



## Section I: Multiple Choice

1. Consider the following program

```
int a;
int b[20];

int foo(int f) {

    int c, *g;
    int d[100];
    static int e;

    g = (int *) malloc(sizeof(int) * 100);

    for (a = 0; a < 100; a++) {
        int h;
        /* ... */
    }
}
```

Which of the following data mappings in memory is consistent with the above program?

- a. Stack: [d, f, \*g, h]; Heap: [g]; Static: [b, e]
- b. Stack: [c, d, f, h]; Heap: [g]; Static: [a, e]
- c. Stack: [d, f, g, h]; Heap: [\*g]; Static: [b, e] **(ANS)**
- d. Stack: [c, e, f, h]; Heap: [g]; Static: [a, b]
- e. Stack: [d, e, f, h]; Heap: [\*g]; Static: [a, b]

2. Consider the following function. Assume that you have 2 caller-saved and 2 callee-saved registers. Pick the best register assignment for h, i, j, k that can lead to the least possible savings and restorings. Assume that each requires its own register.

```
int foo() {
    int h, i, j, k;
    j = 0;
    bar1();
    h = j+1;
    i = j;
    while (i<10) {
        k=...
        ... = h;
        ... = k;
        bar2();
        i += 1;
    }
}
```

- a. caller-saved: [h, j]; callee-saved: [i, k]
- b. caller-saved: [j, k]; callee-saved: [i, h] **(ANS)**
- c. caller-saved: [i, j]; callee-saved: [h, k]
- d. caller-saved: [i, k]; callee-saved: [h, j]
- e. caller-saved: [h, i]; callee-saved: [k, j]

For Questions 3 and 4, consider the following function. Assume that foo is the first function called.

```
void foo(int *data, int first, int last) {
    if(first <= last) {
        foo(data, first+1, last);
        printf ("%d\n", data[first]);
    }
}
```

3. What is the maximum stack depth, in terms of number of frames, of the functional call `foo(data, 0, 4)`?
- a. 2
  - b. 3
  - c. 4
  - d. 5
  - e. 6 **(ANS)**

4. Assume that a new optimization has been included in the compiler: When the last executed statement of a function is a recursive call to the function itself (a.k.a. tail recursion), the compiler will not save the call parameters and return address on the stack. To take advantage of this new compiler optimization, which one of the following new `foo` functions will give the minimal possible stack depth without changing the effect of calling the original function `foo`? Assume that each element of array `data` is different.

a. 

```
void foo(int *data, int first, int last) {
    if(first <= last) {
        printf ("%d\n", data[first]);
        foo(data, first+1, last);
    }
}
```

b. 

```
void foo(int *data, int first, int last) {
    if(first <= last) {
        printf ("%d\n", data[last-first]);
        foo(data, first+1, last);
    }
}
```

c. 

```
void foo(int *data, int first, int last) { (ANS)
    if(first <= last) {
        printf ("%d\n", data[last]);
        foo(data, first, last-1);
    }
}
```

d. 

```
void foo(int *data, int first, int last) {
    if(first <= last) {
        printf ("%d\n", data[first]);
        foo(data, first, last-1);
    }
}
```

e. 

```
void foo(int *data, int first, int last) {
    if(first <= last) {
        printf ("%d\n", data[last]);
        foo(data, first+1, last);
    }
}
```

For Questions 5 and 6, assume the following addressing modes and initial machine configuration.

*Direct:*        `lwd r3, 0x100 # r3=Mem[0x100]`  
*Indirect:*     `lwi r3, 0x100 # r3=Mem[Mem[0x100]]`  
*Base/Disp:*   `lw r3, 5(r1) # r3=Mem[r1 + 5]`

Registers: `r1=0x100 r2=0x200 r3=r4=0x0`

Location	Value		Location	Value
0x100	0x200		0x200	0x100
0x104	0x108		0x204	0x208
0x108	0x204		0x208	0x104
0x10c	0x315		0x20c	0x555

Consider the following program.

```
lwd r3, 0x208
lwi r4, 0x100
lw r2, 8(r3)
swi r2, 0x100
swd r4, 0x10c
```

5. What is the value in r2 after completion?

- a. 0x100
- b. 0x104
- c. 0x108
- d. 0x200
- e. 0x315 (ANS)

6. What is the value of memory location 0x10c after completion?

- a. 0x100 (ANS)
- b. 0x104
- c. 0x108
- d. 0x200
- e. 0x315

7. Consider the following LC-2K5 code:

```
start      lw    0 1 ExitAddr
           jalr  1 1
           beq   3 4 -3
exit       halt
ExitAddr   .fill exit
```

Assuming all registers are initialized to 0, how many dynamic instructions does this code take to finish?

