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.*

```
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;
}
```

⇨

```
int mycode(int w, int z) {
  int temp = w<<8;
  int x = 256; y = 1;
  while (y<256+z)
   {
     z = f(temp, y, z+temp);
     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.

Label	Instruction	Comment
		`# $2: I, $3: C, $9: Count, $8: temp`
	`slt  $8, $3, $0`	
	`bne  $8, $0, Next`	
	`slti $8, $3, 26`	
	`beq  $8, $0, Next`	
	`addi $9, $9, 1`	
`Next:`	`addi $2, $2, 1`	

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

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

Part B (12 points) Turn this C code fragement 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.*

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

Label	Instruction	Comment
	`beq $1, $0, Else`	**`# if !A, branch to Else`**
	`beq $2, $0, Else`	**`# else if !B, branch to Else`**
	`or  $3, $3, $4`	**`# else (if A&&B), C=C|D`**
	`j   End`	**`# jump over Else`**
`Else:`	**`and $3, $3, $4`**	**`# if !A||!B, C=C&D`**
`End:`	**`sll $4, $3, 3`**	**`# D = C*8`**

Problem 3 (3 parts, 24 points) **Associative Sets and 3D Arrays**

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: (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 = 125/5 elements/bucket

Number comparisons = (1200/1500)(5+1)/2 + (300/1500)*5

= (4/5)(3) + (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`.

```
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) **Garbage Collection**

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.

addr	value	addr	value	addr	value	addr	value	addr	value	addr	value
6000	8	6032	12	6064	0	6096	16	6128	12	6160	0
6004	33	6036	28	6068	4	6100	$6052	6132	$6120	6164	0
6008	$6132	6040	$6120	6072	$6132	6104	$6016	6136	$6016	6168	16
6012	16	6044	80	6076	8	6108	5	6140	72	6172	$6016
6016	$6100	6048	16	6080	24	6112	148	6144	20	6176	0
6020	$6172	6052	0	6084	$6172	6116	8	6148	6046	6180	0
6024	25	6056	$6100	6088	4	6120	32	6152	8	6184	0
6028	30	6060	0	6092	80	6124	$6080	6156	26	6188	0

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 Non-Garbage Objects: **6080, 6172, 6016, 6100, 6052**

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) **MIPS and C programming**

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

Original:	Equivalent MIPS statement:
`addi $1, $0, 0xFF`	**`lbu $4, 2($8)`**
`sll $1, $1, 16`	
`lw $4, 0($8)`	
`and $4, $1, $4`	
`srl $4, $4, 16`	

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

```
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.)

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

Problem 6 (40 points) Activation Frames

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).

```
int Bar() {
   int        A[] = {25, 36, 49};
   int        B =  3;
   int        *P;

   (code block 1)

   P = &B;
   A[2] = Foo(A, P, *P);

   (code block 2)

}
```

Bar’s FP	9900	XXX	XXX
	9896	A[2]	49
	9892	A[1]	36
	9888	A[0]	25
	9884	B	3
~~SP~~	9880	P	9884
	9876	RA of Bar	
	9872	FP of Bar	9900
	9868	A	9888
	9864	P	9884
	9860	*P	3
SP, Foo's FP	9856	RV of Foo	

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`