Problem 1 (20 points) **Compilation**

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*** *is a pure function that returns an integer with no side effects to other data structures.*

```
int foo(int g, int h) {

  int p = 1, y, j;

  int x = 0, z = 24;

  for (j=100; j > 0; j--) {

    x += f(j+g+h);

    y = x/z + g*h;

    p *= f(y) - (j+g+h)/128;

  }

  return (p);

}
```

⇨

```
int foo(int g, int h) {

  int p = 1, y, j;

  int x = 0;

  int temp1, t2=g*h t3=g+h;

  for (j=100; j > 0; j--) {

    temp1 = j + t3;

    x += f(temp1);

    y = x/24 + t2;

    p *= f(y) - (temp1)>>7;

  }

  return (p);

}
```

Briefly describe which standard compiler optimizations you applied:

1. **constant propagation (z replaced with 24)**
2. **dead code elimination (declaration and initialization of z)**
3. **common subexpression elimination (temp1 = j + g + h)**
4. **strength reduction (divide replaced with shift)**
5. **loop invariant removal (g*h) and (g+h)**

Problem 2 (2 parts, 30 points) **Packed Pixel Data**

Suppose an image is stored in memory as an array of pixels. As in Homework 2, each pixel is represented as a triple of 8-bit red, green, and blue color components, packed in the lower 24 bits of a 32-bit word, as shown here:

31			0
unused	red component	green component	blue component
31 24	23 16	15 8	7 0

Part A (20 points) Write a MIPS code fragment that reads in the red, green, and blue components of the i^{th} pixel of the image, finds the maximum of these color components, and stores it in register $11. Assume $1 holds *i*, which could be any integer 0, 1, 2,...N-1, where N is the number of pixels in the image. The image is stored in memory starting at base address labeled `Image`. *Modify only registers $1, $2, $3, $4, $5, and $11.*

Label	Instruction	Comment
MaxPixel:	**sll $1, $1, 2**	# Compute address of i_th
	addi $1, $1, Image	# pixel and put it into $1
	lbu $2, 0($1)	# Read R, G, B components of
	lbu $3, 1($1)	# i_th pixel into $2, $3, and
	lbu $4, 2($1)	# $4
	# Find max(R, G, B)	
	# and put it in $11:	
	addi $11, $2, 0	# init max = $2
	slt $5, $11, $3	# is max < $3?
	beq $5, $0, Next	# if not, jump to Next
	addi $11, $3, 0	# else max=$3 ($11: max($2,$3))
Next:	**slt $5, $11, $4**	# is max($2, $3) < $4?
	beq $5, $0, End	# if not, exit
	addi $11, $4, 0	# else max=$4 ($11: max($2,$3,$4))
End:	**jr $31**	# return

Part B (10 points) Suppose we have an image processing application that reads in the pixels in an image array `Image` and unpacks the color components as shown in the C fragment below. Complete the fragment (by filling in the blank) so that it repacks the color components into the same pixel location but with the red and blue components swapped.

```
int  i, Blue, Green, Red;
for (i=0; i<ImageSize; i++){
   /* unpack current color */
   Color = Image[i];
   Blue = Color & 0xFF;
   Green = (Color >> 8) & 0xFF;
   Red = (Color >> 16) & 0xFF;
   /* Repack the pixel color components with Red and Blue swapped. */

   Image[i] = (Blue << 16) | (Green << 8) | Red;

 }
```

Problem 3 (2 parts, 20 points) **Reverse Engineering MIPS Assembly and C**

Part A (10 points) The following MIPS code implements a three-dimensional array access.

```
sll $1, $4, 11
sll $2, $5, 7
add $1, $1, $2
add $1, $1, $6
sll $1, $1, 2
add $1, $1, $3
lw  $2, 0($1)
```

This implements the C code below. The base address of the array **`Video`** is stored in `$3`. Variables **`Frame`**, **`Row`**, **`Col`**, and **`Pixel`** reside in `$4`, `$5`, `$6`, and `$2` respectively. Fill in the array declaration below. Assume a 32-bit operating system.

```
int     Video[8192][   16   ][   128   ]; /* array declaration */

Pixel = Video[Frame][Row][Col]; /* array access */
```

Part B (10 points) Consider the following MIPS code fragment. If $1, $2, $3, and $4 hold variables A, B, C, and D, respectively, what is the equivalent 1-line C statement using compound predicates that this computes? Hint: draw the control flow graph.

