foo` after completing code block 1. Write MIPS assembly code that properly calls `Foo`. Include all instructions between code block 1 and code block 2. Symbolically label all required stack entries and give their values if they are known (below right).

```mips
int Bar() {
    int A[] = {25, 36, 49};
    int B = 3;
    int *P;
    (code block 1)
    P = &B;
    (code block 2)
}
```

```
Bar's FP 9900
9888 A[0] 25
9884 B 3

SP, Foo's FP 9856
9876 RA of Bar
9872 PP of Bar 9900
9868 A 9888
9864 P 9884
9860 *P 3

label | instruction | comment
--- | --- | ---
addi $1, $30, -16 | # compute &B
sw $1, -20($30) | # update P
addi $29, $29, -24 | # allocate activation frame
sw $31, 20($29) | # preserve bookkeeping info
sw $30, 16($29) |
addi $2, $30, -12 | # push inputs
sw $2, 12($29) | # A
sw $1, 8($29) | # P
lw $1, 0($1) | # dereference P
sw $1, 4($29) | # push *P
jal Foo | # call Foo
lw $31, 20($29) | # restore bookkeeping info
lw $30, 16($29) |
lw $1, 0($29) | # read return value
sw $1, -4($30) | # store return value in A[2]
addi $29, $29, 24 | # deallocate activation frame
```