



**Clarifications during the exam:**

Q3-3. The node should free itself also.

- if you can traverse to a node through a series of prev and/or next, that node is reachable

Q4. If you see a 90 under the “Exit” label, please ignore it.

*beargit redux* - In `is_commit_msg_ok()`, `int i` is declared before the for loop.

- if you need to use it, you can use `$sp`, `$ra`, `$gp`, `$fp`

Q6-3 - Give the address in hex

Q6-1 - Instructions are still 32 bits

SID: \_\_\_\_\_

## Q1: A dozen ways to ask about bits (10 points)

1) For a 12-bit integer represented with two's complement, what is the:

a) Most positive value (in decimal):

\_\_\_\_\_ **2047** \_\_\_\_\_

b) Binary representation of that number:

\_\_\_\_\_ **0b011111111111** \_\_\_\_\_

c) Most negative value (in decimal):

\_\_\_\_\_ **-2048** \_\_\_\_\_

d) Hex representation of that number:

\_\_\_\_\_ **0x800** \_\_\_\_\_

e) In general, for an n-bit, two's complement integer:

i) What is the largest value you can represent, in decimal?

\_\_\_\_\_  **$2^{(n-1)} - 1$**  \_\_\_\_\_

ii) What is the smallest value you can represent, in decimal?

\_\_\_\_\_  **$-2^{(n-1)}$**  \_\_\_\_\_

2) Fill in the blank below so that the function mod16 will return the remainder of x when divided by 16. The first blank should be a bitwise operator, and the second blank should be a single decimal number:

```
unsigned int mod16(unsigned int x) {  
    return x _____ & _____ 15 _____;  
}
```

**Q2: Wow! If only you could C the *main memory* (10 points)**

Consider the following C program:

```
int a = 5;
void foo(){
    int temp;
}
int main()
{
    int b = 0;
    char* s1 = "cs61c";
    char s2[] = "cs61c";
    char* c = malloc(sizeof(char) * 100);
    foo();
    return 0;
};
```

1) Sort the following values from least to greatest: `&b`, `c`, `b`, `&temp`, `&a`.

**`b < &a < c < &temp < &b`**

2) For each of the following values, state the location in the memory layout where they are stored. Answer with *code*, *static*, *heap*, or *stack*.

|                    |               |
|--------------------|---------------|
| <code>s1</code>    | <b>stack</b>  |
| <code>s2</code>    | <b>stack</b>  |
| <code>s1[0]</code> | <b>static</b> |
| <code>s2[0]</code> | <b>stack</b>  |
| <code>c[0]</code>  | <b>heap</b>   |

**Q3: Links, Links, and Lists (15 points)**

Here, we have a two-sided linked list, where each node has a reference to both the previous and next node in the list. The HEAD of the list is defined as the node with a NULL prev pointer, and the TAIL of the list is defined as the node with a NULL next pointer.

```
struct ll_node {
    unsigned short id;
    struct ll_node *prev;
    struct ll_node *next;
};
```

For the remainder of the questions, assume that the struct ll\_node is tightly-packed (i.e, all its elements are contiguous in memory).

1) We are given a struct ll\_node **current\_node**. Assuming that the type unsigned short is 2 bytes wide and that we are working with a 32-bit memory address space, what can we expect the function call **sizeof(current\_node)** to return?

\_\_\_\_\_ **10** \_\_\_\_\_

2) Assume that we have access to **id\_addr**, the address of the id of **current\_node** in memory. Using only **id\_addr**, fill in the blank line so that **next\_node** is equivalent to **current\_node.next**.

```
unsigned short *id_addr = &(current_node.id);
struct ll_node *next_node = *(_(struct ll_node **) (id_addr + 3)_);
```

3) Now, fill in the blanks to complete this function that, given a random node in the list, frees all reachable nodes from that given node. Keep in mind that the node may be the HEAD of the list, the TAIL of the list, or a node in between the HEAD and TAIL. You may not need every blank.

```
void free_twosided_ll(struct ll_node *node) {
    if (node != NULL) {
        if (node->prev != NULL) {
            _____ (node->prev)->next = NULL; _____
            _____ free_twosided_ll(node->prev); _____
        }
        if (node->next != NULL) {
            _____ (node->next)->prev = NULL; _____
            _____ free_twosided_ll(node->next); _____
        }
        _____ free(node); _____
    }
}
```

