

**Q1.** (15 points) You are using the “SooperCompiler” to compile the three functions below. function1 is called from main once, while function2 and function3 are both called from function1 only. We report below the relevant snippets of the functions’ code, including all the calls to other functions. The architecture that theSooperCompiler is targeting has 2 caller-saved registers and 2 callee-saved registers

```
int function1(int arnold){
    int a,b,c,d ;
    a = 15 ;
    b = 12 ;
    function2(a)
    c = a + b ;
    while(a > 0) {
        //iterates 15 times
        function3(a) ;
        a-- ;
    }
    d = 9 ;
    d = 4 + arnold ;
    d++ ;
    return 0 ;
}

int function2(int gerald){
    int k,l,m ;
    k = 21 ;
    m = 23 ;
    printf(“ps118\n”) ;
    l = k + m ;
    l = l / 2 ;
    return 0 ;
}

int function3(int helga){
    int w,x,y ;
    w = 4 ;
    x = 9 ;
    y = w + x ;
    printf(“footballhead\n”) ;
    y = w ;
    return 0 ;
}
```

The SooperCompiler has mapped the local variables of the functions as follows:

function1: a and b are in caller-saved, while c and d are in callee-saved

function2: k and l are in caller-saved, while m is in callee-saved

function3: w and y are in caller-saved, while x is in callee-saved

- (a) How many save/restore executed instruction pairs did the compiler have to include in the assembly code of function1 ONLY for the purpose of preserving register values across functioncalls? (6 points)

acceptable solutions computed the number additional **executed instructions**:

c and d are on callee-saved registers → 1 S/R for c and 1 S/R for d

a is alive across the call to function2 → 1 S/R

a is alive across the 15 calls to function3 → 15 S/R

b is alive across the call to function2 → 1 S/R

Total for function1: **19 S/R pairs (or 38 instructions: 19 stores and 19 restores)**

- (b) Can you do better than the SooperCompiler? Provide the best assignment to caller or callee-saved registers on a per-register basis. If caller and callee-save give the same performance for a variable, then put “either” for your answer. Be sure to state the total number of save/restore instruction pairs added to function1’s assembly code. (9 points)

function1 includes calls to function2 (1 call) and to function3 (15 calls).

The variables to be preserved across the function2 call are: **a and b**

The variables to be preserved across the function3 calls are: **a**

function2 includes 1 call to printf.

The variables to be preserved across the call to printf is: **k and m.**

function3 includes 1 call to printf.

The variables to be preserved across the call to printf is: **w.**

	variable	caller-save S/R ops	callee-save S/R ops	answer
<b>function1</b>	a	16 S/R	1 S/R	Callee
	b	1 S/R	1 S/R	Either
	c	0 S/R	1 S/R	Caller
	d	0 S/R	1 S/R	Caller
<b>function2</b>	k	1 S/R	1 S/R	Either
	l	0 S/R	1 S/R	Caller
	m	1 S/R	1 S/R	Either
<b>function3</b>	w	1 S/R	1 S/R	Either
	x	0 S/R	1 S/R	Caller
	y	0 S/R	1 S/R	Caller

With this assignment, the total number of instruction pairs that must be added for preserving register values across function calls is:

function1: 2 S/R instruction pairs

function2: 2 S/R instruction pairs

function3: 1 S/R instruction pairs

**Total of 5 save/restore instruction pairs added. (or 10 instructions)**

**Q2.** (8 points) Given the following C file:

*doug.c*:

```
extern char a[] ;
```

```
int *b ;
```

```
float c[20] ;
```

```
struct {
```

```
    int d ;
```

```
    char e ;
```

```
} f ;
```

```
extern float g ;
```

```
int skeeter(int) ;
```

```
float porkchop(string h) {  
    float i = 3.14 ;  
    float j = patty(h) ;  
    float k = roger(h) ;  
    static float m = j + k ;  
    return m ;  
}
```

Circle all the symbols that are placed in the symbol table of doug.o:

**a**   **b**   **c**   d   e   **f**   **g**   h   i   j   k   **m**  
**skeeter**   **porkchop**   **patty**   **roger**

### Q3. (6 points) C to MIPS assembly compilation

Following is the code snippet for performing matrix addition. A matrix is simply stored as an array in memory. Assume that the matrices a, b and c start from memory locations  $(1000)_{16}$ ,  $(1500)_{16}$ , and  $(2000)_{16}$  respectively. You are required to write the equivalent MIPS assembly code for the code snippet provided. You are free to use any register available for data operations, provided it is not reserved. Be sure to comment every line of your code for partial credit (and ease in debugging).

*Code snippet:*

```
int a[10], b[10], c[10];
```

```
int i;
```

```
for(i=0; i<10; i++)
```

```
    c[i] = a[i] + b[i];
```

**Solution:**

```
addi    $s1,$0,1000    ; load address of 'a' in $s1
```

```
addi    $s2,$0,1500    ; load address of 'b' in $s2
```

```
addi    $s3,$0,2000    ; load address of 'c' in $s3
```

```
addi    $s4,$0,$0      ; load '0' in $s4 for variable 'i'
```

```
addi    $s5,$0,(40)10 ; load '40' in $s5 for number of array items*4 bytes each
```

```

loop1: lw    $t0,0($s1)    ; load contents of mem($s1+0) in $t0
        lw    $t1,0($s2)    ; load contents of mem($s2+0) in $t1
        add   $t2,$t0,$t1    ; add and store result in $t2
        sw    $t2,0($s3)    ; store contents of $t2 in mem($s3+0)
        addi  $s4,$s4,4      ; increment $s4 by 4
        add   $s1,$0,$s4     ; increment $s1 by 4 (next memory location)
        add   $s2,$0,$s4     ; increment $s2 by 4
        add   $s3,$0,$s4     ; increment $s3 by 4
        beq   $s4,$s5,2      ; if $s4 = $s5, end of for loop reached. So exit
        j     loop1         ; else loop back

```

Q4. (12 points)

Consider assigning the variables below to memory locations starting at address 100 (in decimal). The alignment rules are as discussed in class. The size of each data type is listed below.

```

short a;
double b[10];
short c;
struct {
    char d[5];
    double* e;
    short f;
} g[3];
int z;

```

Data Type	Size
char	1 byte
short	2 bytes
int <i>or</i> pointer	4 bytes
float	4 bytes
double	8 bytes

- (a) Calculate the size and starting address for each variable. Show your work by drawing the memory layout. (8 points)

Variable Name	Size (in bytes)	Start Address	End Address
<b>a</b>	2	100	101
<i>(unused)</i>	2	102	103
<b>b[10]</b>	10*8b = 80	104	183
<b>c</b>	2	184	185
<i>(unused)</i>	2	186	187
<b>g[0].d[5]</b>	5*1b = 5	188	192
<i>(unused)</i>	3	193	195
<b>g[0].e</b>	4	196	199
<b>g[0].f</b>	2	200	201
<i>(unused)</i>	2	202	203
<b>g[1]</b>	16	204	219
<b>g[2]</b>	16	220	235
<b>z</b>	4	236	239

The first instance of the struct (shaded grey) shows the layout of its members. The largest member is 4 bytes, so the struct must start and end on a multiple of 4 bytes. Due to this restriction, the struct requires 16 bytes. Unused sections are shown for clarity.

(b) How many bytes are required to layout these declarations in memory? (2 points)

240 - 100 = **140 bytes**

(c) Is it possible to reduce the size of the structure? Explain your reasoning. (2 points)

**Yes.** The struct can reorganized such that it is declared in the following order: pointer e, short f, and the character array. This layout will require 11 bytes for raw data, and will only waste 1 byte to pad the structure up to 12 bytes (multiple of 4 bytes).