Label	Instruction
	`addi $4, $0, 0`
	`beq  $1, $0, TestC`
	`bne  $2, $0, TestC`
	`j End`
`TestC:`	`bne  $3, $0, End`
	`addi $4, $0, 1`
`End:`	`...`

Answer: **`D =    (!A + B)&& !C;`**

Problem 4 (3 parts, 22 points) **Garbage Collection**

Below is a snapshot of heap storage. Values that are pointers are denoted with a “$”. The heap pointer is $6168. 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	6172	6132	$6120	6164	0
6008	$6100	6040	$6120	6072	$6100	6104	16	6136	9	6168	0
6012	16	6044	$6080	6076	8	6108	5	6140	6072	6172	0
6016	80	6048	16	6080	24	6112	148	6144	20	6176	0
6020	8	6052	$6072	6084	$6132	6116	8	6148	6046	6180	0
6024	25	6056	$6080	6088	4	6120	32	6152	8	6184	0
6028	$6004	6060	0	6092	80	6124	$6080	6156	26	6188	0

Part A (10 points) Suppose register $3 holds the address $6004 and the stack holds a local variable whose value is the memory address $6052. 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: 6004, 6052, 6072, 6100, 6080, 6132, 6120

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 $6100?

Reference count of object at $6100 = 2

Part C (9 points) If the local variable whose value is the address $6052 is popped from the stack, which addresses from Part A will be reclaimed by each of the following strategies? If none, write “none.”

Reference Counting:	6052, 6072
Mark and Sweep:	6052, 6072, 6080, 6132, 6120
Old-New Space (copying):	6052, 6072, 6080, 6132, 6120

Problem 5 (2 parts, 18 points) **Linked Lists and Pointers**

Consider a singly linked list whose elements are `Car` structs defined as follows:

```
typedef struct Car {
  int Year;
  int Tag;
  struct Car *Next;
} Car;
```

The global `KnownCars`, which is declared and initialized as follows, holds the head of the list.

```
Car *KnownCars = Null;
```

Part A (10 points) Suppose the list is sorted in order of increasing `Tag` numbers. Complete the C function `Lookup_Car` below that efficiently searches the list for a `Car` that has the `TagNum` given as input. It should return a pointer to the matching `Car` if `TagNum` is found or return `Null` otherwise.

```
Car *Lookup_Car(int TagNum) {
   Car       *ThisCar;
   ThisCar = KnownCars;
   while (( ThisCar != NULL)&&(ThisCar->Tag <= TagNum)){
     if (ThisCar->Tag == TagNum)
       return(ThisCar);
     else
       ThisCar = ThisCar->Next;
   return(NULL);
}
```

Part B (8 points) Consider the procedure `Lookup_Car`. In what region of memory is each of the following allocated? (Put a checkmark in the column of the correct memory region containing each.)

	Static	Heap	Stack	OS
KnownCars (pointer to head of list)	X			
ThisCar (pointer to a Car)			X	
the Car object pointed to by ThisCar		X		
TagNum (integer input to Lookup_Car)			X	

Problem 6 (2 parts, 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;
   int         B[] = {2, 4};

   (code block 1)

   B[0] = Foo(A, &A, B);

   (code block 2)

}
```

Bar's FP 9900	XXX	XXX
9896	A	25
9892	B[1]	4
9888	B[0]	2
9884	**RA**	**n/a**
9880	**FP**	**9900**
9876	**A**	**25**
9872	**&A**	**9896**
9868	**B**	**9888**
SP 9864	**RV**	**n/a**
9860		
9856		

label	instruction	comment
	addi $29, $29, -24	# allocate activation frame
	sw $31, 20($29)	# preserve bookkeeping info
	sw $30, 16($29)	
	lw $1, -4($30)	# read in A
	sw $1, 12($29)	# push A
	addi $1, $30, -4	# compute address of A
	sw $1, 8($29)	# push address of A
	addi $1, $30, -12	# compute base address of array B
	sw $1, 4($29)	# push base address of array B
	jal Foo	# call Foo
	lw $31, 20($29)	# restore bookkeeping info
	lw $30, 16($29)	
	lw $1, 0($29)	# read return value
	sw $1, -12($30)	# store return value in B[0]
	addi $29, $29, 24	# deallocate activation frame