**Q4: *beargit* redux (15 points)**

From project 1, you may remember the function `is_commit_msg_ok()` that you needed to implement in C. Here is a simpler rendition where commit messages are deemed okay *if and only if* those null-terminated commit messages exactly match `go_bears`. Using the **fewest number of empty lines possible**, finish writing the code below. You are only allowed to use the registers already provided **and** registers `$t0-3`, and `$s0-s2` (but you will not need all of them). Assume these registers are initialized to 0 before the call to `ISCOMMITOK`.

```
const char* go_bears = "THIS IS BEAR TERRITORY!";

int is_commit_msg_ok(const char* msg, const char* go_bears) {
    for (int i = 0; msg[i] && go_bears[i]; i++) {
        if (go_bears[i] != msg[i]) return 0;
    }
    if (!msg[i] && !go_bears[i]) return 1;
    return 0;
}
```

```
ISCOMMITOK: _____
               _____
               _lb_ $t0  __0_($a0)
               _lb_ $t1  __0_($a1)
COND:          _____ and $t2 $t0 $t1 _____
               _____ beq $t2 $0 EXIT _____
               _____ bne $t0 $t1 FAILED _____
               addiu $a0 $a0 1
               addiu $a1 $a1 1
               _____ j ISCOMMITOK _____
               _____
EXIT:          _____ _or_ $t2 $t0 $t1 _____
               _____ bne $t2 $0 FAILED _____
               li $v0 1
               _____ j END _____
FAILED:       li $v0 0
END:          _____ jr $ra _____
               _____
```

**Q5: MIPS Sleuth (15 points)**

mystery, a mysterious MIPS function outlined below, is written without proper calling conventions. mystery calls a correctly written function, **random**, that takes an integer *i* as its only argument, and returns a random integer in the range  $[0, i - 1]$  inclusive.

```

1  mystery:  addiu $sp $sp _____ -16 _____
2            _____ sw $s0 0($sp) _____
3            _____ sw $s1 4($sp) _____
4            _____ sw $s2 8($sp) _____
5            _____ sw $ra 12($sp) _____
6            _____
7            addu $s0 $0 $0
8            move $s1 $a0
9            move $s2 $a1
10 loop:    srl $t0 $s0 2
11          beq $t0 $s2 exit
12          subu $a0 $s2 $t0
13          jal random
14          sll $v0 $v0 2
15          addu $v0 $v0 $s0
16          addu $t0 $s1 $s0
17          addu $t1 $s1 $v0
18          lw $t2 0($t0)
19          lw $t3 0($t1)
20          sw $t2 0($t1)
21          sw $t3 0($t0)
22          addiu $s0 $s0 4
23          j loop
24 exit:    _____ lw $s0 0($sp) _____
25          _____ lw $s1 4($sp) _____
26          _____ lw $s2 8($sp) _____
27          _____ lw $ra 12($sp) _____
28          _____ addiu $sp $sp 16 _____
29          _____ jr $ra _____
30          _____

```

1) Fill in the prologue and the epilogue of this MIPS function. Assume that **random** follows proper calling conventions, and that it may make its own function calls. You may not need all of the lines.

2) What operation does this function perform on an integer array? Assume that both the integer array and the length of the array are passed into the function.

**The function shuffles the integer array in place.**

3) Would this function work as expected if a string was passed into the function instead? Write down the line numbers of all lines of MIPS code that must be changed (if any at all), so that the function works correctly on strings. Do not write down any extraneous line numbers.

**10, 14, 18, 19, 20, 21, 22**

**Q6: Registers: bigger is not always better (16 points)**

You decide that instead of having 32, 32-bit registers, you would like to build a machine with 16, 64-bit registers. You also need to make a modified MIPS instruction set for this architecture.

1) In the box below, specify the size of the fields to best utilize the 32-bit instructions on this new architecture. Do not modify the size of the opcode.

| opcode | rs | rt | rd | shamt | funct |
|--------|----|----|----|-------|-------|
| 6      | 4  | 4  | 4  | 6     | 8     |

| opcode | rs | rt | imm |
|--------|----|----|-----|
| 6      | 4  | 4  | 18  |

2) How many different R-type instructions can we now have? \_\_\_\_\_  $2^8$  \_\_\_\_\_

3) If PC = 0x061C, what is the largest address that we can branch to? \_\_\_\_\_ **0x8061C** \_\_\_\_\_  
 $0x61c + 4 + (2^{17} - 1) * 4 = 0x61c + 2^{19} = 0x61c + 2^{(4*4)} * 8$

4) Translate the following machine code into MIPS using your new field sizes. Use register numbers instead of register names, since we'd have to think of a new convention for the names...

0xAE9FFFF8

**101011 | 1010 | 0111 | 111...1000**

**sw \$7 -8(\$10)**

**Q7: After this, you're CALL done! (9 points)**

Connect the definition with the name of the process that describes it.

- Compiler
- Assembler
- Linker
- Loader

- Outputs code that may still contain pseudoinstructions.   a
- Takes binaries stored on disk and places them in memory to run.   d
- Makes two passes over the code to solve the "forward reference" problem.   b
- Creates a symbol table.   b
- Combines multiple text and data segments.   c
- Generates assembly language code.   a
- Generates machine language code.   b
- Only allows generation of TAL.   b
- Only allows generation of binary machine code.   c