Problem 1 (20 points)
Perform at least five standard compiler optimizations on the following C code fragment by writing the optimized version (in C) to the right. Assume \texttt{square}, \texttt{f}, and \texttt{g} are pure functions that each return an integer with no side effects to other data structures.

```c
int square (int n) {
    return (n*n);
}

int f (int m) { ... }

int g (int c, int d) { ... }

int mycode(int a, int b) {
    int x = 100;
    int y = 1, z=64;
    do {
        if (z)
            y += x*\texttt{square}(z);
        else
            y += g(z+a*x, y+a*x);
        printf("x:%d,y:%d\n",x,y);
        x--;
    } while(x>f(y*1024)+g(a, b));
    return g(y, x);
}
```

Optimization

Briefly describe which standard compiler optimizations you applied and how they improve storage and/or execution efficiency in this code example (be specific; e.g., “replaces 2 MIPS operations with 1 operation on every loop iteration”).

1. constant propagation (z=64): reduce storage required and operations to allocate/deallocate it as a local.
2. dead code elimination: “if (64)” – always nonzero, so no need for if or else clause, reduces code length and eliminates branch operation
3. function inlining of square: eliminates overhead of function call and enables additional strength reduction optimization below (y +=x*64*64)
4. strength reduction (y +=x<<12; ... f(y<<10) + tmp): turns multiplication into a simpler shift operation
5. loop invariant removal (g(a, b)): this computation, including the overhead of subroutine call (activation frame allocation/deallocation), is only done once, instead of on every loop iteration.
Problem 2 (2 parts, 30 points)  

Linked Lists

Suppose we have the following definition which is used to create singly linked lists.
```c
typedef struct Link {
    int ID;
    int Value;
    struct Link *Next;
} Link;
```

**Part A** (6 points) Complete the following subroutine which inserts a Link (pointed to by the input parameter NewLink) into the list just after the Link pointed to by the input parameter Before. You may assume that neither input parameter is NULL. Before’s Next field may point to another Link or it may be NULL. NewLink’s Next field is NULL.

```c
void SpliceIn(Link *NewLink, Link *Before){
    NewLink->Next = Before->Next; /* part A*/
    Before->Next = NewLink; /* part A*/
}
```

**Part B** Complete the following recursive subroutine which takes a pointer to the head of a linked list and returns a pointer to a copy of the linked list. Follow the steps specified below.

```c
Link * CopyList(Link *Head) {
    if (Head == NULL) return NULL; /* part B.1 */

    Link *LinkCopy; /* part B.2 */
    LinkCopy = (Link *) malloc (sizeof(Link)); /* part B.3 */

    if (LinkCopy == NULL) { /* part B.4 */
        printf("Error: Insufficient space.");
        exit(1);
    }

    LinkCopy->ID = Head->ID; /* part B.5 */

    LinkCopy->Value = Head->Value; /* part B.5 */

    LinkCopy->Next = CopyList(Head->Next); /* part B.6 */
    return LinkCopy;
}
```

**Part B.1** (3 points) Fill in what should be returned if the list is empty.

**Part B.2** (3 points) Add a local variable called LinkCopy that is a pointer to a Link object.

**Part B.3** (5 points) Allocate space for a Link structure using malloc and make LinkCopy point to the object allocated. Be sure to include appropriate type casting to avoid type errors.

**Part B.4** (3 points) Fill in the test for whether malloc found enough space which controls the print statement.

**Part B.5** (5 points) Copy the values of Head’s ID and Value fields to LinkCopy.

**Part B.6** (5 points) Call CopyList recursively to copy the rest of the list and assign the result to LinkCopy’s Next field.
Problem 3 (3 parts, 25 points)

Consider a hash table that is implemented using the following struct definitions.

```c
#define NUMBUCKETS 5
typedef struct Entry {
    int Key;
    int Value;
    struct Entry *Next;
} Entry;

typedef struct {
    Entry *Buckets[NUMBUCKETS];
    int Size;
} HashTable;
```

**Part A** (6 points) What is the value of each of these (assume a 32 bit system):

- `sizeof(Entry) = ___12___` 
- `sizeof(HashTable) = ___24___`

**Part B** (10 points) Suppose the entries are maintained in a sorted linked in each bucket in order from small to large keys. Complete the C function `Find_Key` that efficiently searches the hash table for an entry corresponding to a specified key (i.e., *it should end the search as early as possible*). It should return a pointer to the matching `Entry` if `Key` is found or return `NULL` if `Key` is not found in the hash table.

```c
Entry *Find_Key(HashTable *HT, int Key) {
    Entry *ThisEntry;
    int Index;
    int Hash(int Key); /* function prototype for hash function */
    Index = Hash(Key);
    ThisEntry = HT->Buckets[Index];
    while((ThisEntry != NULL) && (ThisEntry->Key <= Key)){
        if (ThisEntry->Key == Key)
            return(ThisEntry);
        else ThisEntry = ThisEntry->Next;
    }
    return NULL;
}
```

**Part C** (9 points) Suppose a hash table created using the structs above contains 155 entries total and the entries are evenly distributed across the 5 hash table buckets, each implemented as a sorted linked list of `Entry` structs. An application performs 500 lookups of various keys: 375 of the lookups find the key in the list and 125 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.)

\[ L = \frac{155}{5} = 31 \]

\[ \text{Avg # comparisons} = \frac{(31+1)}{2} = 16 \]

*Average number of comparisons: 16*
Problem 4 (3 parts, 20 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>20</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>$6172</td>
</tr>
<tr>
<td>6008</td>
<td>0</td>
<td>6040</td>
<td>$6120</td>
<td>6072</td>
<td>$6132</td>
<td>6104</td>
<td>6100</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>$6148</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>100</td>
<td>6088</td>
<td>4</td>
<td>6120</td>
<td>32</td>
<td>6152</td>
<td>$6080</td>
</tr>
<tr>
<td>6028</td>
<td>30</td>
<td>6060</td>
<td>0</td>
<td>6092</td>
<td>$6080</td>
<td>6124</td>
<td>$6080</td>
<td>6156</td>
<td>26</td>
</tr>
</tbody>
</table>

Part A (9 points) Suppose the stack holds a local variable whose value is the memory address \$6120 and register $3$ holds memory address \$6016. No other registers or static variables currently hold heap memory addresses. List the addresses of all objects in the heap that would be marked by a mark-and-sweep garbage collection algorithm.

