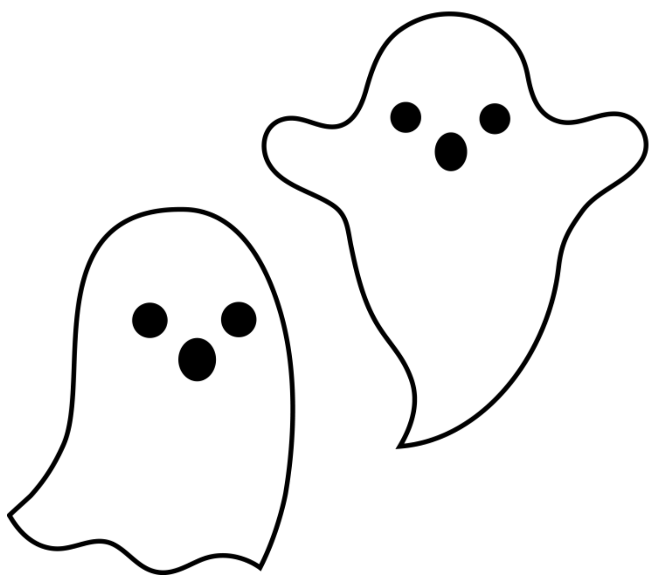
Instructions: This is a closed book, closed note exam. Calculators are not permitted. If you have a question, raise your hand and I will come to you. Please work the exam in pencil and do not separate the pages of the exam. For maximum credit, show your work.
Good Luck!

Your Name (***please print***) ____

This exam will be conducted according to the Georgia Tech Honor Code. I pledge to neither give nor receive unauthorized assistance on this exam and to abide by all provisions of the Honor Code.

Signed ____

1	2	3	total
32	28	40	100



Problem 1 (2 parts, 32 points) **Storage Allocation and Pointers**

Part A (16 points) Assuming a **64-bit system with 64-bit memory interface and 64-bit addresses**, show how the following global variables map into static memory. Assume they are allocated starting at address 4000 and are properly aligned. **For each variable, draw a box showing its size and position** in the double word memory shown below in which byte addresses increment from left to right. **Label the box with the variable name.** Label each element of an array (e.g., M[0]). Note that int and float are still 32-bits.

```
char   Name[] = “Ann”;
double F = 17.75;
int     i = 0;
char   *p = Name;
int    *q = &i;
int     x = 10;
```

4000								
4008								
4016								
4024								
4032								
4040								
4048								
4056								

Part B (16 points) For this part, assume a **32-bit** system, such as MIPS-32.

```
int a = 3;
int b = 5;
char N[] = “Hey!”;
int *p = &a;
char *s = N;
p++;
++s;
printf(“%d\n”, *(p-1);
printf(“%c\n”, N[3]);
printf(“%c\n”, *(s+1));
printf(“%c\n”, *(N+2));
```

Question:	Answer:
How much space (in bytes) is allocated for `p`?	**bytes**
How much space (in bytes) is allocated for `s`?	**bytes**
What is printed by this statement? `printf(“%d\n”, *(p-1);`	
What is printed by this statement? `printf(“%c\n”, N[3]);`	
What is printed by this statement? `printf(“%c\n”, *(s+1));`	
What is printed by this statement? `printf(“%c\n”, *(N+2));`	

Problem 2 (2 parts, 28 points) **Parameter Passing**

Part A (20 points) Consider the following C code fragment.

```
typedef struct {
   int height;
   int width;
} rectangle;

int ComputeArea(int L) {
   int       A = 3;
   int       Scales[] = {2, 4};
   rectangle   R;
   int       ScaleHT(int, rectangle *, int []);
   R.height = 10;
   A = ScaleHT(R.height, &R, Scales);
   return(L+A);
}

int ScaleHT(int h, rectangle *P, int S[]) {
   int         w, area;
   w = S[1]*h;
   P->width = w;
   area = h*w;
   S[0] += 8;
   h++;
   return(h+area);
}
```

For each statement below

from `ScaleHT` (as called from ComputeArea), list the resulting value. If the result is an address, just list "address". Also determine if it changes any of `ScaleHT` activation frame variables, `ComputeArea`'s activation frame variables, or both.

Statement in `ScaleHT`	Result (*assigned value*)	`ComputeArea`'s AF variables changed?	`ScaleHT`'s AF variables changed?
`w = S[1]*h;`		Yes No	Yes No
`P->width = w;`		Yes No	Yes No
`area = h*w;`		Yes No	Yes No
`S[0] += 8;`		Yes No	Yes No
`h++;`		Yes No	Yes No

Part B (8 points) Consider the MIPS code on the left which implements the array declaration and access on the right, where the variables **Z**, **Y**, **X**, and **Value** reside in $4, $5, $6, and $7 respectively.

```
sll $1, $4, 6
sll $2, $5, 4
add $1, $1, $2
add $1, $1, $6
sll $1, $1, 2
sw $7, Array($1)
```

```
int Z, Y, X, Value;
…

int Array[_________][_________][_________];
…
Array[Z][Y][X] = Value;
```

What does this code reveal about the dimensions of Array? Fill in the blanks in the array declaration with the size of each dimension that can be determined from the code. If a dimension cannot be known from this code, put a "?" in its blank. Assume a 32-bit operating system.