**Q5.** (8 points)

CAEN is planning to install a new capacity monitor for the computing labs on campus. This monitor will contain a single light, which will be yellow, orange, or red depending on how many computers are occupied. Due to CAEN's lack of funding, however, this light will be operated manually using two switches, one which controls the yellow LED's, and one which controls the red LED's. Each switch has two positions: "on" and "off". Turning either switch "on" while the other switch is "off" will cause its respective color to be displayed, while turning both switches "on" will display orange. There is also a switch for the light itself, which simply turns the light "on" or "off".

You will analyze and design a state machine for the monitor light in the following series of steps. The possible inputs are (6 total):

- Flip yellow switch on or off
- Flip red switch on or off
- Flip light switch on or off

The possible outputs are (4 total):

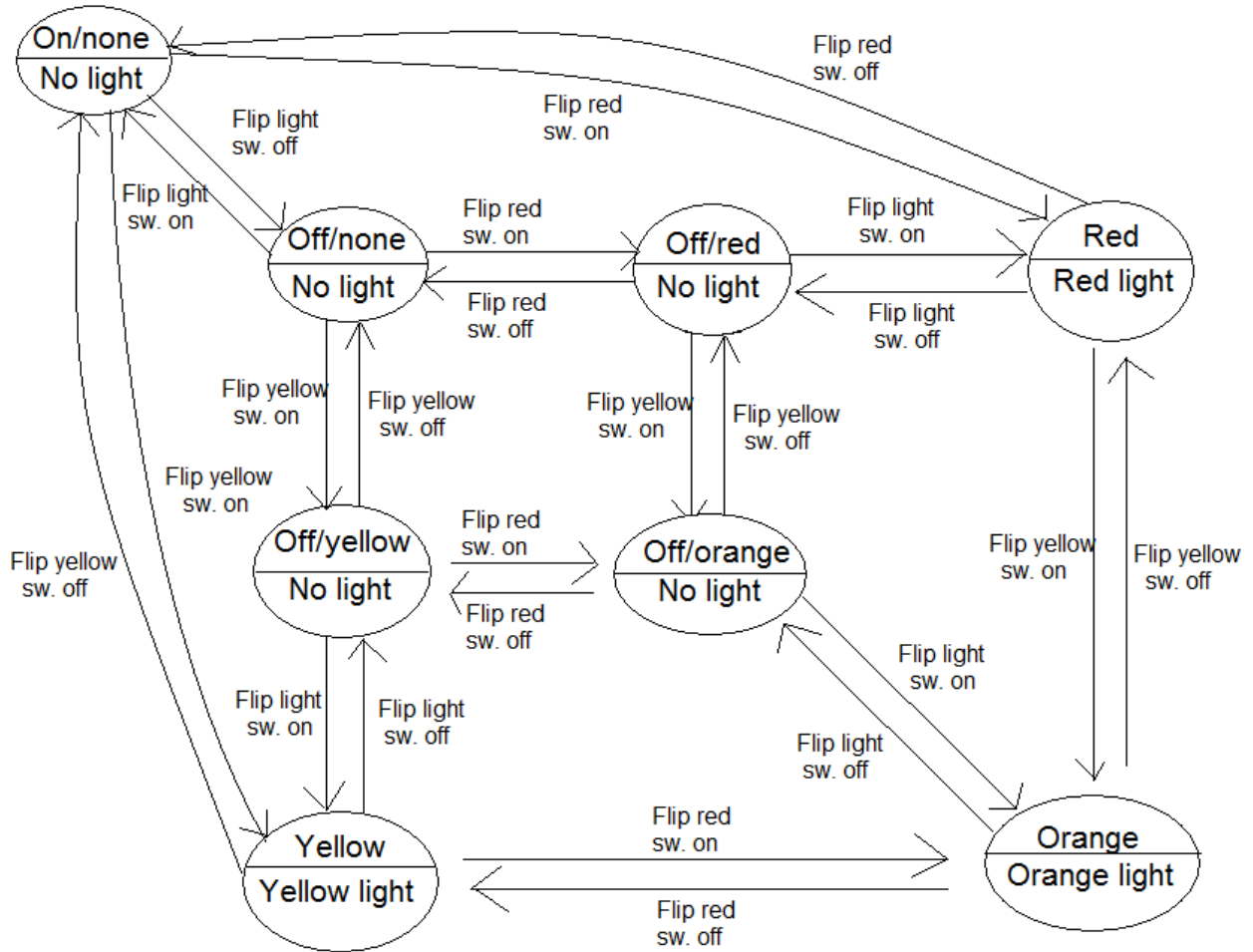
Red/orange/yellow light

No light

You may ignore vacuous transitions, such as attempting to flip a switch on when it is already on. Assume that only one switch may be flipped at a time, and that the initial position of all the switches is "off".

- a. (3pt) Draw a state diagram (with input and output values labeled on all arcs)
- b. (2pt) Provide the state table (with state encoding)
- c. (2pt) Determine the output functions (write the corresponding equations); you may omit "no light" as an output
- d. (1pt) Is this a Mealy or Moore machine?

a. State Diagram



b. RO = red switch on, YO = yellow switch on, LO = light switch on, RF = red switch off, YF = yellow switch off, LF = light switch off, RL = red light, YL = yellow light, OL = orange light, NL = no light