**Addresses of Marked Objects:** \$6120, 6080, 6148, 6000, 6172, 6052, 6016, 6100

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 \$6172?

Reference count of object at \$6172 = \textbf{2}

Part C (8 points) If the local variable whose value is the address \$6120 is popped from the stack, which addresses will be reclaimed by each of the following strategies? If none, write “none.”

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Reclaimed Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Counting:</td>
<td>$6120</td>
</tr>
<tr>
<td>Mark and Sweep:</td>
<td>$6120, 6080, 6148</td>
</tr>
<tr>
<td>Old-New Space (copying):</td>
<td>$6120, 6080, 6148</td>
</tr>
</tbody>
</table>
Problem 5 (3 parts, 30 points)  
MIPS and C programming

Part A (8 points) Suppose the instruction “jal Foo” is executed which changes the values of the following registers to:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$31</td>
<td>3216</td>
</tr>
<tr>
<td>PC</td>
<td>3620</td>
</tr>
</tbody>
</table>

What is the address of the first instruction of the subroutine Foo and what is the address of the jal Foo instruction?

Subroutine Foo starts at address: 3620
Address of jal Foo instruction: 3212

Part B (12 points) Suppose variables A, B, and C are of type int and are stored in registers $1, $2, and $3. Write a MIPS code fragment that computes \( C = A \times \text{min}(A, B) \). Use only registers $0, $1, $2, and $3 and for maximum credit, use a minimal number of instructions and include comments.

<table>
<thead>
<tr>
<th>Label</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min:</td>
<td>slt $3, $1, $2</td>
<td># is A&lt;B?</td>
</tr>
<tr>
<td></td>
<td>bne $3, $0, Min</td>
<td># if so, branch to Min</td>
</tr>
<tr>
<td></td>
<td>mult $1, $2</td>
<td># if not, compute A*B</td>
</tr>
<tr>
<td></td>
<td>j End</td>
<td># and end</td>
</tr>
<tr>
<td></td>
<td>mult $1, $1</td>
<td># Min: A*A</td>
</tr>
<tr>
<td>End:</td>
<td>mflo $3</td>
<td># put result in $3</td>
</tr>
</tbody>
</table>

Part C (10 points) What does the following code fragment print?

```c
int i, j;
for(i=1; i<10; i=i+2) {
    for (j=1; j<10; j=j+2) {
        if (A[i] == B[j]) {
            printf("C: %d, As: %d, Bs: %d.\n", A[i], A[i-1], B[j-1]);
            break;
        }
    }
}
```

C: 1, As: 1999, Bs: 1993
C: 1, As: 2005, Bs: 1993
Problem 6 (40 points)

The function Bar (below left) calls function Foo after completing code block 1. Write MIPS code that properly calls Foo. Include all instructions between code block 1 and code block 2. Note that code block 1 may change the values of the local variables (e.g., assume i can be any value from 0 to 2). Symbolically label all required stack entries and give their initial values if known (below right).

```mips
int Bar() {
    int   A[] = {9, 25, 27};
    int   i = 0;
    int   y = 5;

    (code block 1)
    (code block 2)
}
```

<table>
<thead>
<tr>
<th>label</th>
<th>instruction</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>addi $29, $29, -24</td>
<td># allocate activation frame</td>
</tr>
<tr>
<td></td>
<td>sw $31, 20($29)</td>
<td># preserve bookkeeping info</td>
</tr>
<tr>
<td></td>
<td>sw $30, 16($29)</td>
<td># compute A</td>
</tr>
<tr>
<td></td>
<td>addi $1, $30, -12</td>
<td># push A</td>
</tr>
<tr>
<td></td>
<td>sw $1, 12($29)</td>
<td># compute &amp;y</td>
</tr>
<tr>
<td></td>
<td>addi $2, $30, -20</td>
<td># push &amp;y</td>
</tr>
<tr>
<td></td>
<td>lw $2, -16($30)</td>
<td># compute A[i] by loading i,</td>
</tr>
<tr>
<td></td>
<td>sll $2, $2, 2</td>
<td># scaling it, and adding it to</td>
</tr>
<tr>
<td></td>
<td>add $2, $2, $1</td>
<td># the base address of A</td>
</tr>
<tr>
<td></td>
<td>lw $3, 0($2)</td>
<td># push A[i]</td>
</tr>
<tr>
<td></td>
<td>sw $3, 4($29)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jal Foo</td>
<td># call Foo</td>
</tr>
<tr>
<td></td>
<td>lw $31, 20($29)</td>
<td># restore bookkeeping info</td>
</tr>
<tr>
<td></td>
<td>lw $30, 16($29)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw $2, 0($29)</td>
<td># read return value</td>
</tr>
<tr>
<td></td>
<td>sw $2, 8($1)</td>
<td># store return value in A[2]</td>
</tr>
<tr>
<td></td>
<td>addi $29, $29, 24</td>
<td># deallocate activation frame</td>
</tr>
</tbody>
</table>

Note that code block 1 may change the values of the local variables (e.g., assume i can be any value from 0 to 2). Symbolically label all required stack entries and give their initial values if known (below right).