Problem 3 (2 parts, 40 points) **Activation Frames**

Consider the following C code fragment:

```
int ComputeSQ(int Max) {
   int         Sum = 0;
   int         Sqs[3];
   int         M[] = {2, 3, 4};
   int         i;
   for(i=0; i<Max; i++){
     SoS(M[i], &Sum, Sqs, i);
   }
   return(Sum);
}

int SoS(int side, int *Total, int S[], int j) {
   int         square;
   square = side*side;
   *Total += square;
   S[j] = square;
   return(square);
}
```

Part A (18 points) Suppose `ComputeSq` has been called with input `Max`=3 and it calls `SoS` 3 times in its `for` loop. Describe the state of the stack at the end of the *first* execution of `SoS`, just before `SoS` deallocates locals and returns to `ComputeSq` for the first time. Fill in the unshaded boxes to show `ComputeSQ's(CSQ's)` and `SoS's` activation frames. Include a symbolic description and the actual value (in decimal) if it has been assigned. For return addresses, show only the symbolic description; do not include a value. *Indicate the value of the frame pointer ($fp) and stack pointer($sp) at this point in execution*. Assume a 32-bit system.

	address	description	Value
	9880	**RA of CSQ's caller**	
	9876	**FP of CSQ's caller**	
	9872	**Max**	**3**
SP, ComputeSQ's FP	9868	**RV**	
	9864		
	9860		
	9856		
	9852		
	9848		
	9844		
	9840		
	9836		
$fp:__________?	9832		
$sp:__________?	9828		
	9824		
	9820		
	9816		
	9812		
	9808		
	9804		
	9800		

Part B (22 points) Write MIPS code fragments to implement the function `SoS` by following the steps below. *Do not use absolute addresses in your code; instead, access variables relative to the frame pointer.* Assume no input parameters are present in registers (i.e., access all parameters from `SoS`'s activation frame). If you assign a register in one part, you may assume it still has that value in a later part. However, changes to variables must update memory.

First, write code to properly set `SoS`'s frame pointer and to allocate space for `SoS`'s local variables and initialize them if necessary.

label	instruction	Comment
SoS:		

 # square = side*side;

label	instruction	Comment

 # *Total += square;

label	instruction	Comment

 # S[j] = square;

label	instruction	Comment

 # return(square); (store return value, deallocate locals, and return)

label	instruction	Comment

MIPS Instruction Set (core)

instruction	***example***	***meaning***
	arithmetic	
add	add $1,$2,$3	$1 = $2 + $3
subtract	sub $1,$2,$3	$1 = $2 - $3
add immediate	addi $1,$2,100	$1 = $2 + 100
add unsigned	addu $1,$2,$3	$1 = $2 + $3
subtract unsigned	subu $1,$2,$3	$1 = $2 - $3
add immediate unsigned	addiu $1,$2,100	$1 = $2 + 100
set if less than	slt $1, $2, $3	if ($2 < $3), $1 = 1 else $1 = 0
set if less than immediate	slti $1, $2, 100	if ($2 < 100), $1 = 1 else $1 = 0
set if less than unsigned	sltu $1, $2, $3	if ($2 < $3), $1 = 1 else $1 = 0
set if < immediate unsigned	sltui $1, $2, 100	if ($2 < 100), $1 = 1 else $1 = 0
multiply	mult $2,$3	Hi, Lo = $2 * $3, 64-bit signed product
multiply unsigned	multu $2,$3	Hi, Lo = $2 * $3, 64-bit unsigned product
divide	div $2,$3	Lo = $2 / $3, Hi = $2 mod $3
divide unsigned	divu $2,$3	Lo = $2 / $3, Hi = $2 mod $3, unsigned
	transfer	
move from Hi	mfhi $1	$1 = Hi
move from Lo	mflo $1	$1 = Lo
load upper immediate	lui $1,100	$1 = 100 x 2^16
	logic	
and	and $1,$2,$3	$1 = $2 & $3
or	or $1,$2,$3	$1 = $2 | $3
and immediate	andi $1,$2,100	$1 = $2 & 100
or immediate	ori $1,$2,100	$1 = $2 | 100
nor	nor $1,$2,$3	$1 = not($2 | $3)
xor	xor $1, $2, $3	$1 = $2 ⊕ $3
xor immediate	xori $1, $2, 255	$1 = $2 ⊕ 255
	shift	
shift left logical	sll $1,$2,5	$1 = $2 << 5 (logical)
shift left logical variable	sllv $1,$2,$3	$1 = $2 << $3 (logical), variable shift amt
shift right logical	srl $1,$2,5	$1 = $2 >> 5 (logical)
shift right logical variable	srlv $1,$2,$3	$1 = $2 >> $3 (logical), variable shift amt
shift right arithmetic	sra $1,$2,5	$1 = $2 >> 5 (arithmetic)
shift right arithmetic variable	srav $1,$2,$3	$1 = $2 >> $3 (arithmetic), variable shift amt
	memory	
load word	lw $1, 1000($2)	$1 = memory [$2+1000]
store word	sw $1, 1000($2)	memory [$2+1000] = $1
load byte	lb $1, 1002($2)	$1 = memory[$2+1002] in least sig. byte
load byte unsigned	lbu $1, 1002($2)	$1 = memory[$2+1002] in least sig. byte
store byte	sb $1, 1002($2)	memory[$2+1002] = $1 (byte modified only)
	branch	
branch if equal	beq $1,$2,100	if ($1 = $2), PC = PC + 4 + (100*4)
branch if not equal	bne $1,$2,100	if ($1 ≠ $2), PC = PC + 4 + (100*4)
	jump	
jump	j 10000	PC = 10000*4
jump register	jr $31	PC = $31
jump and link	jal 10000	$31 = PC + 4; PC = 10000*4