Problem 1 (2 parts, 30 points) **Storage Allocation, Strings, and Pointers**

Part A (20 points) Assuming a **32-bit system with 32-bit memory interface and 32-bit addresses**, show how the following global variables map into static memory. Assume they are allocated starting at address **6000** and are properly aligned. **For each variable, draw a box showing its size and position** in the 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]) and struct (e.g., M.part). Assume all alignment restrictions imposed by the hardware are obeyed, and the compiler does not add additional alignment restrictions. Note: `int` and `float` are 4 bytes, and `double` is 8 bytes.

```
typedef struct{
  int x;
  int y;
  double *scale;
} coord;

char f[] = "Buzz";
char  *h = &f[2];
double z = 4.8;
char   k = '!';
coord  c = {5, 6, &z};
coord *d = &c;
int    e = c.x;
```

6000	**f[0]**	**f[1]**	**f[2]**	**f[3]**
6004	**EOS**	**slack**	**slack**	**slack**
6008	**h**	**h**	**h**	**h**
6012	**z**	**z**	**z**	**z**
6016	**z**	**z**	**z**	**z**
6020	**k**	**slack**	**slack**	**slack**
6024	**c.x**	**c.x**	**c.x**	**c.x**
6028	**c.y**	**c.y**	**c.y**	**c.y**
6032	**c.scale**	**c.scale**	**c.scale**	**c.scale**
6036	**d**	**d**	**d**	**d**
6040	**e**	**e**	**e**	**e**
6044				

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

```
char *S = “twelve+one”;
int  A[] = {2, 3, 5, 4, 9, 8, 6, 0, 1, 7, 10};
int  *V = A;

do {printf(“%c”, *(S + *V++));
   } while(*V != 10);
```

eleven+two

Problem 2 (2 parts, 30 points)	**Accessing Arrays and Structs**

Assuming a 32-bit system, consider the following C fragment:

```
typedef struct{
  int R;
  int G;
  int B;
} Pixel;

Pixel Image[1024][768] = {…};
Pixel *P = Image;

int GetG(int i, int j){
  int Gij = Image[i][j].G;
  return (Gij);
}
```

Part A (10 points) Replace the assignment of `Gij` with a statement that uses only the identifiers `P`, `i`, `j`, and `G` to access the value of `Image[i][j].G`. *Do not use the identifier* `Image` *in your answer*.

`int Gij = (P + 768*i + j)->G` OR `int Gij = (*(P + 768*i + j)).G`

Part B (20 points) Write MIPS code to implement the assignment statement

`int Gij = Image[i][j].G;`

from the code above. Assume that the base address of array **`Image`** is given in **$1**, and the values of variables **i**, and **j** are given in **$2**, and **$3**, respectively. Store the result (**`Gij`**) in register **$4**. (Note: there are more blank lines provided than you need.)

Label	Instruction	Comment
	`addi $5, $0, 768`	
	`mult $5, $2`	
	`mflo $5`	**`# 768*i`**
	`add  $5, $5, $3`	**`# 768*i + j`**
	`addi $6, $0, 12`	
	`mult $5, $6`	**`# mult by |Pixel|:`**
	`mflo $5`	**`# 12(768*i + j)`**
	`add  $5, $5, $1`	**`# + Image base address`**
	`lw   $4, 4($5)`	**`# read G part`**

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

Consider the following C code fragment:

```
int Bar() {
   char        x = ‘i’;
   int         y = 1;
   int         z;
   char        Name[] = “Tom”;
   int         Foo(char [], int, char *);

   z           = Foo(Name, y, &x);
   return(z);
}

int Foo(char S[], int n, char *c) {
   int         a = 3;

   if (S[n]) {
      S[n] = *c;
      n++;
   }
   return(n);
}
```

Part A (18 points) Suppose `Bar` has been called so that the state of the stack is as shown below. Describe the state of the stack <u>just before</u> `Foo` deallocates locals and returns to `Bar`. Fill in the unshaded boxes to show `Bar's` and `Foo's` activation frames. Include a symbolic description and the actual value (in decimal) if known. For return addresses, show only the symbolic description; do not include a value. *Label the frame pointer and stack pointer*. Assume a **32-bit system** and maintain word alignment.

	address	description	Value
	9900	**RA of Bar’s caller**	
	9896	**FP of Bar’s caller**	
SP, Bar’s FP	9892	**RV**	
	9888	**x**	**‘i’** (or **105** ascii)
	9884	**y**	**1**
	9880	**z**	
	9876	**Name**	**T i m EOS**
	9872	**RA**	
	9868	**FP**	**9892**
	9864	**S**	**9876**
	9860	**n**	**~~1~~ 2**
	9856	**c**	**9891**
Foo’s FP	9852	**RV**	
	9848	**a**	**3**
FP: **9852**	9844		
SP: **9848**	9840		
	9836		
	9832		

Note: The ‘o’ in ‘Tom’ (at address 9877) is replaced with ‘i’ due to the instruction ‘S[n]=*c;’.
The value of ‘n’ at address 9860 changes to 2 due to the instruction ‘n++;’. The address of x (passed in as Foo’s parameter c) is 9891 because the stack grows downward: RV spans addresses 9895-9892, x is at 9891, then 3 bytes of slack, y spans 9887-9884.

Suppose we have the following:

```
int Bar() {
   char       x = 'i';
   char       w = 'j';
   int        y = 1;
   ...
```

Then RV spans addresses 9895-9892, x is at 9891, w is at 9890, then 2 bytes of slack, y spans 9887-9884.

Part B (22 points) Write MIPS code fragments to implement the subroutine `Foo` by following the steps below. *Do not use absolute addresses in your code; instead, access variables relative to the frame pointer.* Assume no parameters are present in registers (i.e., access all parameters from `Foo`'s activation frame). You may not need to use all the blank lines provided.

label	instruction	Comment
`Foo:`	`add  $30, $29, $0`	`# set Foo's FP`
	`addi $29, $29, -4`	`# allocate space for locals`
	`addi $1, $0, 3`	`# initialize locals`
	`sw   $1, -4($30)`	`# a = 3;`

`# if S[n] == 0 branch to End.` *Be sure to use load/store byte for values of type char.*

	`lw   $2, 12($30)`	`# $2 = S`
	`lw   $3,  8($30)`	`# $3 = n`
	`add  $4, $2, $3`	`# $4 = S + n`
	`lbu  $5, 0($4)`	`# load S[n] (a char)`
	`beq  $5, $0, End`	`# if S[n] == 0, skip to End`

`# otherwise, do Then clauses: S[n] = *c;`

	`lw   $6, 4($30)`	`# $6 = c`
	`lbu  $6, 0($6)`	`# $6 = *c`
	`sb   $6, 0($4)`	`# S[n] = *c`

`# n++;`

	`addi $3, $3, 1`	`# n+1`
	`sw   $3, 8($30)`	`# n = n+1`

`# return(n);      (store return value, deallocate locals, and return)`

`End:`	`sw   $3, 0($30)`	`# store RV`
	`add  $29, $30, $0`	`# deallocate locals`
	`jr   $31`	`# return to caller`