- a. 2
- b. 3
- c. 4
- d. 5
- e. Infinite (ANS)

8. Two characteristics of typical RISC instruction set architectures are:

- a. variable instruction length, register-memory operations
- b. variable instruction length, many registers
- c. fixed instruction length, few registers
- d. fixed instruction length, register-memory operations
- e. fixed instruction length, register-register operations (ANS)

9. Suppose there is a MIPS-like instruction set architecture that supports 100 opcodes and 64 registers. Instructions are 32 bits wide. Assume there is a class of instructions in this ISA that takes two register arguments and one signed (2's complement) immediate. What is the maximum possible range of values of this immediate?

- a. 0 to  $2^{13}$
- b.  $-2^{13}$  to  $2^{12}$
- c.  $-2^{13}$  to  $2^{13}-1$
- d.  $-2^{12}$  to  $2^{12}-1$  (ANS)
- e.  $-2^{12}-1$  to  $2^{12}$

10. How many bytes does the C data structure below require?

```
struct foo {  
    int a, b;  
    char c;  
    char d[10];  
};
```

- a. 10
- b. 13
- c. 16
- d. 19
- e. 20 (ANS)

Questions 11 and 12 refer to the two snippets of pseudo C code below. Assume that each file is compiled and assembled into its own object file.

**file1.c**

```
1:  int c;  
  
2:  void foo(int a) {  
3:      int b;  
4:      reference to b  
5:      reference to c  
6:      bar(a);  
7:  }
```

**file2.c**

```
8:  void bar(int x) {  
9:      int y[5];  
10:     reference to y  
11:     reference to x  
12:     reference to c  
13: }
```

11. What symbols will appear in the symbol table for **file2.c**?

- a. c
- b. bar, c (ANS)
- c. bar, x, y, c
- d. bar, x
- e. No symbols

12. Which of the numbered lines are referenced in the relocation table of **file1.c**?

- a. 1, 2, 6
- b. 1, 6
- c. 5, 6 (ANS)
- d. 4, 5
- e. 4, 5, 6

13. The following MIPS code performs some computation and stores a value in register \$v0. What does the value in register \$v0 represent? Assume \$s0 and \$s1 are initialized to 0, and \$a0 and \$a1 are incoming parameters that contain the base address of an array and the number of elements in the array, respectively.

```
loop:  slt  $t0, $s0, $a1
       beq  $t0, $zero, end
       lw   $t2, 0($a0)
       andi $t3, $t2, 1
       bne  $t3, $zero, skip
       add  $s1, $s1, $t2
skip:  addi $s0, $s0, 1
       addi $a0, $a0, 4
       j    loop
end:   move $v0, $s1
```

- The sum of elements at even indices of the array.
- The sum of elements at odd indices of the array.
- The sum of even elements in the array. **(ANS)**
- The sum of odd elements in the array.
- The number of even elements in the array.

14. Consider the following C declaration:

```
struct node_t {
    int      data;
    struct node_t *next;
} *node;
```

Which sequence of MIPS assembly instructions implements the following C while loop?  
Assume that the variable “node” is in register \$s1, and that the constant NULL is equal to 0.

```
while (node->next != NULL)
    node = node->next;
```

- a. Loop: lw \$t0, 4(\$s1) **(ANS)**  
beq \$t0, \$zero, End  
move \$s1, \$t0  
j Loop  
End:
- b. Loop: lw \$t0, 0(\$s1)  
addi \$t1, \$t0, 4  
lw \$t2, 0(\$t1)  
beq \$t2, \$zero, End  
move \$s1, \$t2  
j Loop  
End:
- c. Loop: lw \$t0, 0(\$s1)  
addi \$t1, \$t0, 4  
beq \$t1, \$zero, End  
move \$s1, \$t1  
j Loop  
End:
- d. Loop: lw \$t0, 4(\$s1)  
lw \$t1, 0(\$t0)  
beq \$t1, \$zero, End  
move \$s1, \$t1  
j Loop  
End:
- e. Loop: addi \$t0, \$s1, 4  
beq \$t0, \$zero, End  
lw \$t1, 0(\$t0)  
move \$s1, \$t1  
j Loop  
End:

## Section II: Design Questions

Consider the 8-bit, stack-based, byte-addressable instruction set named Sue. Sue contains a single register, the 8-bit accumulator (ACC), to assist with all computation.

There are 3 instruction formats (bit 0 is the least-significant bit, bit 7 most-significant).