Current state		Inputs						Next state	Outputs			
Name	Encoding	RO	YO	LO	RF	YF	LF	State	RL	YL	OL	NL
Off/none	000	0	0	1	0	0	0	001	0	0	0	1
Off/none	000	0	1	0	0	0	0	010	0	0	0	1
Off/none	000	1	0	0	0	0	0	011	0	0	0	1
On/none	001	0	0	0	0	0	1	000	0	0	0	1
On/none	001	0	1	0	0	0	0	100	0	1	0	0
On/none	001	1	0	0	0	0	0	101	1	0	0	0
Off/yellow	010	0	0	0	0	1	0	000	0	0	0	1
Off/yellow	010	0	0	1	0	0	0	100	0	1	0	0
Off/yellow	010	1	0	0	0	0	0	111	0	0	0	1
Off/red	011	0	0	0	1	0	0	000	0	0	0	1
Off/red	011	0	0	1	0	0	0	101	1	0	0	0
Off/red	011	0	1	0	0	0	0	111	0	0	0	1
Yellow	100	0	0	0	0	0	1	010	0	0	0	1
Yellow	100	0	0	0	0	1	0	001	0	0	0	1
Yellow	100	1	0	0	0	0	0	110	0	0	1	0
Red	101	0	0	0	0	0	1	011	0	0	0	1
Red	101	0	0	0	1	0	0	001	0	0	0	1
Red	101	0	1	0	0	0	0	110	0	0	1	0
Orange	110	0	0	0	0	0	1	111	0	0	0	1
Orange	110	0	0	0	0	1	0	101	1	0	0	0
Orange	110	0	0	0	1	0	0	100	0	1	0	0
Off/orange	111	0	0	0	0	1	0	011	0	0	0	1

Off/orange	111	0	0	0	1	0	0	010	0	0	0	1
Off/orange	111	0	0	1	0	0	0	110	0	0	1	0

c. These have not been optimized using K-maps, but give the general idea.

Let STATE[i] = the ith bit of the current state (zero-indexed)

Output Functions:

Red light (RL) = (STATE[0] • ~STATE[1] • ~STATE[2] • RO) + (STATE[0] • STATE[1] • ~STATE[2] • LO) + (~STATE[0] • STATE[1] • STATE[2] • YF)

Yellow light (YL) = (STATE[0] • ~STATE[1] • ~STATE[2] • YO) + (~STATE[0] • STATE[1] • ~STATE[2] • LO) + (~STATE[0] • STATE[1] • STATE[2] • RF)

Orange light (OL) = (~STATE[0] • ~STATE[1] • STATE[2] • RO) + (STATE[0] • ~STATE[1] • STATE[2] • YO) + (~STATE[0] • STATE[1] • STATE[2] • LO)

d. This is a Moore machine because the output function is based only on current state.