Problem 1 (2 parts, 30 points) **Storage Allocation, Arrays, 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: int and float are 32-bits.

```
int      a   = 66;
int     *b   = &a;
int    **c   = &b;
float    f   = 9.23;
float   *p   = &f;
char    S[]  = "Sally";
char    *q   = &(S[0]);
float    g   = 8.26;
double   z   = 917.25;
```

4000	a	a	a	a	s	s	s	s
4008	b	b	b	b	b	b	b	b
4016	c	c	c	c	c	c	c	c
4024	f	f	f	f	s	s	s	s
4032	p	p	p	p	p	p	p	p
4040	S	S	S	S	S	S	s	s
4048	q	q	q	q	q	q	q	q
4056	g	g	g	g	s	s	s	s
4064	z	z	z	z	z	z	z	z

Part B (14 points) Assuming a 32-bit system, consider the following declarations:

```
int A[8][16][16] = {...};
int *q = A;
```

B.1 Complete the assignment statement below using only `q` to assign to `x` the value of `A[1][10][i]`.

`int x = *(q + 1*16*16 + 10*16 + i);//an expression is ok`

B.2 Write the MIPS code implementation of the following assignment statement in the smallest number of instructions. A pointer to the array **A** is stored in **$3** and variables **j**, **k**, and **y** reside in **$4, $5,** and **$6**, respectively. Modify only registers **$6** and **$7**.

`int y = A[k][j][4];`

Label	Instruction	Comment
	sll $7, $5, 8	# k * 256
	sll $6, $4, 4	# j * 16
	add $7, $7, $6	# k * 256 + j *16
	addi $7, $7, 4	# + 4
	sll $7, $7, 2	# scale by 4
	add $7, $7, $3	# add A
	lw $6, 0($7)	# read from memory and assign to y

Problem 2 (2 parts, 30 points) **Accessing Structs, Activation Frame Allocation**

Consider the following C code fragment.

```
typedef struct {
   int   A;
   int   B;
   char  C;
} struct_t;

struct_t myStruct1= {10,20,0x2F};
struct_t myStruct2;
int j = 42;
struct_t * p = &myStruct1;
int *q = &myStruct1.A;

myStruct2.A = j;
(*p).B = 10;
p->C = 0x2A;
```

Part A (15 points) Assuming a **32-bit system with 32-bit memory interface and 32-bit addresses**, fill in the table with the values of the given expressions or U if they are unknown. Each expression should be evaluated independently only given the above code. Please assume variables are allocated beginning at address 1000.

Expression	value
`&myStruct2`	**1012**
`p->A`	**10**
`*(q+1)`	**10**
`myStruct1.C+1`	**0x2B**
`(p+1)->A`	**42**

Part B (15 points) Consider the following function:

```
int foo(char *s,char *d){
       int cnt=0;
       while (*s){
          *d++ = *s++;
          cnt++;
      }
      *d = 0;
      return cnt;
}
```

B.1 What does the function do?

It copies one EOS delimited string s to another string d and counts the number of characters in s.

B.2 Describe two things that must be true for this function to execute correctly.

The space allocated for d must be at least as big as the space allocated for s.

The string x must end with EOS (null character).

B3. What is the size of foo's activation frame? **24 bytes**

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

Consider the following C code fragment:

```
typedef struct {
   int Start;
   int End;
} trip_info_t;

int TripAdvisor() {
   int         odometer = 981005;
   int         Gallons[] = {16, 6};
   trip_info_t TI;
   int         rate;
   int         Update(trip_info_t, int [], int *);
   TI.Start    = 180;
   TI.End      = 420;
   rate = Update(TI, Gallons, &odometer);
   return(odometer);
}

int Update(trip_info_t Trip, int G[], int *OD) {
   int         miles, MPG;
   miles       = Trip.End - Trip.Start;
   MPG         = miles/G[1];
   *OD        += miles;
   return(MPG);
}
```

Part A (18 points) Suppose `TripAdvisor` has been called so that the state of the stack is as shown below. Describe the state of the stack just before `Update` deallocates locals and returns to `TripAdvisor`. Fill in the unshaded boxes to show `TripAdvisor's` and `Update'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.*

	address	description	Value
	9900	RA of TA's caller	
	9896	FP of TA's caller	
SP, TripAdvisor's FP	9892	RV	
	9888	odometer	981245
	9884	Gallons[1]	6
	9880	Gallons[0]	16
	9876	TI.End	420
	9872	TI.Start	180
	9868	rate	
	9864	RA	
	9860	FP	9892
	9856	Trip.End	420
	9852	Trip.Start	180
	9848	G	9880
	9844	OD	9888
	9840	RV	
FP: 9840	9836	miles	240
SP: 9832	9832	MPG	40
	9828		

Part B (22 points) Write MIPS code fragments to implement the subroutine `Update` 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 `Update`'s activation frame). You may not need to use all the blank lines provided.

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

label	instruction	Comment
`Update:`	`add $30, $29, $0`	`# set FP`
	`addi $29, $29, -8`	`# allocate locals`

`# miles = Trip.End – Trip.Start;`

label	instruction	Comment
	`lw $1, 12($30)`	`# read T.Start`
	`lw $2, 16($30)`	`# read T.End`
	`sub $1, $2, $1`	`# T.End-T.Start`
	`sw $1, -4($30)`	`# store in miles`

`# MPG = miles/G[1];`

label	instruction	Comment
	`lw $2, 8($30)`	`# read G (base address)`
	`lw $2, 4($2)`	`# read G[1]`
	`div $1, $2`	`# miles/G[1]`
	`mflo $3`	`# result in $3`
	`sw $3, -8($30)`	`# store in MPG`

`# *OD += miles;`

label	instruction	Comment
	`lw $4, 4($30)`	`# read OD (address)`
	`lw $2, 0($4)`	`# dereference it`
	`add $5, $2, $1`	`# *OD + miles`
	`sw $5, 0($4)`	`# *OD = *OD+miles`

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

label	instruction	Comment
	`sw $3, 0($30)`	`# put MPG in RV slot`
	`add $29, $30, $0`	`# deallocate locals`
	`jr $31`	`# return to caller`