R-type instructions (add, push, pop, halt):

bits 7-5: opcode

bits 4-0: unused (should all be 0)

I-type instructions (pushi, beq)

bits 7-5: opcode

bits 4-0: value (a 5-bit, 2's complement number with a range of -16 to 15)

U-type instructions (loadb, storeb)

bits 7-5: opcode

bits 4-0: address (a 5-bit, unsigned number with a range of 0 to 31)

Assembly language instruction	Opcode in binary (bits 7, 6, 5)	Action
add (R-type)	000	ACC = ACC + (top of stack) Pop value off top of stack, and place the sum of the ACC and the popped value into the ACC.
push (R-type)	001	Copy the value in the ACC onto the stack.
pop (R-type)	010	Remove value at top of stack and place it into ACC.
halt (R-type)	011	Increment the PC (as with all instructions), then halt the machine (let the simulator know that the machine halted).
pushi (I-type)	100	Place the sign-extended value stored in the value field onto the stack.
beq (I-type)	101	If the value on the top of the stack is equal to the value in ACC, pop it and branch to PC = PC + 1 + offset. Otherwise do nothing (don't change anything on the stack).
loadb (U-type)	110	ACC = MEM[addr] Take the byte value stored at the memory location specified by the address field and put it in ACC.
storeb (U-type)	111	MEM[addr] = ACC Place the byte value contained in ACC into the memory location specified by the address field.

15. Translate the following program into *hexadecimal*. [8 pts]

pushi 2	0x82
loadb 21	0xD5
add	0x00
push	0x20
beq -4	0xBC
halt	0x60

16. Translate the following C statement into Sue assembly. Assume variables a, b, and c are stored at memory locations 15, 16, and 17, respectively. Do not assume anything about the initial value of ACC. [8 pts]

a = (a + 3) + (b + c);

```
loadb 15
pushi 3
add
push
loadb 16
push
loadb 17
add
add
store 15
```

17. Suppose that Susan is the 32-bit version of Sue, in which all data are 32 bits. List one advantage and one disadvantage of a processor that uses the Susan instruction set over the LC-2K5 used in class. [6 pts]

Advantages: smaller encoding, larger opcode / immediate fields

Disadvantages: larger number of dynamic / static instructions

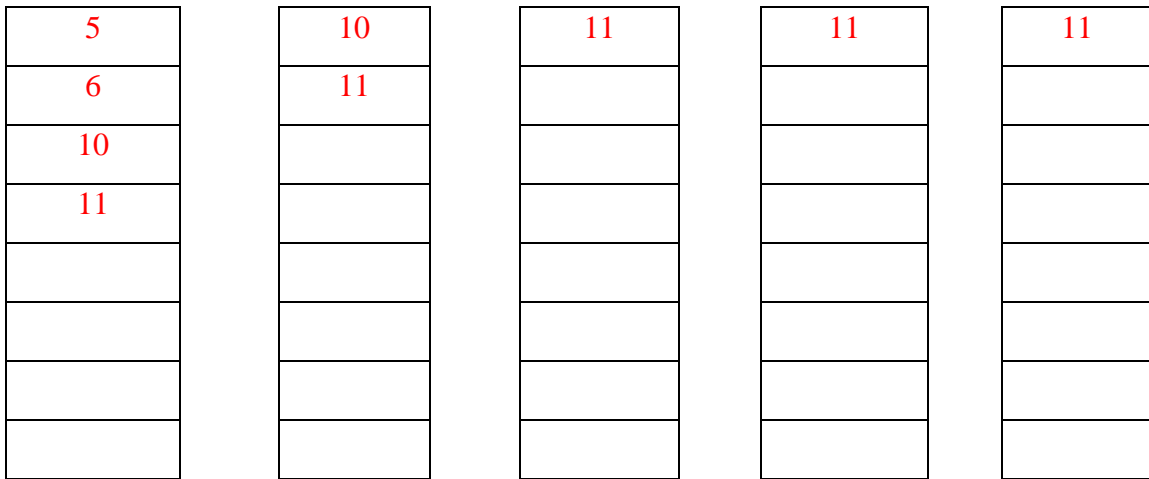
18. Show value(s) stored in the stack *after* each time the following program encounters a beq *as well as* when the program halts. Assume the PC starts as zero initially. We have provided space for your answer, but be sure to label each stack. Draw your stack so the “top-of-stack” is on top. Hint: Correct execution should require no more than 5 stack snapshots. [8 pts]

```

pushi 11
pushi 10
pushi 6
pushi 5
pushi 1
pop
beq 2
add
beq -2
push
storeb 29
beq 1
push
halt

```

Top of stack



Bottom of stack